Secret Key Generation on a FPGA using a PUF

Justin Cox and Tyler Travis
Department of Electrical and Computer Engineering
Utah State University
Logan, Utah 84322
email: justin.n.cox@gmail.com, tyler.travis@aggiemail.usu.edu

Abstract— **Encryption has become an important standard for data storage and protection. In order to encrypt or decrypt the information, a secret key must be used. This secret key is usually stored somewhere in memory where it potentially can be stolen. To avoid this, new methods of secret key generation have been developed. One of these methods is using a Physical Unclonable Function (PUF) to securely generate the secret key. This paper will cover the design, implementation, and experimental results of a PUF implemented on a FPGA.**

***Index Terms*—secret key generation, security, PUF, FPGA.**

I. INTRODUCTION

A lot of research has been done on the use and applications of Physical Unclonable Functions which will be referred to for the remainder of this paper as PUFs. Some of these applications include, but are not limited to, authentication, secret key generation, true random number generation, and counterfeit chip identification. This paper will look into the effectiveness of a PUF used for secret key generation. This PUF will be developed on an FPGA. The results will be given at the end of the paper.

A. Previous Work

A lot of research has been done on PUFs. More important to this paper, research has been done on improving the reliability and effectivness of PUFs on FPGAs [1].

B. Contribution

This paper will try to reproduce the PUF designed and analyzed in [1] on a different FPGA. This paper aims to verify the claims made in the previous work. Our main contribution is to test the PUF and see if it will work well enough as a secret key generator for the DES algorithm.

II. OVERVIEW

A. Data Encryption Standard

A brief overview of DES will be given so that the reader has a better understanding of how the algorithm works and will better understand where DPA can be used. If the reader would like an in-depth understanding of DES, it is recommended that the reader look to other sources [2].

The DES algorithm takes a 64-bit plaintext input. It is then run through an initial permutation that outputs 56-bits which are then split into two halves. The data goes through sixteen rounds that each have a sub-key that is generated for each round based on the original 64-bit DES key. After the sixteenth round, the output is run through a finial permutation and the algorithm outputs a 64-bit encrypted ciphertext. The PUF will be used to generate the 64-bit secret key used to create the subkeys for the sixteen rounds.

B. Physical Unclonable Function

A Physical Unclonable Function (PUF) is a physical entity used to produce information that can only be reproduced using the same physical device. Since physical imperfections and deviations are normal in the manufacturing and design process of circuits and devices, each different device of the same family should have different physical characteristics that only pertain to said device. These imperfections, whether they be voltage levels, temperature, or delay times, can be used to produce information.

PUFs are especially effective when used in security. Since PUFs are generally difficult to model, the only way an attacker could recover the information generated by the PUF is to steal the actual physical device. Even then, if the attacker where to inspect the PUF they would risk changing the physical characteristics of the device and the PUF's output may change.

There are many different types of PUFs such as Optical PUFs, Delay PUFs, and SRAM PUFs. This paper will focus on a design which uses a Delay PUF, more specifically, a type of Arbiter PUF.

III. PUF DESIGN

Arbiter PUFs, which will be referred to as APUFs, are a well known delay PUF but they are infamous for having troubles being implemented on FPGAs. It also has been proven that an APUF can be modeled and simulated allowing an attacker to guess the response message from a given challenge.

There are different variations of the APUF and some of these variations improve the reliability and effectiveness of the APUF. One variation that has shown improvements is the Double APUF or DAPUF [3]. The DAPUF is made by duplicating a single APUF on an FPGA. Another variation that has improved the resistance to modeling of an APUF is the XOR APUF [4]. The XOR APUF taks multiple APUFs' output and XORs them together. This decreases an attackers ability to model the APUF but it requires more APUFs to be built on the FPGA.

As mentioned in [1], the PUF that this paper will be using for secret key generation is the 3-1 DAPUF. This APUF is a combination of the DAPUF and the XOR APUF. It duplicates three APUFs on a FPGA and XORs their respective outputs together. This means that for each response bit, 3 APUFs are needed. The overall structure of the 3-1 DAPUF is shown in Figure 1.

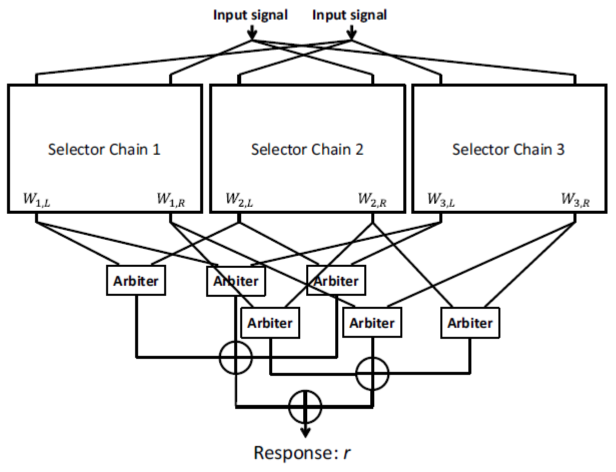


Fig. 1. 3-1 DAPUF figure taken from [1].

The APUF is split into three distinct parts. These three parts are the input signal, the Selector Chain, and the Arbiter. In order for the reader to better understand the design, the Selector Chain is shown in detail in Figure 2. The Selector Chain is a chain of MUXes that will either cross the signal or send it straight through. The travel direction of the signal depends on the challenge bits. The Selector Chain creates a physical delay. The outputs of the Selector Chain are sent to an Arbiter which determines which signal arrives first.

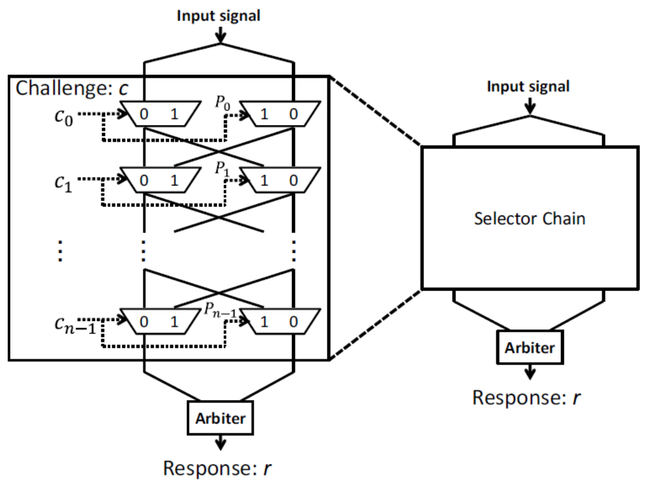


Fig. 2. Details of the Selector Chain taken from [1].

Normal APUF designs have a difficult time working well on an FPGA because of the problem of symmetry. If the Selector Chains paths are not symmetric, the output will be heavily biased to either a 1 or a 0. This will make the APUF have a low uniqueness among other APUFs. The 3-1 DAPUF was designed specifically to work well with FPGAs. The duplicated APUF needs to be copied to a neighboring SLICE on the FPGA. Since the duplicated APUF is a copy of the original, the neighboring SLICE should have close to the same wire lengths. This paper will try to emulate that design using Xilinx ISE and Xilinx Plan Ahead.

IV. SECRET KEY GENERATION

One of the difficulties of using a PUF for secret key generation is the value of $\mu intra$. The measurement $\mu intra$ is used to determine how much the output response changes of a certain PUF when given the same input challenge. Ideally, a PUF should generate the same response when given the same challenge. Two different PUFs should also generate a different response when given the same challenge. This measurement is referred to as $\mu inter$. A good PUF should have a $\mu inter$ close to 50% and a $\mu intra$ close to 0%.

The reason $\mu intra$ is important is because in a encryption/decryption algorithm, even the slightest change to the secret key will change the cypher-text dramatically. This means that in order to use a PUF for secret key generation, the PUF's challenge-response pairs need to be consistent. Since this is almost impossible to do using a PUF, the design of this paper will use Error Correcting Code (ECC) to fix the response output. A flow chart of this design is shown in Figure 3.

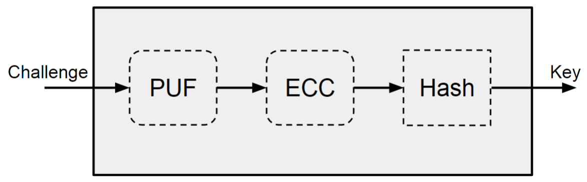


Fig. 3. Flow chart of the secret key generation process.

Majority voting will be used to determine what bits of the response output typically change given the same challenge input. This process is explained in detail in [5], but a brief overview of the process will be given. A chosen challenge message will be sent through the PUF multiple times. It will then be determined which bits have a tendency to change. These bits will be thrown out and the new challenge message will be sent in the ECC. The ECC encoding process will return parity bits associated with the certain challenge message. During normal operation, these parity bits will be used to fix the response output for a given challenge. This process will make sure that the secret key will remain the same for a given challenge.

As shown in Figure 3, it is also a good idea to hash the output from the ECC to inhibit the attacker from knowing the response message. The design can be made even more secure by adding a hash function to the challenge input as well. The hash function will not be implemented in this paper's design because the main focus of the paper is to prove that a PUF can be used for secret key generation.

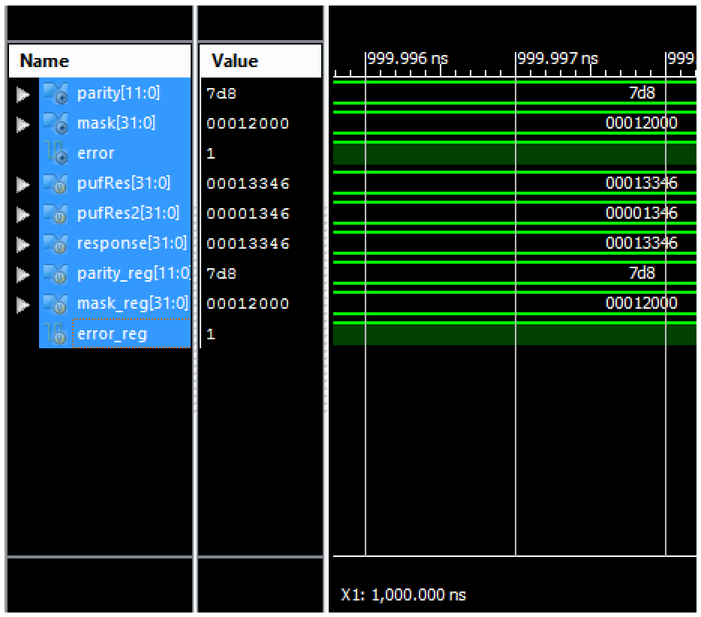
V. RESULTS

The 3-1 DAPUF was successfully implemented on a Digilent Nexys 2 FPGA board. The system clock was running at

50 MHz. The initial design was planned to have a challenge of 64-bits. However, the 3-1 DAPUF requires over 20,000 Look Up Tables (LUTs) and the Nexys 2 only has 17,000 LUTs. As a result of this limitation, a key of 32-bits was chosen.

We tested 10 different challenges 100 times each to measure $\mu intra$ and $\mu inter$. We also measured $\mu intra$ at the normal FPGA operating temperature of 85 degrees Fahrenheit and at a hotter temperature of 185 degrees Fahrenheit. To measure $\mu inter$ we ran the same challenge messages on 2 different FPGAs. Our $\mu intra$ at normal temperature was 4.29% and at the hotter temperature it was 7.03%. A slight decrease in the quality of $\mu intra$ was observed at higher temperatures, but overall it retained good results. The $\mu inter$ that we measured was 23.82%. This $\mu inter$ could be a lot better, but it should be sufficient for secret key generation because the output of the ECC is typically sent through a HASH function. One reason we achieved a low $\mu inter$ is because we were not able to enforce symmetry. This would be fixed in a later revision.

A free open source BCH IP core was found on the internet and it was utilized in the design. The BCH core is able to correct 2 error bits. In order to make sure that the challenge message has 32 reliable bits, the 3-1 DAPUF for this design produces 40 response bits. This allows the output response to have 8 bits that can be tossed out after the majority voting. The BCH core was verified to correct two errors and this can be seen in Figure 4.



Name	Value
parity[11:0]	7d8
mask[31:0]	00012000
error	1
pufRes[31:0]	00013346
pufRes2[31:0]	00001346
response[31:0]	00013346
parity_reg[11:0	7d8
mask_reg[31:0]	00012000
error_reg	1

Fig. 4. *pufRes* is the correct output, *pufRes2* is the output with 2 error bits, and *response* is the corrected output response

VI. Conclusion

References

[1] J. Orlin Grabbe. The DES Algorithm Illustrated. *Laissez Faire City Times*, 2006.

[2] Thomas S. Messerges and Ezzy A. Dabbish, ”Investigations of Power Analysis Attacks on Smartcards,” *USENIX Workshop on Smartcard Technology*, Chicago, Illinois, USA, 1999.