1 Physical Unclonable Functions

Evaluation and Environmental Variation

A Introduction

Integrated circuits (ICs) are comprised in large part of transistors - switches that control the flow of electricity throughout the circuit. Modern transistors are *very small*, on the order of 10s of nanometers, and moving towards the single digits. Even if they are designed to be a specific size, such as 14 nm or 22 nm, they don't necessarily come out that size after production; they'll be close, but not exact. Every transistor, on the same IC, and on different ICs, will be just a little bit different. Importantly, these variations are too small for us to control, so every IC ends up just a *little bit* different from the others, even if they are made from the same design. This is called **process variation**.

The tiny variations in the transistor size will impact the electrical properties / characteristics of the transistor. One such property is the delay, or how long it takes the transistor to turn on or off. Transistors are arranged in different ways to implement different logic functions (NAND gates, NOR gates, etc.) and that switching time manifests as propagation delay - the time it takes to see an output change after one or more input signals change. Usually, the variations among individual components will not be very significant, and on the whole, most of the ICs produced from the same design will end up having the same overall behavior, within some acceptable tolerance. However, some circuits can actually be designed to *amplify* the effect of these tiny variations so that we can more easily measure them. In turn, we can use this principle to create a signature that is unique to that circuit, like a digital fingerprint. And because these variations are too small to be intentional, it is physically impossible to perfectly copy them into another IC. This is the basis for Physical Unclonable Functions, or PUFs, which are one of the basic "building blocks" or "security primitives" that are used in hardware security.

B Background

Now, we will explore the structure of one particular PUF - an Arbiter PUF (APUF), and characterize its behavior under different conditions. An APUF is a circuit consisting of two "racing paths" - a "top" path, and "bottom" path - that pass through a series of stages. At each stage, a multiplexor (MUX) can potentially swap the top and bottom paths; different paths include different transistors, and hence, have very slightly different delays. The input to the circuit branches off to the top and bottom paths; a pulse, or transition from logic 0 to logic 1, starts the race. The pulse travels along the two paths, occasionally swapping sides, until finally reaching an "arbiter" at the end of the line. The purpose of the arbiter is to decide which signal arrives first (which one "wins" the race). The arbiter is simply a D flip-flop (DFF). Recall that the DFF has two inputs - `D` and `CLOCK`, and one output, `Q`. If the pulse on the top path arrives first, this means that there is a logic 1 present at the D input, so by the time the pulse arrives at the clock input, a logic 1 is stored in the DFF. If the pulse arrives at the clock input first, this means that there is a logic 0 present on the D input, so a 0 is stored in the DFF. This result - a 0 or 1 - is propagated to the DFF's `Q` output - forming one bit of response for the PUF circuit.

Hence, the output of the circuit – 0 or 1 – depends on which signal arrives first to the arbiter. The slight differences in delay between the top and bottom path play a role in this. The other

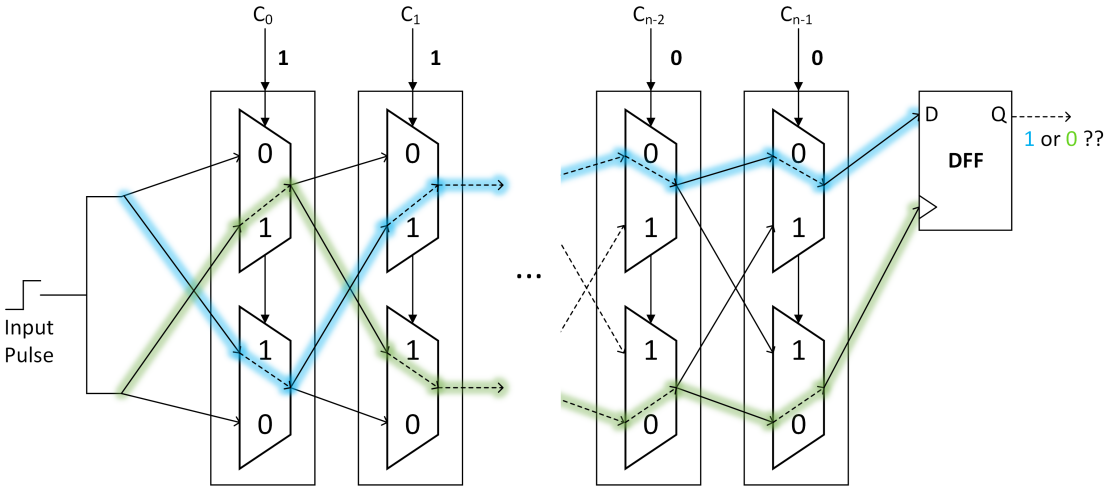


Figure 1: Example Arbiter PUF with n stages (0 to $n-1$). Challenge bits C_0 through $C_n - 1$ control whether or not the top and bottom paths swap at a particular stage. A single pulse travels through the circuit, experiences different delays through each stage. Ultimately, the arbiter will record either a 0 or 1, depending on which signal arrived first.

important aspect is the number of times the signal paths get swapped – top to bottom, or bottom to top. This is controlled by the MUXes in each stage. Each MUX has 3 inputs – the signal from the top path, the signal from the bottom path, and a *SELECT* input which decides whether or not to feed the top signal to the output of the MUX, or the bottom signal to the output of the MUX. This *SELECT* bit on each stage of the circuit is one of the *challenge* bits. For a PUF with 64 stages, there are 64 challenge bits, and 2^{64} different possible challenges. Because these challenge bits decide whether or not the signals swap paths, and the different paths have different delays at different stages, two different challenges could result in two different outputs – a 0 or a 1. Ideally, it should be unpredictable whether a given challenge will result in a 0 or 1 (but, as we will see in Lab 2, this is not necessarily the case!). The fact that there are many different possible challenges (2^n for an n-bit challenge) makes the APUF a kind of *strong PUF* that is useful for *authentication*, which is an important aspect of many security protocols. In order to use an APUF for authentication, it should first be characterized by undergoing an *enrollment* process. Essentially, this means providing a number of random challenge vectors to the PUF, and recording how it responds. In the future, the authenticity or identity of the chip can be confirmed by sending these challenges again, and confirming the response matches what has been previously observed during enrollment.

The *quality* of a PUF can be evaluated in two main ways: first, its *reproducibility*, and second, its *uniqueness*. A PUF with good *reproducibility* will output the same value (0 or 1) for the same challenge consistently *on the same chip*. On the other hand, if you take the same PUF design on a *different chip*, a PUF with good *uniqueness* will output a 0 or 1 with close to 50% probability, which would in no way be related to the output on the original chip. In other words, you should not be able to use information from one chip to predict the output on a second chip, even if you know the PUF design and have access to the PUF responses from the first chip. Both of these features - reproducibility and uniqueness - can be easily measured using the Hamming distance metric. In a binary string, Hamming distance (HD) refers to the number of bits that differ between two strings. For example, 00001111 and 11001111 have a HD of 2, because the first 2 bits differ. HD can also be reported in terms of a percentage; in this case, 2 out of 8 bits differ, so the HD is equal to 25%.

The output for the same challenge on the same chip should have a HD close to 0% - meaning it is highly reproducible. This is also known as the *intra-chip Hamming distance*. The output for the same challenge on two different chips with the same PUF design should have a HD close to 50%. This is also known as the *inter-chip Hamming distance*. A strong PUF should, at the very least, have good reproducibility and uniqueness.

For example, take two FPGAs, FPGA0 and FPGA1, and a 64-bit challenge vector c. If c_0 is applied to the PUF in $FPGA_0$ three times, the responses may be `0xF2A4`, `0xF2A5`, and `0xF3A4`. If c_0 is applied to the PUF in $FPGA_1$ three times, the responses may be `0xB31A`, `0xB30A`, and `0xB71A`. To find the intra-chip HD, compute the pairwise HD on $FPGA_0$ by computing $HD(0xF2A4, 0xF2A5) = \frac{1}{16} = 6.25\%$, $HD(0xF2A4, 0xF3A4) = \frac{1}{16} = 6.25\%$ and $HD(0xF2A5, 0xF3A4) = \frac{2}{16} = 12.5\%$. Hence, the average intra-chip HD is about 8.3%. The same can be done for $FPGA_1$, and the results can be averaged to measure the intra-HD performance across all chips. Next, three *different* challenges c_0, c_1, and c_2 can be applied to $FPGA_0$ and $FPGA_1$ to obtain a response set r_0, r_1, and r_2 for $FPGA_0$ and $FPGA_1$. A pairwise comparison can once again be performed, computing the HD between $FPGA_0$ r_0 to $FPGA_1$ r_0, $FPGA_0$ r_1 and $FPGA_1$ r_1, and so on. Averaging the computed HDs will yield the inter-chip HD for the PUF.

C Hardware and Software Tools

This experiment requires the Cmod S7 board, the PHS-VM (Appendix C), and the provided lab module files stored in "PHS-Lab-01.zip." The lab module files include a starting Jupyter Notbook scripts with details on how to connect and program the board. Two bitfile `APUF_CmodS7.bit` are available in the assignment directory. This file contains an implementation of a 64x16 APUF, which takes as challenge 64-bits, and returns a 16-bit response. The bitfile also implements a universal asynchronous receiver / transmitter (UART) module that understands a few basic commands which allows you to interface with the APUF from a program on your computer. Implementing an APUF in an FPGA is a bit tricky - if the paths between stages are different lengths, this could result in bias, which would make the output *less* unpredictable. The provided implementation makes use of constraints which try to limit this bias, but it is not bias-free; hence, it is meant for educational purposes only, and should not be used in projects where a secure, unbiased FPGA-based PUF is required.

To send and receive data from the PUF, connect the Cmod S7 board to your computer via USB. This development board has an IC that converts USB messages to Serial messages that can be read by a Universal Asynchronous Receiver / Transmitter (UART) circuit. Be sure to connect to the USB-UART port on the development board - this in turn connects to a USB to UART conversion chip on the development board, which interfaces with the FPGA itself. The provided PHS-VM should have the appropriate FTDI VCP drivers installed; this will create a Virtual Port that allows you to send messages using a program on your computer (for example, PuTTY) or through a programming language and a serial library. You will use the latter - PySerial, in this case - to send challenges to the APUF, and read back the response. For more details on communicating with the Cmod S7 board, see the provided Jupyter Notbook scripts in the "PHS-Lab-01.zip" file.

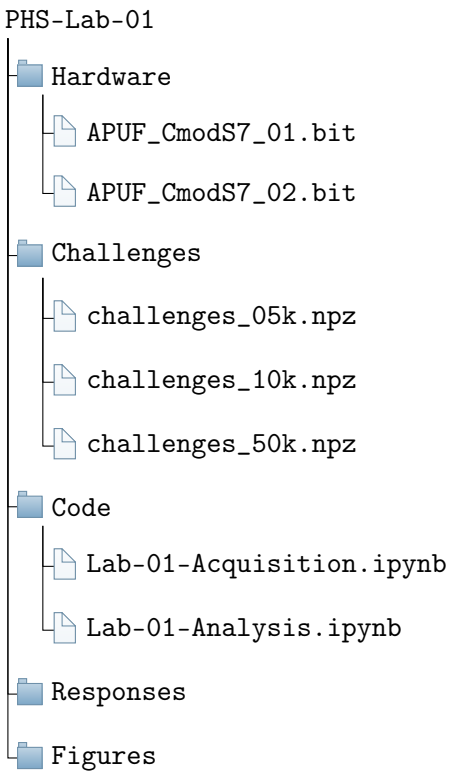
Once programmed, the implemented state machine first enters a reset state (S0), then waits to receive 8 bytes for the challenge, 1 byte at a time (S1). Once all 8 bytes are received, the challenge is loaded into the PUF circuit (S3). A single rising edge pulse is applied to the signal input of the PUF (S4). The state machine then enters a wait state to allow the PUF output to settle (S5). The output is then read and transmitted back to the PC over the serial connection (S6). Finally, the device returns to S1 to await another 8 byte challenge.

Reading Check

1. What is the purpose of the arbiter in the APUF?
2. Why might the PUF response bits change when different challenges are applied to the *same circuit* on the *same FPGA*?
3. Why might the PUF response bits change when *the same* challenges are applied to the *same circuit* on different FPGAs?

D Getting Started

Upload and unzip the "PHS-Lab-01.zip" file in the corresponding directory with the same name. The directory is structured as follows:



The **Hardware** directory contains 2 *bit* files that can configure the FPGA. Each bit file includes the same APUF circuit, just shifted into a different physical placement in the FPGA. **Undergraduate** students only need to implement this lab on "APUF_CmodS7_01.bit." While, **Graduate** students needs to implement this lab on *both* APUFs. We will use Digilent Adept 2 v2.26.1 tools pre-installed in the PHS-VM to configure the FPGA with these *bit* files.

The **Challenges** directory contains 3 *npz* files that contains a set of challenges to be sent to the APUF. The value on each the challenge's file name indicates the number of challenges such file have. For instance, "challenges_05k.npz" have 5,000 (5k) challenges, "challenges_10k.npz" have 10,000, and "challenges_50k.npz" have 50,000. While testing, you can just use any (5k is faster to load/run), but **your final results for your report must use the 50k**.

The **Code** directory contains the Jupyter Notebook starting scripts that you need to complete. You will use these script to send challenges to and collect responses from the FPGA (aquisition), then generate plots and statistical results for your report (analysis).

Store all your response files collected from "Lab-01-Acquisition.ipynb" in the **Responses** directory and all your figures generated from "Lab-01-Analysis.ipynb" in the **Figures** directory.

E Lab Assignment

The assignment will consist of writing 2 Jupyter Notebook scripts - one to characterize the APUF, and one to analyze the results. When saving lists to file, it is convenient to convert to a numpy array, and use the `savez_compressed` function to store the array as a npz file. You must use the provided `challenge_x.npz` files found in the Challenges directory.

Part A: Acquisition Write a script using the PySerial library that sends challenges to the APUF over a serial connection, then reads back and stores the response using the `read()` function. Be sure to specify the number of bytes to read back.

1) Connect and configure the FPGA with the provided "APUF_CmodS7_01.bit" file found in the Hardware directory.
2) Create an empty `response` list. Send each challenge to the PUF, read back the response, and append it to the list. Use any challenge file while testing, but your final deliverable must use the "challenges_50k.npz" file. It may take 10-20 minutes to get all 50k responses, while only 1-2 minutes for the 5k.
3) Save this list to file. *Ensure your specify a unique filename.*
4) Repeat this process two more times, so that you have a total of three response files (r_0, r_1, r_2) per FPGA ($FPGA_0$, $FPGA_1$) generated using the same challenge set. Total of 6 response files: "fpga0_resp0.npz", "fpga0_resp1.npz", "fpga0_resp2.npz", "fpga1_resp0.npz", "fpga1_resp1.npz", and "fpga1_resp2.npz". For instance, "fpga1_resp2.npz" contains the response file r_2 for $FPGA_1$.

Part B: Analysis Write a second script using Python that analyzes this data to assess the reproducability and uniqueness of the PUF. Your code should perform the following functions:

1) Load the saved response lists, e.g. "fpga0_resp0.npz", "fpga0_resp1.npz", "fpga0_resp2.npz", "fpga1_resp0.npz", "fpga1_resp1.npz", and "fpga1_resp2.npz".
2) Using `scipy.spatial.distance.pdist`, find the Hamming distance (HD) between responses. Graph the result as a histogram using `matplotlib.pyplot.hist` or `matplotlib.pyplot.bar`.
3) Swap response lists with your teammate. For the same challenge, find the HD for each response. Graph the results as you did in the previous step.

Graduate Only: Repeat both parts but program the FPGA using the "APUF_CmodS7_02.bit" instead.

F Badges and Achievements

This experiment involves **Machine Learning**, **Data Processing**, and **Data Visualization**. By successfully completing the lab you can unlock the following achievements:

- FGPA1: Machine Learning
- COM1: Serial Communication
- FILE1: File Manipulation
- STAT1: Statistical Analysis
- PLOT1: Data Visualization

G Deliverables

Follow all report guidelines as detailed in Appendix A. The following result(s) must be included in the report:

- Answers to the reading check questions.
- Average and standard deviation intra- and inter-chip Hamming distance
- Histogram showing intra- and inter-chip Hamming distance

Submit your report along with all code and your response list. All analysis code should be runnable and should produce the output submitted with your report.