Side Channel Analysis of an Embedded/Hardware Crypto Device

Dallin Marshall, Sam Mitchell, and Nathanael Weidler
Deptartment of Electrical and Computer Engineering
Utah Stat University
Logan, Utah 84322
e-mail: geekbott@gmail.com, samuel.alan.mitchell@gmail.com, NWeidler@gmail.com

***Abstract*—This paper describes the design and implementation of a physically unclonable function (PUF).**

***Index Terms*—Physically unclonable Function, Device Authentication.**

I. Introduction

A physically unclonable function (PUF) is one method of testing user authentication. The server sends a challenge to a device, and the device responds using the output to the PUF; this is called a challenge response pair. If the PUF is truly unclonable, this authentication method is effectively secure.

The analysis of PUFs in authentication relies on 2 characteristics of the PUF: the variation between devices, μ_{intra}, and the reliability of the device to reproduce the same bitstream given a challenge, μ_{inter}.

A. Related works

Some of the main usages of PUFs are to provide reliable device identification, device authentication, key storage, and true random number generation [1].

Not all PUFs can be implemented on every type of device (FPGA, ASIC, microcontroller). There are various types of PUFs that can be implemented on an FPGA [2], some of which are SRAM, Butterfly [3], flip-flop [4], [5], Buskeeper [6].

The Butterfly PUF was analyzed in [7] with results inconsistent with [3]. The authors determined that this was due to the difficulty to define the routing distance between gates.

B. Structure of paper

The organization is as follows: in Section II, the development of the PUF is presented. Section III contains the analysis of the data. Conclusions are detailed in Section IV.

II. Design

The butterfly PUF consists of wiring 2 NAND gates together in positive feedback mode, as shown in Figure 1. Upon excitation, the gates are put into a race condition until the voltage level converges (less than 1 second). The butterfly should result in a consistent output for each device it's implemented on, while the output of the devices have low correlation. When implemented on a Spartan IV, the majority of the butterfly PUFs gave consistent but unique output, but the logic of 12% never converges to one voltage.

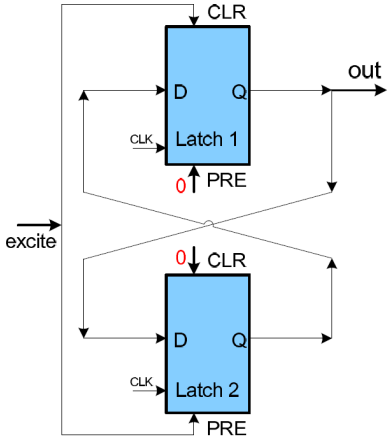

Figure 1. Butterfly PUF: This works because of cross-coupled latches.

This paper utilizes a modified butterfly PUF that accepts an input. The excite signal is used as a switching signal through negation, as shown in Figure 2. This allows the PUF to accept an input, which results in 2 potential random outputs. Testing showed that some PUFs would consistently toggle, some would consistently remain 1 value, and some weren't consistent, which is consistent with the 12% of the standard butterfly PUF that wouldn't converge. This PUF is very compact, requiring only 0.03% of the FPGA space.

A. Butterfly in authentication

An authentication device accepts a challenge sequence and sends back a device-specific response. We designed a modified-butterfly PUF authentication device that accepts a 64-bit key input and responds with a 64-bit key.

The device is a 64 by 64 grid of modified-butterfly PUFs. Each column uses an XOR gate to accept 1 bit of the 64-bit key; column 1 accepts bit 1, the output of column 1 is XORed with bit 2 and fed to column 2. The row order corresponds with the response bits; row 1 produces bit 1.

This design is valid because each bit of the key can alter the response up to 50%, which means that the corresponding response bit can only be predicted with 50% accuracy. Each row has a unique string of PUFs, so each response bit will be random to the other bits.

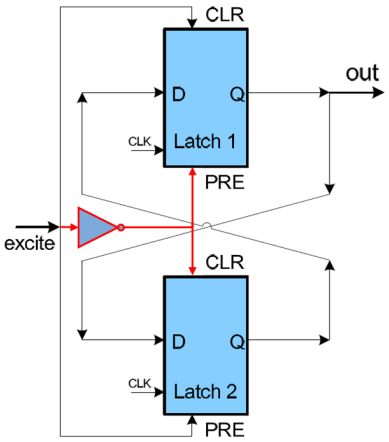

Figure 2. Modified-butterfly PUF: The cross-coupled latches are altered by the input.

A simple verification was performed on this system by implementing 1 row of our modified-butterfly authenticator PUF. Two of the 64 stages are represented in Figure 3. However, unique keys did not successfully produce unique responses. It seemed that the output was almost completely random data. There was nothing deterministic about the result of this string of 64 butterfly pufs.

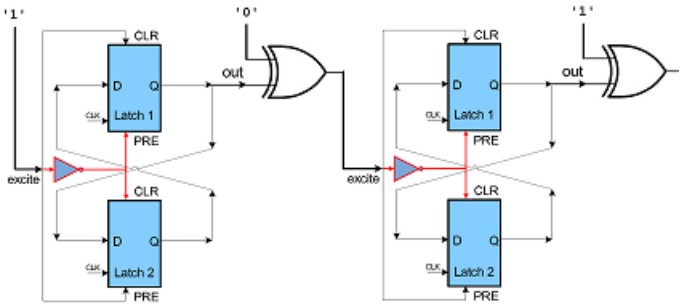

Figure 3. The first two stages of a 64 stage puf.

It is believed that without manually length matching the feedback nets in FPGA editor a single indeterministic butterfly in the chain can cause the entire chain to be unstable (depending on the challenge.) Through our testing it seems that 1 in 6 butterfly pufs would be random. That means in a string of 64 there are 10 or 11 that are random. The modified-butterfly PUF was discarded for authentication applications, but a new idea emerged.

B. Butterfly in TRNG

TRNG requires random processes to extract numbers from. The butterfly puf setup for this is the same as seen in Figure 3. It was decided to determine if the string of 64 butterfly pufs produced truly random data. In order to accommodate this there were two values used to excite the 64 butterfly pufs. The first was the pattern "10101010101010101010101010..." repeated out to 64 bits. The second was the pattern "01010101010101010101010101..." repeated out to 64 bits. Each pattern is held for 5000 clock cycles as the butterfly is allowed to settle. The frequency of the FPGA clock is 50MHz so each value is held for 100 micro seconds.

The output of the last butterfly in the string is read and if is was a '1' a 0x31 would be sent over the UART transmitter. If it was a '0' a 0x30 would be sent over the transmitter so that the data would be simple to digest. The transmitter was tied high so that it would always transmit. This way the last puf would be ringing as it settles and we would see random data. Once the pufs settled we would see the output settle to a value and then the pufs would be excited again with the opposite 64 bit pattern. This was accomplished by toggling a slide switch on the FPGA board.

With a baud rate of 9600 with 8 data bits, 1 stop and 1 parity bit we get 960 bits per second output from the TRNG. However these come in bursts of about 1000 before the butterfly settles and needs to be excited again by toggling the slide switch. This means that the overall rate of the TRNG depends on how efficiently you can re-excite the pufs once they settle as the constant values are discarded in between the random numbers.

III. Data Analysis

To verify the integrity of a TRNG we needed to verify that the steams produced by the PUF were indeed random. The two tests historically used to verify these streams were the Diehard tests and the NIST (STS) tests. Each of these test packages use multiple tests to verify the randomness of a stream. Presently, these tests are both part of the Dieharder test package, which was created to be the Swiss Army Knife of random number testing. These packages use statistical models to try and predict future numbers in the sequence.

To test our PUF we first captured a bit stream from the PUF and saved this stream to a file. Next we ran this file through the Dieharder package. Throughout our testing phase we never had a stream that failed a single test within the the Dieharder package. There were a few false positives that the test identified with a weak correlation to a specific test. As the length of our streams increased these false positive results became much less common.This positive feedback from the Dieharder tests allowed us to feel particularly good about the viability of the Butterfly PUF as a TRNG.

IV. Conclusion

The theory behind using a butterfly puf for authentication is a sound theory. However the implementation would require more time than we had for this project. We still believe that if the lengths of the feedback nets were matched inside the FPGA for each puf this would make a very reliable means of authentication. This will be left to future work.

The fact that butterfly pufs are unstable for a time make them very good building blocks for TRNGs. Our results showed that our puf solution to the TRNG is very promising. There could be ways to improve the design but we are pleased with the results attained above. An improvement may be to profile the

puf and excite it again before it settles to a steady value. This would greatly improve the rate of random number generation as you would not need to discard the steady values seen between the random values. Also simply increasing the baud rate would increase the throughput and would be worthwhile if the other optimization was made as well.

REFERENCES

[1] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[2] A. Van Herrewege, "Lightweight puf-based key and random number generation," 2015.

[3] S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls, "Extended abstract: The butterfly puf protecting ip on every fpga," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, June 2008, pp. 67–70.

[4] R. Maes, P. Tuyls, and I. Verbauwhede, "Intrinsic pufs from flip-flops on reconfigurable devices," in *3rd Benelux workshop on information and system security (WISSec 2008)*, vol. 17, 2008.

[5] V. van der Leest, G.-J. Schrijen, H. Handschuh, and P. Tuyls, "Hardware intrinsic security from d flip-flops," in *Proceedings of the fifth ACM workshop on Scalable trusted computing*. ACM, 2010, pp. 53–62.

[6] P. Simons, E. van der Sluis, and V. van der Leest, "Buskeeper pufs, a promising alternative to d flip-flop pufs," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, June 2012, pp. 7–12.

[7] S. Morozov, A. Maiti, and P. Schaumont, "An analysis of delay based puf implementations on fpga," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2010, pp. 382–387.