An Efficient Implementation of an Arbiter PUF on a Lightweight Microcontroller

Eric Y. Li
Princeton University
eyli@princeton.edu

Physical Unclonable Functions (PUFs) have emerged as a novel lightweight alternative to traditional cryptographic techniques. These devices differ from traditional cryptography in that there is no secret key stored in non-volatile memory. One type of PUF, the Arbiter PUF, applies a race signal to two identical paths and determine which is faster, making use of the process variations that occur in manufacturing transistors. However, these PUFs have been proven to be insecure via side channel and machine learning attacks. This paper explores implementing a software arbiter PUF on an 8-bit microcontroller based upon information collected from these attacks and the timing operations of the various models intoduced in (Becker & Kumar, 2014).

Introduction

PUFs have gained widespread attention as a cryptographic alternative to traditional cryptography. Indeed, one of the benefits of using a PUF over traditional cryptographic techniques is that there is no need for a secret key to be stored in non-volatile memory. Rather, PUFs make use of the process variations inherent in manufacturing two identical chips as a way of ensuring unique identification of each chip when participating in challenge-response authentication.

Because each chip does not contain any secret keys and rather depends on the physical characteristics of the chip to generate a secret, PUFs are much more resilient to physical attacks and reverse engineering on a hardware level. PUFs are sensitive to a variety of environmental factors as well, such as temperature, supply voltage, and electromagnetic interference. Therefore, just accessing a PUF is enough to alter characteristics of the chip, thereby modifying the PUF itself and changing the generated secret.

Currently, strong PUFs[1] suffer from several major pitfalls that prevent them from being used in practice. The first is that they are unreliable. Because PUFs are subject to environmental variations, error correcting codes must be put into place in order to make them viable. Secondly, these PUFs can be modeled in software, with the needed paramters determined based on machine learning techniques, meaning that currently, PUFs are not secure.

Nevertheless, PUFs are seen as novel methods for use in cryptographic security and many protocols have been proposed that in such PUFs in them even though they are not considered cryptographically secure.

Thus, it is of interest whether it is possible to efficiently replicate such hardware based PUFs in software so that an attacker might be able to forge a PUF using a cheap smartcard after a successful attack.

Arbiter PUFs

Arbiter PUFs (Suh & Devadas, 2007) are the most commonly used PUF implementation in current literature and the focus of this this research. The basic design of an Arbiter PUF is to apply a race signal between two identical paths and determine which is faster. The two paths have an identical layout so that the delay difference ΔD is only based on process variations produced when manufacturing the transistors. This ensures that each chip will have a unique delay behavior.

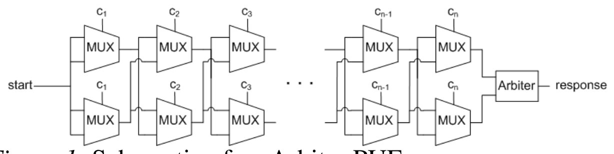

Figure 1. Schematic of an Arbiter PUF

An Arbiter PUF gets a challenge as an input, which defines the paths of the race signal. An Arbiter PUF consists of a top and bottom signal fed through delay stages, with each stage consisting of two 2-bit multiplexers (`MUX`). If the challenge bit for that stage is a '1', then the multiplexers will swap signals, otherwise the signals will not change. Each individual transistor in the multiplexers will differ slightly due to the process variations in manufacturing. Hence, the delay difference of each stage between the top and bottom signals are different for a '0' and a '1'. Thus, a race signal can take on many different paths based on the challenge; an n-stage Arbiter PUF will have $2n$ different possible paths. Finally, an Arbiter consisting of two cross-coupled `AND` gates form

[1]A strong PUF has a challenge space that is large enough so that it is computationally infeasible to store all possible challenges in order to forge a PUF with a memory look up.

a latch at the end of the PUF determines which of the two signals was faster. If the top signal arrives first, then it will output a '0' and it the bottom signal arrives first, then it will output a '1'.

By applying a maximal length Linear Feedback Shift Register (LFSR) to a given challenge, Arbiter PUFs can also be used to generate a n-length response bit string. A maximal length LFSR will ensure that the derivative input challenges will be pseudorandom when being fed into the Arbiter PUF. In additional, there can be multiple Arbiter PUFs that are fed the same challenge, with an XOR at the very end that produces a single output bit, which will make it harder to break these PUFs. However, by adding more Arbiter PUFs, the reliability will decrease due to the Arbiter PUF's dependence on environmental factors.

Modeling an Arbiter PUF

An Arbiter PUF can be easily modeled once delay values of each stage are known. Though each stage has four delay values, the delay for the top and bottom MUXes for challenge bits '1' and '0', we are primarily interested in the overall delay difference. Thus, rather than computing the total delay of both the top and bottom race signals, we can focus on the delay difference between these signals. This also reduces the number of delay values per stage i to two, the delay difference $\delta_{1,i}$ between the top and bottom stages for a challenge bit '1' and $\delta_{0,i}$, the delay difference between the top and bottom stages for a challenge bit '0'. This means that an Arbiter PUF can be modeled in just $2n$ delay values.

The delay difference is positive if the top signal is faster, and negative if the bottom is faster. The total ΔD of a challenge $\mathbf{C} = c_1, \ldots, c_n$ can be computed by adding up the relevant delay for each stage. If the current challenge bit c_i is a '1', the bottom signal switches to the top and vice versa. This can be modeled mathematically by making the current delay difference negative. Thus, the delay difference after stage i can be effectively written as:

$$\Delta D_i = \Delta D_{i-1} * (1 - 2c_i) + \delta_{c_i,i} \tag{1}$$

The overall delay difference between the two signals is simply the final difference ΔD_n after the last stage n. The response bit can then be computed using the following equation:

$$r = \begin{cases} 0 & \text{if } \Delta D_n < 0 \\ 1 & \text{if } \Delta D_n > 0 \end{cases} \tag{2}$$

However, a more efficient approach was allows for an n-stage Arbiter PUF to be modeled with just $n + 1$ parameters, thereby saving space. To do this, a delay vector $\mathbf{w} = (w_1, \ldots, w_{n+1})$ is created based on the given delay values:

$$w_1 = \delta_{0,1} - \delta_{1,1} \tag{3a}$$

$$w_i = \delta_{0,i-1} + \delta_{1,i-1} + \delta_{0,i} - \delta_{1,i} \text{ for } 2 \leq i \leq n \tag{3b}$$

$$w_{n+1} = \delta_{0,n} + \delta_{1,n} \tag{3c}$$

Likewise, a feature vector Φ is created based upon the input challenge vector $\mathbf{C}$ as follows:

$$\Phi_i = \prod_{l=i}^{n} (1 - 2c_l) \text{ for } 1 \leq i \leq n \tag{4a}$$

$$\Phi_{n+1} = 1 \tag{4b}$$

From these two vectors, the delay difference ΔD_n can easily be computed by taking their dot product:

$$\Delta D_n = \mathbf{w}^T \Phi \tag{5}$$

These two Arbiter PUF implementations were the two chosen to be implemented on a lightweight 8-bit microcontroller.

Implementation of an Arbiter PUF

The purpose of this research was to determine if it was possible to efficiently replicate a hardware based Arbiter PUF in software, and therefore spoof a real Arbiter PUF with an attacker's. Though an STM8S Discovery Microcontroller was initially chosen as the target microcontroller, we shifted to using a cheap ATMega163 8-bit microcontroller on a FunCard for several reasons. First, the STM8S did not have an easily accessible communication protocol, whereas the FunCard was a smartcard that used an application protocol data unit (APDU) to communicate back and forth with the client. Additionally, the FunCard was a smartcard, a device which attackers might actually target with a spoofing attack in order to gain unauthorized entry.

There were several steps within this research, with a focus of optimization always in mind. Code for the client was written in Java whereas the code for the smartcard was written in C. The first goal was to establish a reliable form of communication between the smartcard and client (in this case, a computer) where challenges and responses could be easily communicated. The second goal was to implement both implementations of the Arbiter PUF and determine which was faster. The third goal was to implement a basic LFSR that would allow for generation of a full length response rather than a single bit. The final goal would be to implement a maximal length LFSR in conjunction with the chosen Arbiter PUF as efficiently as possible, as the whole goal was to complete all tasks in as little time as possible so as to reduce the time difference between the spoofed version and hardware version.

Delay and Challenge Vector Generation

The delay values and vector Φ as well as the challenge vectors $\mathbf{C}$ were generated using a MATLAB script that generated the values on a Gaussian distribution. The script also calculated reference output bits for each given challenge vectors and later was modified to include a maximal length LFSR. In addition, the script supported both 64-bit and 128-bit bit strings and was capable of generating multiple challenge strings at once that could later be read by the associated client program. The corresponding delays and challenges were then stored as text files that were parsed by the client.

Client Program

The client was written in Java and allowed for efficient communication between the smartcard and the computer via APDUs. The client also converted all binary bit strings and delay values to hexadecimal values, which was the format supported by the APDU protocol. Two versions of the client were produced to support both the standard implementation and alternate implemenation of the Arbiter PUF, with both versions supporting 64 and 128 bit strings and read/write functionality for the challenge and response pairs. Finally, the client timed the amount of time it took to send a list of challenges and receive their corresponding responses.

Communication

Communication between the smartcard and the client was based on the APDU protocol with modifications to suit the project. Different instruction (INS) codes were given depending on if the card was being programmed with delay values or being sent a challenge. In addition, the parameters (P1,P2) contained information on the bitlength of the challenge or the index of the delay values being sent. Since an APDU could only support 255 bytes of data, the delay values (stored as shorts) had to be sent in several batches, making it crucial for the starting index of the current delay value to be known. Finally, the input length of each challenge read would vary depending on if it was 64 bits (8 bytes) or 128 bits (16 bytes). The smartcard would the respond accordingly directly to the client, whereby the response was converted back to binary and written to a text file.

Methods were written on both the client and smartcard side that would allow for efficient conversions from binary to hexadecimal and vice versa, as well as splitting a short into two hexadecimal bytes and vice versa.

Arbiter PUF Implementation

The delay values for the Arbiter PUF were stored in two n-length short arrays. Though the initial delay values in the MATLAB script were decimals, floating point arithmetic is expensive on microcontrollers, so a change in magnitude was necessary to ensure efficiency while still mantaining a decent amount of precision.

Once received, the challenge bit would be recovered from the hexadecimal based APDU and sent to a method that would compute the Arbiter PUF based on the given delay values, returning an output bit as a character. This was accomplished by converting the recursive function defined in (1) to an explicit function and running a for loop n times to calculate the final output delay. Timing tests were run for both 64 and 128 bit implementations at varying sample sizes.

Alternate Arbiter PUF Implemenation

The delay values were stored in one $n + 1$ length short array, containing values precalculated in the MATLAB script. The transformation of the challenge vector was done on the simultaneously with the calculation of the final delay value ΔD_n. This combination of equations (4) and (5) was a clever way of optimizing the method to reduce running time further.

LFSR Implementation

Maximal Length Linear Feedback Shift Registers were used to generate pseudorandom sequences based upon the input challenge seed. By using a maximal length LFSR of $2^n - 1$ possible sequences, n "independent" pseudorandom sequences could be recovered by clocking the LFSR n steps and taking the current sequence after those steps (as any remnants of the previous sequence would no longer be retained).

The type of LFSR implemented was a Fibonacci LFSR, which operates by performing an `XOR` on several tap bits and then right shifting the entire sequence and appending the generated bit to the left. Doing this n steps ensures a completely new sequence. For a 64-bit maximal length LFSR, tap bits $\mathbf{t} = \begin{bmatrix} 64 & 63 & 61 & 60 \end{bmatrix}$ were used. For a 128-bit maximal length LFSR, tap bits $\mathbf{t} = \begin{bmatrix} 128 & 126 & 101 & 99 \end{bmatrix}$ were used.

Methodology

When simulating the Arbiter PUF, the first step was to generate delay values and challenge vectors using the MATLAB script. Following this, the delay values were communicated to the smartcard.

For each challenge, the initial seed was run through the Arbiter PUF once to generate the first response bit. Following this, the challenge vector was sent to an LFSR and clocked n times before being sent again to the Arbiter PUF, creating the next response bit. This continued until the length of the response bit was equal to the length of the input bit, n. The full response string would then be communicated back to the client and written to a file. This process was repeated for all the randomly generated challenge sequences.

Timing

Timing was done by using Java's built in `System.currentTimeMillis()` method and began when the first challenge was read and sent to the smartcard. Though the overall time included the time needed to open the file and write to it on the client side, these were negligible compared to the time required to compute the responses for the challenges. Timing was done for challenge files of varying input sizes $I = 100, 500, 1000, 2000$ for both $n = 64$ and $n = 128$.

In addition, timing was done on just the communication, PUF computation, and LFSR operation to access which areas had the most overhead and which methods could be further optimized.

Optimizations

Optimization was done on the code in several ways. First, loops and conditionals were looked at to see if they could be reduced or eliminated as they took extra clock cycles. Following this, the disassembly was also analyzed to determine which operations were unecessary.

Several key optimizations were made that helped to reduce running time. To reduce space, the smallest integer size was used; many of the for loops that initially used `int` to store values now used `unsigned char`. Additionally, the LFSR methods used to be called n^2 times, meaning that each time it was called, the smartcard had to load all values array values into its registers and modify them before storing them again, causing an unnecessary inefficiency. As an alternative, the LFSR clocked n times everytime it was run, reducing the need to load and unload registers by a factor of n. Another optimization was modifying the output array to be a $8/n$ size array instead of a binary n size array. This would save space and operating time, since the conversion would happen simultaneously and was accomplished by left shifting the current output value by one and using `OR` with the next computed PUF value.

Results

Results from various implementations and optimizations are shown here.

PUF Implementation Comparisons

Initial timing tests before implementation of the pseudorandom LFSR was done to determine which of the two implementations were faster. Tables 1 and 2 show that clearly, the standard implementation yields much faster results for both 64 and 128 bit lengths. The alternate implementation had issues completing calculations for $n = 128, S = 2000$ but it became clear that the standard implementation was much faster. Therefore, it made sense to focus on optimizing the standard version and implementing a pseudorandom LFSR based on that.

Table 1
Timing (ms) for 64-Bit of Both Arbiter PUF Implementations With Simple LFSR

$S =$	100	500	1000	2000
Standard	9561	47697	95304	190618
Implementation	9545	47716	95428	190644
Alternate	18304	91506	183026	366061
Implementation	18310	91521	183013	365936

Table 2
Timing (ms) for 128-Bit of Both Arbiter PUF Implementations With Simple LFSR

$S =$	100	500	1000	2000
Standard	27731	138700	277307	554743
Implementation	27735	138703	277385	
Alternate	62029	310045	620166	N/A
Implementation	62036	310025		

Pseudorandom LFSR Implementation

After implementing a pseudorandom LFSR and applying the various optimizations discussed, the following timings were achieved in Table 3. Unfortunately, when attempting to run timing tests with $n = 128$, the smartcard produced an error `Unknown error 0x45d`, which indicates that within the APDU protocol, it is taking too long for the smartcard to respond.

Table 3
Timing (ms) for 64-Bit of Standard Arbiter PUF With Pseudorandom LFSR

100	500	1000	2000
21205	105998	212049	424101
21243	106036	212066	424136

Timing Breakdown

Breaking down the timing into individual sections, table 4 shows that communication took 19.37% , LFSR computation took 36.60%, and Arbiter PUF calculations took 44.03% of the overall time. Breaking this data down into each challenge seed, LFSR computation took 77.614 ms and PUF calculations took 93.359 ms. Overall, it takes 212.059 ms to send a challenge, run a maximal length LFSR and the Arbiter PUF, and return the response.

Table 4
Breakdown of Timing (ms) for 64-Bit of Standard Arbiter PUF With Pseudorandom LFSR with $S = 2000$

$S = 2000$	Time (ms)
Communication	82173
LFSR	155227
Arbiter PUF	186718
Total	424118

Conclusions and Further Directions

From the timing results, it is clear that it is more than feasible to implement a software based Arbiter PUF within the time limits of the APDU protocol with an average challenge time 0f 0.212 seconds. This means it is perfectly feasible for an attacker to recover delay values and then program a cheap 8-bit smartcard to clone the original Arbiter PUF, thereby gaining access to a previously secure system.

From these results, it is clear that further optimizations can be made, particularly to LFSR computation. Currently, the smartcard will error out because its latency is too high when calculating 128-bit pseudorandom LFSRs. This can be rectified by implementing the LFSR and possibly other functions in assembly rather than C. Another direction to go in is to streamline the overall process of generating a response bit by reducing the number of number system conversions. Unfortunately, it is not possible to stream a response back while concurrently receiving a challenge; this is due to the nature of the APDU protocol. Another possible improvement would be to find a card reader with a faster latency, which would further reduce the communication overhead.

Nevertheless, it is clear from this framework that it is perfectly feasible to efficiently implement an Arbiter PUF on an 8-bit microcontroller in a manner that replicates the characteristics of a hardware Arbiter PUF. This work further diminishes the viability of an Arbiter PUF as a cryptographic building block and shows that it currently is not usable in production protocols.

Acknowledgement

This research was conducted under the direction of Dr. Georg T. Becker of the Horst Görtz Institute for IT-Security at Ruhr Universität Bochum. Without his support and advice, this project would likely be nonexistant.

I would also like to thank Princeton University and the Keller Center for providing me with the opportunity to spend a Summer researching embedded security at the Horst Görtz Institute for IT-Security run by Dr. Christof Paar at Ruhr Universität Bochum.

References

Becker, G. T., & Kumar, R. (2014). Active and passive side-channel attacks on delay based puf designs.

Suh, G. E., & Devadas, S. (2007). Physical unclonable functions for device authentication and secret key generation. *ACM*.