Hardware Security (NWI-IMC065)

Tutorial: Physically Unclonable Functions

Goals: After completing these exercises, you should be able to:

- Simulate PUFs in software.
- Evaluate the quality of PUFs.
- Understand the impact of noise on the PUF behavior.

Handing in the tutorial report:

- You should upload your report to Brightspace by the end of the tutorial.
- You can work in teams of at most **two**. Ensure that your name and student numbers for both team members are mentioned on top of the first page.
- The report should contain your answers to the exercises in this tutorial. It is okay if you did not finish all the exercises; we want a best-effort approach. The report is not graded.

Before you start: You need to install the `pypuf` package. This package will be used in all the tasks of the tutorial. You can install it using the following command
`pip3 install pypuf`

Follow the documentation provided in `https://pypuf.readthedocs.io/en/latest/` for more details.

1 PART A

Simulation and Evaluation of Arbiter PUF (depicted in Figure 1) using `pypuf`.

Task 1: Simulate Arbiter PUF instances and generate challenge-response pairs.

1. Create a Python file named "arbiter_puf.py".
2. Import the necessary classes:

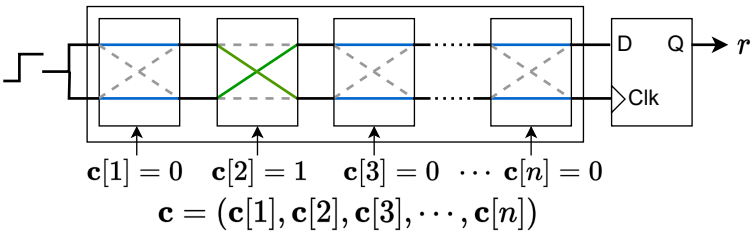

Figure 1: n-bit Arbiter PUF

```
from pypuf.simulation import ArbiterPUF
from pypuf.io import random_inputs
```

3. Simulate an instance of the Arbiter PUF that takes a 64-bit challenge using:
   `puf1 = ArbiterPUF(n=64, seed=21, noisiness=0)`

   where the seed is the sum of your student number digits (replace '21' with the sum of your digits). Set the 'noisiness' parameter to 0 for noiseless responses.

4. Create another instance by changing the seed to the sum of the last two digits of your student number. For example:
   `puf2 = ArbiterPUF(n=64, seed=11, noisiness=0)`

5. Generate 10,000 random challenges, each 64 bits in length:
   `challenges = random_inputs(n=64, N=10000, seed=7)`

6. Feed the challenges to both PUF instances:
   ```
   resp1 = puf1.eval(challenges)
   resp2 = puf2.eval(challenges)
   ```

7. Compare the responses by computing their Hamming distance (HD). To do this, create a function `puf_hd()` that computes the normalized HD by dividing the number of differing responses by the size of the challenge set. Note that the response generated by `pypuf` are -1 or 1. To compute Hamming distance, you can either convert the responses to $\{0, 1\}$ or compute the number of challenges for which the responses differ, without the conversion.

   ```
   def puf_hd(resp1, resp2):
       return ...
   ```

   This will give the normalized Hamming distance between the two response sets.

Task 2: Analyze the Statistical Quality of the Responses Produced by the APUF Instances.

In this task, we compute three key performance metrics of PUFs: *uniqueness*, *uniformity*, and *reliability*. We will use these metrics to assess the quality of the responses generated by the simulated APUFs. *Mention these metrics in the report along with the design parameters (challenege length and number of challenges)*

1. *Uniformity:* The uniformity metric measures the balance of 1's and -1's in the PUF responses. Compute the fraction of 1's and -1's in the responses from both instances.

2. *Uniqueness:* Compute the uniqueness of the PUF responses by measuring the Hamming distance (HD) between different instances. For example, generate 5 PUF instances with different seeds and compute the uniqueness:

   ```
   from pypuf.metrics import uniqueness
   instances = [ArbiterPUF(n=64, seed=seed) for seed in range(5)]
   uniqueness_value = uniqueness(instances, seed=31415, N=5000)
   ```

   Then, increase the number of instances to 50 and compute the HD between every pair of instances. The number of pairs for 50 instances is $\binom{50}{2} = 2450$. Plot the frequency distribution (histogram) of the HDs. *Add the histogram and mention the distribution in the report.*

3. *Reliability:* Reliability is a measure of how consistent the PUF responses are when repeated measurements are taken under the same conditions. High reliability means that the PUF produces the same or similar responses across multiple measurements of the same challenge. To calculate the reliability:

   - Simulate a PUF (for example, with noise rate `noisiness=0.1`) to introduce noise into the responses. We refer to it as `noisy_puf`.
   - Generate a set of 20,000 random challenges (referred to as `chal` in the code).
   - For each challenge, repeat the measurement 20 times (e.g., 20 different response measurements for each challenge) and store the responses in a 2-D array.
   - Perform *majority voting* over the first 15 measurements to compute a reference response for each challenge. The reference response is the most frequent response (either 1 or -1) for each challenge across the 15 measurements. This majority-voted response is referred to as the *reference response* or the *golden response*. Refer to it as `reference_response` in your code. The reference response is considered the "true" response under ideal conditions (i.e., without noise). In practice, it is not possible to compute completely noiseless responses, hence, this is a standard practice to generate the reference response. Here the number of measurements considered in the computation is directly proportional to the stability of the response.

- Once the reference response is computed, compute the Hamming distance between the remaining 5 measurements and the reference response. This gives a measure of how different the deployed responses are from the reference response.
- The average HD across all challenges and remaining measurements gives the reliability score. A lower average HD indicates higher reliability, meaning the responses are consistent across multiple measurements.

Task 3: Understand the Contribution of the Number of Measurements on PUF Reliability. Temporal majority-voting is a standard practice to improve the reliability of responses. Herein, multiple measurements are collected and the majority-voted response is used in applications. In this task, we will analyze how the number of measurements in majority-voting affects the final reliability of the PUF response by comparing it with the "true" response. For this, we continue with `noisy_puf` instance and the `reference_response`.

1. Compute the temporal majority-voted response for `noisy_puf` over set `chal` across 5, 15, and 25 measurements. For this, you need to re-generate responses for 5, 15 and 25 measurements. These correspond to the responses generated post deployment.
2. Compute the Hamming distance of the majority-voted response (in the previous step) from the `reference_response` (the reference response generated in Task 2).
3. Plot the HD to show how the reliability improves as the number of measurements increases.

2 PART B

Simulation and Evaluation of XOR Arbiter PUF (depicted in Figure 2) using `pypuf`.

Task 4: Performance evaluation for XOR Arbiter PUF

1. **Simulate 10 instances of k-XOR Arbiter PUF for different values of k.** The values of k to be considered are $k = 2$, $k = 4$, and $k = 8$. Each PUF will take a challenge of length 64 bits, and the noise level will be set to 0.1.
2. **Report:** For each value of k (i.e., $k = 2$, $k = 4$, $k = 8$), report the following for a set of 20000 challenges:
   - The uniformity of the responses, the fraction of 1's and -1's.
   - The uniqueness across all pairs of 10 instances.

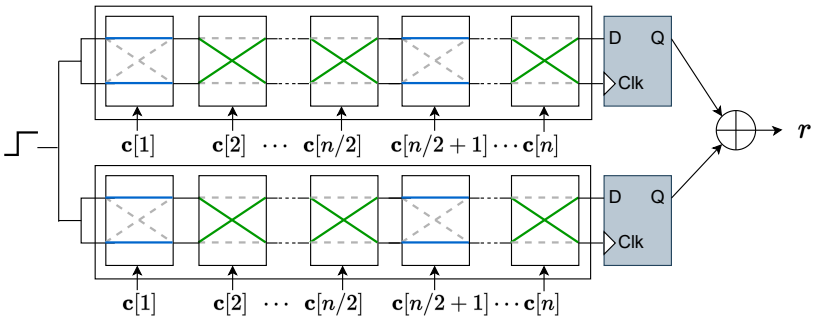

Figure 2: XOR Arbiter PUF

- The reliability based on the Hamming distance between the majority-voted reference response and 5 separate measurements. Compute the reference response over 15 measurements.

Provide a brief description on the observed, noting how the value of k affects the performance of the XOR Arbiter PUF, particularly in terms of reliability.