Experiment 7
Hardware-based Security Primitives and Their Applications
on HaHa v3.0 Board

We introduce two most important hardware-based security primitives: physical unclonable functions (PUFs) and true random number generator (TRNG). We also introduce the applications of these security primitives to defend counterfeiting attacks at the chip level.

Instructor: Dr. Swarup Bhunia
Co-Instructors/TAs: Reiner Dizon-Paradis, Chris Vega, and Shuo Yang

Theory Background

1. PUF

Physical unclonable function (PUF) is an emerging potential security primitive for generating volatile secret keys in cryptographic applications. There are different types of PUFs implemented in FPGA [1-8]. A PUF is described as unclonable due to its uniqueness being derived from the uncontrollable variations introduced during the manufacturing process. PUFs offer a high level of protection in cryptographic applications with strong volatile key storage. PUFs are issued a challenge and (ideally) produce a unique and reliable response in return. Since this response is unique to the device, it can, therefore, be used as a device ID or key. In this experiment, we will learn two most popular PUFs: SRAM-based PUF and ring-oscillator (RO)-based PUF.

1.1 SRAM-based PUF:

The usage of SRAM as the PUF medium is appealing for a variety of reasons. Most notably, SRAM is commonly available in most systems and therefore does not require additional hardware. For a standard 6T SRAM (Figure 1), every memory cell is composed of six transistors that are two cross-coupled CMOS inverters and two access transistors. The inverters are designed to be symmetric, match in size, etc., but random variations incurred during manufacturing will result in random mismatches. SRAM PUFs exploit the mismatch which results in each SRAM cell being biased (or skewed) toward a zero or one at power-up. Due to uncontrollable variations in the manufacturing process, different CMOS devices have different physical parameters (e.g., doping-levels, transistor oxide thickness, etc.). When an SRAM is powered-up, these variations affect the power-up state of their associated cells. It has been observed that certain cells have a strong preference to power-up to a '1' or '0' state. Cells that have no preference are deemed neutral and power up at random depending upon the influences of system noise [5]. The more useful cells for PUF output are the ones that strongly prefer '0' or '1'.

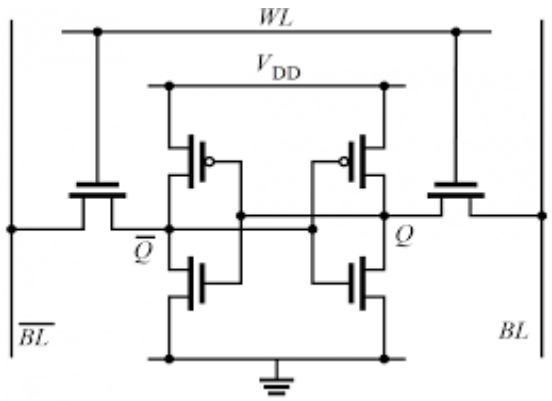

Figure 1 SRAM memory cell used as PUF

1.2 RO-based PUF:

Figure 2 shows a conventional RO-based PUF which generally consists of N identically laid out ROs, two counters, a comparator, and two N-bit multiplexers. Each RO consists of an odd number of inverters and oscillates with a particular frequency. Because of process variation and inherent random noise, each RO oscillates with a slightly different frequency than others even though they are designed to be identical. A response bit is generated by comparing the frequency of two identical ROs, where one RO is faster than the other and generates one (or zero) by a comparator. A pair of ROs is chosen, in order to generate a bit, by applying a digital sequence of length log2(N), known as a challenge. The applied challenge in both MUXs selects RO from each MUX and forms a pair. The counters connected to the MUXs count the number of oscillations of each of the paired ROs for a fixed time interval. This time interval is known as comparison time and impacts the reproducibility of a key. Usually, the comparison time needs to be larger when the difference between the frequencies of ROs in a pair is very low. The comparator generates a bit from the counters' outputs by comparing the corresponding values and set `0' or `1' based on which oscillator from the selected RO pair is faster, i.e., has a greater count for the given comparison time.

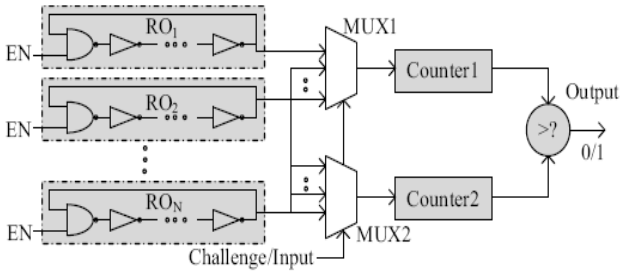

Figure 2 RO-PUF

1.3 PUFs Evaluation:

Reliability: The Reliability of a PUF determines how often a PUF can generate the same response to a given challenge. A PUF must generate the same response at all operating conditions.

Uniqueness: Uniqueness measures how well a single PUF is differentiated from other PUFs based on its challenge-response pair. Different PUFs must generate different responses for a given challenge in order to separate one from another. The average inter-chip fractional hamming distance for an ideal PUF must be 0.5. If a PUF circuit is instantiated on several different chips, then each of the PUF instantiations is expected to produce unique responses when supplied with the same challenge C. In general, C is an m-bit binary input which is supplied to the each of the PUF instantiations. The responses R1, R2, R3, and R4 are n-bit binary strings (Figure 3).

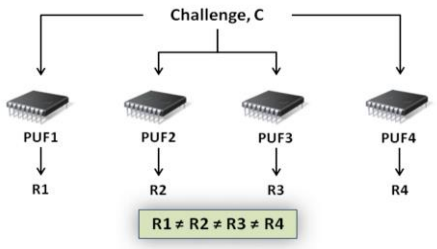

Figure 3 PUF Uniqueness

Randomness: Entropy can be used to measure the randomness of a PUF. It's the reflection of the amount of 1s and 0s in a sequence.

Diffuseness: The diffuseness, as discussed earlier, is very similar to the uniqueness. How responses of the same PUF are different for different challenges.

2. True Random Number Generator:

2.1 TRNG and an example: RO-based TRNG

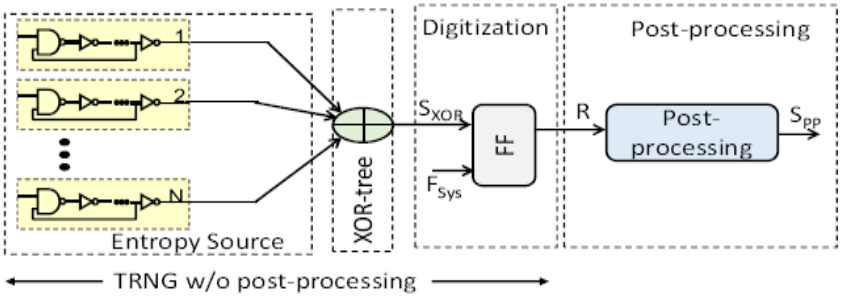

Figure 4 RO-based TRNG

A random number generator (RNG) is an important security block widely used in most cryptographic applications such as one-time pads, session, and temporary keys, nonce, seeds, challenges for authentication, zero-knowledge protocols, hardware metering, generation of primes, secure communications, secured servers and processors, VPN access, and customer-facing web access. A quality RNG generates statistically independent and unpredictable sequences of random numbers. Compromising an RNG often means compromising an entire system. A true RNG (TRNG) translates random physical phenomena such as thermal noise, atmospheric noise, shot noise, radio noise, flicker noise, clock jitter, phase noise, noise in a compact memory etc. into random digits. A TRNG must have uniform statistics; non-uniform statistics due to biased random sequences help attackers to guess the random numbers. Generally, random numbers are generated by comparing two symmetric devices which possess some process variation (PV) and random inner noise [9-11].

TRNG consists of a source of randomness (entropy) and a randomness extractor. Metastability of logic cells and timing jitter of Ring Oscillators (ROs) are the most common sources of entropy for a TRNG. The simplicity of implementation and entropy collection have made the RO-based TRNG most popular. Figure 4 shows a conventional RO-based TRNG where the RO outputs are combined by an XOR-tree. The output of the XOR-tree, S_{XOR}, is sampled constantly by a synchronous D flip-flop driven by the system clock to convert RO jitter into a random digital sequence. Jitter, in this case, represents the deviation from ideal RO behavior caused by random process variation and temporal variations such as random physical noise, environmental variations, and aging. Without jitter, the ROs will possess almost identical phase and the output of XOR-tree would be almost constant which is undesirable. Among the variations, only the random physical noise (thermal noise, atmospheric noise, shot noise, radio noise, flicker noise, etc.) improves the randomness/entropy of the bitstream output by the TRNG.

2.2 TRNG Evaluation:

Unlike the PUF output, the TRNG output has to be different from measurement to measurement so that attacker cannot guess future output from previous TRNG output. NIST test is used to measure the randomness of a TRNG output [4]. NIST's statistical test suite is popularly used to evaluate the quality of randomness for random and pseudorandom number generators designed for cryptographic applications and can also be used for PUF [9-11]. There is a total of 15 NIST tests and different tests require a different minimum length of bitstreams. For example, rank test, linear complexity test, and overlapping template matching test require at least 38912, 106, and 106 bits long bitstreams respectively. On the other hand, frequency test, block frequency test, and runs test require a minimum of 100-bit long bitstream

3. Application of PUFs in Supply Chain Integrity

PUFs can serve as a root of trust and can provide a key which cannot be easily reverse engineered. One of the biggest issues in any supply chain management is being able to verify that the product ordered is the product received. The electronics industry makes great strides to verify the integrity of their product lines. For example, the distributor your group used to complete the Bill of Materials for the previous experiment, Digi-Key, is a member of the trade association Electronic Component Industry Association, or ECIA, which strives to ensure that no counterfeit devices enter the supply chain. However, one can't always order from verified suppliers. In this case, it is necessary to order components from unauthorized resellers, increasing the risk of acquiring counterfeit goods. One emerging research area to counteract this problem is in the area of Physically Unclonable Functions. The gist of the idea of a PUF is that the manufacture of the goods will use some sort of intrinsic process variation to generate a unique signature for every device. This signature is entered into a database so that the end user can, using the same signature generation process, generate and verify the signature of their device.

Applications of PUF includes cryptographic key generation memoryless key storage, device authentication, PUF-based RFID for anti-counterfeiting, Intellectual Property (IP) protection etc.

Experiment Set-up: Configuration

The instruments needed for this experiment are the HAHA V3.0 Board, a USB A to B cable, and a computer. Figure 5 shows the instruments connections.

The software needed to program the microcontroller is avrdude. Use GOWIN FPGA Designer to program the FPGA. Python is recommended to process the data.

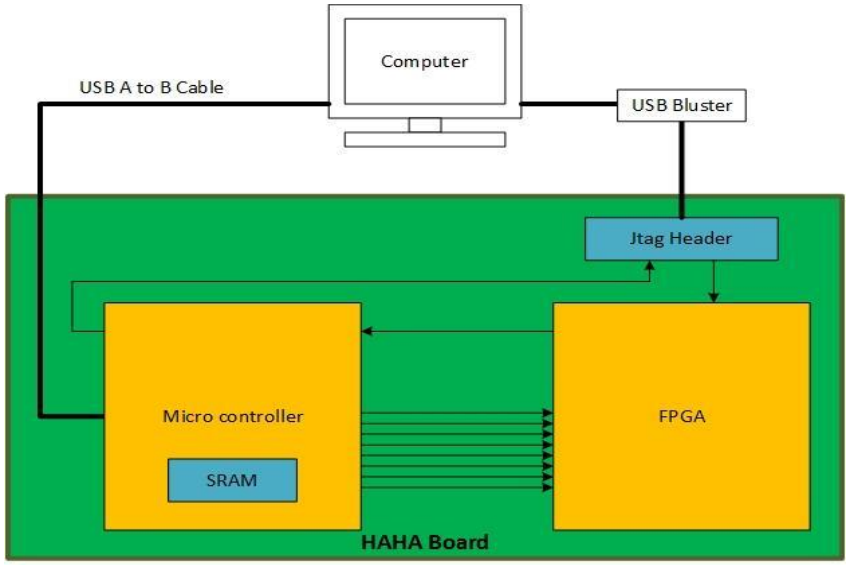

Figure 5 Experiment set-up.

Instructions and Questions

Part I SRAM PUF

In this part, you will implement an SRAM PUF on the HAHA V3.0 Board. Read 64 bytes of the SRAM power-up states from the Microcontroller to generate a signature.

Program the Microcontroller. It needs to do two things: read and send. Refer to the website to find the address range for the SRAM inside the Microcontroller. In this range, read 64 bytes power-up states. Note that when the Microcontroller is running, some SRAM addresses will be occupied, and these addresses will no longer hold their power-up states. Therefore, chose the addresses for your 64 bytes carefully and make sure they won't be occupied by running your program.

Print the values to the terminal over UART.

Process your data with a custom script (eg. Python, etc.), according to the requirements given below and answer questions.

1) Refer to the datasheet of the Microcontroller, what is the size of the SRAM in it?
2) You should read the power-up states of the SRAM cells, i.e., the values are not changed by the program after power up. Can you tell what addresses will be occupied by your program? Find an address to use for your 64-byte signature? (Hint: for affected addresses, their contents will look "much less random" than the power-up values.)
3) Indicate the address you used in your report.
4) Commit your code for the Microcontroller (should be in lab2/part1/main.c).
5) Copy the terminal output to your analysis script and perform analysis as described next.
6) The first 64-byte value you collect is called signature S1. Include your S1 in the report in hex format.
7) For S1, how many of the bits are 0 and how many are 1? What is the mean value? (eg. what percentage of bits are high?

8) Power OFF the whole board when taking different readings. Leave the board off for 30 seconds between readings. Power ON the board and read out the signature again. Repeat this two more times. These will be your signature S2, S3, and S4. Include thse in the report in hexadecimal format.
9) Are S2, S3, and S4 the same with S1? For each new sample, how many bits are different? What is the fractional ***intra*** hamming distance? [6] (ie. percentage of bits that flipped)
10) Use the compressed air to change the temperature of the microcontroller. Take another sample, S5. (Make sure to cool the microcontroller while powered off by blowing air for ~10-15 seconds, then turn it on).
11) Again report how many bits are different than S1, and the fracitional intra hamming distance.

12) Turn in your code for calculating the mean value and the ***intra*** hamming distance for this part.

Part II SRAM as a TRNG

In this part, you will modify your design for Part I so that it can work as a TRNG instead of a PUF. You can repeat your steps in Part I to collect multiple S1's. See if all the bits of S1 will stay the same whenever you collect them and find if there are bits that act more randomly than others. Use these bits to generate an 8-bit TRNG.

1) What type of SRAM cells is a good candidate for TRNG? Select 8 bits from them. Include their addresses in your report and explain the procedure that you used to find them.
2) Write code to generate an 8-bit random number with your SRAM in the Microcontroller. Turn in your code.

References and Further Reading

[1] http://rijndael.ece.vt.edu/puf/background.html

[2] https://en.wikipedia.org/wiki/Hamming_distance

[3] Gassend, Blaise, et al. "Silicon physical random functions." Proceedings of the 9th ACM conference on Computer and communications security. ACM, 2002.

[4] Rukhin, Andrew, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Booz-Allen and Hamilton Inc Mclean Va, 2001.

[5] Holcomb, Daniel E., Wayne P. Burleson, and Kevin Fu. "Power-up SRAM state as an identifying fingerprint and source of true random numbers." *IEEE Transactions on Computers* 58.9 (2009): 1198-1210.

[6] Böhm, Christoph, Maximilian Hofer, and Wolfgang Pribyl. "A microcontroller sram-puf." *Network and System Security (NSS), 2011 5th International Conference on*. IEEE, 2011.

[7] Suh, G. Edward, and Srinivas Devadas. "Physical unclonable functions for device authentication and secret key generation." *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007.

[8] A. Maiti et al., “A systematic method to evaluate and compare the performance of physical unclonable functions,” In Embedded Systems Design with FPGAs, P. Athanas, D. Pnevmatikatos, and N. Sklavos (Eds.). Springer, New York, 245267.

[9] Md. Tauhidur Rahman et al., "TI-TRNG: Technology Independent True Random Number Generator. Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference (DAC), Article 179, pp. 1-6, 2014.

[10] Mehrdad Majzoobi, Farinaz Koushanfar, Srinivas Devadas, FPGA-Based true random number generation using circuit metastability with adaptive feedback control, Proceedings of the 13th international conference on Cryptographic hardware and embedded systems, September 28-October 01, 2011, Nara, Japan.

[11] Berk Sunar, William J. Martin, Douglas R. Stinson, A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks, IEEE Transactions on Computers, v.56 n.1, p.109-119, January 2007.

[12] GOWIN User Guide, http://cdn.gowinsemi.com.cn/SUG550E.pdf

[13] GOWIN Design Physical Constraints, http://cdn.gowinsemi.com.cn/SUG935E.pdf

[14] GOWIN FPGA Primitive, https://www.gowinsemi.com/upload/database_doc/39/document/5bfcff2ce0b72.pdf