Practical Introduction to Hardware Security

Task 1: Physically Unclonable Functions

Chair of Dependable Nano Computing, Karlsruhe Institute of Technology
Instructors: Dennis Gnad, Jonas Krautter, Mehdi Tahoori

October 31, 2019

1 Introduction and Supplementary Information

Physical Unclonable Functions (PUFs) are used in various security applications. The lecture looked into generally describing PUFs, concentrating on the idea of *challenges* and *responses*.

PUFs that are based on challenges and responses are called *strong PUFs*, because they should be able to reply to a significant number of challenges (often exponential) with an unpredictable response. However, for many implementations it is still often possible to generate a prediction model for responses to unknown challenges (i.e. based on machine learning), if enough challenge-response pairs are collected. For more details see III. in [1].

Another category are *weak PUFs*. They are called weak, because they typically just offer a single response. In that way, they can still be used as a hardware-based unique identifier, often only accessible when the chip is powered up. In this lab you will implement such a weak PUF, based on the power-on state of SRAM memory. The bits of SRAM flip to either 0 or 1 during power-up of the transistors in the SRAM. However, typically some bits always flip to just 0 or just 1. These bits can then be used for PUFs.

More details on weak and strong PUFs can be found in sections II. and III. of [1].

1.1 Examination of this and following Tasks

When working on these tasks, feel free to discuss it with your classmates. But you need to understand every part of what you implemented, since we will ask you on that.

After the task is finished, you will have to hand it in by sending us all your material and appear in the lab. You should then explain us all the steps that you implemented, if possible by actually showing how/that it works. That means, in this first task you have to send the finished hardware and software project source files and the final evaluated results for the four metrics. That means, send both the averages of all four metrics, as well as the detailed results.

1.2 Questions / Help

Do it! If you are really stuck and can't find your problem, it is usually better to ask us for help. As in the initial tests, feel free to write mails / come by to get assistance.

2 Hardware Project

We supply the required source files which still have to be completed. This project already has a combined module that makes all SRAM of the FPGA accessible through one addressing scheme. Additionally, a rough outline for the state machine to transfer the data to the host PC is available. This state machine has to be completed accordingly.

2.1 Source Code Task

Please check the `puf_module.v` file and search for `TODO`, and finish all the TODO items, following these recommendations:

1. Fulfill some basic TODOs required for initializattion (TODO-BASIC items)
2. Make the UART communication work by sending dummy data to the PC (TODO-UART items).
3. Connect the SRAM to the design and send that data to the PC (TODO-SRAM items). Consider that we do not need to write to the memory.
4. Check that the UART transmission works correct from start to end (no repeated or zero bytes at the start/end of the overall data transmission).

Please note, you might not need to fill all of "???" (but we needed to fill most of them in our sample solution).

2.2 Adjusted Bitstream Packing Task

Secondly, we have to compile our bitstream in a specific way, to not initialize all SRAM cells to '0', to be able to use them as a PUF. The open-source FPGA tools are per default initializing the SRAM to '0', thus we have to use a patched version of the sub-tool 'icepack'. A pre-compiled version is included, please either use it in the Makefile by giving its relative path (i.e. use "./icepack" in the Makefile instead of just "icepack"), or add it to your PATH.

Alternatively you can compile a recent icepack version from git yourself, which already contains this feature. Git repository: https://github.com/cliffordwolf/icestorm

After you integrated this new icepack, **please adjust the Makefile entry** below the line `%.bin: %.asc` with an additional icepack parameter to skip initializing BRAM (which is based on SRAM). You will find the required switch by running the patched icepack binary with the `--help` command-line parameter.

2.3 Note on the UART transmission to the PC

Sending with the UART TX simply works by waiting until it is ready. Then it can be enabled while the data to transmit is kept stable at its TX data input, until it signals to be ready again.

3 Software for Communication

We supply a basic python script that use the usb-to-serial port on the Lattice board. Please use a respective python installation with the serial package (in Linux "`sudo apt install python3-serial`", in Windows you can for example install Anaconda Python 3.x, and make sure the respective serial module is installed). The script might need to be adjusted for the correct COM/ttyUSB port.

The script basically sends an 's' to your FPGA board and receives the uninitialized SRAM data as the reply. This reply is split into 512 blocks of 256 bit each. Each 256 bit are saved as lines in the text file of 32 byte in ascii hex characters (=64 characters, since each represents a 4bit nibble). You can also save the file as a binary file, if it suits your following statistics implementation. In the end, each of these 512 blocks should be considered as one PUF instance, where each has 256 bit.

To later get the PUF *Reliability* metric, you need to collect data with this script multiple times. We know that this involves some manual effort, so we suggest the following:

1. You will see that your implementation works if you receive a similar file as the puf_data-_example.txt that we already added. If in doubt, just send us one of your puf_data.txt files.

2. When everything works, program the flash memory of the board (taking significantly longer time than the SRAM). For that, **while the board is disconnected from USB**, connect the left jumper and rotate the two right jumpers 90 degrees (also explained in the board documentation). Then, set your software accordingly to program the flash memory. In Linux with icestorm-iceprog, just use the 'iceprog' command without the '-S' flag, which works also by using 'make prog-flash'. For Lattice Diamond in Windows this is more complicated, please contact us if you can't get iceprog to work.

3. Do the following procedure 20 times: 1. Connect the board to the PC 2. Launch the script python script and save the puf_data.txt to a new file (or adjust the script respectively) 3. Disconnect the board from the PC, 4. back to 1.

This procedure is required, because the SRAM can initialize differently on each power-up. For a good reliability metric usually more than 20x is required, but in this lab we consider it sufficient. Please see the next point on how to get the metrics.

4 Software for Statistics

In this task, it is on your own to analyze the collected PUF data. You can choose your favorite way to do that. That means, the programming language or statistics software. Keep in mind that you handle each bit separately, not just each byte. A good overview of the statistics can be found from Section 3.2. in the paper by Maiti et al. [2].

For software, we suggest to consider writing a program in C++, Python, Perl or Matlab if you are familiar with that – python is probably a reasonable choice if you don't know better. But feel free to choose any other favorite programming language. Consider that bit and byte operations on individual bits are not easily handled in all languages. Also remember you have to show it to us in the end. So please make it in a way that it can be easily explained to us by you.

So what you have to do:

- Please read from Section 3.2. onwards in the paper by Maiti et al. [2].
- Consider each line of all 20 files of puf_data.txt(or similarly named) to contain one PUF of 256 bit. (Sometimes called "one chip")
- That means, a specific line number across all the files contains the data from that same PUF. You will realize that some bits/bytes are changing from readout to readout because they are unstable/unreliable. Thus, to get an impression of the *reliability* metric, use a diff tool that shows individual bytes (e.g. "diffuse", available for Windows/Linux/Mac)

  You can use that diff tool to compare two of the puf_data.txt files to see the differences on a byte-level. For example, you can compare puf_data_example.txt and puf_data_example2.txt.
- Read this data into your software and process the metrics of *Uniqueness*, *Reliability*, *Uniformity* and *Bit-Aliasing*.
- Next to the detailed results per metric, please also summarize your results with an average Uniformity, Bit-aliasing, Uniqueness, and Reliability.

Again, to clarify. When it is talked about "intra-chip" it means for you across multiple acquired puf_data.txt files (where you power-cycled the board inbetween). When it is talked about different "chips" or "devices", it means different lines in one file (then just repeat the whole statistic for all the files). We essentially "simulate" different multiple chips on one, by dividing the entire available SRAM into multiple PUF instances/"chips"/"devices".

For getting an idea if your final metric results are reasonable, please see Table 6 in [2] for expected *Ideal Values*, where your SRAM PUF should also not be too far away from.

References

[1] U. Rührmair and D. E. Holcomb, "Pufs at a glance," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 347. [Online]. Available: https://spqr.eecs.umich.edu/papers/holcomb_PUFs_date14.pdf

[2] A. Maiti, V. Gunreddy, and P. Schaumont, "A systematic method to evaluate and compare the performance of physical unclonable functions," in *Embedded systems design with FPGAs*. Springer, 2013, pp. 245–267. [Online]. Available: https://eprint.iacr.org/2011/657.pdf