Remote Integrity Checking using Multiple PUF based Component Identifiers

Harsha Mandadi

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Patrick Robert Schaumont, Chair

Peter M Athanas

Leyla Nazhandali

April 27, 2017

Blacksburg, Virginia

Keywords: Physical Unclonable Functions, Fuzzy Extractors, Authentic Protocol

Remote Integrity Checking using Multiple PUF based Component Identifiers

Harsha Mandadi

(ACADEMIC ABSTRACT)

Modern Printed Circuit Boards (PCB) contain sophisticated and valuable electronic components, and this makes them a prime target for counterfeiting. In this thesis, we consider a method to test if a PCB is genuine. One high-level solution is to use a secret identifier of the board, together with a cryptographic authentication protocol. We describe a mechanism that authenticates all major components of PCB as part of attesting the PCB. Our authentication protocol constructs the fingerprint of PCB by extracting hardware fingerprint from the components on PCB and cryptographically combining the fingerprints. Fingerprints from each component on PCB are developed using Physical Unclonable Functions (PUF).

In this thesis, we present a PUF based authentication protocol for remote integrity checking using multiple PUF component level identifiers. We address the design on 3 different abstraction levels. 1)Hardware level, 2)Hardware Integration level, 3)Protocol level. On the hardware level, we propose an approach to develop PUF from flash memory component on the device. At the hardware Integration level, we discuss a hardware solution for implementing a trustworthy PUF based authentication. We present a prototype of the PUF based authentication protocol on an FPGA board via network sockets. This research is supported by CISCO systems Inc.

Remote Integrity Checking using Multiple PUF based Component Identifiers

Harsha Mandadi

(GENERAL AUDIENCE ABSTRACT)

Electronic devices have become ubiquitous, from being used in day to day applications to device critical applications (defense, medical). These devices have valuable electronic components integrated on it. Because of its growing importance, they have attracted many counterfeiters. Counterfeiters replace a genuine component with a substandard component. In this thesis, we discuss a method to identify if an electronic device, a Printed Circuit Board in this case, is genuine.

We present a solution to remotely verify authenticity of the board by extracting fingerprints from all the major components on the board. Fingerprints from each major component on the board are extracted using Physical Uncloanable Functions (PUF). These fingerprints are crypographically combined to develop an unique fingerprint for the board.

Our design is addressed in 3 different abstraction levels 1) Hardware level 2) Hardware Integration level 3) Protocol level. In the Hardware level, we discuss an approach to extract fingerprints from flash memory component. In the Hardware Integration level, we discuss a hadware approach for trustworthy PUF based solution . In the Protocol level, we present a prototype of our design on FPGA using network sockets.

Dedication

In dedication to my family. Daddy, Amma and Tammudu.

Acknowledgments

First, I thank my advisor Dr. Patrick Schaumont. It is a privilege to work and learn under his guidance. I am extremely grateful for all his support and help. My journey during my masters was smooth and memorable because of his support and motivation from time to time. I would also like to thank Dr. Leyla Nazhadali and Dr. Peter Athanas for sparing their time to serve on my thesis committee.

I take this opportunity to thank my family and friends who are with me throughout my journey. Thank you, Blacksburg for giving me a memorable start to my journey in The United States of America.

I thank all my SES lab mates Aydin, Shravya, Carol, Chinmayi, Hemendra, Bilgiday, Nahid, Abhishek, Archanaa, Conor, Yuan for their support during the course of my project. I will miss working in Secure Embedded Systems lab.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Electronic devices have become ubiquitous due to its growing applications in every part of human life. Growing economic importance and popularity of electronic devices has attracted many counterfeiters. Counterfeit devices are cheap substitutes that do not meet quality requirements, leading to substandard or non functioning products. Such contaminated devices can affect mundane applications like dropping a phone call to application critical systems. Failure of a faulty counterfeit device in critical applications like military, aerospace, medicine and defense industry can lead to catastrophic results [8].

Electronics supply chains are complex and counterfeit products are passed down to the user along the supply chain. Hardware can be tampered with along the supply chain from

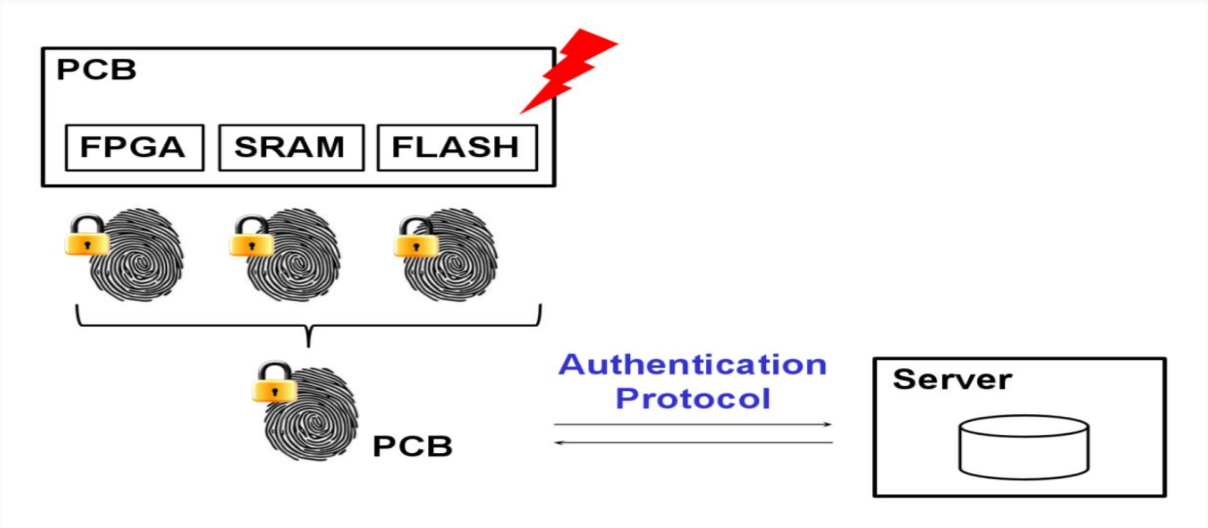

Figure 1.1: Hardware attestation solution for a PCB

manufacturer to remote installation. Such devices incur huge monetary losses and also damage the reputation of the manufacturer. It is important for the manufacturer to ensure that a remote, unattended hardware device and its components are authentic.

The manufacturer can verify the authenticity of printed circuit board (PCB) by designing a hardware attestation solution for complex electronic components. The manufacturer should be able to detect the tampered device at the consumer end, from a server installed at the remote location. One naive solution for this approach is by adding a fingerprint to the board and using it in an authentication protocol. Fingerprints are coded string of binary digits that uniquely identifies a device. A hard coded fingerprint can be added to a non-volatile memory and used in a protocol for authenticating a remote device. This will only demonstrate the authenticity of the nonvolatile memory on the device and not the authenticity of all the digital components on the board.

Extracting fingerprints from each critical component on the board(FPGA, SRAM, Flash)

is an alternative to this problem. Fingerprints for each of these components are extracted using Physical Unclonable Functions(PUF) [6]. PUF are constructions that convert chip-unique properties into stable and unique fingerprints. The fingerprint of an entire board is constructed by cryptographically combining PUF based fingerprints from its components. The first part of this thesis is on investigating a novel approach to generate fingerprints from Flash memory component using PUF.

Fingerprints obtained from PUF are noisy and need an error correction mechanism to validate fingerprints. The second part of this thesis provides hardware support for trustworthy PUF based authentication.

We provide a hardware attestation solution with a secure authentication protocol and PUFs for remote verification. The third part of this thesis is a system demonstration of our PUF based authentication protocol on a PCB using network sockets.

This thesis is demonstrated on two platforms - Altera DE1-SoC and Altera DE2-115 with SRAM memory, FPGA and Flash memory components.

1.2 Contributions

Figure:1.2 presents an overview of our PUF based authentication protocol for hardware attestation. The design has 3 abstraction levels. The hardware level provides proof of identity for the board using PUF technology. PUF is constructed on most of the critical components on the board. The system software level consists of drivers which securely obtain

PUF data from individual components on hardware and cryptographically merge it. This method of cryptographically merging PUF components is called PUF Fusion [1]. Reverse Fuzzy Extraction Module(FEModule) also includes a mechanism for error correction of PUF data obtained from Fusion PUF. The application software protocol level provides a secure authentication protocol for remote integrity verification. This thesis contributes solutions at each of these abstraction levels, providing a design with better security objective.

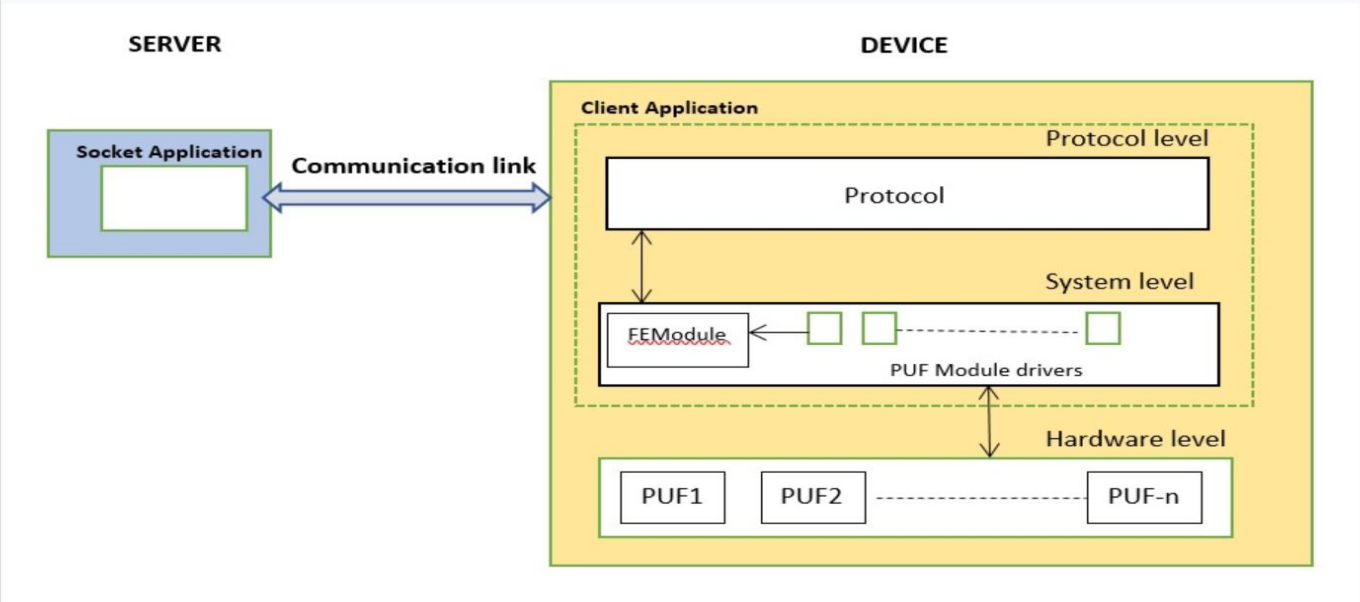

Figure 1.2: PUF based authentication protocol

The contributions of this thesis are as follows:

- We present a method to extract PUF output from NOR Flash memory chips using partial programming. We describe the practical implementation of this approach and evaluate reliability and uniqueness of the designed PUF.

- We propose a hardware solution for trustworthy PUF based authentication. The secret

key from PUF is noisy and requires a correction mechanism to eliminate noise and to improve security. The error correction design is a Reverse Fuzzy Extraction, an algorithm comprising of BCH encoder, decoder and SHA256 hash function.[1]

- We present a system demonstration of our PUF based authentication protocol design on DE1-SoC board. We implement device-side hardware components using Altera Quartus II tool chain version 16.0. The protocol communication is handled through Network Sockets.

A part of this work is described in the paper:

- A. Aysu, S. Gaddam, H. Mandadi, C. Pinto, L. Wegryn, P. Schaumont, A Design Method for Remote Integrity Checking of Complex PCBs, Design, Automation & Test in Europe (DATE 2016), Dresden, Germany, March 2016.

1.3 Organization

This thesis is structured as follows. Chapter 2 gives the necessary background and preliminaries for this thesis. Chapter 3 describes our design for Flash PUF. Chapter 4 provides hardware interface for Fusion PUF. Chapter 5 demonstrates the protocol on PCB using network sockets. Chapter 6 concludes the thesis

Chapter 2

Background

2.1 Physical Unclonable Functions

An onchip Physical Unclonable Function (PUFs) is a chip unique Challenge-Response mechanism exploiting manufacturing process variation inside Integrated Circuits(ICs). For a given m-bit challenge C, the n-bit response R is determined by complex variations inside an IC. This challenge-response pair is unique to the device, even though all these devices are manufactured using identical design files. As PUF technology uniquely distinguishes digital chips and can be used as a source of fingerprints in the remote authentication protocol.

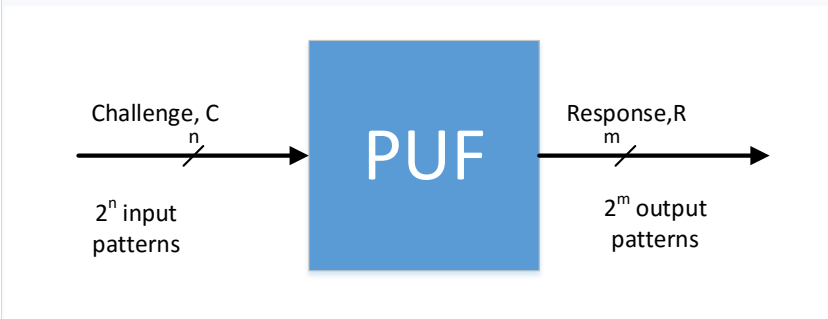

Figure 2.1: PUF representation

2.1.1 Types of PUF

Strong PUFs and Weak PUFs

Strong PUFs have the large set of Challenge-Response pairs making it unpredictable. In other words, an adversary cannot build an accurate model of the PUF just by observing challenge-response pairs.

Weak PUF exhibit low number of challenge-response pairs. An extreme case of weak PUF has a single challenge. Our work is based on weak PUFs and hence we assume only 1 challenge for each PUF [12].

Memory based PUF and Delay based PUF

Memory based PUF uses initial state of a memory cell upon startup and uses it as a fingerprint. Memory based PUF do not require special designs to obtain PUF response. SRAM is one example of memory based PUF. Turn-on value of SRAM cells tend to be device unique, due to manufacturing variations of SRAM latch cells. A SRAM PUF response is obtained by

reading a (non-initialized) section of SRAM cells, after power-cycling the SRAM chip [10].

Delay based PUF compares delays between two identical controlled paths to extract PUF response. Arbiter PUF and Ring Oscillator PUF(RO PUF) are examples of delay based PUF. RO PUF exploits delay variations in the logic elements to produce a unique n-bit identifier [5]. The design works by chaining an odd number of inverters and connecting the output of the last inverter to the input of the first inverter. Each RO produces an oscillating output with frequency dependent on how quickly the looping signal propagates through the logic elements. The output frequency of RO(A) is compared to the output of another RO(B), and either a 1 or 0 is produced depending on whether A or B has a faster frequency.

2.1.2 PUF Properties

PUF quality metrics are measured in terms of Hamming Distance(HD). For 2 different bit strings, Hamming Distance is the number of positions at which these strings differ.

Reliability

Reliability estimates the noisiness of a PUF response. A PUF is expected to return the same response, for same challenge, multiple times. But we observe noise in PUF responses due to environmental variations such as temperature, aging. Reliability is measured using average Intra hamming distance between m-bit responses to identical challenges on same

PUF instance.

$$Reliability = (100 - IntraHD)\% \tag{2.1}$$

Uniqueness

Uniqueness estimates the diversity of PUF responses. The uniqueness of a PUF shows how unique are the signatures generated by the PUF from different chips [5]. Uniqueness is measured using Inter hamming distance over the output responses to identical challenges on different PUF instances of the same component.

$$Uniqueness = (InterHD)\% \tag{2.2}$$

2.2 Reverse Fuzzy Extraction Module

The first step of PUF based authentication protocol is enrollment. Manufacturer extracts the secret PUF response from the device and stores it in his database. After deployment of the device in the field, the server will test the device authenticity by testing its ability to regenerate an earlier generated response.

However, a PUF is a noisy function. Due to environmental variations and electrical noise, every response to a challenge may show small variations. This prevents one from directly using a PUF response as a unique identifier or a cryptographic key. The noise effects needs to be removed, typically using an error correction mechanism. Furthermore, a PUF may show bias. For example, certain response bits may stick to a logic-1 with high probability.

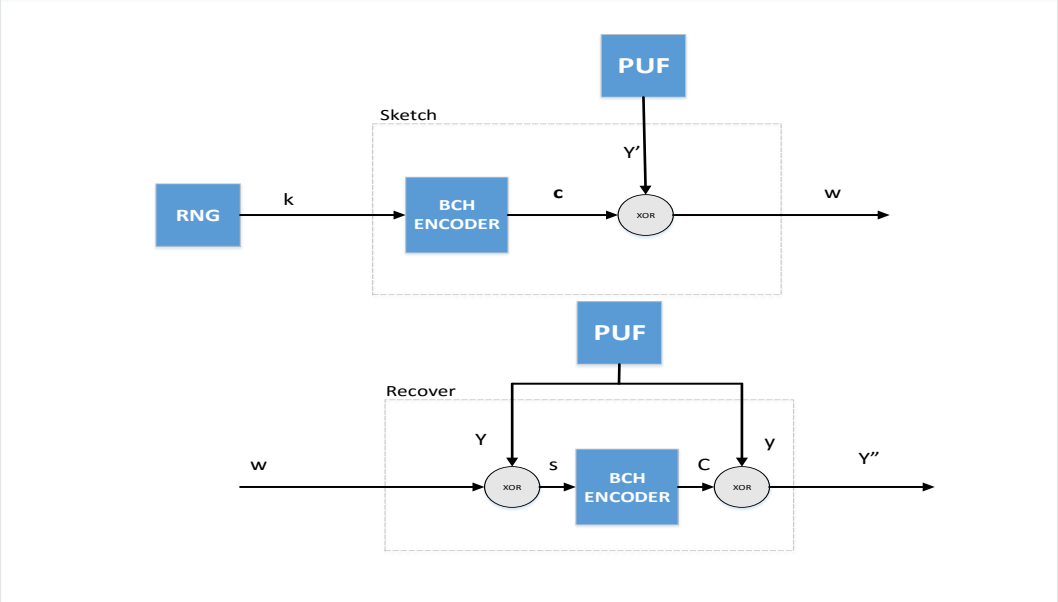

Figure 2.2: Fuzzy Extraction

This effect needs to be corrected as well, using a strong extractor. Strong extractor is a function that restores randomness in a PUF output, at the expense of the number of output bits.

Reverse Fuzzy Extraction

Reverse Fuzzy Extraction is an error correction mechanism used to remove noise. Reverse Fuzzy Extractors [9] are used to compare two PUF responses which are noisy. The fuzzy extractor design comprises of a Sketch and Recover procedures. Fuzzy extractor design makes use of BCH encoding and decoding algorithms.

Sketch procedure takes the noisy PUF response,y' and a random number, k as inputs. The random number is encoded using a BCH encoding algorithm into a codeword,c. Helper data,

w is calculated by performing a XOR operation on codeword and PUF response, $w = y' \oplus c$.

Recover procedure runs on the server. Recover procedure takes the original PUF,y response stored in its database and the helper data,w as its inputs. A syndrome,s is calculated by performing xor on helper data and PUF response, $s = y \oplus w$. Syndrome is decoded using BCH decoder algorithm to a value e. PUF response, $y"$ is reconstructed by $y" = e \oplus y$.

When using a fuzzy extractor, we have to ensure that the number of bit errors between the original and noisy PUF is within the error capacity of the BCH code.

Strong Extractor

Strong Extraction is a technique used to reduce the biased output rate of a PUF to the true information rate. Since PUF responses are biased, we need to use strong extraction to ensure maximum security. Cryptographic hash functions are used to build strong extractors [3]. In our implementation, we use SHA-256 hash function.

2.3 Mutual Authentication Protocol

Mutual Authentication Protocol [13] has 2 phases- Setup Phase and Authentication phase [13]. The server at the manufacturer end has access to secure database. The server is responsible for authenticating device at the user end.

Setup Phase

In the setup phase, manufacturer enrolls one PUF response along with the board serial number in its database. This is a one-time enrollment. The PUF enrollment process is secure and the interface used in setup phase is destroyed after enrollment.

Authentication Phase

After the enrollment, the device is sent to the user. The device is now installed at the remote location. The server initiates the mutual authentication protocol whenever the manufacturer wishes to verify the authenticity. After initiation from server, device responds with serial number, helper data and a random nonce($r1$). Server recovers the noisy PUF response using the helper data. The server and device now authenticate by comparing the hashes computed on noisy PUF data, (u_2) and recovered noisy PUF data, (u_1).

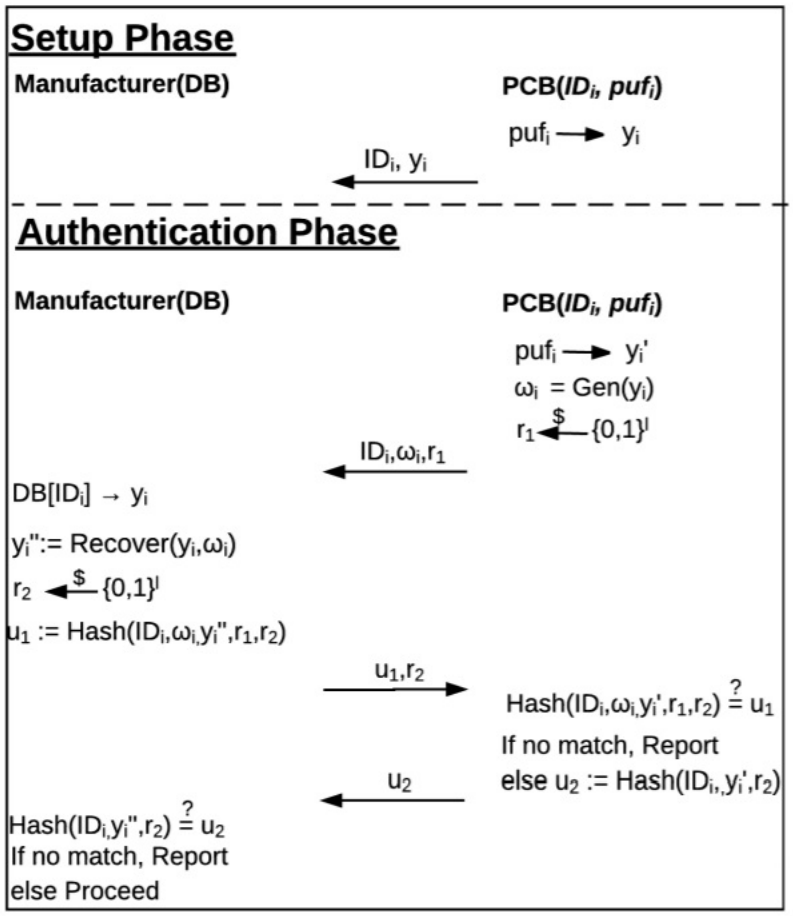

Figure 2.3: Mutual authentication protocol on a PCB with a public identity ID_i. r_1 and r_2 are true random numbers; y_i, y_i' are PUF outputs (on a fixed challenge); y_i'' is a reconstructed PUF output; ω_i is helper data.

Chapter 3

Flash PUF

3.1 Flash Memory

In this chapter, we present a novel PUF design for NOR based Flash memory. Before offering the design, we will briefly discuss Flash memory operating principles and source of manufacturing variation in flash memory.

3.1.1 Flash Cells

Flash memory is a type of non-volatile memory i.e. they retain information even if the power supply is switched off. Flash memory is composed of an array of floating gate transistors. An extra layer, a floating gate (FG), is sandwiched between the control-gate (CG) and body. The floating gate is an insulated conductor surrounded by oxide and is electrically isolated.

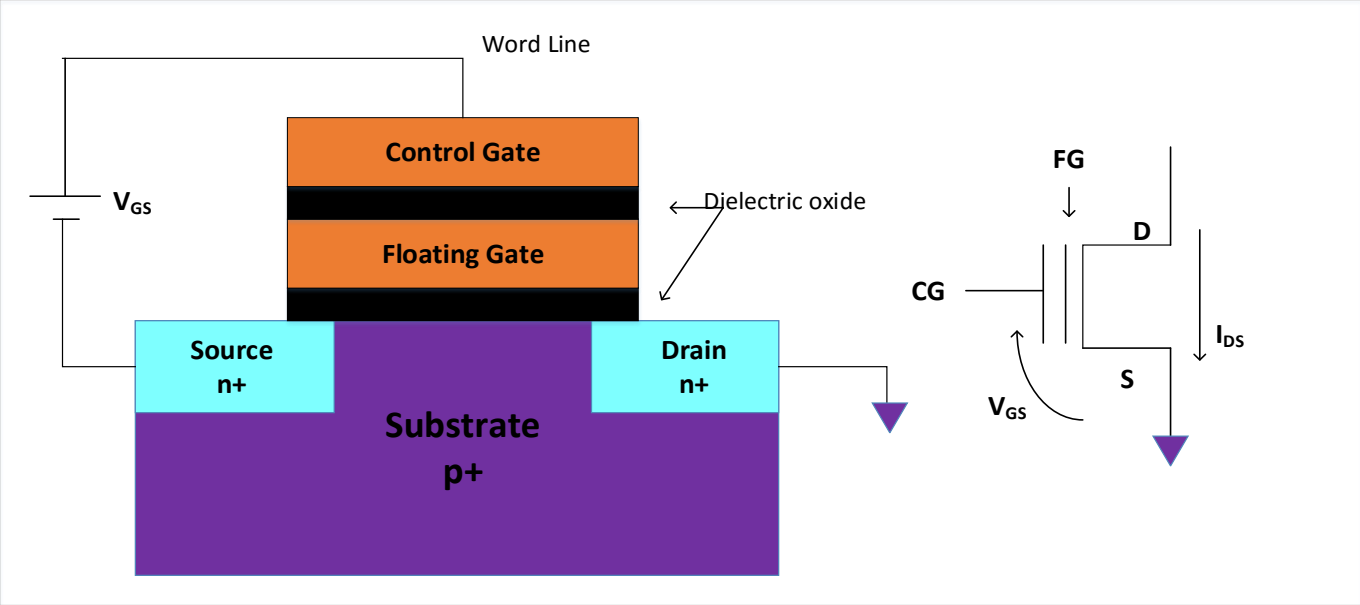

Figure 3.1: Floating Gate transistor

This allows any charge stored on floating gate to remain for a long time, maybe years.

Flash memory stores information as presence or absence of a trapped charge on FG. Flash memory cells with the negative charge on the FG are encoded as bit value '0' and cells without a charge corresponds to a bit value '1'. The presence or absence of charge on the FG controls the drain current flowing from drain to source when a gate voltage, V_{GS} is applied. Depending on whether I_{DS} is low or high, a cell is encoded as either 0 or 1.

Threshold Voltage

The threshold voltage, V_{th} is the minimum voltage required to be applied at the control gate to make transistor conductive. In other words,V_{th} is the minimum voltage required to be applied at control gate, for the current to flow from drain to source, I_{DS} . The threshold voltage is influenced by the presence of negative charge on the floating gate. The threshold

voltage is proportional to the amount of charge on the FG. In other words, a cell at logic state "0"(program state) has a higher threshold voltage than at logic state "1"(erase state). Fig. 3.4 shows the threshold voltages(V_{T1} and V_{T0}) at both the logic states.

3.1.2 Array Organization

Based on the arrangement of cell matrix, there are 2 types of Flash memory- NOR Flash memory and NAND Flash memory. Our work is focused on generating PUF data from NOR Flash memory. Existing approaches till today are focused on NAND based Flash PUF. Our work is the first contribution to the design of PUF from NOR based Flash memory.

Fig.3.2 illustrates the NOR Flash memory array organization. Due to the array organization, each cell can be accessed by its specific row and column address. NOR Flash layout connects the floating gate transistors in parallel [4]. This resembles the layout of a NOR gate.All the floating gate transistors have connections to bit line(BL), word line(WL) and ground.

In general, a NOR flash memory is grouped into blocks of size 64Kbytes to 128Kbytes. Thousands of such independent blocks make up a NOR Flash memory chip. The erase operation is performed on an entire block at a time while programming can be performed on a single byte at a time. In Fig.3.2, the drain terminals are connected through a shared bit line (BL), while the gate terminals are connected through a shared word line (WL).

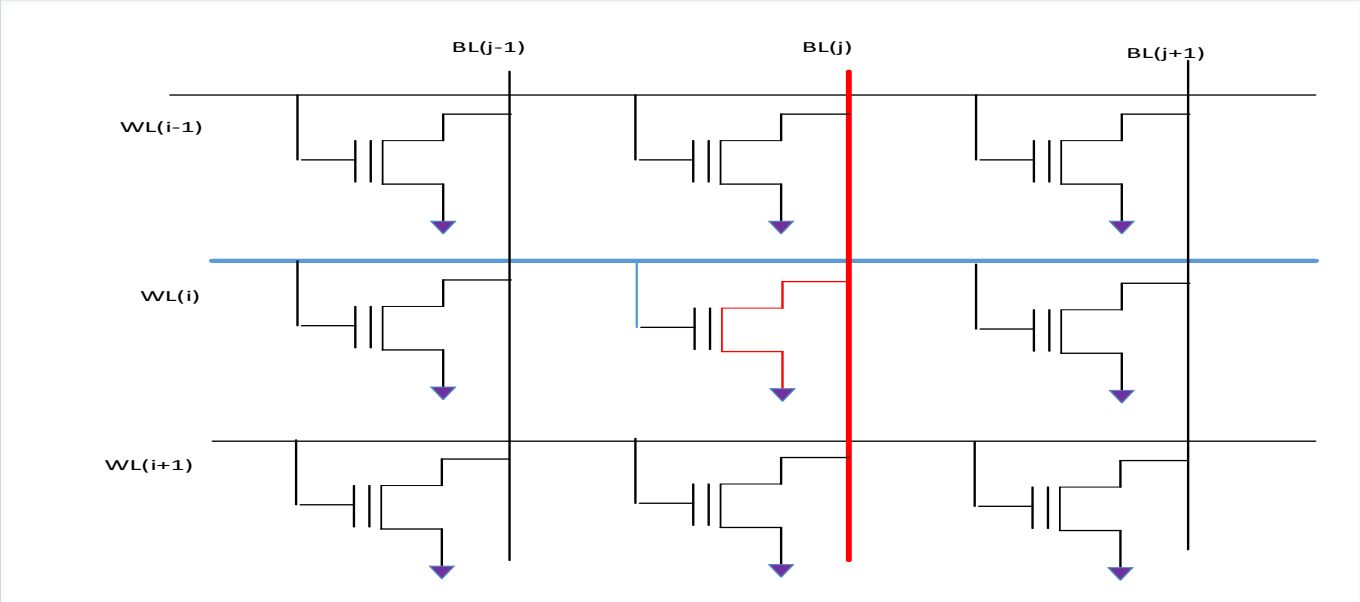

Figure 3.2: NOR memory architecture showing the biasing of wordlines and bitlines during the program operation; the cell under program is W_{ij}

3.1.3 Operations on NOR Flash

Reading a Flash Memory

The presence of a negative charge on FG implies that the cell is at state "0". A negative charge on the floating gate screens off some positive charge on the control gate. This implies that we need more charge on the control gate to get the device reach threshold. For a bit value "0", we need higher voltage to reach the threshold than when bit value is '1'.

Fig. 3.4 is plot for channel current, I_{DS} vs gate to source voltage, V_{GS} for a fixed drain to source voltage. Since the threshold is higher for a bit of value 0, the curve is shifted right to higher voltages. In order to read value on any floating gate transistor, we apply an intermediate voltage in between two threshold voltages as a gate voltage and measure the

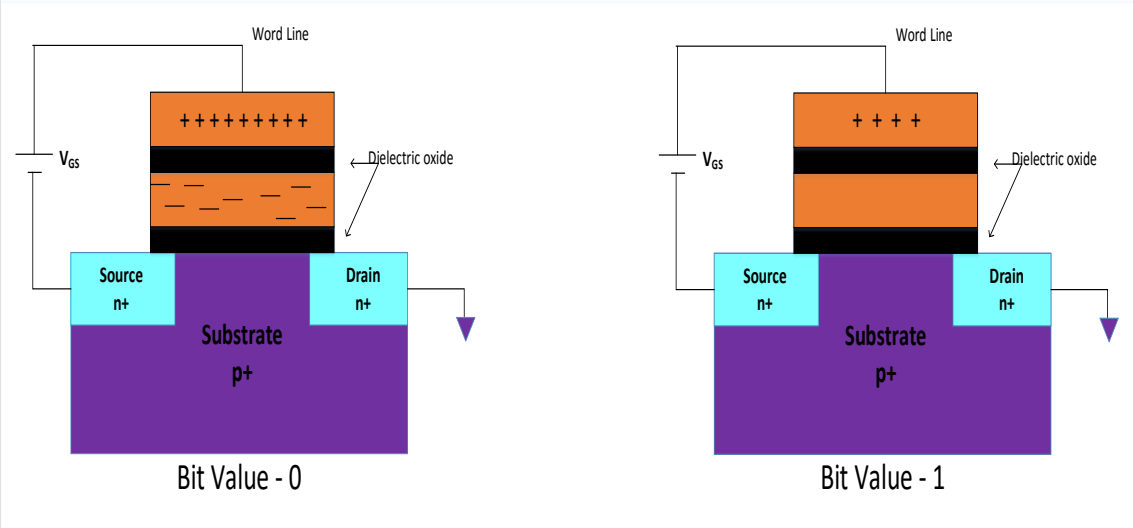

Figure 3.3: Flash memory cell at different states

current. No current implies that we are on the right side of the curve and the bit value is 0 whereas positive current implies that we are on left curve and bit value is 1.

To read from an array: In order to read a single byte at an address location, we apply an intermediate voltage to all the gates of transistors on a Word Line and measure the current at the Bit line. We apply a low voltage to other Word Lines to block the current from other transistors. This is to ensure that current we detect is only through the WL transistors we are trying to measure.

Programming a Flash Memory

Programming a flash memory requires writing a value "0". Since there is no electrical contact to the floating gate, charges must be sent through the oxide layer. NOR flash uses an approach called Hot Electron Injection [2] to get electrons on the floating gate.

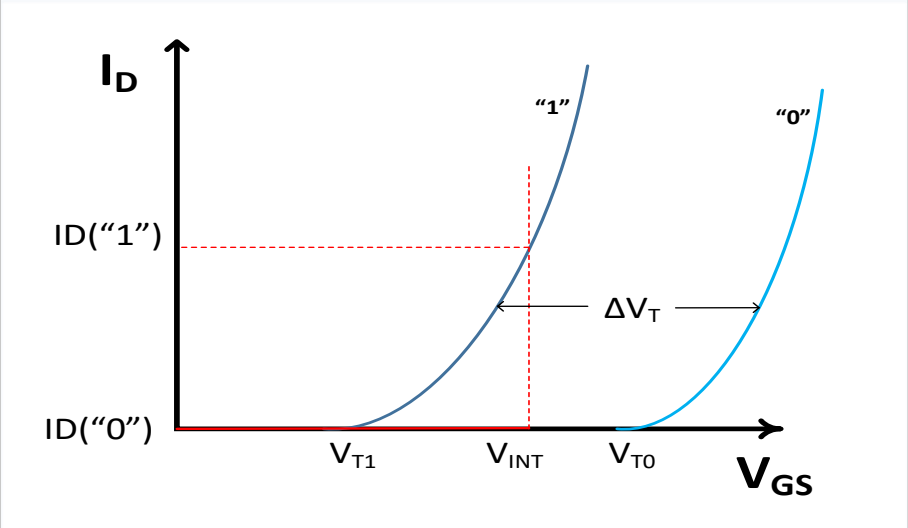

Figure 3.4: Threshold voltage graph while reading

In Hot Electron Injection method [2], we apply source to drain voltage. The electric field between source and drain accelerates the electrons. If they are accelerated enough, they will gain enough kinetic energy to hop over the oxide layer.The kinetic energy of the electrons is a function of the free mean path d between collisions and the electric field E. It is given by E_k = qEd. If the kinetic energy exceeds the barrier energy than electrons make it across the barrier. Electrons move across the oxide layer and end up on the floating gate. Just applying high source to drain voltage is not enough. We also have to apply a gate to source voltage for two reasons. First, there have to be electrons in the channel for charges to end up on floating gate. Second, there needs to be some electric field to draw electrons onto the floating gate.

To Program an array: To program a single bit at position,ij in Fig.3.2, we apply high voltage to the bit line,$BL(j)$ and gate voltage, V_{ON} to transistors on word line,$WL(i)$. To avoid programming of other transistors on bit line,$BL(j)$, we apply gate voltage, V_{OFF} (0

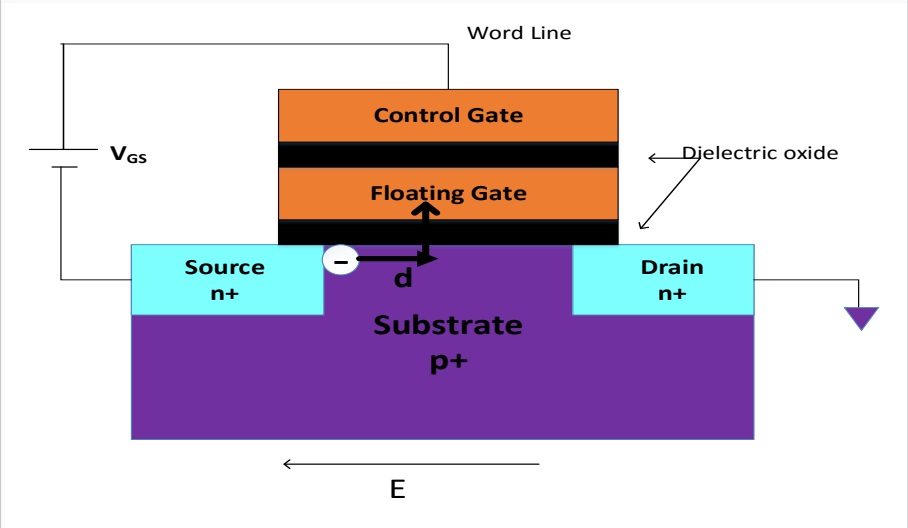

Figure 3.5: FG while programming i.e storing a value 0

V), to all the other word lines. Fig. 3.2 depicts programming of a single cell location in an array.

Erasing a Flash Memory

Erasing a flash memory requires writing a value 1. Programming a flash memory requires performing erase operation first followed by the program operation.

During erase, all the bits are pushed to a state "1" by removing all the negative charge from the FG. This is done through quantum tunneling for NOR [2]. When a high gate voltage is applied to all the FG transistors, the electrons on the FG are attracted to CG and tunnel through the oxide layer into the control gate. All the charge on the floating gates ends up as 0, which implies that transistor is in a state "1".

All these operations involve high voltage and high electric field. This limits the number of

times we can write to floating gate transistor. Electrons gain a lot of energy and dissipate that energy by colliding with oxide layer lattice and that damages the oxide lattice. This damage builds up over time. Flash Memory usually operate for 100k write cycles.

3.1.4 Sources of Variation

Process variations make every bit unique. Threshold voltage responsible for programming or erase is one such variation. The distribution of threshold voltages over all the cells in a flash memory is different owing to process manufacturing variations [16]. However, digital interfaces are built to hide such variations. One way of exposing such variation is by using *Partial Programming*.

Partial Programming implies programming a cell for a fixed amount of time, T. This time, T is much less than the programming time required for the flash memory cell to be programmed. So a cell requires certain number of partial programming operations to flip its states from logic state "1" to logic state "0" [11].

Flash PUFs can be extracted using this technique called *Partial Programming*. The initial and after-erase threshold voltage,V_{th} for a flash device are different from cell to cell owing to process variations. Hence, each cell in flash memory will require a different program time to change state. This implies that each cell will require different number of partial programming pulses. Cells with lower after-erase("1") threshold voltage (V_{th}) require more iterations of partial programming to flip the cell value(programmed to "0").

The number of partial programs required for a partial program time, T varies significantly from bit to bit, page to page and chip to chip. Therefore, this can be potentially used as a unique PUF function.

3.2 Existing Approaches

Yang et al. introduced a fingerprinting scheme based on partial programming [15]. In this approach, a page on a flash chip is partially programmed repeatedly. After each partial program, some bits will have been programmed enough to flip their states from 1 to 0. For each bit in the page, we record the order in which the bit flipped. The order of bit flip is the unique fingerprint. A short partial programming time provides a better resolution to distinguish different bits with the cost of increased fingerprinting time. Algorithm 1 shows the procedure used to obtain fingerprints from flash memory.

The resulting signature with this approach is noisy. Extracting signature from a single address location repeatedly did not yield the same signature. The observed reliability for this approach, when implemented on NOR flash memory was calculated using average intra hamming distance. The minimum and maximum intra HD obtained was 22.5% and 37.5% respectively. As a result of high uncertainty of device fingerprints, the above mentioned technique could not be used to extract unique and reproducible secret keys.

One reason for high noise in this implementation may be due to array matrix arrangement and methods adopted to program, erase, read Flash memory. Electrical influences exist

Algorithm 1 Extract Order in which bits in a page reach programmed state

```
1: procedure MyProcedure
2:     Choose a partial programming time T (below the rated program time).
3:     Nbits ← number of bits in one page;
4:     Order ← 1;
5:     Initialize BitRank[Nbits] to 0.
6:     do {
7:         Partially program a page for T;
8:         For all programmed bits do
9:             BitRank[programmed bit] ← Order;
10:        End for
11:        Order = Order + 1;
12:    }while(atleast 99% of bits in the page are programmed)
13: end procedure
```

between adjacent NOR Flash memory cells. Due to the memory architecture, there will be some cells with high voltage on the gate but 0 V on drain, and some other cells with a high voltage on drain but 0 V on gate. These cells may suffer disturbance. For instance, erased cells(no charge on FG) sharing the same word line(with another cell being programmed) might be programmed due to Fowler-Nordheim tunneling [2] and can show greater threshold voltage at the end of programming. In other words, a higher threshold voltage is observed due to the movement of electrons to FG during program operation in adjacent cells. This inconsistency in the threshold voltage, due to read, program, erase operations on adjacent cells, can be one of the reason for the noise causing uncertainty in device fingerprints.

From this, we concluded that noise on the adjacent cells can be reduced by limiting the program and read operations on a flash cell after erasing a block so that tunneling effect is minimum. Our approach discussed in the next section extract fingerprint by limiting the partial program operations. Instead of partially programming the address location till all the bit line cells are flipped, we only program it till at least one bit line cell is flipped.

3.3 Proposed Method

In this section, we propose a flash memory based PUF implementation. Our proposed design repeatedly performs partial programming on an address location to exploit the threshold voltage variations in flash memory. In our PUF design, the challenge is the address location of flash memory and response is the bit position of the cell with minimum threshold voltage transistor. Unlike the previous approach, in which partial program operations are performed till all the bits are flipped, we perform limited number of partial program operations. We assume that this should limit the noise in flash memory to a great extent.

The first step in the algorithm is to choose an address location and partial program time T. The block which has address location, A is erased and then programmed for a duration of T. After each partial program, the address location is read and the output is stored. We observe if any of the bit locations are flipped. If none of the bits are flipped then we perform repeated partial programs till atleast one bit has been flipped. If multiple bit locations are flipped, then we decrement the partial program time and start the process again. When only

one bit has been flipped, then the location of the flipped bit is encoded as 3-bit binary form and this is stored as the response of our PUF.

The core idea of our approach is that, with a good precision in the partial programming time, a floating gate transistor with minimum threshold voltage will be the first one to be programmed. This procedure is explained in Algorithm 2.

In the proposed approach, for an example of 24 bit challenge(8MB flash memory), the output response is 3 bit(Flash memory with 8 bit mode).To obtain a fingerprint with more bits, we perform the above operation on a continuous sequence of address locations and concatenate the responses. Algorithm 3 gives details to extract fingerprints.

Algorithm 2 Proposed Algorithm for Flash PUF

```
1: procedure MyAlgorithm
2:     Choose an address location A.
3:     Choose a partial programming time T (below the rated program time).
4:     top:Perform Erase operation on the block consisting of address A
5:         Initialize Output to 0.
6:     loop: Partially program at address A for T;
7:         Read the data at the address A
8:         if atleast one bit posistion is flipped
9:             if only one bit has been flipped
10:            output ← Bit position of flipped bit
11:            exit
12:            endif
13:            elseif more than one bit have been flipped
14:            T ← T-1;
15:            goto top
16:            endif
17:        endif
18:        elseif No bit position has been flipped
19:            goto loop
20:        endif
21: end procedure
```

Algorithm 3 Generate a n-bit binary signature from Algorithm 2

```
1: procedure FlashPUFAlgorithm
2:     Choose an address location A.
3:     address ← A
4:     Choose number of address locations(N) based on n-bit response required;
5:     Order ← 0
6:     Choose a partial programming time T (below the rated program time).
7:     Perform Erase operation on the block consisting of address A
8:     Initialize finalOutput[N*3] to 0.
9:         do {
10:        Perform MYALGORITHM on address A
11:        Output[A]
12:        Order ← Order+1
13:        A ← A+1
14:        } repeat till Order is equal to N
15:    finalOutput ← {Output[N-1].....,Output[1],Output[0]}
16: end procedure
```

3.4 Implementation and Results

3.4.1 Implementation Details

We have implemented our design on a Spansion S29GL064N, 8MB NOR flash memory which is CFI (Common Flash Interface) compliant. Spansion provides a CFI flash control interface that describes flash programming, erase and read algorithms and provide device size and block configuration. CFI provides set of commands to be communicated to flash memory via address and data bus to perform erase, program, read on flash memory.

Our design comprises flash memory controller, JTAG-UART on FPGA and the proposed Algorithm 2 & Algorithm 3 implemented in tcl on host system. We designed a flash controller to performs erase, program and read operations for flash.

Software on the host system inputs Partial program time,T, address and operations(erase,write,read) to be carried out on flash memory. This information is communicated to the flash controller via JTAG-UART through avalon memory mapped interface.

Our flash controller is implemented on Altera FPGA and provides an interface to 8 bit data bus flash memory using Common Flash Interface(CFI). Controller communicates asynchronously. Flash memory requires specific time duration for which `oe,ce,we,reset,dq,addr` signals should be kept active to perform flash operations. This information is provided in the datasheet.We implement a state machine according to the timing specifications for each command set provided by CFI. We designed the controller to perform partial program by pro-

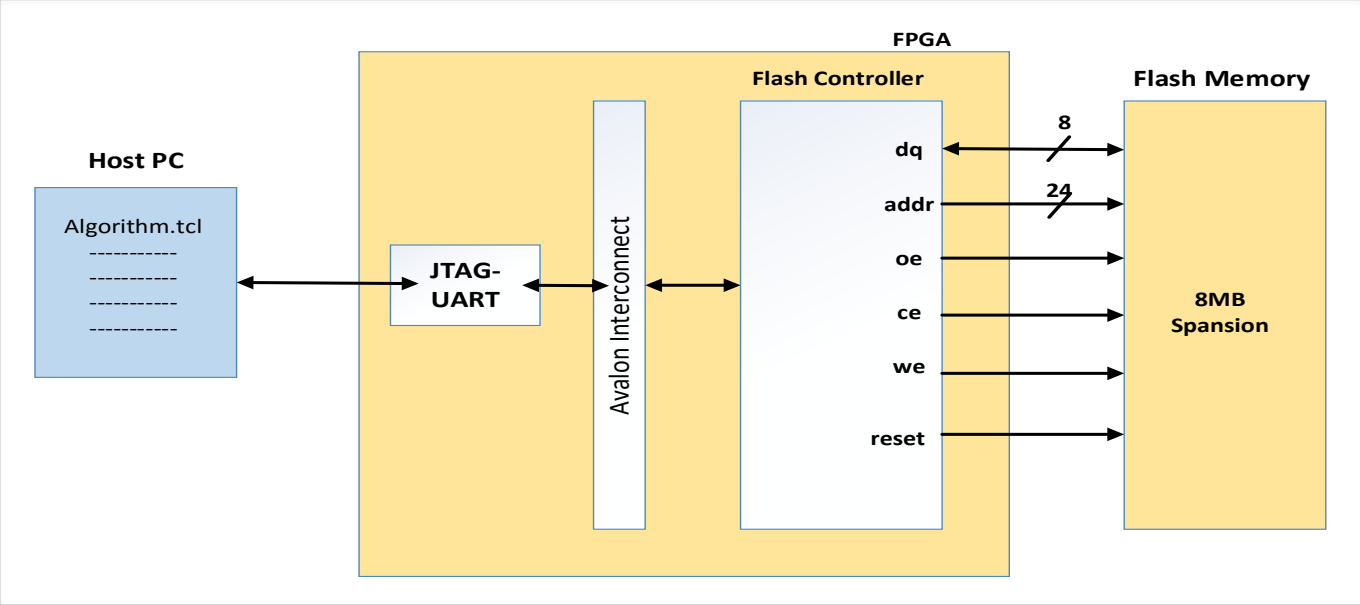

Figure 3.6: Design Implementation of Flash PUF

gramming the flash for a given time duration T and then giving a hardware reset. Hardware reset influences the power consumption(voltage on wordline and baseline), thus aborting the program.

Our design uses a 50MHz system clock. System clock is to ensure that signals are active for the required write and read access times and also to control partial program time. NOR Flash memory in our design has a read cycle time and write cycle time of 90ns(approx 5 cycle counts).

3.4.2 Results

This section presents results of implementing proposed algorithm. First, we present the quality metrics of the designed PUF and then we discuss the hardware implementation cost.

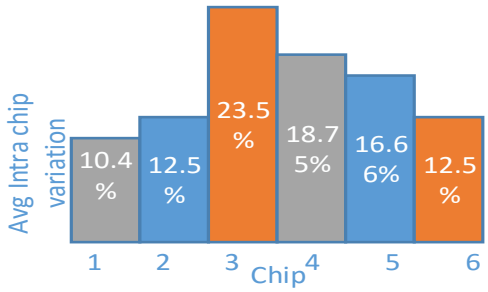

Figure 3.7: Intra chip Variation for Flash PUF design

We analyzed the data collected from 6 NOR based Spansion Flash memories. We collected 5 samples of data(at 21 address locations) for each board.

Reliability

Fig: 3.7 shows the distribution of the intra-chip Hamming distance among all the samples taken from 6 different chips. The average is 15.7% with a maximum value of 23.5% and minimum value of 10.4

Uniqueness

Fig: 3.8 shows the distribution of the inter-die Hamming distance between each of the 6 NOR Flash memory chips. In Fig: 3.8 x-axis gives the comparison made between 2 different samples among 6 flash memories. The average is 41.63% with a maximum value of 56.2% and a minimum value of 31.25%.

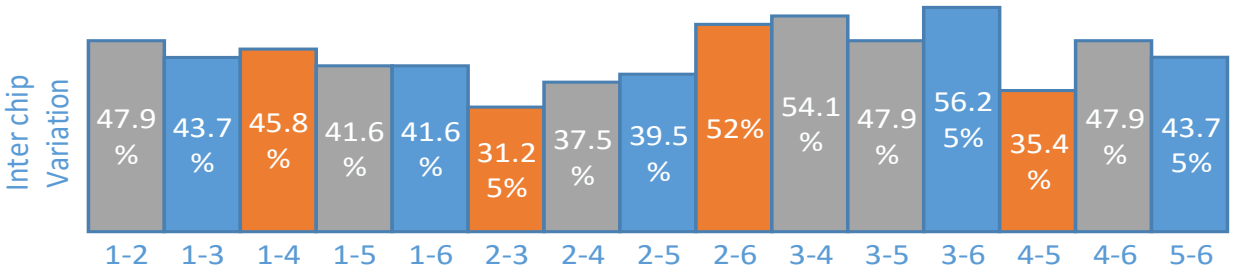

Figure 3.8: Inter chip Variation for Flash PUF design

Hardware Cost

Module	LC combinational	LC Registers	Memory Bits
Flash Controller	383	108	0
JTAG-UART	318	57	0

It is observed that this approach has eliminated noise in the PUF design to some extent compared to existing designs. But compared to other PUF designs (SRAM and RO PUF), Flash PUFs remain highly noise sensitive, and they cannot be used for practical applications yet.

Comparison with NAND Flash memory

Our work is the first attempt on producing PUF from NOR Flash memory. All the existing work on Flash memory PUFs was performed on NAND flash memory. The major difference between NAND flash memory and NOR flash memory is in their array architecture. NAND flash memory has bitline transistors in series and NOR has them in parallel.

Yang et al proposed an approach for deriving PUF and measured the reliability and uniqueness in terms of Pearson correlation coefficient [14]. His approach was implemented on couple of different NAND flash memory chips. The average intra chip correlation coefficient was 0.9673[15]. The intra chip variations in NAND Flash memory chips through partial program approach was typically 14.42% [11].

The results obtained for NOR Flash memory in our approach, when compared to the existing work on NAND flash memory, were found to be a lot noisier. NOR Flash memory was more erroneous in terms of reproducibility when compared to NAND. The cause of error in PUF data can be due to different array architectures influencing differently on a single cell when operation are performed on an address location for PUF design. Further research needs to be done, by thoroughly analyzing the physics behind flash memory cells, to study the cause of error in NOR flash memory than in NAND flash memory.

Chapter 4

Hardware Interface for Fusion PUF

In this chapter, we provide a hardware based correction mechanism to eliminate noise in PUF based authentication. We initially propose a hardware interface and then discuss operations that can be performed on the hardware. We then provide implementation costs for the design.

In the authentication protocol, manufacturer extracts the secret PUF response from the device and stores it in his database. After remote installation, server at manufacturer end tries to verify the authenticity of device by regenerating the PUF response. Since the secret key from PUF applications are noisy, the regenerated PUF response is noisy. This requires additional analysis to come up with a best available mechanism to eliminate the noise.

One solution is to use Reverse Fuzzy Extraction mechanism [9]. Fig.4.1 shows the working of a Reverse Fuzzy Extractor. Sketch procedure is implemented on the device. It takes the

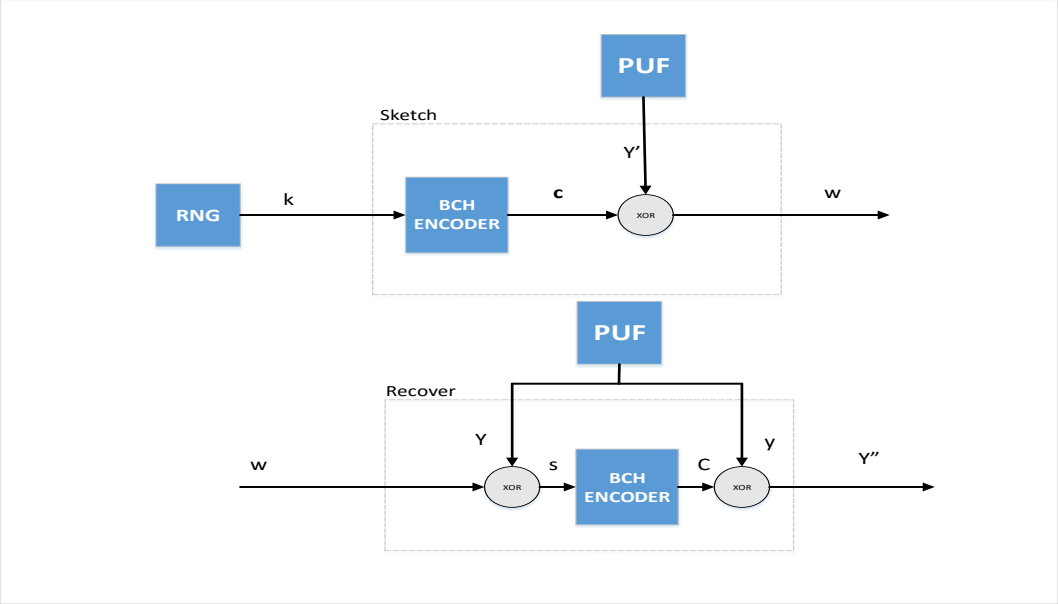

Figure 4.1: Fuzzy Extraction

noisy PUF response and calculates helper data using BCH Encoding and XOR operation. This helper data is communicated to the verifier via an authentication protocol.

Recover procedure, at the server, reconstructs the PUF response using original PUF response stored in its database and the helper data. If the number of bit errors between original and noisy PUF are within the error correcting capacity of the BCH code, then the reconstructed PUF response matches the noisy PUF response and the device is authenticated.

In our PUF based authentication protocol, a fingerprint for the board is constructed by cryptographically combining PUF based fingerprints from its components, such as performing XOR or concatenating bits. This merged PUF data is the input to the sketch procedure in Reverse Fuzzy Extraction Module. Mutual Authentication Protocol introduced in chapter 2 authenticates the device by comparing the hash on the PUF response, board ID and random

nonce.

In our solution for PUF based mutual authentication protocol, we propose a hardware implementation comprising a sketch procedure in the Reverse Fuzzy Extraction Module to generate helper data from the merged PUF response and a SHA256 hardware implementation to obtain hash data.

In our implementation design, a 255 bit helper data is constructed from 255 bit PUF response obtained by concatenating 128-bit SRAM PUF and 127-bit RO PUF with an error rate of 25 bits.

4.1 Hardware Interface

Fig. 4.2 depicts a hardware interface to the module. The hardware interface is a synchronous interface with input/output samples at the rising edge of the clock. Module has a BCH Encoder and SHA256 implemented on hardware.

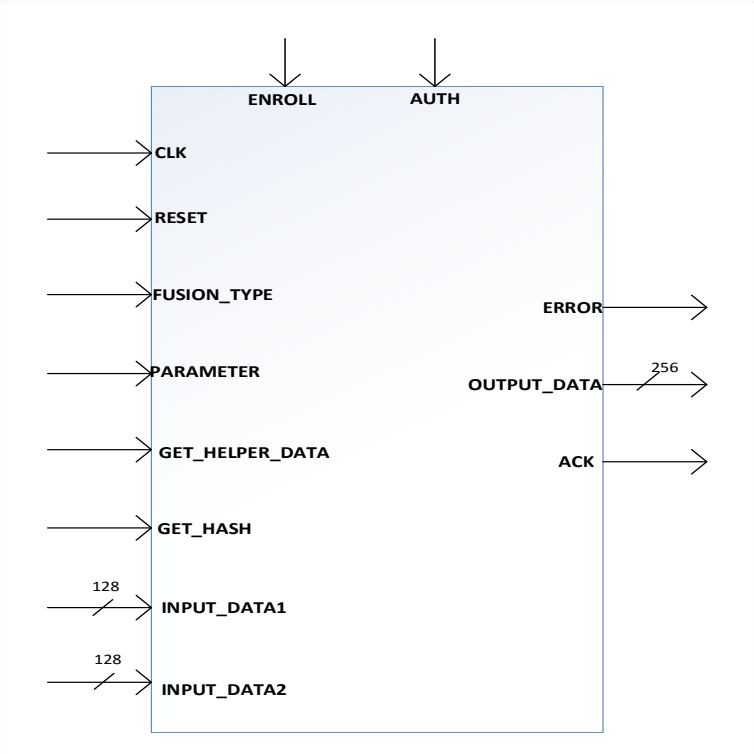

Figure 4.2: Hardware Fusion PUF Interface

Port Definitions

CLK : Clock signal. All signals are sampled at the rising edge of this clock.

RESET : Reset signal.

FUSION_TYPE : When FUSION_TYPE is high, PUF bits from different components are concatenated. When low, bits are XORed.

PARAMETER : When PARAMETER is high, signals sampled on INPUT_DATA are parameters to BCH encoder and SHA.

GET_HELPER_DATA: When GET_HELPER_DATA is high, it signals the hardware to

perform Reverse Fuzzy Extraction on inputs.

GET_HASH : When GET_HASH is high, it signals hardware to perform a hash on inputs.

INPUT_DATA1 : INPUT_DATA1 inputs the PUF data from one of the components when GET_HELPER_DATA is high. When GET_HASH is high, input to hash are sampled.

INPUT_DATA2 : INPUT_DATA2 inputs the PUF data from one of the components when GET_HELPER_DATA is high.

ACK : When ACK is high, it signals the availability of data on OUTPUT_DATA.

ERROR : When ERROR is high, it indicates error inside the module.

We have implemented the module on Altera DE2-115 board with Cyclone IV. Our hardware design makes use of Avalon Memory Mapped interface with a master-slave connection. The proposed hardware fusion PUF design is implemented as a slave interface to JTAG_UART, wrapped around BCH module and SHA256 module.

4.2 Operations

Mutual Authentication protocol has 2 Phases. Enrollment Phase and Authentication Phase. During enrollment, merged PUF response from different components is obtained without error correction mechanism. In the authentication phase, helper data is generated initially and hash output is obtained in the later stages of the protocol.

Parameters for BCH Encoder (size, error correction capability) and SHA256 (input size) are

predefined and communicated to the design by setting PARAMETER high and sampling data at INPUT_DATA1.

Enrollment Phase

When the ENROLL goes high, the device merges PUF data from INPUT_DATA1 and INPUT_DATA2 according to the signal at FUSION_TYPE. After the merge, module sets DONE high signaling the availability of PUF response at OUTPUT_DATA.

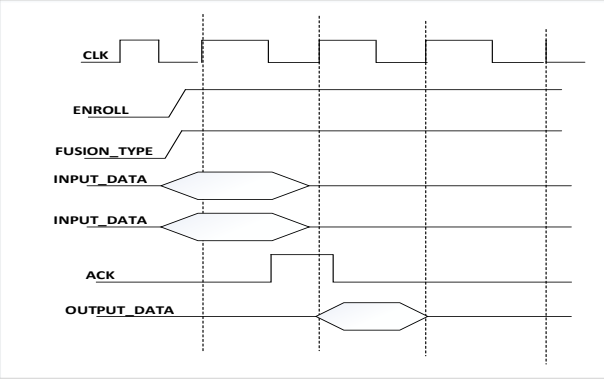

Figure 4.3: Timing diagram during Enrollment

Authentication Phase

GET_HELPER_DATA: When the server initiates the authentication, it is communicated to the module on AUTH. When AUTH is high and GET_HELPER_DATA is high, we merge PUF data from INPUT_DATA1 and INPUT_DATA2 according to the signal at FUSION_TYPE. This PUF response is given to the sketch module in Reverse Fuzzy Extraction. After wait cycles, the module signals DONE high signaling the availability of helper data at OUTPUT_DATA.

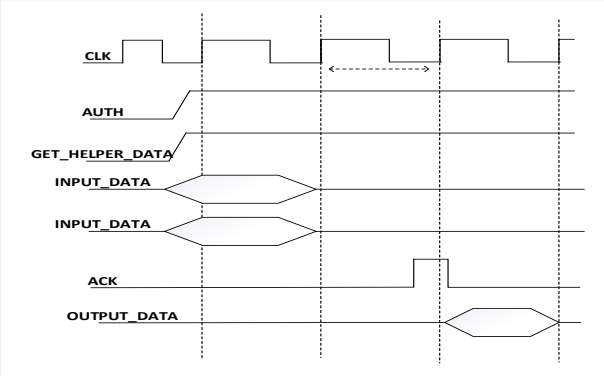

Figure 4.4: Timing diagram during Authentication: To extract helper data

GET_HASH : When AUTH is high and GET_HASH is high, the input sampled at INPUT_DATA1 is given to SHA256 module. High on DONE indicates the availability of hash output on OUTPUT_DATA.

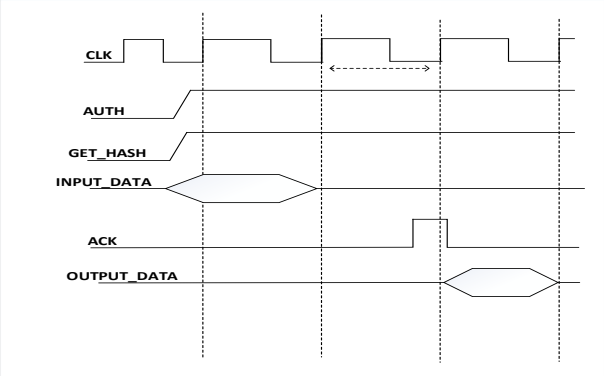

Figure 4.5: Timing diagram during Authentication: To get hash value

4.3 Results

Hardware Utilization

Module	Combinational ALUTs	Dedicated Logic Registers	Memory Bits
BCH module	546	648	1296
SHA module	1453	3882	0
RFEM-Total	2179	4676	1296
JTAG_UART	142	112	1024

A prototype for our PUF based authentication protocol was performed on Altera DE2-115 Cyclone IV FPGA board running uClinux [1]. This prototype implements RFEM module on software which requires a Nios II processor and SDRAM for running uClinux OS. The total hardware utilization for FPGA components was found to use 7112 LE (Nios-II - 2254, SDRAM controller - 327, JTAG-UART - 146, PLL - 8, RO PUF- 4385, SRAM controller - 5) and 9 BRAM. Similarly, the memory requirements for software components was 1665.9KBytes(RO-PUF driver-38KB, RFEM module - 116.6KB, Board application - 35KB, uClinux Kernel- 1475KB). A Software implementation of RFEM requires a processor and memory resources. Reverse Extraction Module implemented on hardware eliminates the need to use software on the board for post processing of PUF responses.This replaces the use of Nios-II processor, SDRAM controller with RFEM module. Our hardware module implementation of Reverse Fuzzy Extractor can be used for resource constraint devices with no software overhead and less hardware overhead.

Chapter 5

System Demonstration

In this chapter, we demonstrate our PUF based authentication protocol design using RO PUF and network sockets. We first present basics to socket programming and then show implementation details. We discuss this implementation on DE1-SoC board.

The mutual Authentication protocol can be programmed using server-client communication model of computer networks. A client-server model is concurrent programming technique in which one of the programs (may be client) requests a service from the other program (maybe server) and the server complies with the request. In computer networks, it is common to create sockets at each end point of communication and use socket programming for server-client communication. We intend to use a similar approach for implementing our Authentication protocol. In our design, the verifier at the server end (similar to a client in client/server model) initiates the protocol and requests information (PUF data) from the device. The

device here acts as a server in client-server model.

5.1 Introduction to Sockets

Sockets were developed by University of California, Berkley as a programming primitive for networking. This concept is now expanded and is being used widely in network communications. Every network endpoint needs to create a socket before exchange of message with other network hosts. In the scenario of server/client model, sockets are created by the server program to listen to a connection request from a socket driven by a client program. Client application also creates a socket to connect and communicate with server socket.

When a socket is created, it has no identity. The identity of a socket is defined by the address, protocol and port number. In our design, the address is Internet address, transmission protocol is TCP, and the port number is assigned by the application for the lifetime of the socket. Both server socket program and client socket program assign the identity to their respective sockets. After identity assignment, all the connection requests are sent through socket from client and arrive through socket at server end.

5.2 Socket API

Sockets are supported as part of operating systems using system calls. The socket API is implemented through library functions, which provide the necessary abstraction to program-

mers and which allow them to abstract the internal operation of the network protocol [7]. API provides an interface at each communication end points to setup, modify and close sockets. The application developer uses this API to develop communication protocols on top of sockets, independently of the underlying network protocol. A socket can be created, read from, written to, and closed. Socket communication is defined using 4 parameters: Source Identifier (IP address); Source Port; Destination Identifier; and Destination Port. Each socket link is defined through these four parameters. Our design uses the following Socket API.

1. **socket**()

SOCKET *socket*(int *family*, int *type*, int *protocol*);

A protocol is initialized by calling the socket function. *socket*() opens a new socket and returns a non negative descriptor if OK (or) -1 if encountered with an error. The *family* is specified by one of the constants that are defined in sys_socket.h. Our implementation uses AF_INIT, which belongs to an internet family. Socket types are also defined in emph_sys_socket.h, which should be assigned as SOCK_STREAM- for stream socket, SOCK_DGRAM- for datagram socket. We use SOCK_STREAM for our implementation.

s = socket(AF_INIT, SOCK_STREAM, 0);

socket function initiates protocol value to a default 0.

2.**bind**()

int *bind*(SOCKET s, struct *sockaddr *addr*, int *namelen*);

socket() creates a socket without a name or a address value. *bind*() is used to bind the socket to a specified address in sockaddr datastructure. The address parameter specifies the IP address and port number. *bind*() is called on server side and then it accepts connection requests from client. *bind*() returns zero on success and SOCKET_ERROR when socket parameters are not valid.

sockaddr data structure

address family: AF_INIT
host IP: 172.168.1.249
port: 5000

3. **listen**()

int *listen*(SOCKET s, int *backlog*);

listen() is a function called on the server side for server socket to listen and accept connection requests from clients. *backlog* indicates number of connection that can be pending in queue. *listen*() flags an indication that server is ready to accept requests and allocates a new connection queue. It returns zero on success and SOCKET_ERROR on failure.

4. **accept**()

int *accept*(SOCKET s, struct *sockaddr* **addr*, int *addrlen*);

accept() extracts the connection request from the queue for the listening socket and creates a new connected socket for this client. This new temporary socket has a different name to that of parent socket and will be closed when the connection is no longer required. *accept*()

returns a descriptor to a new socket if successful else INVALID_SOCKET.

5. **connect**()

int *connect*(SOCKET s, struct *sockaddr* **addr*, int *addrlen*);

when *connect*() is called, it connects the socket s to the address specified in *sockaddr* structure (address, port, protocol). *connect* return zero on success and SOCKET_ERROR on failure.

6. **read**()

ssize_t read(SOCKET s, void **buf*, size_t *count*);

read() waits to read message on the connected socket. It reads up to *count* bytes into buffer with starting address *buf*.

7. **write**()

ssize_t write(SOCKET s, void **buf*, size_t *count*);

write() function writes up to *count* bytes from the buffer with starting address *buf* to the file specified by socket file descriptor. *write*() does not specify destination address as the function calls called before *write*, *connect*(for a client) or *accept*(for a server) already specify it.

8. **close**()

close(sockfd);

close() is called after all the communication has been done and we are ready to close the

connection on the socket descriptor.

5.3 Implementation - DE1-SoC board

5.3.1 System Design

We implement our design on Altera DE1-SoC board. It integrates Altera SoC FPGA with an embedded ARM processor. We use Altera 28nm Cyclone V FPGA for our hardware design(RO PUF). On the ARM cortex-A9 processor side, we have a HPS Gigabit Ethernet for high speed networking, 2 HPS USB host and a micro SD card. For memory on HPS we have 1GB DDR3 SDRAM and 1GB of QSPI Flash memory.

Hardware Architecture: Our design implements 256 ROs to generate 255 bit RO PUF data on the FPGA. The RO PUF driver communicates with the design using Avalon memory mapped interface. The driver has access to each RO through this interface to get the frequency of every RO and process it for PUF output.

Client Application: DE1-SoC has ARM core processor. Using the Linux system image on SD card(a peripheral to ARM), the board is setup to boot Linux. FPGA fabric and ARM are connected through AMBA bus. Software running on ARM comprises of RO PUF driver and Reverse Fuzzy Extractor Module (RFEM module introduced in Chapter 2) wrapped around a client side socket program (server program in client-server model).

Server Application: Application comprises BCH decoding scheme and a SHA-256 hashing

program with socket program wrapped around it. Requested Helper data received from the device is decoded and hashed as part of the authentication protocol.

5.3.2 Protocol Implementation

Fig. 5.1 illustrates the working of the protocol using socket calls. As already mentioned before, due to the nature of the protocol implementation, our verifier on the server end is a client and device is a server in the client/server model.

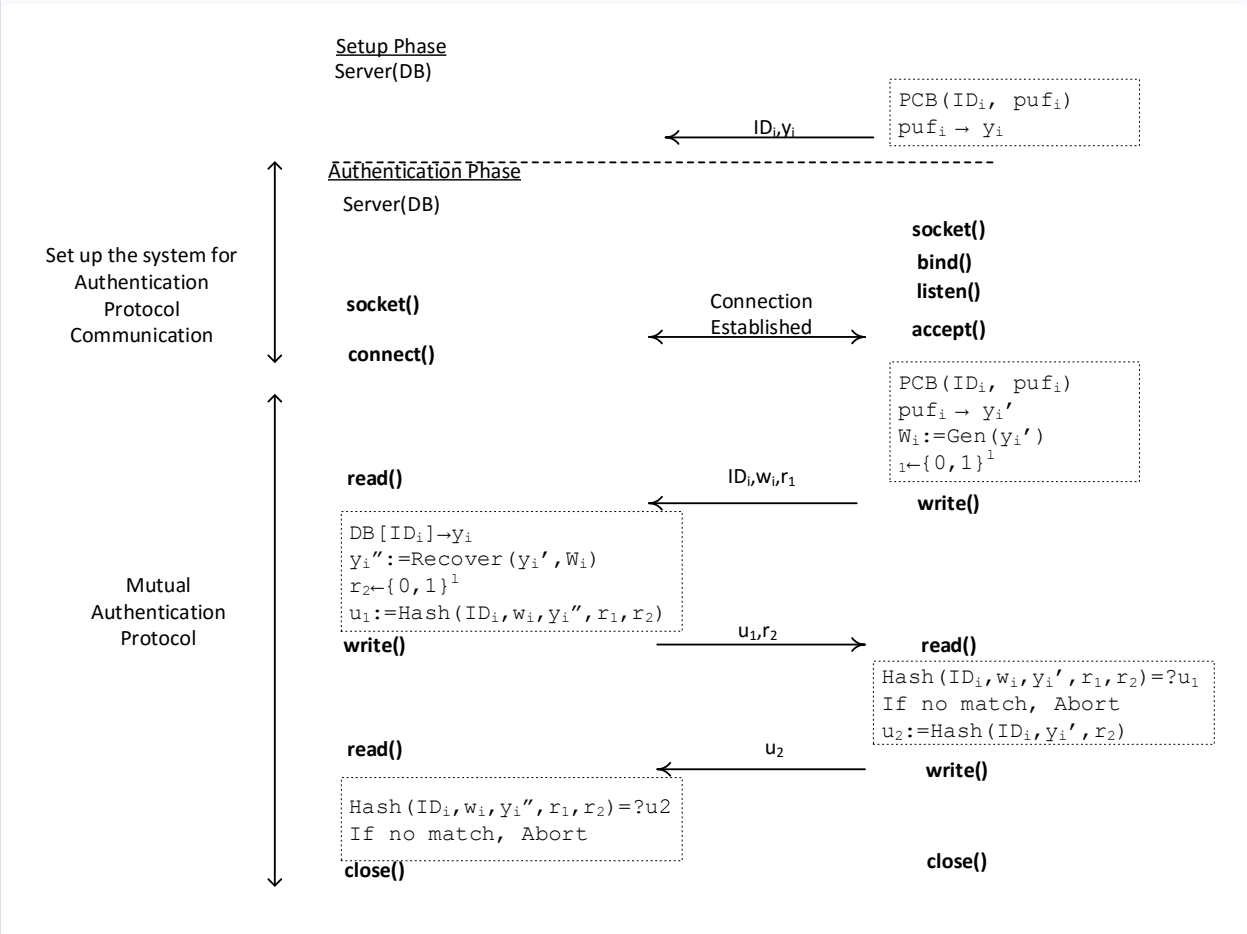

Figure 5.1: Protocol Communication using Sockets

In the first part of the protocol, device PUF data is securely enrolled into the server DataBase [13]. Client application is launched before the device is taped out to the consumer. By launching client application, we create a socket at the device end and bind it to address, port number. This enables the device to be setup for protocol communication. Device remains in wait mode till it receives a connect request from the verifier.

Serving a Verifier request: There are two major steps in the PCB authentication protocol. **STEP I**: When a verifier wishes to authenticate the device, a server application is launched. This sends an initiation to the device by first creating a socket and requesting a network connection with the device. When a device receives request, it establishes a connection with server and starts reconstructing the PUF data.

The 255 bit RO PUF data is communicated to the RO PUF driver through memory mapped interface. The RFEM module interacts with the RO PUF driver to generate PUF response. This response is then processed by RFEM to generate the helper data (w_i). This information along with board ID (ID_i)and random nonce (r_1) is sent to server using *write*() function call.

STEP II: Server process the received information by decoding and reconstructing the noisy PUF response ($y"_i$) from helper data. It computes hash value (u_1) from this information. Device requests the hash information and a random nonce (r_2) on the same connection by calling *read*() and server complies with this request. RFEM module is also responsible for hash data used in verification protocol. Device computes hash over the identification data (u_2) and sends it to the server for verification. Once the authentication phase is done, we

destroy both the sockets by calling *close*(). Figure 3.2 gives an overview implementation of our model on DE1-soc board.

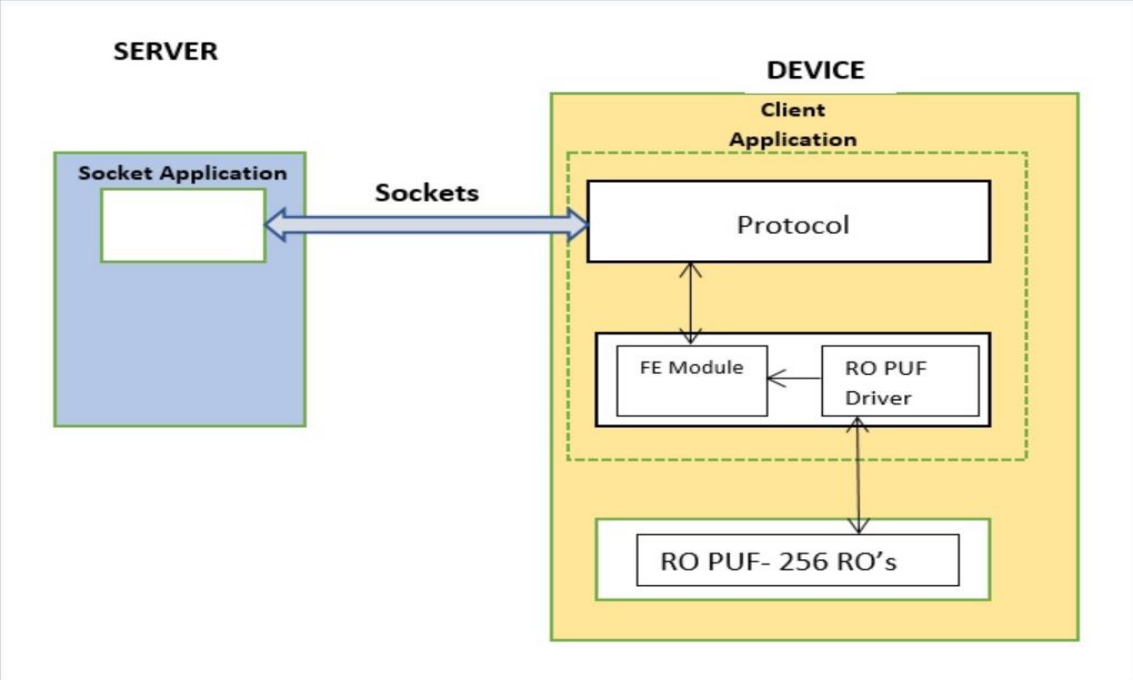

Figure 5.2: Overview of the implementation

5.4 Results

Hardware Utilization

Module	LC combinational	LC Registers	Memory Bits
RO PUF	4263	159	0

Software Utilization

Module	Codesize(Bytes)
RO PUF	3881
Client Application	48600
Server Application	38100

The prototype for our PUF based authentication protocol was performed on Altera DE2-115 Cyclone IV FPGA board running uClinux [1]. The device receives messages sent by server application over JTAG-UART. The total hardware utilization for FPGA components was found to use 7112 LE and 9 BRAM. Similarly the memory requirements for software components was 190.9KBytes(RO-PUF driver-38KB, RFEM module - 116.6KB, Board application - 35KB). We observed that our implementation using sockets minimized the software overhead required for an application to implement the protocol. On the hardware, our approach eliminated the need to use SDRAM controller, JTAG-UART and thereby minimizing the resource utilization on the FPGA. Using sockets over UART makes it a more practical approach as we do not require a physical link between server and client, which is typical scenario expected for server-device remote authentication.

Chapter 6

Conclusion

In this thesis, we present a design method for PUF based authentication protocol in 3 abstraction levels. At the Hardware level, fingerprints are extracted from various components on a PCB. We proposed a novel approach to extract PUF based fingerprints from NOR Flash memory component. We proposed a mechanism to reduce the noisiness of the Flash PUF, but we conclude that the noise level remains high compared to other types of PUF. In the hardware integration level, fingerprints obtained from various components on a PCB are merged cryptographically to obtain a Fusion PUF. The response from Fusion PUF was noisy and needs an error correction mechanism to eliminate noise. We proposed a hardware solution by providing a hardware interface for Reverse Fuzzy Extraction Module. The hardware interface eliminates the need for software support and this is useful in memory constrained environments. In the protocol level, we implemented a mutual authentication protocol using network sockets. We show performance improvements in terms of speed and

memory utilization while using network sockets for protocol communication.

Future Work

This thesis proposed a novel method to obtain PUF from flash memory component. There are many such critical components on PCB and generating fingerprints by combining all components will demonstrate the presence of these components on PCB. This research can be extended to find PUF from other components on PCB like SDRAM PUF, DDR PUF etc. This research can also be extended to perform better coding analysis for noisy PUF responses. The goal should be to minimize the hardware and software utilization for error correction modules.

Bibliography

[1] A. Aysu, S. Gaddam, H. Mandadi, C. Pinto, L. Wegryn, and P. Schaumont. A design method for remote integrity checking of complex pcbs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1517–1522. IEEE, 2016.

[2] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[4] L. Crippa, R. Micheloni, I. Motta, and M. Sangalli. Nonvolatile memories: Nor vs. nand architectures. In *Memories in Wireless Systems*, pages 29–53. Springer, 2008.

[5] L. Feiten, A. Spilla, M. Sauer, T. Schubert, and B. Becker. Analysis of ring oscillator pufs on 60nm fpgas. *European cooperation in science and technology*, 2013.

[6] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*,

pages 148–160. ACM, 2002.

[7] W. W. Gay. *Linux socket programming: by example*. Que Corp., 2000.

[8] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris. Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain. *Proceedings of the IEEE*, 102(8):1207–1228, 2014.

[9] A. V. Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A. Sadeghi, I. Verbauwhede, and C. Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 374–389, 2012.

[10] D. E. Holcomb, W. P. Burleson, K. Fu, et al. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *Proceedings of the Conference on RFID Security*, volume 7, page 2, 2007.

[11] S. Jia, L. Xia, Z. Wang, J. Lin, G. Zhang, and Y. Ji. Extracting robust keys from nand flash physical unclonable functions. In *International Information Security Conference*, pages 437–454. Springer, 2015.

[12] R. Maes. *Physically Unclonable Functions - Constructions, Properties and Applications*. Springer, 2013.

[13] R. Maes. Puf-based entity identification and authentication. In *Physically unclonable functions*, pages 117–141. Springer, 2013.

[14] P. Prabhu, A. Akel, L. M. Grupp, S. Y. Wing-Kei, G. E. Suh, E. Kan, and S. Swanson. Extracting device fingerprints from flash memory by exploiting physical variations. In *International Conference on Trust and Trustworthy Computing*, pages 188–201. Springer, 2011.

[15] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 33–47. IEEE, 2012.

[16] S. Q. Xu, W.-k. Yu, G. E. Suh, and E. C. Kan. Understanding sources of variations in flash memory for physical unclonable functions. In *Memory Workshop (IMW), 2014 IEEE 6th International*, pages 1–4. IEEE, 2014.