School of Electrical & Electronic Engineering

EEE8097 : Dissertation

Physically Unclonable Functions for Networked Device Authentication

---

Author:

Michael Walker (130623935)

Supervisor:

Dr. Alex Bystrov

September 29, 2014

Declaration

I declare that this dissertation represents my own work and that I have correctly acknowledged the work of others. This dissertation is in accordance with University and School guidance on good academic conduct (University guidance is available at `www.ncl.ac.uk/right-cite`).

Michael Walker, September 29, 2014

Summary

Physically Unclonable Functions are a relatively recent area of active research which can be seen as analogous to hardware fingerprints that potentially offer a way to provide low-cost, automated, trustworthy authentication of embedded systems. As the infrastructure of networked devices grows in our homes and workplaces so too will the need to protect ourselves from 'Hardware Trojans' and other threats. In reviewing the state-of-the-art of 'PUF's' and their potential incorporation into modern ciphers as cryptographic primitives using fuzzy extractors, the general significance and worth of an investigation into the implementation of practical Ethernet message authentication codes using Static RAM PUFs and Fuzzy Extractors as the core components is to be shown.

Contents

List of Figures **vi**

List of Tables **vii**

Acronyms **viii**

1 Introduction **1**

2 Background Research **5**

2.1 Identity and Authentication 5

2.1.1 Challenge-Response 5

2.1.2 Authentication Factors and Inherited Identity 6

2.2 PUFs & SRAM 8

2.3 Fuzzy Extractors 10

2.3.1 Introduction 10

2.3.2 Secure Sketch with BCH encoding 14

2.3.3 Privacy Amplification with SHA-256 16

2.4 PUF Enhanced Network Authentication Protocols 17

3 System Design **20**

3.1 Introduction 20

3.2 SRAM-PUF interface implementation on FPGA 21

3.2.1 SRAM wrapper 22

3.2.2 UART Implementation 25

3.2.3 SRAM-PUF Implementation 28

3.3 Fuzzy Extractor in MATLAB 28

3.3.1 Introduction . . . 28
3.3.2 Helper Data . . . 29
3.3.3 Secure Sketching . . . 32
3.3.4 Randomness Extracting . . . 33
3.3.5 Bus Width Considerations . . . 34
3.3.6 Components Required by the Fuzzy Extractor . . . 35
3.4 BCH Encoder and Decoder . . . 36
3.4.1 Introduction and Theory . . . 36
3.4.2 Implementation . . . 38
3.5 SHA-256 Algorithm Implementation on FPGA . . . 42
3.5.1 Theory of Cryptographic Hash Functions . . . 42
3.5.2 Implementation . . . 45
3.6 Design of New Ethernet Authentication Protocol Method . . . 45
3.6.1 The EAP-PUF Protocol Sequence . . . 47
3.6.2 Ethernet Packet Contents . . . 51

4 Results **55**
4.1 SRAM-PUF . . . 55
4.2 Fuzzy Extractor . . . 56
4.3 SHA-256 . . . 59
4.4 Ethernet Authentication . . . 59
4.4.1 CRC . . . 60
4.5 Overall System Results . . . 60

5 Conclusions **63**
5.1 Achievements . . . 63
5.1.1 Goals Met . . . 63
5.1.2 Benefits of a Modular Approach . . . 64
5.2 Limitations . . . 64
5.3 Implications . . . 65
5.4 Future Research . . . 65

6 References **66**

A Appendix **70**
A.1 SRAM-PUF Implementation in VHDL 70
A.1.1 puf.vhd - Top Level Design Entity 70
A.1.2 uart.vhd - Implementation of UART module 75
A.1.3 sram.vhd - Wrapper function for SRAM 82
A.2 Fuzzy Extractor implementation in Matlab 91
A.2.1 FuzzyGenerator.m - Implementation of Generation process of Fuzzy Extractor 91
A.2.2 FuzzyReproducer.m - Implementation of Reproduction process of Fuzzy Extractor 94
A.3 EAP-PUF Protocol Simulation in Matlab 96
A.3.1 GenerateRequest.m - Generates a full EAP-PUF Ethernet Packet 96
A.3.2 ProcessRequest.m - Extracts Challenge from EAP-PUF Packet 98
A.3.3 crc.m - Generates FCS of Ethernet packet 99
A.4 SHA-256 hash function implementation in Matlab 101
A.4.1 sha256.m - Main SHA-256 Matlab Implementation code 101
A.4.2 Various files - SHA-256 Matlab Implementation Helper functions 106
A.5 Full Project Demonstration code 112
A.5.1 pufdemo.m - Fuzzy Extractor Demonstration code . . . 112
A.5.2 FuzzySim.m - Fuzzy Extractor error correction testing 116
A.5.3 FuzzyLengthTest.m - Fuzzy Extractor length parameter testing 119
A.5.4 challengegen.m - Generates random hexadecimal challenge string 122
A.5.5 Validator.m - Validates two Responses 122

List of Figures

2.1 Six transistor SRAM diagram 9

3.1 RS-232 data transmission scheme for 8 data bits (D0-D7) . . . 25
3.2 Fuzzy Extractor Generation Process 30
3.3 Fuzzy Extractor Reproduction Process 31
3.4 SHA-256 Block Diagram 46
3.5 EAP-PUF authorization sequence diagram 50
3.6 EAP-PUF encapsulated in an Ethernet Packet 51

4.1 Full system testing demonstration flowchart 62

List of Tables

3.1 Number of correctable errors for BCH code length of 63 40
3.2 Number of correctable errors for BCH code length of 127 . . . 40
3.3 Number of correctable errors for BCH code length of 255 . . . 41
3.4 EAPoL Type Codes 53
3.5 EAP Types 53
3.6 Selected EAP Method Codes 54

Acronyms

(

c.

AES

advanced encryption standard.

ASCII

american standard code for information interchange.

ASIC

application-specific integrated circuit.

CRA

challenge-response authentication.

CRC

cyclic redundancy check.

CRP

challenge-response pair.

DoS

denial of service.

EAP

extensible authentication protocol.

EAPoL

EAP over LAN.

ECC

error-correcting code.

FCS

frame check sequence.

FPGA

field-programmable gate array.

GSM

global system for mobile communications.

HTTPS

hypertext transfer protocol secure.

IANA

internet assigned numbers authority.

ICV

MACSec message authentication code.

IP

intellectual property.

LAN

local area network.

LBA

lower byte access.

LFSR

linear feedback shift register.

LSB

least significant bit.

MAC

message authentication code.

MAC

medium access control.

MACSec

MAC Security.

MitM

man-in-the-middle.

MOSFET

metal-oxide semiconductor field-effect transistor.

MSB

most significant bit.

NAK

negative-acknowledgment.

NSA

national Security Agency.

OTP

EAP one time password/pad.

OTP

one time pad.

PAE

port access entity.

PIN

personal identification number.

PISO

parallel in, serial out.

pots

plain old telephone system.

PPP

point to point protocol.

PPPoE

PPP over ethernet.

PRNG

pseudo-random number generator.

PUF

physically unclonable function.

RADIUS

remote authentication dial in user service.

RF

radio frequency.

RFC

request for comments.

RFID

radio-frequency identification.

RNG

random number generator.

RTL

register transfer level.

SHA

secure hash algorithm.

SHA-2

secure hash algorithm, version 2.

SIPO

serial in, parallel out.

SLIP

serial line Internet protocol.

SRAM

static random-access memory.

SSL

secure socket layer.

TLS

transport layer security.

TRNG

true random number generator.

UART

universal asynchronous receiver/transmitter.

UBA

upper byte access.

UMTS

universal mobile telecommunications system.

VHDL

VHSIC hardware description language.

VHSIC

very high speed integrated circuit.

WEP

Wired Equivalent Privacy.

WPA

wi-fi protected access.

WPA2

wi-fi protected access version two.

WPS

wi-fi protected setup.

XOR

exclusive disjunction.

Chapter 1

Introduction

Physically Unclonable Functions (PUFs) are a relatively recent concept currently receiving a great deal of research interest. By harnessing physical properties that are extremely difficult to simulate or copy, yet are easy to sample from, distinctive entropy can be created. This can then be used as a source of cryptographic key information for identification, authentication and auditing purposes. This function can then be packaged into any device that contains or incorporates that physical property and, with some necessary extensions, offers an extremely secure and trustworthy means of uniquely identifying a device and detecting any attempts at forgery. This is analogous to a biometric such as a fingerprint or iris, but one which can be found in embedded electronic devices rather than human biology.

In this dissertation, I will discuss my findings regarding the implementation of a PUF authentication system; one that would be of practical use if incorporated into a small electronic embedded networked device for providing the means to authenticate that device without requiring passwords or any other type of user intervention. The value of such a device is clearly growing [1] given the increasing numbers of network enabled electronic devices available for home and office settings. These all currently require the installer to be physically present for manual, individual authentication into their network through wi-fi protected access (WPA) passwords or more recently wi-fi protected setup (WPS) buttons. A naive preliminary set-up for

this would involve the extraction of data from a physical device with the properties of a PUF in response to a challenge specifying the parameters used for extraction, with all the data sent in plain-text via a standard network protocol.

This first notion does provide a simple challenge-response authentication (CRA) by which the device can provide identity information. However, over the course of the project, this has proven to be inadequate for a complete authentication solution for two reasons. Firstly, both the challenge and the response are in plain-text, thus are completely open to an adversary's use of attacks such as the replay attack or the man-in-the-middle (MitM) attack hence any practical system requires some use of modern cryptographic methods to secure the data.

Secondly, the meta-stable nature of static random-access memory (SRAM) PUFs means they provide somewhat unreliable data, this must be accounted for and mitigated. The concept of /emphfuzzy extraction [2] shall therefore be introduced and explored as a means of providing both of these necessary additions to the PUF-based authentication system by utilising the concepts of error-correcting codes (ECCs) and cryptographic hashes.

In this scenario, it is necessary that the PUF's device authenticates itself as a *supplicant* over an open access network to some *authenticator* which issues the challenges to the device to prove its identity. The networked device must then respond to the challenge correctly otherwise all or part of the networking medium required by the device is in some way withheld by the authenticator. In this way, network security can be achieved without any human intervention. This is surely an important and necessary step in facilitating the likely future integration of innumerable, inexpensive embedded systems into home, office and factory wireless networks. This concept is often named as the 'internet-of-things' [3]. This is because, without burdening the user with any increased installation inconvenience, adequate security provision can be provided. Therefore, the practical issues surrounding the use of PUFs in this security provision by the establishment of a prototypical network security protocol were investigated.

In chapter 2, I will report my background research into the subjects of

PUFs and the use of SRAM as a PUF source. This will be followed by research on fuzzy extraction with focus on the usage of a class of ECC called BCH codes and a family of cryptographic hash functions called SHA-2. Finally, research into techniques to add PUF-based security provision into the Ethernet protocol will be discussed.

In chapter 3, this initial research is built upon with an explanation of the design of systems and simulations that were built to analyse and experiment with the concepts gleaned from the initial research. These designs broadly fit into four areas of design effort.

- An implementation of a RS-232 based CRA system for data extraction from a physical SRAM device.
- The simulation of a complete fuzzy extractor system in MATLAB© .
- The design of a modified Ethernet protocol for PUF authentication.
- The joining together of the above systems into a cohesive system to allow for fully demostrating the concepts and as a framework for future research.

In chapter 4 the results of the practical implementations of the designs are presented. The implementation details of fully end-to-end demonstration will be discussed and other testing for an optimal set of parameters for the process will be shown.

The concluding chapter includes a discussion of the implications of this study, some of which indicate the feasibility of developing a commercially viable new wireless security system. Further work that would be needed to both develop the demonstration design as it stands to a basically marketable point will be enumerated. However, more intriguingly, the avenues of further research that could be explored to advance from this first attempt as a PUF-authentication design to something much more commercially attractive are discussed.

It should also be noted that this dissertation has been typeset in LaTeX, the use of which is a new experience to the author. The experience of producing this document using the de-facto standard for the publication of scientific

documents as opposed to Microsoft Word is considered an integral part of the project.

Chapter 2

Background Research

The intention of this project is to investigate and implement in full - or in part - a prototypical networked device that can be authenticated using a PUF. As explained in the introduction, there are three areas under investigation. In this chahpter are the findings from preliminary research undertaken for each of the three areas. This is the foundation which supported the eventual system design that will be discussed in the next chapter.

2.1 Identity and Authentication

Verificaton of identity is fundamental to this project, and so related concepts and foundations should be explored. Firstly, it should be noted that authentication is a different concept to authorization and accounting, although the three together (AAA-security) are often considered as a whole, the focus in this project is Authentication only.

2.1.1 Challenge-Response

In this project, CRA is foundation for authentication of device identity. CRAs isa commonly used concept in the authentication of devices on a network. The concept is that the device to be authorized (hereby known as the *Supplicant*) is challenged by the device that controls access to the network (hereby known as the *Authenticator*). For each device to be identified cor-

rectly, there must be at least one challenge-response pair (CRP) associated with each device and the supplicant must respond with its correct response, otherwise access to the network is denied.

In the simplest case this could be as simple as a basic password check with a unique password associated to each device/user. Thus, mutual authentication by a shared secret is obtained. However, if an attacker can eavesdrop the communications channel, they can also see the password sent and use a replay attack to gain access by spoofing the response of the original supplicant. To make the system more secure, many CRPs can be generated, thereby requiring an attacker to record all possible pairs to be guaranteed access to the system. The probability of one-time access is inversely proportional to the number of pairs, such that with sufficient pairs the reply attack can be made statistically impossible.

There is one important complication; if a response can be partially guessed solely through analysis of patterns in the ciphertext a challenge generates (called information deduction) this weakens the system. Therefore correlation of the challenge to the response should be as near to the Shannon entropy limit as possible.

2.1.2 Authentication Factors and Inherited Identity

In network communications within a shared medium, the identities of the communicating devices must be explicitly made so that they can be read by the intended recipient. However, without security procedures in place, it is very simple and entirely possible for any device to claim the identity of another since there is no way to verify its provenance. Thus, in mature networks some form of authentication protocol is introduced. Authentication is generally performed using authentication factors, these can be stated succinctly as one of three types;

- Knowledge - what you *know*.
- Ownership - what you *have*.
- Inheritance - what you *are*.

In most cases of network authentication the first two factors are used, sometimes in combination. In the case of knowledge factors examples include passwords and personal identification numbers (PINs). In the case of ownership factors, examples include radio-frequency identification (RFID) tags and smart-cards. These are extensively used for current local area network (LAN) authentication systems, for example the old Wired Equivalent Privacy (WEP) security scheme for the IEEE 802.11 standard (branded as 'Wi-Fi') required a password 10 to 26 Hexadecimal digits to be physically entered by hand. In fact, secret keys of this type form an integral part of hardware cryptographic primitives in conventional methods of authentication. These necessarily rely on the digital storage of secret keys in non-volatile memory, which has proven vulnerable to reverse engineering and side channel attacks.

Although less common, due to its more physical nature, Ownership factors include the 'VideoGuard' viewing cards that must be inserted into a set-top-box in order to authenticate the device over a point to point protocol (PPP) connection through the plain old telephone system (pots) to access encrypted digital satellite or cable television channels. Ultimately these systems also rely on the digital storage of keys, however, they are stored within the smart-card and not the device itself. If an exploit is found, new protocols with changed smart-card designs can be implemented and replacements distributed to upgrade the security of the system. This incurs a significant additional expense, but allows for far greater longer-term security.

In contrast, inheritance factors are very uncommon in authentication of network devices, and more usually are used to authenticate users themselves in the form of biometrics such as fingerprints, iris-scans and facial recognition. The advantages of inheritance factors over traditional methods for networked devices should be clear;

- Increased Convenience - Automates process, nothing to be forgotten or misplaced by the users
- Increased Security - No passwords on scraps of paper, or smart-cards in wallets to be stolen, and nothing is easily guessable or easily duplicated.
- Increased Accountability - Auditing and post-factum reporting are stronger,

as identities are harder to forge or alter.

While humans and other biological systems have biometrics that can be used in this way, it is less obvious that electronic hardware systems can make use of Inheritance factors. PUFs are an enabling technology that can allow this. By making use of them we can transfer much of the beneficial techniques found in the application of biometrics to user-centric security systems to the topic of of device-centric authentication.

2.2 PUFs & SRAM

The first thing required for a PUF based authentication system is a PUF. Introduced in 2002 [4], PUFs can be built from a wide variety of technologies. Some have electronic construction, some not, and new designs seem to appear every year. Of those PUFs that are created from electronic circuits, there are some that can use off-the-shelf components and some that must be custom made in silicon. There are some that are delay based [5] and some that are based on memory state meta-stability. SRAM PUFs that use the meta-stability of the standard six transistor SRAM cell (see Figure 2.1) were initially proposed by Guadardo [6], and are the focus of this project.

An SRAM cell is conventionally constructed from metal-oxide semiconductor field-effect transistors (MOSFETs) in the ubiquitous, modern photolithography process. Any imperfections in the fabrication of the cell[1] will result in a cell that is predisposed to initialise to a particular binary value. This meta-stability is a central underlying principal for the SRAMs functionality. Much research has been made into how the physical properties of SRAM relate to it's use in PUFS [7–9]. This project, however, will focus simply on the use of the unclonable entropy provided by a PUF once it is created. It should be noted that authentication is only one possible application of PUFs, and they have potential application in a wide array of future technology including true random number generators (TRNGs).

[1] such as subtle variances in the transistors size, tiny changes in performance characteristics due to silicon impurities, or any other of minute variance in any of a thousand manufacturing steps

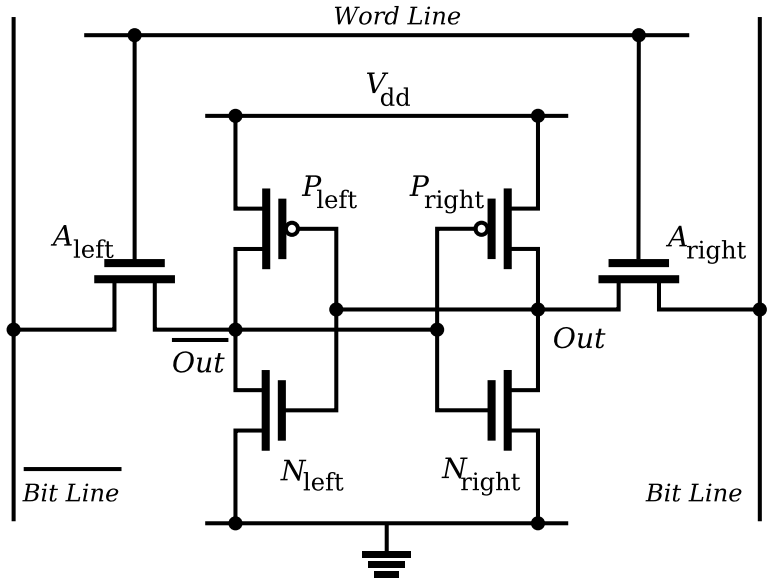

Figure 2.1: Six transistor SRAM diagram

To construct an SRAM based PUF either the SRAM must be incorporated into the design of an intellectual property (IP) core on the application-specific integrated circuit (ASIC) itself, or it could be accessed from outside. In a practical PUF core of an embedded device, the former may be easier to implement, but the latter would take less space. In the case of this project, where a physical IP core will not be implemented, the functionality is intended to reside in an field-programmable gate array (FPGA). The acquisition of a separate SRAM device is required as a basic FPGAs chip itself is unlikely to have the memory quantity required. For example, in this project a Cyclone II 2C20 FPGA device is used which contains 32 M4K RAM blocks yielding a total of 240 Kilobits of RAM which need to contain the logic look up tables used in the design and are likely to be reset undesirably to zero upon memory initialization. A device that does not have the property of initialising the memory cell contents to a set value upon power-up is essential, and the ability of the PUF implementation to control the power supply to the SRAM device is desirable.

Unfortunately, in the case of many FPGA development boards, such as

the Altera DE-1 [10], the SRAM chip on the board itself is not completely compatible with this usage. This is because the wire that powers the chip in operation is simply wired to the required voltage rail of the board. Without the ability to physically turn the SRAM chip off and back on, its initial values are available only once without manual intervention, which makes repeated testing of the device in an automated way difficult. However the simplicity of the interface and its relative ease of availability make it a good choice for preliminary work such as that carried out in this project, with the understanding that further work requiring the capture of averaged performance data will require a modified system with a more appropriate SRAM device.

If other parts of the system are to be simulated externally, the device itself also needs to communicate with those external modules. This implies the need for a simple serial communication channel and protocol. A suitable candidate is the RS-232 protocol, for which transceiver and capacitive voltage generator hardware happen to be included on the DE-1 Development board in question. This necessitates the implementation of a universal asynchronous receiver/transmitter (UART) module.

While the UART module will handle conversion of serial input into byte length words, it will be necessary to build an interface module around the SRAM if it does not have similar single byte word length for addressing and data.

2.3 Fuzzy Extractors

2.3.1 Introduction

The manner in which PUFs are implemented means that their raw output is often, and perhaps necessarily, *fuzzy* (or *noisy*). That is to say that it is far from guaranteed that any individual bit or symbol within the data extracted from the PUF will remain the same when checked again. This could be due to variations in conditions (such as temperature or radio frequency (RF) interference) or by degradation or wear of the physical substance that the PUF is created out of. In naive solution, this could be solved by accepting

some set threshold of errors in the response of the PUF with respect to any particular challenge. However, while easy to implement, this is inefficient and severely limits the cryptographic security of the device. It would be far preferable to create some sort of cryptographic *wrapper* around the device that corrects the errors internally and can be guaranteed (with some degree of certainty) to output the same response to a challenge even if conditions have changed or the device has degraded over time. It would also prevent the leakage of keys, hardening the device from replay attacks.

As it turns out, such a wrapper exists. This is called a **Fuzzy Extractor**, which was first proposed by Dodis et al. [11] for handling biometric data for cryptographic situations. In laymans terms, this involves three separate components.

2.3.1.1 Components

Firstly, and most essentially, a *secure sketch* which extracts ('sketch process') and recovers ('recovery process') stable cryptographic key data in a noise tolerant way. Secondly, a *privacy amplification* component is implemented through a *randomness extractor*. In more specific terms, this obscures the key by hashing the initial output of the secure sketch - which could be quite regular, and increasing its randomness, thus making the message much more difficult for an adversary to break. In any practical implementation of the two above components of a Fuzzy extractor, it is at least necessary to implement one further component; a source of entropy.

2.3.1.2 Processes

If the authentication system employs CRA when verifying the identity of a device it should be obvious that there needs to be challenges stored ready to use with the correct responses expected. These CRPs need to be generated in a fuzzy extractor process called the 'Generation Process' in which a secure *sketch* is made, and the response is carefully stored with the challenge as pairs in a secure database for later use. This process could be performed at any time, but in practice it would be most suitably performed at the factory

when the device is first created. It is this process that requires an additional component to provide a source of entropy.

The verification process also involves the use of the fuzzy extractor, but in a different configuration called the 'Reproduction Process'. This uses a secure sketch *recovery* to mitigate errors. No further entropy source is required. However both processes make use of a identical randomness extractor.

These components can be implemented and connected in a variety of ways, in creating the two related processes of *Generation* and *Reproduction* but generally the secure sketch requires the implementation of some kind of ECC. The randomness extractor requires the implementation of a cryptographic hash function. The source of randomness needs to be practically non-deterministic. Once these three components are found, a method by which they can be integrated securely into a cohesive whole must also be designed. This design will

2.3.1.3 Source of Randomness

Randomness can be sourced from the implementation of a random number generator (RNG). Many different types have been proposed in the literature and can be separated into two classes, TRNGs and pseudo-random number generators (PRNGs). The difference between the two is that true randomness requires sampling from a noisy physical process and are (in theory) completely unpredictable. PRNGs which are deterministic, and as such can be considered mathematical functions which can be replicated by an attacker. Thus, it is far more secure to use a true random number generator if possible. In Electronics a hardware clock is created from a harmonic oscillator such as a quartz crystal whose output flips high to low and back at a regular interval. The basics of the implementation of a random number generator to be used by the fuzzy extractor is by sampling the output state of one hardware clock at times controlled by another. The two clocks are required to originate from two independent clock crystals (one cannot be the Phased Locked Loop result of the other). Since a clocks crystals are not perfectly precise in their oscillation due to thermal noise effects ageing and variances

in supply current and voltage it can be seen to be unknown at a given time whether two independent clocks are in phase or not.

If the frequency of one clock is very much higher than the other (1000s of times), it can be seen as a type of digitised noise source in respect to the readings that will be obtained at the sampling frequency imposed by the slower clock. However, this implies that the bit rate of the output of the random number generator will have to be very much lower than the fastest clock. For the purposes of the Generation procedure in a Fuzzy extractor that is used only once in a factory setting this issue may well not be a problem. In reference to the implementation used, the two RNGs used need only to generate a few hundred bits of random data for each CRP. Another issue is that this is a cyclical process and as such produces data with relatively low entropy, as many repeated patterns can be found in the data stream due to harmonic resonances between the clocks. To mitigate this the stream could be passed through a hash function to increase the entropy while keeping its true randomness.

Given that each generation procedure also required many clock cycles of activity for BCH encoding and SHA-256 Hashing it would be possible to generate random data of required length in the time it takes to complete a response. However this random data is required at the start of the generation process, therefore it would be expedient to generate the randomness required for the first Fuzzy Extractor Generation Process during the devices initialisation routines and store the result in a buffer. Then on subsequent generation procedures, after the buffer contents has been accessed the random data generation process can function in parallel, filling the buffer with new randomness.

Interestingly the PUF itself, being a source of entropy, could be used to provide the randomness required. The benefits of this would be decreased complexity and reduced size and power requirements. This could prove highly advantageous in situations of limited resources. However, it presents the issue that all the cryptographic security relies upon the foundation of the PUF only, metaphorically placing all eggs in a single basket which increases the power of any exploit or attack on the physical PUF implementation. Also, it

increases the amount of SRAM required for adequate security. In the case of the design in the next chapter, this means approximately between 200% to 300% the amount of SRAM would be required to achieve the same amount of possible keying material. It would also be interesting for further research to look into separating the usage of an SRAM array. Where areas of highest unpredictability of an SRAM array are used for TRNGs, the predictable areas that are constant from device to device are limited to storing data used in fuzzy extractor calculation purposes, leaving the best areas of meta-stability for PUF identity purposes.

Further TRNG implementation alternatives include the use of ring-oscillators and many other concepts that, not coincidentally, are often applicable as components in PUFs, as they share much of the same property requirements as TRNGs. Implementing a TRNG on a FPGA was attempted using the same SRAM metastability as that for the PUF as this was relatively simple to perform using three raw puf requests instead of just one and passing the extra data into the fuzzy extractor instead of that coming from a MATLAB© PRNG. However, for the final demonstration model the MATLAB© `randi()` function was employed for this purpose as it was simple and eliminated a potential source of complication.

2.3.2 Secure Sketch with BCH encoding

ECCs, in simple terms, use extra redundant information to provide a scheme for some number of errors in a message to be detected and eliminated. Generally these have been most usefully applied in the field of network communications. In the fuzzy extractor, this ability to correct errors can similarly be used to eliminate inherent noise to produce cryptographic key data with a high probability of correctness. This is exactly what is needed for implementation of the secure sketch process.

There are many types of error-correcting code, some more suitable in the implementation of a fuzzy extractor than others. This is especially true when implementing for embedded devices where electronic complexity needs to be minimised. The theory of error-correcting codes is often introduced by the

Hamming code. This most simple of codes can be generalised as a cyclic linear block code.

Block codes, as opposed to convolution codes such as the Viterbi algorithm, work on fixed sized packets of symbols rather than streams of data. The raw data coming out of the PUF is of a fixed size, therefore block codes are the obvious choice for implementing a secure sketch. However, Hamming codes can only correct one error, the raw output of a PUF could potentially contain more than one error, and therefore it is necessary to explore other schemes.

BCH Codes where first proposed by Alexis Hocquenghem, and independently Raj Bose and D. K. Ray-Chaudhuri (the name BCH being an acronym of their surnames). It is a type of cyclic linear block error-correcting code. They historically build upon (and can be seen as a generalisation and refinement of) the Hamming codes proposed by Richard Hamming in 1950. BCH codes are binary in the nature of their symbols, unlike other block codes such as the more famous Reed-Solomon code used for error-correction for compact discs. This binary nature allows for a more compact implementation in embedded hardware and is easier to implement in VHSIC hardware description language (VHDL).

Again, the implementation of complimentary BCH Encoders and Decoders in FPGA was investigated, but for reasons of expediency this was not completed, and a MATLAB© simulation was used. Due to their central nature in the design, it was important to undertake further research into the mathematical nature and algorithmic process by which BCH encodes redundancy into the coded message and decodes an original message out. This is important in making reasonable and educated decisions, as parameters chosen in the design of BCH Encoder and Decoder have as direct impact on the design and security of the fuzzy extractor, which in turn, affects the construction of the entire authentication system. The next chapter therefore contains a section on the theoretical knowledge that was vital to designing the system, even though in reality all encoding functionality was performed by calling high-level functions in the MATLAB© communications system toolbox.

2.3.3 Privacy Amplification with SHA-256

Hash functions are any function that maps variably sized input data (message) to output data (the hash, or message digest) of a fixed size. They are primarily used in hash tables to speed up data searches. However, they have a wide variety of other uses such as use in cryptography. A cryptographic hash function can be defined as any hash function where it is infeasible to generate the input given the output, i.e. it is impossible to invert the function in a practical sense.

In finding a suitable cryptographic hash function for privacy amplification purposes, there are also three other properties that would be ideal for it to have. These are; that it is simple to implement the hash algorithm, that the likelihood of finding two different inputs that map to the same output data is microscopically small (emphcollision resistant) and finally, that it is practically impossible to generate the required input data from a given output data (*one-way function* or *pre-image resistant*).

Such cryptographic hash functions are widely suggested in the literature. However, in the context of an embedded system, a balanced compromise between the cryptographic strength of the hash function and the complexity of its implementation is paramount. Bogdanov [12] presents a lightweight implementation that is used by other instances of fuzzy extractors in the literature [13, 14] which could be used. Yet rigorously assessing the cryptographic security of such a relatively novel hash is deceptively difficult, beyond the scope of this project and may lead to compromises if not fully understood. Therefore, implementation of a relatively new hashing scheme would be ill-advised. It is, in the view of the author, far better to tailor a long-standing existing and well-understood and tested scheme for the purpose.

The standard cryptographic hash function set called SHA-2. secure hash algorithm, version 2 (SHA-2) was designed by the National Security Agency (NSA) in 2001 [15]. It avoids some flaws in its predecessor SHA-1, yet is relatively simple to implement in hardware. The set consists of hash functions differentiated by their digest size (224, 256, 384, and 512 bits). They all operate using a similar structure, however, for easy implementation in VHDL

the 256-bit implementation (SHA-256) is the most appropriate choice. This is because it is one of the smallest and the digest size is a power of two, which is often beneficial in simplifing hardware implementations.

2.4 PUF Enhanced Network Authentication Protocols

In providing a PUF authentication mechanism, consideration should first be given to where in the protocol stack would be most effective. For an embedded device, a high layer protocol such as one in the application or session layers of the traditional OSI model is both inefficient and unnecessary. Even when going down the stack, it would also seem that establishment of the transport and network levels presuppose that the device has already obtained access to a point-to-point connection or LAN. This means that authentication at this layer may well be too late to prevent certain attacks and exploits that would compromise a network. This would suggest that a suitable place to establish authentication through PUF is at the *data link layer*.

The link layer consists of both direct connection protocols involving only two devices, such as serial line Internet protocol (SLIP) and PPP and multi-node protocols, where the communication channel is shared, such as Ethernet and Wi-Fi. In invisioning a future 'Internet-of-Things' protocol relying on PUFs, it would seem advisable to look at the multi-node protocols, specifically those for wireless networking. However, in attempting to keep the project focused, it would also be preferable to investigate protocols in a simpler setting than that of wireless communications, as the addition of high-levels of channel noise and comparatively new and complex layer 2 protocol implementations would hinder the project in many areas. Thus, a compromise of investigation of modifications to the older wired Ethernet protocol was reached.

Any modern link-layer network authentication system that is intended to replace or improve upon current standards necessitates adhering to much the same high security methods utilized in modern cryptographic protocols

such as WPA and more recently wi-fi protected access version two (WPA2) in Wireless protocols. This is because those methods have been both proved secure through exposure to the 'real-world' and provision for those methods is readily available to industry. Modern standards such as WPA2 and glsppoe all support the use of different methods for security provision at the data link layer by employing the extensible authentication protocol (EAP) framework. The underlying authentication method is encapsulated by EAP which provides for the safe transfer of any parameters and secret key information for the method itself. Methods used include EAP-TLS using the transport layer security (TLS) protocol (which is the most widely supported) and is the successor to the secure socket layer (SSL) asymmetric public-key encryption protocol, commonly used in e-commerce through the hypertext transfer protocol secure (HTTPS) protocol. Other methods include the simple EAP one time password/pad (OTP) and EAP-MD5 which were heavily referenced in the design of the EAP-PUF method outlined in this project, and EAP-SIM and EAP-AKA used in the mobile telephony standards; global system for mobile communications (GSM) and universal mobile telecommunications system (UMTS) respectively.

While EAP can be encapsulated into various protocols, it can mostly be found being utilized over Ethernet (IEEE 802.3) and Wi-Fi (IEEE 802.11) networks using the ISO/IEC/IEEE standard 802.1X [16] for encapsulating it. In the case of Ethernet, it uses the standard Ethernet packet format with a *EtherType* set to the value `0x888E`. The idea is that those devices requiring authentication to the network known as *supplicants* are not allowed to send Ethernet packets of any other EtherType, including general ones such as `0x0800` IPv4 packets. Thus, an attacking device can do no harm to any other device on the network. Any and every attempt to communicate will be dropped until authenticated.

Paramount to cryptographic security is the issue of running out of keying data for a protocol. This is a serious issue in this project, given the ultimately finite nature of SRAM memory and therefore the finite number of challenges that can be made without repeating the data - a fundamental issue in cryptography. A completely secure system could be achieved using a one time

pad (OTP) system whereby the system would never use the same memory twice. Any reuse of keying data will make the system less secure, but the problem can be mitigated by ensuring reuse is handled effectively. Firstly, by using different ordering of keying data in the challenge. Secondly, by including a time-stamp in the challenge and thirdly by limiting the number of challenges necessary to maintain authentication through the integration of glsplmac2 into the protocol. The first method was implemented in the project extrinsically, the second as intrinsic property of the use of glseapol and the third will be considered in as an avenue for further research in the concluding chapter.

Chapter 3

System Design

3.1 Introduction

Ideally, a complete PUF-based authenticatable device could be implemented on a single FPGA, containing all the necessary components required. However, given the time constraints of a Masters project, while this approach was initially favoured, it was deemed necessary to simulate parts of the system in the MATLAB© programming environment. The system area considered most difficult to synthesise was the error correction encoder and decoder. These are critical components, without which a Fuzzy Extractor could not be implemented. As a result, both the generator and reproduction implementations of the Fuzzy Extractor were not implemented in FPGA but simulated.

The Ethernet protocol is also comparatively complex and so modification of this for the purposes of device authentication was also removed from the design. Instead, a study of possible protocol schemes was conducted with a simulation of only one message of the complete scheme developed to a level for demonstration.

The PUF functionality of SRAM on the other hand, was possible for full implementation in FPGA, due to the easy availability of multiple DE-1 development boards, each with integrated SRAM chips. Extraction of specific PUF data from the SRAM chip would require the design of a 'wrapper' function that could issue memory addresses and retrieve memory contents

of the SRAM chip. It would then be necessary to link that wrapper with a communication system that would allow external access to the device.

It was decided to implement this system using the RS-232 protocol standard. This is because the necessary hardware was available on the DE-1 board and it was deemed the simplest to implement. The development of a UART that could receive memory address locations encoded serially and pass them in full to the wrapper was thus required. Likewise, it would be necessary to send the memory contents passed from the wrapper serially back to the requester. This requester would be a full PC running MATLAB© which contains the ability to integrate serial communications data into its programs.

3.2 SRAM-PUF interface implementation on FPGA

The SRAM-PUF interface was designed in three sections each in a seperate VHDL file. These files can be found in the appendix and can be summarized as follows:

SRAM.VHD

Used as a 'wrapper'. Required to access the SRAM memory.

UART.VHD

Used to interface between the SRAM wrapper and the physical RS-232 connection.

PUF.VHD

Used to connect the above two sections together and connect their inputs and outputs to the appropriate pins of the FPGA to communication with the peripherals which included the SRAM chip, the MAX232 RS-232 driver/receiver chip, master clock, reset button and LED array.

3.2.1 SRAM wrapper

The DE-1 development board contains a IS61LV25616 chip. This contains 2^{22} SRAM cells organised into 2^{18} words of 16-bits each. A memory of this size provides more than sufficient quantity of memory for testing our design. In fact, for the purposes of simplicity, it is easier to use only a quarter of the address space the chip provides, to allow 16-bit addressing which reduces the complexity of the communications (via RS-232) part. The implementation of the SRAM 'wrapper', as it came to be known, involved coding a component in VHDL to be synthesised for use on the FPGA of the DE-1 development board This can be split into two areas of responsibility; a *transmit* and a *receive* section. Both were implemented in a single VHDL process sensitive to the master clock to make register transfer level (RTL) synthesis less complex to design for and debug.

The receive section was required to convert multiple 8-byte words received from the UART module into one 16-byte memory address word to be placed on the memory bus connected to the SRAM chip. The transmit section was required to perform the opposite function of converting a single 16-byte data word output into multiple 8-byte words.

To access the SRAM chip as a PUF, we need only to concern ourselves with the read cycle of the device, as writing data is counter-productive. There are 5 control lines to the chip (all use active-low); Chip Enable ($\overline{CE}$), Write Enable ($\overline{WE}$), Output Enable ($\overline{OE}$), Lower Byte Access ($\overline{LB}$) and Upper Byte Access ($\overline{UB}$).

Whilst the first three can remain in a constant state for our purposes ($\overline{CE} = 0, \overline{WE} = 1, \overline{OE} = 0$), the byte access signals can be utilized to output the full 16-bit memory data onto just one 8-bit bus by cross-wiring the low order bits lines (0-7) to the high order bits (8-15) sequentially. Therefore, a full reading can, in theory, be sent via the rs-232 link in two transmissions. In a similar way, the 16-bit address can be received in two pieces. Timing is important, and the particular chip on the development board is the fastest in the range, with an access time of 10ns rather than 12ns or 15ns in other chips in the family. Thus, with careful reading of the datasheets [17], from

the initial setting of the address to the point at which the data output is valid[1] is 10ns.

It can be seen that a wrapper function is required that buffers the two part address as it is received, then *forwards* a complete 16-bit address onwards to the address bus of the SRAM memory, with the upper byte access (UBA) signal active. After some delay greater than 10ns, the upper byte can be forwarded and the upper and lower byte access (LBA) signals toggled. After at least another identical delay, and if a strobe has been received from the communication module to indicate the previous byte has been sent, the lower byte can be forwarded. This creates a system for converting challenges in the form of memory addresses into responses in the form of memory data.

This was indeed the first configuration of the wrapper function implemented, however over time it became apparent that due to the encoding of some memory addresses as ASCII escape characters used in RS-232, certain memory locations were effectively unavailable. Specifying them in a challenge would cause unspecified behavior (bugs). Thus, a more redundant encoding scheme for the addresses needed to be used.

It would be preferable to implement the system such that all memory addresses could be specified without accidentally encoding a 'system bell', 'carriage return' or, most dangerously, the RS-232 software flow control sequences 'XOFF' and 'XON' which could end the communication session abruptly. For this purpose, the wrapper was extended to separate a 16-bit memory address into four 4-bit values (nibbles) each rather than just two 8-bit values. A simple scheme was used whereby these 4-bit values could be encoded as an american standard code for information interchange (ASCII) character corresponding to their value in conventional hexadecimal notation. This turned out to be very useful in testing as faults could be detected in short order if non-hexadecimal characters appeared in any terminal output from the device. Note that ASCII is a 7-bit encoding, but for the purposes of this project it is considered as 8-bits where the most significant bit (MSB) is always '0'.

In encoding 4-bits of binary data to and from ASCII a conditional statement, two possibilities could be used:

[1] t_{AA}, or Address Access Time

Numbers

The binary 4-bit values '0000' to '1001' (0-9) could be encoded as ASCII numerical characters. This means that the upper nibble must be set to '0011' and the lower bit is the binary value (a useful property of ASCII)

Letters

Binary values greater than '1001' (decimal 9) need to be represented as the ASCII characters A-F which in binary are represented in the 8-bit values; '01000001' (decimal 65) to '01000110' (decimal 70). So, for conversion, the upper nibble must be set to '0100' and the lower nibble is the binary value plus the value '1001' (decimal 9).

Similarly, converting an ASCII byte to a binary nibble requires another conditional statement in VHDL. This time it is slightly more complicated, as one more condition is added to accommodate for lowercase characters (a-f) in the input, where the upper nibble is '0110' instead of '0100'.

This final implementation abandoned the use of the UBA and LBA toggling and instead used a loop structure to access each nibble of the full 16-bit output in turn. This structure was applied to both the transmission and receiving sections. This meant that upon receiving a strobe signal from the glsuart component called 'STR', which indicated a byte of data had been received, a nibble in the address was updated and a counter was updated.

Once that counter indicated a full address had been received (after 4 nibbles) both the receive counter and a transmission counter are reset and a corresponding strobe signal for transmission 'STT' was raised to send the first of four bytes of ASCII data. While sending the data, another loop is entered which mostly waits on the master clock to ensure sufficient time is given to the UART component to send the current byte, but it too loops 4 times raising the 'STT' each time before entering an idle state to await the next 4-byte memory address to be received in full. A full description of the functionality can be found in the well-commented source code for the file 'SRAM.vhd' in section A.1.

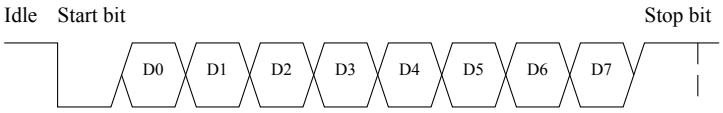

Figure 3.1: RS-232 data transmission scheme for 8 data bits (D0-D7)

3.2.2 UART Implementation

The DE-1 development board contains a MAX232 chip that can handle the conversion of on-board logic levels to the higher voltages required by the RS-232 standard, so this was not required to be implemented manually. Instead the timing requirements of the protocol needed to be met.

A Universal Asynchronous Receiver/Transmitter (abbreviated UART) provides serial communication between devices. In essence, the UART controls the process of converting data arriving from a parallel data bus into a form that can be sent sequentially (one bit at a time) over a communication channel.

A fundamental component of a UART is the shift register. In the case of reception a Serial-In, Parallel-Out (SIPO) shift register is used. Similarly, in the case of transmission a Parallel-In, Serial-Out (PISO) type is required.

Synchronisation of the rate at which the bits are sent (baud) is critical. While clock skew is not an issue, as there is no master clock, data recovery still depends on both devices being set to operate at the same speed. Common bit rates supported range from 75 to 115,200 bits/s.

The RS-232 protocol uses binary signaling. When idle, the channel is left at a logical high. Data send is framed by sending an initial low start bit and ended with a high stop bit. The stop 'bit' isn't really a bit a all, just the convention that the channel returns to a logical high for at least one clock cycle after the transmission of data. It can therefore be specified 1.5 or 2 'bits' in length, but this is unusual. The data itself can have a data word length of 5 to 9 bits, but is usually 8 (1 byte) and an optional parity bit (Even, Mark or Space parity) can be appended.

Our implementation will focus on just one possibility for all these values, in the common shorthand, we use '115200/8-N-1', meaning that the baud

is 115,200 bits per second, there are 8 data bits, no parity bits and 1 stop bit. The timing digram for this implementation can be seen in Figure 3.1. While RS-232 can utilise extra handshaking signals for flow control, such as software flow control with 'XOFF' and 'XON' signals, these are not required and so are not implemented. However, without any flow control the timing of the transmission is of even greater importance to the protocol. With a baud chosen of 115,200 bits per second we need to generate a clock pulse with a period of $\frac{1}{115200}$ seconds, or approximately $868\mu s$. A master clock of 50MHz is provided by the development board. Using this as a basis for keeping syncronised to the baud rate necessitates the implementation of a clock divider which implements a counter that counts 434 rising edges of the master clock ($\frac{50MHz}{115200}$) before reseting. There is also a counter for tracking progress through the 8 separate data bits processed.

The design of a UART is split into two distinct halves; the transmitter and the receiver. Each operates somewhat independently. Each half requires a separate set of two of counters; *frequency* and *bit*, counting down to 0 before resetting back to 434 or 8 respectively:

- `FRQ_CNT_RX` - Baud Counter for Receiver
- `FRQ_CNT_TX` - Baud Counter for Transmitter
- `BIT_CNT_RX` - Bit Counter for Receiver
- `BIT_CNT_TX` - Bit Counter for Transmitter

3.2.2.1 Receiver

The receiver is required to convert serial input into an 8-bit parallel output. Hence a serial in, parallel out (SIPO) shift register is used in this half of the UART. Transmission is handled through a case analysis, triggered every rising edge of the master clock which sets output depending on its two counters:

If `BIT_CNT_RX = 0 AND FRQ_CNT_RX = 0`

In normal circumstances this is the idle state. However, if the receiving

signal is low it means that a start bit has been encountered and the module starts receiving a byte by resetting both counters.

If `BIT_CNT_RX < 8 AND BIT_CNT_RX < 0`

Writing the current bit of the bit of the SIPO it indexes, noting that the RS-232 protocol is little-endian. If this is the last bit, then set the strobe (STR) and decrement the bit counter for the final time this cycle.

Default Case

Decrement the frequency counter.

3.2.2.2 Transmitter

The transmitter is required to convert an 8-bit binary input into a serial output. Hence, a parallel in, serial out (PISO) shift register is used in this half of the UART. Transmission is also handled through a case analysis triggered every rising edge of the master clock as follows:

If Transmission Strobe (STT) is high

Set the RS-232 output signal low to indicate the start bit and reset counters.

If `FRQ_CNT_TX = 0 AND BIT_CNT_TX > 0`

Set the output to the bit of PISO register indexed by the bit counter, decrement the bit counter and reset the frequency counter. Again, note that the RS-232 protocol is little-endian.

If `FRQ_CNT_TX=0 AND BIT_CNT_TX=0`

Last data bit was previously sent, so set the RS-232 output signal to high to indicate the stop bit, with both counters zeroed, the idle state is entered.

Default Case

Decrement the frequency counter.

3.2.3 SRAM-PUF Implementation

With the two components (SRAM wrapper and UART module) created, integration of control signals becomes a simple matter of connecting the two strobe signals (STR and STT) together so that they interact appropriately. Then the 8-bit data lines between the two were joined (with a separate output to drive the LEDs for debugging purposes). The VHDL code implementing the SRAM-PUF can be found in section A.1.

3.3 Fuzzy Extractor in MATLAB

3.3.1 Introduction

The fuzzy extractor has two processes, the generation process and the reproduction process. The generation process is only used for initial set-up, whereby the expected responses for all possible challenges are generated and securely stored. This process is likely to happen during manufacturing of the device at the factory. The reproduction process on the other hand, is the commonly employed procedure whereby for a given challenge a response is generated that can be tested against the initial response from the generation process. These are implemented differently because in the generation process, both a response and helper data must be generated. In the reproduction process, helper data is consumed and only a response is generated.

The core difference between the processes occurs in the secure sketch process. In generation, a sketch is made and the required response is recorded. This sketch results in helper data to be used in the reproduction procedure and is stored in such a way that it can be sent with all challenges used in the challenge-response protocol. In reproduction, a recovery is performed, whereby the previously generated helper data and the noisy input from the PUF are processed to produce an input for the randomness extractor.

3.3.2 Helper Data

The helper data is required both to allow for differences in the raw responses of the PUF and to keep the data cryptographically secure. One is associated with the secure sketch components (let us call this 's'). The other is associated with the randomness extractor (let us call this 'x'). As the design is both quite complex and nuanced, it is difficult to explain at once. However, much of the overarching functionality of the fuzzy extractor can be explained following the path of the helper data as shown diagrammatically in figure 3.2 and figure 3.3 before exploring the major components in detail.

In generation the secure sketch outputs redundant data n which is the same length as the raw data from the puf p. These are combined with the XOR operator and the result is the helper data s ($p \oplus n = s$). In reproduction s is used twice, once when it is combined with new (fuzzy) PUF data p' (note the dash) using an XOR operation. Due to the information preserving, cyclical nature of the disjunctive operator (XOR) the result is therefore a potentially fuzzy version of the encoded message ($s \oplus p' = n'$). It is XORed again to reproduce the original PUF data by using the XOR operation on the encoded message, but after it has been 'cleaned up' (ie. $s \oplus n'' = p'' \approx p$). Thus s is, in effect, the secure *sketch* itself, encrypted by a key analogous to a OTP made from the PUF data.

The other helper data is used in the randomness extractor and is used to generate a Privacy Amplified response. In generation, random data is captured from a TRNG and stored as helper data x. Simultaneously, input from the PUF is combined with it through an XOR operator (let us call the output of this operation xs, so $p \oplus x = xs$) which is applied as the input a cryptographic hash function which outputs the original response.

In reproduction, the same input to the randomness extractor is used to output the same response. If it is assumed for now that the secure sketch recovery component provides the original PUF data, by XORing it with x again we should create the same input to the cryptographic hash function as in the generation process, thus the same response will be generated. This means that x is analogous to a OTP used to encrypt in original PUF data.

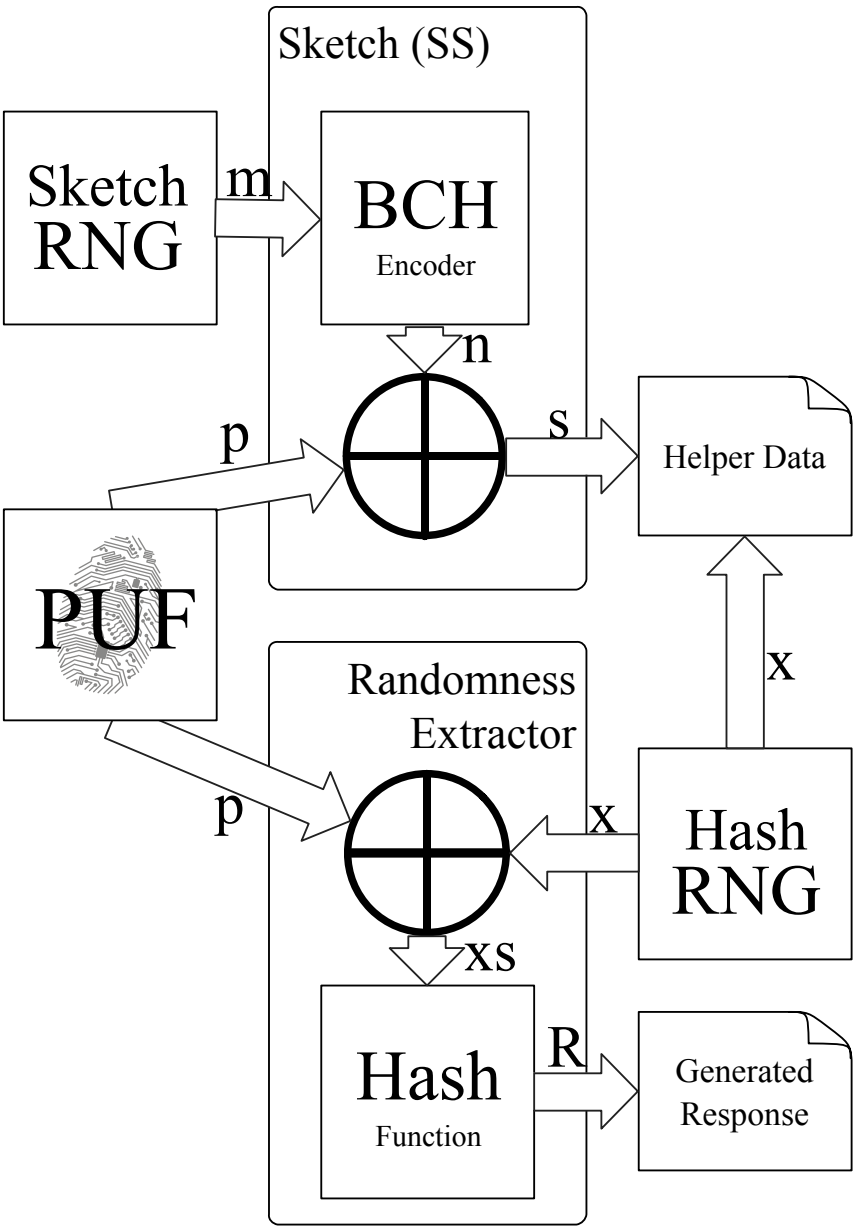

Figure 3.2: Fuzzy Extractor Generation Process

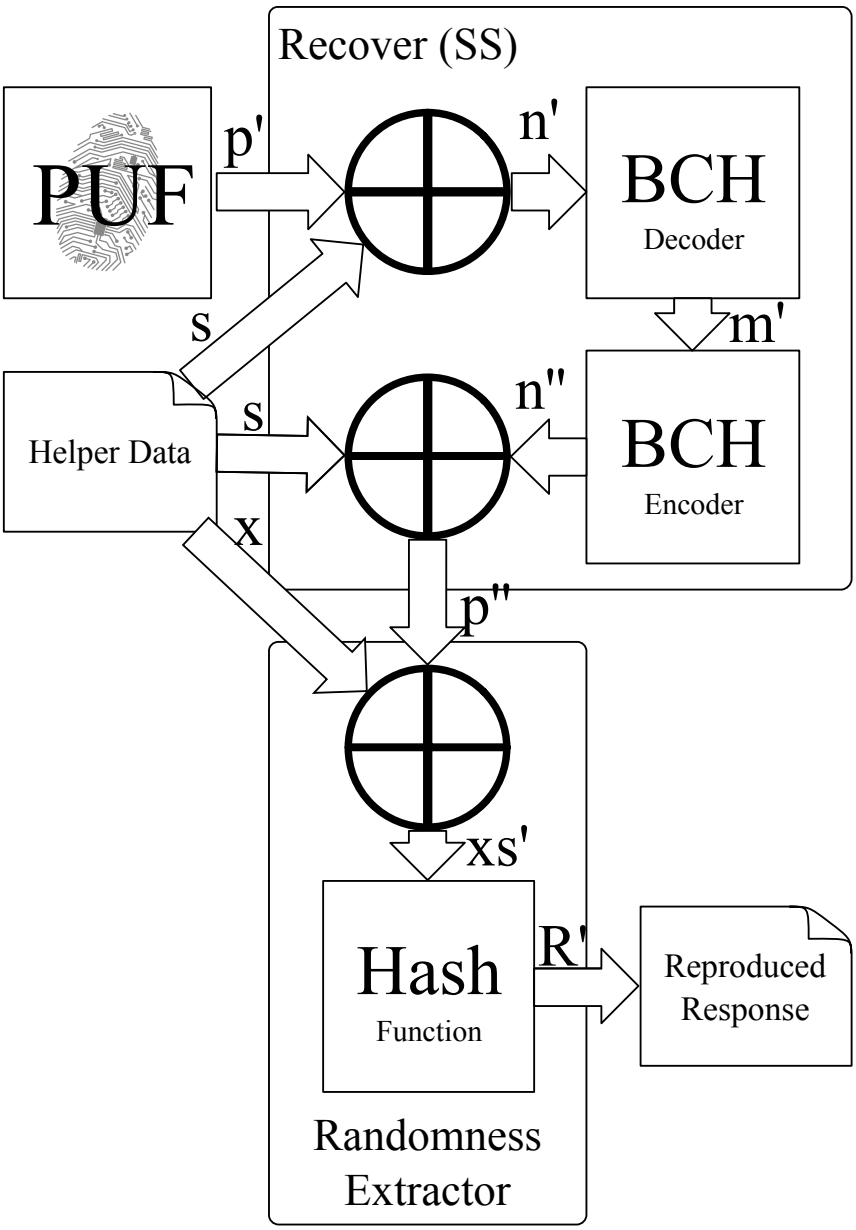

Figure 3.3: Fuzzy Extractor Reproduction Process

3.3.3 Secure Sketching

More accurately, the *sketch* used in the generation process takes in two inputs; the first is the raw data from the PUF that is known to be noisy and the second is random data generated by the first of two independent TRNGs (we shall name this the 'Sketch RNG' in future as it is used by the Secure Sketch part of the fuzzy extractor). The random data, which we shall call 'm' for 'message', is passed into a BCH-encoder sub-component that therefore generates a redundant encoding of that random data which we shall call 'n' which can later be error-corrected during the recovery stage. The length of the message must always be less than the encoding. The smaller it is, the more errors that can be corrected.

This encoded message is then XORed with the PUF data (p) to create the first of the two pieces of helper data s. It is therefore important that the output of the BCH encoder is identical in size to the raw input data from the PUF. The other part of the helper data is a second random data block generated by the second of the TRNGs (we shall call this the 'Hash RNG' in future as it is used in the *privacy amplification* part of the fuzzy extractor which primarily involves hashing). Again, x must be the same size as p. Ultimately, this process securely generates helper data from the raw PUF (and therefore noisy) data by which the second process, the recovery, will be able to reproduce as output the same PUF data as that encountered in the initial generation procedure, even if the PUF outputs slightly different data.

The *recovery* also takes in two inputs. The first is raw input from the PUF (p') which, to reiterate, is likely to be different from the first time (hence the dash). The second is the helper data s. By XORing them together we produce a potentially noisy version of the original output of the BCH encoder (n) and then using that as the input to a BCH-Decoder, we can use the properties of forward error correction to duplicate the original message created by the sketch RNG used in the generation procedure. If we then pass that into a BCH-encoder, we will get exactly the same data that was originally combined with the PUF output. If we then XOR this with the

same helper data s a second time, we reproduce the original output of the PUF.

Thus, we have potentially recovered, error-free, the data used in generating the response. This means that the fuzzy extractor works as desired, and slight changes in raw PUF data can be mitigated. However, if there are more errors than can be corrected by the BCH decoder used, the data will not be recovered correctly. This has two important consequences. Firstly, the BCH encoder should be capable of correcting enough errors such that all but the most extreme degradation of the PUF can be accounted for. Secondly, it can be seen that without the functionality of the randomness extractor, i.e. if xs was taken as the response of the fuzzy extractor instead of passing it through a hash function, a security risk occurs. This is due to the BCH-encoding process 'correcting' errors in the response. This means that an attacker can use repeated guessing of a response to a particular challenge in what is known as a 'bit-flipping attack' and each time it will be capable of learning some parts of the original response through pattern analysis.

3.3.4 Randomness Extracting

When the recreated PUF data 'p' is XORed with the helper data 'x'' (which is the original output of the Hash-RNG) this should be expected to match the message that was applied to the SHA-256 cryptographic hash function in the original generation procedure. The nature of a hash function is such that, given the same message, and the exact same initial parameters, the function *must* produce the same digest. Thus the response in the reproduction procedure will be the same as in the generation procedure even with slight differences in raw PUF data.

However, the purpose of the hash function is to throughly scramble the data in such a way that the input cannot be deduced from the output alone. This means that even a slight change (one bit flipped) in the input should result in a radically different output. This is commonly known as the *avalanche effect* [18].

Through this, the PUF data is hidden and never revealed in plain-text. It

is cryptographically assured and thus authentication of the PUF is performed securely. A incremental 'bit-flipping attack' is prevented as no patterns can be gleaned from the response. Even though helper data (s and x) is available to the attacker, this will not give away the PUF key, as we can trust the mathematically rigorous cryptographic security of the SHA-256 hash function to obfuscate the response/digest output such that the original PUF cannot be retrieved from it, as it is the 'key' used to 'encrypt' the sketch output 'n' keeping it secret prevents any attack.

3.3.5 Bus Width Considerations

The response produced by both fuzzy extractor processes is fixed by the nature of the SHA-256 hash function, which has a 256-bit output. Furthermore, assuming that the output of the PUF is a block of data of length w and the SHA-256 hash function and BCH encoder are used the length w is somewhat limited. For purposes of cryptographic strength (to be looked into further in section 3.5 on Cryptographic Hashing) the length of the encoded message n should be one less than an integer power of 2. Since the PUF data p is bit-wise combined with n it should be the same length. However, due to the 16-word length of the data it would be sensible to use an exact power of 2 output and simply drop the MSB.

It also simplifies the implementation to limit the input of the SHA-256 component to a single 512-bit block. By avoiding the development of complex functionality, necessitating extra memory requirements and adding greater potential latency a multi-block implementation would involve, the design is simplified while ensuring that little cryptographic strength is lost. This means that the upper limit for PUF data is actually no greater than 440 bits. This is because the SHA-256 input message is appended with a stop bit and 64 bit length field and as the PUF output comes in 16-bit words, the largest multiple of 16 that will fit into $512 - 65$ bits is 440. This therefore limits the reasonable choices for the size of PUF data to be either 64, 128 or 256 bits in length.

Due to the contraction entailed in matching the raw PUF input to the

code length of the BCH encoder the s and x helper data will both also be initially created one bit smaller than the corresponding PUF data length (ie. 62, 127 or 255 bits). However a '0' MSB is added to conform the data to conventional power of 2 bus widths, hence they will also be the same size. In the main implementation of the system, a data length of 128 was chosen, but the modular nature of the design was used to allow for experimentation with the other two options. The exact size of the sketch-RNG output, which corresponds to the length of the plain, uncoded message 'm' is all that is now left to define. The nature of BCH encoding limits the efficient choices to a small list (see the correctable errors tables - Table 3.1, Table 3.2 and Table 3.3). Even limited to a finite list, there is no obvious 'correct' value for this and necessarily involves a compromise. A smaller message allows more errors to be corrected, conversely, a larger one provides more entropy and adds greater security to the system. The main implementation demonstrated uses a message length of 64 bits, but again this is set by a parameter that can easily be changed in the code.

3.3.6 Components Required by the Fuzzy Extractor

To implement our complete fuzzy extractor in MATLAB© the following sub-components must be created, correctly tested and connected together correctly:

BCH Encoder

Used once in generation and once in the reproduction procedure.

BCH Decoder

Used only once in the reproduction procedure.

SHA-256 Hash Function

Used once in the generation and once in the reproduction procedure.

True Random Number Generator

Used twice, both times in the generation procedure (*Hash-RNG & Sketch-RNG*).

Bitwise XOR

Used twice in the generation procedure and three times in the reproduction procedure.

The code implementing the fuzzy extractor can be found in section A.2.

3.4 BCH Encoder and Decoder

3.4.1 Introduction and Theory

Forward error correction codes are methods for the transmission and reception of data such that noise is mitigated and the original data sent is transferred without error. All implementations require an encoder which adds redundancy to the message to be transmitted and a decoder which can use that redundancy to recover the original message even if the transmitted data is disturbed and unwanted alterations are introduced. It is important to contrive a system that is both efficient in the resources it uses (both time and area it requires) and is accurate and powerful enough to correct the peak amount of errors expected due to noise.

Error correcting codes can be classified into a hierarchical tree of types. One of the broadest branches of that tree are linear block codes, of which BCH is one form. Linear codes are a class of error correction codes in which addition is closed this means the code is cyclic in the same sense as modular arithmetic. BCH codes are in the sub-class of linear codes called block codes. These function opon the original message one block at a time. An alternate scheme is to use convolution codes such as the Viterbi [19], which function continuously on streams of data. However, for the purposes of the fuzzy extractor such codes simply adds extra complexity since the responses are fixed in size which suits block encoding. Recent research has been directed to even more efficient codes (such as Turbo or Raptor code) these approach the theoretical limits of coding theory (Shannon limit) but are similarly out of the scope of this project.

BCH codes are of a specific sub-class of cyclic linear block codes those of which utilise a binary encoding, this is also the case for the simpler Hamming

codes, but is unlike other block codes such as Reed-Solomon or Reed-Muller which use a larger symbol set. For brevity, we adopt the convention that the encoder takes a block of symbols length m and that it translates into a larger block of length n by adding redundancy. In any encoding system the symbols used in the message block are taken from an alphabet A which has q symbols (therefore there are q^m and q^n possible message and digest block sequences). Therefore we can define any block code in the form: $a(n, m)$-block code called C over the alphabet A with q symbols. An encoder for the block code C (let us call it E) is a bijective mapping such that there is an exact one-to-one correspondence between the original message (M) and the encoded message (T).

The simplest block codes are the Hamming codes. These are capable of correcting only one random error, and are therefore not useful in this project. The BCH code is more sophisticated and can be seen as a generalisation of the Hamming codes for multiple-error correction. Mathematically BCH codes operate over finite fields (in simple terms this as the same as modular arithmetic) these can also be called Galois fields after their discoverer the 19th century French mathematician Évariste Galois. The order of the field is the number of elements it contains, and for BCH codes the size of the Galois field is two or $GF(2)$. The key property of finite fields is that they allow for the basic operations of addition, subtraction, multiplication and division to be used while holding to six conditions:

Closure

For all operations on two operands that are elements of the finite field F the result is also an element of the field (i.e. $c = a+b$ where $a, b, c \in F$).

Associative

Two like operations applied in either order will cause the same result (i.e. $a + (b + c) = (a + b) + c$)

Identity

There exists identity elements such that for all operations the result is the same as the other elements (i.e. $0 + a = a$ where 0 is the identity for addition and $1 \times b = b$ where 1 is the identity for multiplication)

Commutative

Where order of operands does not change the result (i.e. $a + b = b + a$)

Inverse

There is an inverse element for every element in the field that the result is the operators identity element. (i.e. $a+b = 0$, where b is the additive inverse, and $a \times c = 1$, where c is the multiplicative inverse)

It has been shown that the set of integers $0, 1, .., p-1$ where p is a prime number and where all operations are performed modulo p are valid finite fields of order p. As 2 is the first prime number the Galois field used by BCH is the simplest finite field, it is also much easier to map to the digital domain of binary numbers. Larger fields can be used in error correcting codes. These can often result in greater encoding efficiency, but are harder to implement and for our purposes the efficiency increase is of rather limited usefulness due to the small size of the responses expected.

3.4.2 Implementation

As previously stated, the implementation of a BCH encoder in FPGA was not attempted, neither was the implementation of the MATLAB© code necessary to perform the calculations. Instead, the BCH implementation in the MATLAB© Communications Systems Toolbox was employed. Adequate understanding of the mathematical principals of the BCH code was required for informed usage of the BCH functions and tools provided in the MATLAB© environment. It was important to understand the errors that could be corrected by the different parameters under which a BCH encoded message could be generated. Efficient BCH code lengths are somewhat set by the properties of the cyclic codes used. Efficient implementation entails BCH code lengths ('n') that have the form $2^m - 1$, where m is an integer i.e.

$\{n | \exists m \in \mathbb{N} \wedge n = 2^m - 1\}$.

Further MATLAB© computations limitations limit the value of m to between 3 and 16.

Thus, considering anything below 16-bits is wasteful as that is the minimum memory data retrieved and anything extremely large would be expensive in terms of system resources. Hence, 31, 63, 127, 255, 511 and 1023 are the choices available for efficient code length. as the corresponding outputs from the PUF will necessarily be multiples of 16, the MSB of the PUF was dropped for it to conform to the sketch system. For the purposes of PUF implementation a width of 127 for the input bus was chosen. Testing was also carried out in simulation of other widths for comparison.

The size of the message encoded by BCH has direct effects on the number of errors that can be corrected in the decoder. The MATLAB© function `bchnumerr(N)` can be used to calculate these values; where `N` is the code length. The results of running the function for code lengths of 63, 127 and 255 are shown in Table 3.1, Table 3.2 and Table 3.3. Note that n is the code length, m is the message length and d is the number of errors that can be corrected. Using these tables it was decided that for project demonstration purposes it would be best to use a message size of 64 bits which is approximately half the size of the code length. The reasons for this were three-fold: Firstly it covered for the possibility of one complete byte of memory data being corrupted in the RS-232 communication channel (seen as a possibility given the lack of parity checking). Secondly, it would ensure adequate entropy, as with a small message size a brute-force attack could be mounted. Whereas with 2^{64} possible messages it would be infeasible. Thirdly, it was also admittedly, simply an aesthetic choice of a power of 2. This last point demonstrates that further investigation into the consequences and inherit compromises made in any particular message length choice would be of interest.

n	m	d
63	57	1
63	51	2
63	45	3
63	39	4
63	36	5
63	30	6
63	24	7
63	18	10
63	16	11
63	10	13
63	7	15

Table 3.1: Number of correctable errors for BCH code length of 63

n	m	d
127	110	1
127	113	2
127	106	3
127	120	1
127	113	2
127	106	3
127	99	4
127	92	5
127	85	6
127	78	7
127	71	9
127	64	10
127	57	11
127	50	13
127	43	14
127	36	15
127	29	21
127	22	23
127	15	27
127	8	31

Table 3.2: Number of correctable errors for BCH code length of 127

n	m	d
255	247	1
255	239	2
255	231	3
255	223	4
255	215	5
255	207	6
255	199	7
255	191	8
255	187	9
255	179	10
255	171	11
255	163	12
255	155	13
255	147	14
255	139	15
255	131	18
255	123	19
255	115	21
255	107	22
255	99	23
255	91	25
255	87	26
255	79	27
255	71	29
255	63	30
255	55	31
255	47	42
255	45	43
255	37	45
255	29	47
255	21	55
255	13	59
255	9	63

Table 3.3: Number of correctable errors for BCH code length of 255

3.5 SHA-256 Algorithm Implementation on FPGA

3.5.1 Theory of Cryptographic Hash Functions

Hash Functions in simple terms, are a mapping from an input data called the *message* to output data called the message *digest*. Mathematically, this mapping is *surjective*, i.e. it guarantees that **all** possible inputs map to some valued output. However it is not a *bijection* because it is not *injective*. Therefore, in a hash, there may be *more than one* input that maps to the same output. This possibility means it is conceivable for a hash *collision* to occur, although this can be made extremely unlikely in practice through the use of universal hashing algorithms.

Cryptographic hash functions are a subset of all hash functions. Their defining attribute is that the function must be as **one-way** as possible. It should be, if not theoretically then practically impossible to perform the inverse function of creating a valid message given some digest. The degree of difficulty for an adversary breaking a given system will increase in some complexity order related to the digest length. Making this order as large as possible is key to a strong hash function. For the SHA-2 hash family, which is to be implemented in the project, a brute force attack can only be done in exponential time ($O(c^n), c > 1$)). Hence, by using a large enough digest size (and due to compounding nature of exponential growth the size in bits need not be very large at all) a *pre-image* attack (i.e. one try to find a message for a given hash) will be prevented. This only holds true as long as serious security flaws in the SHA-2 algorithm allowing some shortcut are not found in the near future; which would seem somewhat unlikely; excluding advances in quantum computing.

Another form of attack that needs consideration is the *second pre-image* attack., whereby a second message is found with the same digest as a set first message. Again this form of attack is secured against by SHA-256 well. The final style of attack considered is that of a *collision* attack, whereby two messages (neither set) are found which result in a collision. This is the

easiest attack to mount, and generally any attacks found in the literature are of this type. So far, a collision attack of the SHA-2 family has not been found, although non-standard reduced versions of the SHA-2 family and the older, related SHA-1 standard are vulnerable [20].

An explanation of the functionality of the SHA-256 hash function would be appropriate to explain the choices made in its implementation. Although the design (as shown in figure 3.4) seems rather convoluted, this is a product of the nature of the function. Its purpose is to jumble the data in the message as thoroughly as possible in the minimum amount of time and space. Hence it uses a lot of complex logical operations in a complex sequence, but there is no other meaning or purpose behind the design other than that it has been found through experimentation or mathematical analysis to jumble well.

The SHA-2 family use six different logic functions (operating on 32 bit inputs and outputs). These are:

- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\Sigma_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
- $\Sigma_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
- $\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
- $\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

Where $ROTR^n$ means a *rotate right* function by n bits, $\oplus$ is the exclusive or operator, $\wedge$ is the bitwise AND operator and finally, note well that the last x in the first function is negated ($\neg$).

Note that all these functions can be synthesised on an FPGA, as should be expected, as the SHA-2 was designed to function well in hardware. The SHA-256 function operates on block sizes of 512-bits, and produces a 256-bit digest. In operation, that 256-bit digest is stored in 8 32-bit buffers ($H_0 - H_7$). Also to be stored are 8 intermediate 32-bit hash registers (note $8 \times 32 = 256$) labelled alphabetically a to h. Both buffer and registers are first initialised

to a set of constant values (generated from calculating the square roots of the first 8 prime numbers). The message block is split into 16 32-bit values ($W_0 - W_{63}$), whereby the message gets fed into the main part of the function one at a time in a queued fashion, each feed occurring during a round of operation. Added complexity is introduced by also feeding the front value back into the queue using the two bottom s operations such that for the current round j:

$W_j \Leftarrow sigma_1(W_{t-2}) + W_{t-7} + sigma_0(W_{t-15}) + W_{t-16}$.

This sounds complicated, but can be implemented as a large linear feedback shift register (LFSR). Note, there is usually a padding step, but padding is not necessary in our implementation as we can ensure our values fit exactly the 512-bit message block required.

Normally the function is used to hash large messages that are split into multiple blocks. In our implementation only a single block is sufficient for security, which simplifies the implementation by removing an outer loop from the standard algorithm. The important step is to apply the SHA-256 compression function for 64 rounds, in each of which updates of all the registers are made and more of the message block is added piece by piece. The updates are as follows:

- $T_1 \leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_j + Wt$ (n.b.. where K_j is one of 64 standardised 32-bit constants)
- $T_2 \leftarrow \Sigma_0(a) + Maj(a, b, c)$
- $h \leftarrow g$
- $g \leftarrow f$
- $f \leftarrow e$
- $e \leftarrow d + T_1$
- $d \leftarrow c$
- $c \leftarrow b$

- $b \leftarrow a$
- $a \leftarrow T_1 + T_2$

Once the registers are set, the buffers are set to a new intermediate value by ANDing each of the 8 registers to its corresponding buffer (i.e. $H_0 \leftarrow a + H_0, H_1 \leftarrow b + H_1$ etc.). After all 64, rounds the buffer contains the hash of the message.

3.5.2 Implementation

The SHA-256 function was implemented in MATLAB© for mostly pedagogical reasons. There were no serious performance constraints and as such the implementation is intentionally extremely inefficient to retain as much clarity of intent as possible. Interestingly, there is no official library implementation of the SHA algorithms in MATLAB© . The matlab code can be found in section A.4.

3.6 Design of New Ethernet Authentication Protocol Method

EAP is only a framework in which authentication methods can be incorporated. By using the technique of following the design of a minimal pre-existing method, while using different PUF specifics where appropriate, a new method could be created. This approach leverages an existing framework for PUF based authentication.

Each EAP method is indicated through a code in the EAP packet. A new method would require a new method code. The official types are maintained by internet assigned numbers authority (IANA) [21]. At the time of writing the official list contains 57 reserved codes, out of the possible 256, meaning there is room for more methods. While actually registering a new method is out of the scope of this project, the demonstration uses the unassigned value

Figure 3.4: SHA-256 Block Diagram

192 for the new method type that is designated 'EAP-PUF'. [2]. For expediency, it was determined to develop a full working *simulation* of sequence of Ethernet packets required in the 'EAP-PUF' authentication procedure. This is to avoid practical implementation difficulties while at a prototype stage and gain an overview of the technologies and overarching design decisions required.

The design of a complete authentication procedure intertwines two broad areas; that of protocol packet sequencing and that of packet contents specification.

3.6.1 The EAP-PUF Protocol Sequence

Figure 3.5 most clearly shows a complete authentication sequence. As can be seen this involves three entities:

- The Supplicant - A device containing a PUF for verification.
- The Authenticator - A device which represents the point of connection for nodes on the network, most likely a network switch.
- The Authentication Server - A server that can issue and verify EAP-PUF CRPs

In a typical sequence for a basic EAP method (such as EAP-MD5, which was most often referenced in the design of EAP-PUF) the authenticator acts a middle man or translator in a handshake exchange between the authentication server and the supplicant. This two step approach is used because the authentication server is likely communicating with the authenticator using a high layer protocol. A typically used protocol is remote authentication dial in user service (RADIUS), which happens to run in the application layer. As an intended supplicant for PUF authentication is a low-power embedded device it may have insufficient resources to communicate with a RADIUS server directly. EAP, being a versatile authentication framework can be transported

[2]Usefully, 192 is also unassigned as a RADIUS Attribute Value Pair

via any layer protocol, in this case it can be encapsulated in Ethernet between the supplicant and authenticator using the EAP over LAN (EAPoL) protocol, then translated by the authenticator into a RADIUS packet and visa-versa.

The process of authentication can be initiated by the authenticator sending a EAP-Request Identity packet, but typically this can be ignored and the supplicant can, in any case, preempt this by simply sending a EAPoL Start packet. This packet has an empty body without any EAP data. By sending this packet the supplicant confirms that it can use the EAP protocol for authentication.

Before Authentication using the EAP-PUF the supplicant needs to provide its claimed identity so that the appropriate authentication method can be used, with the appropriate CRP for that identity selected. This identity retrieval step generally involves the use of a separate EAP method (type 1) used solely for the communication of identity. Although in advanced methods *identity protection* is promoted by bypassing this method and using custom versions instead. First, upon receipt of the start packet, the authenticator generates a new identity byte for this authentication exchange to allow it to keep track of multiple simultaneous authentication sessions and sends the identity request packet. The supplicant then replies with its identity. These are both full EAP packets. It is interesting to note that the authentication server is not required up to this point. The specifics of the identity protocol body are authentication method specific, but it is recommended in the relevant request for comments (RFC) to piggy-back a method request in the identity response packet.

When the Authenticator receives the identity response it needs to initiate RADIUS communication with the authentication server with a Access-Request packet, this includes the medium access control (MAC) address of the supplicant and a EAP-PUF method request. How this is formatted in RADIUS is out of scope of this project, but the authentication server is expected to respond with an Access-Challenge packet containing an appropriate Challenge for the identity of the supplicant which is forwarded by the Authenticator in an EAP-PUF Request packet. This needs to contain the

challenge data and the helper data. At this point the reproduction process can be initiated on the device, as explained in the preceding parts of this chapter.

Once a response has been reproduced, the supplicant replies to the request with it EAP-PUF Response Packet containing just the 256-bit digest. It should be noted that the Ethernet packet specification requires a minimum packet size of 512 bits. Including the headers both the EAP-PUF request and response packets are smaller than this, and hence padding is required to be added to the packet.

The authenticator translates and forwards the response to the authentication server where validation occurs. This is a simple matter of comparing the received bit sequence with that generated in the factory and stored in the database. Only if the response is an exact match is an Access-Accept packet sent back to the authenticator. At this point the supplicant is considered authentic by the authenticator and full network access is granted. The authenticator also sends a EAP Success packet (no other data contained) informing the supplicant of its new status.

If the response is invalid a RADIUS Access-Reject message is sent instead, the authenticator can act upon this in anyway it seems fit, but usually will in turn send a EAP Failure packet (again no other data contained). Often the supplicant will then request authentication by another method.

A successfully authenticated device can then send a EAPoL-Logoff packet when it no longer requires network access. It should be noted that this exposes a vulnerability of EAP protocol as this packet is sent in plain-text and is easy for an attacker to spoof, thereby the attacker can implement a denial of service (DoS) attack by continually issuing a EAPoL-Logoff packet for any MAC addresses it discovers using EAP through packet sniffing.

To review, a supplicant using EAP-PUF to authenticate needs to be able to interpret four receivable Ethernet packet types, and generate three types to transmit as a functional minimum, these packet are summarised as follows:

- Received Packets
  - EAP Identity Request

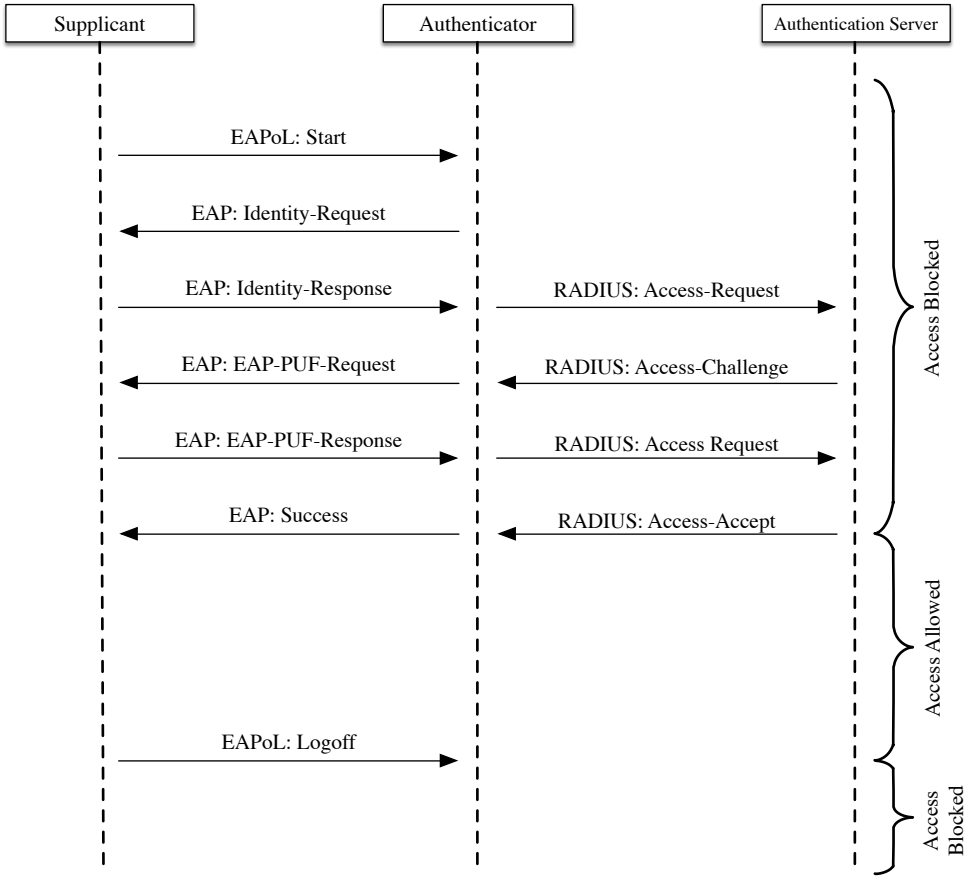

Figure 3.5: EAP-PUF authorization sequence diagram

  - EAP EAP-PUF Request
  - EAP Success
  - EAP Failure

- Transmitted Packets
  - EAPoL Start
  - EAPol Logoff
  - EAP Identity Response
  - EAP EAP-PUF Response

The structure of these packets are explained in the next subsection.

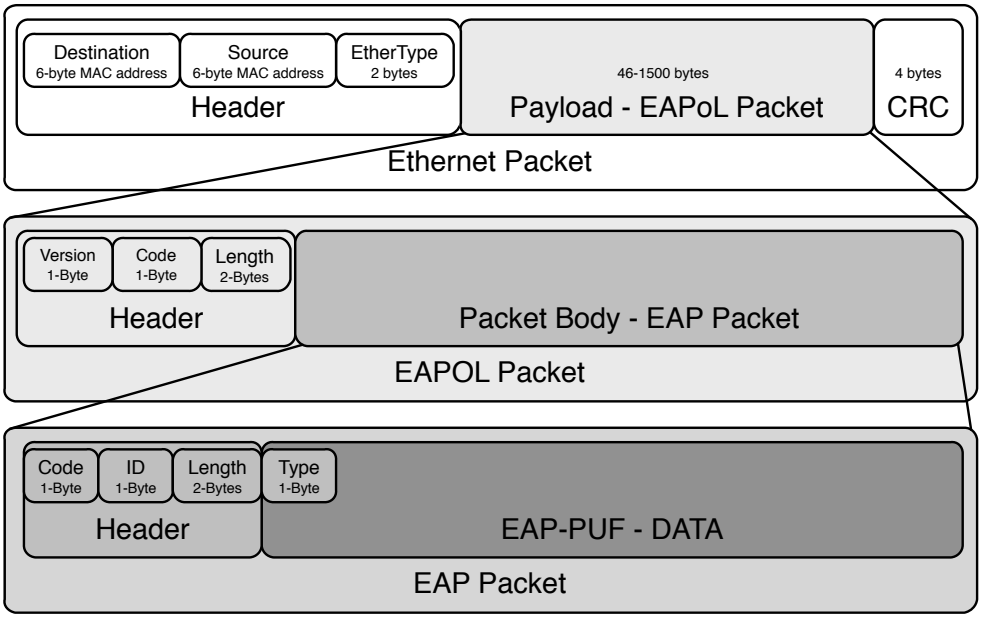

Figure 3.6: EAP-PUF encapsulated in an Ethernet Packet

3.6.2 Ethernet Packet Contents

The formating of the Ethernet packets used in EAP-PUF requires three levels of encapsulation:

- The Ethernet packet contains a EAPoL packet
- The EAPoL packet contains a EAP packet
- The EAP packet contains a EAP-PUF body

This nesting can be clearly seen in Figure 3.6, which also shows all of the heading fields that are required. The implementation of the EAP-PUF protocol requires the supplicant to be able to interpret all these fields in packets received from the authenticator and generate the appropriate fields in its responses.

3.6.2.1 Ethernet Header and FCS

Ethernet packets consist of a header, body and a 32 bit frame check sequence (FCS). The header contains the destination and source as 6-byte MAC addresses which should be globally unique identifiers. In the EAPoL protocol an additional address is used in EAPoL start packets which is a reserved group-multicast address assigned specifically to port access entitys (PAEs) (which are synonymous with EAP Authenticator devices). This is used by the supplicant at the start, as it does not know the MAC address of authenticator of the network at first. The Address is `0x0180C2000003`. The Ethertype is 2 byte field which for EAP is always set to `0x888E`.

After this comes the encapsulated EAPoL payload followed by the FCS. The FCS is used to detect communication errors through the use of a cyclic redundancy check (CRC). The header and body of an Ethernet packet are checked before sending and the CRC is computed and appended to the packet. The receiver also computes the glscrc when it receives a packet and if it is different to the one attached discards that packet without any further action being taken.

3.6.2.2 EAPoL Packet Header

EAPoL is, for the most part, a wrapper around EAP packets when sent via Ethernet. It does also contain functionality necessary to initiate and end EAP authentication sessions. Its header consists of *version*, *type* and *length* fields. The version field is 1 byte long and, as version 2 was standardised in 2004, it can in all likelihood be considered a static field, always containing the value `0b00000010`.

The type field represents whether the EAPoL packet encapsulates a EAP packet or not. It it does not then it is a start or stop packet (other codes exist, but are rarely used and not considered for the purposes of this project). The code values used can be seen in Table 3.4.

The length field specifies the length in bytes of the EAP payload (it is zero if the EAPoL packet is not a 'EAP-packet' type).

EAPoL Type	Name
0000	EAP-Packet
0001	EAPOL-Start
0010	EAPOL-Logoff

Table 3.4: EAPoL Type Codes

EAP Type	Description
1	Request
2	Response
3	Success
4	Failure

Table 3.5: EAP Types

3.6.2.3 EAP Packet Header

The EAP packet header consists of the fields; *code*, *identity* and *length*.

The code field is 1 byte long and the main codes available can be seen in Table 3.5, although there are other less used extensions which are not considered in this project. The success and failure types contain no body content and are sent from the authenticator to the supplicant at the end of the validation process. The Request type is also sent from the authenticator and contains a EAP-PUF challenge in the body.

The body of the EAP packet first contains a method type (which in certain respects can be seen as part of the header). This is 1 byte in length, and is set to either the identity type (1) in the first part of the authentication sequence or the EAP-PUF method type (192) when using EAP-PUF. A selection of common methods are listed in Table 3.6.

The request packet for identity contains only an additional 1 byte method type indicating EAP-PUF should be used in the response packet. The supplicant can indicate a desire to use another method by sending a response with a negative-acknowledgment (NAK) method type, but this feature is not considered in the final design. For an actual EAP-PUF request the challenge data (16 bytes) and helper data (32 bytes) make up the rest of the body. This data is extracted by the supplicant and passed into the SRAM-PUF

Method Value	Description
0	Reserved
1	Identity
2	Notification
3	Legacy Nak
4	MD5-Challenge
5	One-Time Password (OTP)
13	EAP-TLS
18	GSM Subscriber Identity Modules (EAP-SIM)
21	EAP-TTLS
25	PEAP
26	MS-EAP-Authentication
43	EAP-FAST
192	EAP-PUF

Table 3.6: Selected EAP Method Codes

and fuzzy extractor to generate a response.

The identity response packet consists of a one byte method type containing the code for EAP-PUF, as agreement to the usage of this method and the supplicants identity which is its own MAC address (6-bytes). This is deemed sufficient identification of the device, and is identical to the scheme used in EAP-MD5 but could easily be spoofed in practice. Further extensions to the protocol could include extra protection from MAC address spoofing. The actual EAP-PUF response packet contains the response data (256 bit digest) and nothing else, again this follows the pattern used in EAP-MD5.

3.6.2.4 simulation

Only a simulation of the EAP-PUF authentication scheme was implemented, the code can be found in section A.3.

Chapter 4

Results

4.1 SRAM-PUF

In testing on various boards the extracted memory was found to exhibit large differences in output given the same challenge, hence it c ould informally be seen that there is a large *inter-distance*. Contrastingly, very little change was ever detected on any one single board given multiple identical challenges, hence, informally, it would seem that there is a small *intra-distance*. The purpose of this project is not to explore the performance of the IS61LV25616 SRAM chip, so this was not systematically tested, but it is reassuring to know that the chip had the desired performance characteristics.

While the design was simulated in Mentor Graphic's Modelsim application using a testbench during initial development, once minimal functionality was achieved outside simulation the testbench was no longer used. After further development of the system the testbench code no longer matches with the implementation and so was deprecated and the earlier waveforms produced are not included in this dissertation as they are no longer relevant.

The SRAM-PUF was extensively tested simply by connecting the serial cable to a PC running a common serial terminal emulator. It was found to function well when data was manually typed by a user, producing the correct 4 digits of hexadecimal output (the memory contents of an SRAM address) each time 4 corresponding hexadecimal digits were entered (the

memory address of SRAM to select for output).

Timing issues were found to occur when data was sent automatically, either through MATLAB© serial functions or through a line-oriented terminal mode. These were mostly corrected, but issues still occurred when using different setups, lack of time meant that these issues were never completely eliminated. The SignalTap application was used to try and detect the source of the issues, but a lack of understanding of its functionality meant that triggering never occurred correctly and the investigation proved unfortunately fruitless. It is certain that more work is required to bring this component up to the sought level of functionality, but its ability to function manually was sufficient to complete the objectives of the project.

4.2 Fuzzy Extractor

The results from the fuzzy extractor show that it functions correctly. Below is a printout from a MATLAB© which demonstrates the fuzzy extractor running eleven times. The first run (w) is the generator process providing the original response (R) and the helper data (s and x). The next seven runs (wa05 - wa15) are simulated reproduction runs wherein 5, 8, 9, 10 11, 12 and 15 errors are introduced into the raw response. The last three (wr1 - wr3) simulate the reproduction runs on non-authentic devices with different randomly generated raw responses. Next the corresponding responses are given (R00-R15 & RR1-RR3) and the simulation concludes with the validation checking which shows that up to ten errors can be mitigated. It also shows that when a raw response has a hamming distance from the original greater than 10 the given digest response of the fuzzy extractor has a hamming distance close to 50%, i.e. the avalanche effect of the cryptographic hash in the privacy amplification procedure is effective in securing the response and preventing replay attacks:

```
w = D9DA7BEA1A31D8ABE2A27B4E855C5C5C
wa05 = D9DA6BEA1B31D8A3E2A27B4E857C5E5C
wa08 = D1D27BEA3A21D8ABE2A27B4E955D545E
```

```
wa09 = F9CA7BE21A71D8ABE2C27B4E855CDDDC
wa10 = DBEA7BCA1E21F8ABE2B27B4CA55C5C5C
wa11 = D0DA7BA81CB0D8ABE2E23B6E855C5C5C
wa12 = D1DA7BEA1A35DCBBE6B37B4AA15DDC5C
wa15 = D8DBFBEA5AB580ABE2B27B2E815C4C5E
wr1 = 7B86B5F833E9BF6A0EE6538261C5A7AE
wr2 = DD4DC9E1AAD2BB30051F0F6F2E1CB7F9
wr3 = BBD3AE3E9D8B3CE8867A027F0979F8F6

R = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
s = 35B5595F99E4A446741CAFEE3F1C08C4
x = 37982001400E20B7A67AB050F073F429

R00 = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
R05 = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
R08 = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
R09 = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
R10 = 5204AD19BDFA759B36D1CEF842B924C15A977F112E246CC4A5336E149058BDE6
R11 = CB3B16EFADAB739429A5381A5F9C21E112D23487E933B0A7B6FCA4BDF4809B99
R12 = B01F5CFF14677D560D1CEBD5C876A458BA761B563622CD4C1413690454E8464F
R15 = 5225B4E08066191C29395423B0DCDA8F4B4C4DFB73CD254B1F57CAE5DD455E13
RR1 = 23DA1F0810BD6D6D73C1C61E659267F2132DF43BB729D1702A53944F754E941A
RR2 = DF0BC75A7A5D346B4E67FDD1A4CC67C0807E4ABFFD3F90A5BFDD70689B623B5D
RR3 = 43737B59470BFBA2056FD47B239536389EDAFF3B3AAA2FDBD63D8472269FA7D1

The R00 device passed verification
The R05 device passed verification
The R08 device passed verification
The R09 device passed verification
The R10 device passed verification
The R11 device failed verification, distance: 49.2%
The R12 device failed verification, distance: 44.1%
The R15 device failed verification, distance: 53.1%
```

```
The RR1 device failed verification, distance: 48.8%
The RR2 device failed verification, distance: 51.6%
The RR3 device failed verification, distance: 48.0%
```

Next the ability to alter attributes of the fuzzy extractor is shown. The test is the same, but this time the SRAM-PUF data width is halved to 64 bits and the message length is correspondingly reduced to 30 bits. Thus only 6 bits can be corrected. The output shown the fuzzy extractor performing accordingly:

```
w = 673E9620164BD71E
wa04 = 633E9660164BF71F
wa05 = 673A12201249D71E
wa06 = 672A8660364BD79E
wa07 = 6736DEA0024BD31E
wr1 = 410BCBA3D7C877AE
wr2 = 9B05581D6F8CEC6C
wr3 = F3BB6A432F7D979F

R = 3DAD18154B7010A439AA68448151F61E74A727229DB50D8576AC59EF2F353C81
s = 3C7F8142BE595539
x = 390F18C57D17BB8E

R00 = 3DAD18154B7010A439AA68448151F61E74A727229DB50D8576AC59EF2F353C81
R04 = 3DAD18154B7010A439AA68448151F61E74A727229DB50D8576AC59EF2F353C81
R05 = 3DAD18154B7010A439AA68448151F61E74A727229DB50D8576AC59EF2F353C81
R06 = 3DAD18154B7010A439AA68448151F61E74A727229DB50D8576AC59EF2F353C81
R07 = 741373A3B67B00242E02F0EE90D07A47C08DAB741086D39F7C0D46DF8CF37C71
RR1 = 0F1069EB2913962B55B410EA144BDC594AC97BA8A33CA94EA789FA539348A0BD
RR2 = 5A28A8EB40D64B96730CB9A45D26B0019EFAC0ECD0A09ABF4A994BD7349AEBB2
RR3 = 39B7AC78B67FEBA7C20273E24F178DB9B68985B16B22D621914B1D524F54CD60

The R00 device passed verification
The R04 device passed verification
```

```
The R05 device passed verification
The R06 device passed verification
The R07 device failed verification, distance: 43.8%
The RR1 device failed verification, distance: 52.3%
The RR2 device failed verification, distance: 53.1%
The RR3 device failed verification, distance: 53.9%
```

4.3 SHA-256

The sha-256 implementation was tested to ensure it conformed with the standard specification given in [22] exactly. It was tested against the worked example included in an official description document [23] and implementations available on the web and iPhone apps. It was shown to follow the specification exactly, as long as the message fits into one block.

The output generated from MATLAB© when running the implementation of SHA-256 against the hexadecimal encoding of the ASCII text 'abc':

```
>> sha256('616263')
ans = BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
```

This can be seen to be the same result as that given on page 11 of [23].

4.4 Ethernet Authentication

Code to simulate the generation and extraction of data from Ethernet packets with EAP-PUF protocol bodies were developed (see section A.3). The resulting hexadecimal packets were constructed as follows:

EAPoL Start Packet (From Supplicant)

`0180C200000300005E005301888E02010000` + 4 Byte CRC

EAP Identity Request Packet (From Authenticator)

`00005E00530100005E0053FF888E020000060155000601B0` + 4 Byte CRC

EAP Identity Response Packet (From Supplicant)

`00005E0053FF00005E005301888E0200000B0255000B0100005E005301` + 4 Byte CRC

EAP EAP-PUF Request Packet (From Authenticator)

`00005E00530100005E0053FF888E0200003501550035B0` + Challenge(16 Bytes) + Helper Data(32 Bytes) + 4 Byte CRC

EAP EAP-PUF Response Packet (From Supplicant)

`00005E0053FF00005E005301888E0200002502550025B0` + Response(32 Bytes) + 4 Byte CRC

EAP Success Packet (From Authenticator)

`00005E00530100005E0053FF888E0200000403550004` + 4 Byte CRC

EAPoL Logoff Packet (From Supplicant)

`0180C200000300005E005301888E02020000` + 4 Byte CRC

These Ethernet packet stubs have been checked and verified to correctly follow the specification proposed.

4.4.1 CRC

The CRC algorithm is used to generate the FCS appended to every Ethernet packet. Below is the output of running the MATLAB© implementation developed for this project against the EAPoL Start Packet as generated above:

```
>> crc('0180C200000300005E005301888E02010000')
ans = F7D19377
```

This can be seen to be the same as the specification by checking using the online FCS generator at `http://www.lammertbies.nl/comm/info/crc-calculation.html`.

4.5 Overall System Results

The overall system was tested through a demonstration as shown in Figure 4.1. While not all parts of the EAP-PUF protocol were simulated and

there are still issues with the UART implementation in communicating effectively with MATLAB© , in general terms the system can be said to be functional in a prototype form.

There is much that could be done to improve and develop the implementation, but it shows in broad terms that the design proposed works, which was the intention of the project.

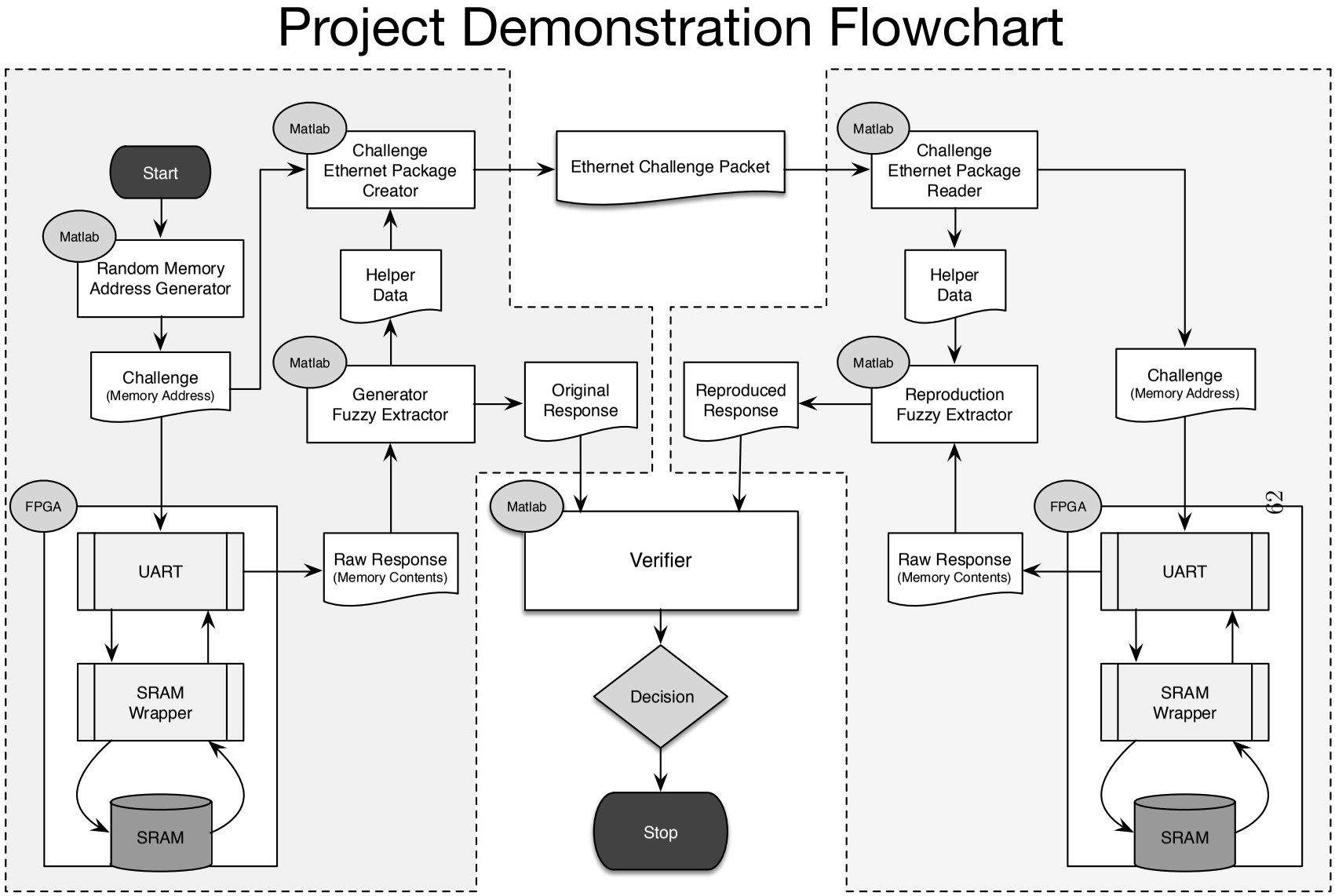

Figure 4.1: Full system testing demonstration flowchart

Chapter 5

Conclusions

5.1 Achievements

5.1.1 Goals Met

In the time alloted a functioning prototype of an achievable PUF-based network authentication system has been achieved. A working example of a PUF was created. Its potential for use in cryptographically secure applications was fulfilled through encapsulating it with a fuzzy extractor. A novel approach for harnessing the properties of a secure PUF was demonstrated through wedding it to an extension of an existing authentication framework. Most importantly, the potential for a real, practical solution to the problem of automating embedded device authentication at the data-link layer has been shown, and a path to its realisation has been marked out.

In learning terms, this project has surpassed expectations. It has exposed many topics of further interest and increased understanding of both many technical areas and of good design approaches to be used in future projects. A greater familiarity of the use of MATLAB© , FPGA synthesis using HDLs, LaTeXand other tools has been fostered. Technical and theoretical experience with communication protocols (both RS-232 and Ethernet), cryptographic hash functions, error correction & detection codes, SRAM, RNGs, LFSRs, network security and other technologies has been gained. Gaining a fuller understanding of vital parts of the Academic research process; such as the

best ways to find relevant academic papers, standards documentation and other literature has also come about through the process of completing this project.

5.1.2 Benefits of a Modular Approach

In tackling a novel, wide-ranging problem such as that underlying this project, the agile 'Do the simplest thing that could possibly work first.' mantra espoused by experienced developers [24] has been used in carrying out the work necessary.

At the outset of this project, the ambitious goal of fully implementing a working FPGA implementation of an authentication solution utilising the unique properties of PUFs was provisionally set. In the initial design stages it became readily apparent that this was too ambitious a task and the project was sensibly scaled back to a semi-simulated, hybrid approach. This turned out to be a virtuous decision, as all the components required for a complete system could be investigated to a appropriate level given the time and experience constraints. This modular nature allows for incremental development towards the full design. This, in turn, allowed for changes to the specification as previous assumptions were found to be wrong. In contrast, if the decision had been taken to develop a fully functional design in one single effort the probability of restricting the ability to redesign when better solutions were found was dangerously high. This advantageous property can also serve as the basis for wide-ranging further research into the topics of PUFs, fuzzy extractors and authentication.

5.2 Limitations

While the project has met a satisfactory level of completion, there is much that is missing. Given time it would be preferable to have implemented more of the design in VHDL rather than MATLAB© . While simulation is valuable, it leaves some potential issues unexplored. This is due to the lack of resource constraints that any real system would encounter. Without knowing

the true bounds of a final implementation from the outset, none could be encountered when simulating parts of the system. This could jeopardize the ultimate feasibility of the solution delivered. Also, the exploration of the EAP-PUF protocol design could have been improved through use of Ethernet simulation tools which are available in MATLAB© through the Simulink platform. These were not used due to lack of training, confidence and familiarity with the software.

5.3 Implications

This project could be used as the basis from which a practical system could be made. One which, if turned into a reality, could have benefits the way in which embedded devices are used in the home or office. By removing the need for manually authenticating devices onto wireless networks in particular a potential barrier-to-entry to the future expansion of the 'Internet-of-Things' could be removed. Once removed, inexpensive networked devices and sensors could become ubiquitous sooner and in greater quantity. Ultimately this could have wide-ranging benefits to society as a whole.

5.4 Future Research

While the objectives of the project were achieved, there is a great deal of scope for further work on this topic. All of the technologies and concepts encountered in this project could be explored in much greater depth. As the project covered so many areas it is difficult to think which would offer the greatest research potential. While active research is still focused on the properties of PUFs, the level of research into their applications is very much less intense. There is virgin research territory to explore in the current gap between PUF devices and the security of electronic networks.

Chapter 6

References

[1] G. Press, "Internet of things by the numbers: Market estimates and forecasts," *Forbes Magazine*, 2014. [Online]. Available: http://www.forbes.com/sites/gilpress/2014/08/22/internet-of-things-by-the-numbers-market-estimates-and-forecasts/

[2] P. Tuyls, B. Škoric, and T. Kevenaar, *Security with Noisy Data: On Private Biometrics, Secure Key Storage and Anti-Counterfeiting*. Springer, 2007. [Online]. Available: http://books.google.co.uk/books?id=psWUQ5cDsI0C

[3] K. Ashton, "That internet of things thing," *RFiD Journal*, vol. 22, pp. 97–114, 2009.

[4] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical One-Way Functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, Sep. 2002. [Online]. Available: http://dx.doi.org/10.1126/science.1074376

[5] A. Maiti, J. Casarona, L. McHale, and P. Schaumont, "A large scale characterization of ro-puf," in *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 94–99.

[6] J. Guajardo, B. Škorić, P. Tuyls, S. S. Kumar, T. Bel, A. H. Blom, and G.-J. Schrijen, "Anti-counterfeiting, key distribution, and key storage

in an ambient world via physical unclonable functions," *Information Systems Frontiers*, vol. 11, no. 1, pp. 19–41, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10796-008-9142-z

[7] R. Van Den Berg, B. koric, and V. Van Der Leest, "Bias-based modeling and entropy analysis of pufs," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, ser. TrustED '13, 2013, Conference Proceedings, pp. 13–20, export Date: 14 April 2014 Source: Scopus. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84889016630&partnerID=40&md5=7a37c991d1c9f64f2cef4e5cbc7f07c3

[8] A. Maiti, V. Gunreddy, and P. Schaumont, *A systematic method to evaluate and compare the performance of physical unclonable functions*. Springer, 2013, pp. 245–267.

[9] R. Maes, V. Rozic, I. Verbauwhede, P. Koeberl, E. Van der Sluis, and V. van der Leest, "Experimental evaluation of physically unclonable functions in 65 nm cmos," in *ESSCIRC (ESSCIRC), 2012 Proceedings of the*. IEEE, 2012, Conference Proceedings, pp. 486–489.

[10] Altera. (2006) De1 development and education board user manual. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE1/DE1_User_Manual.pdf

[11] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Advances in cryptology-Eurocrypt 2004*, Springer. Springer, 2004, Conference Proceedings, pp. 523–540.

[12] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "Spongent: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems–CHES 2011*. Springer, 2011, pp. 312–325.

[13] R. Maes, *Physically Unclonable Functions: Constructions, Properties and Applications*. Springer Berlin Heidelberg, 2013. [Online]. Available: http://books.google.co.uk/books?id=dPjMngEACAAJ

[14] A. Van Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids," in *Financial Cryptography and Data Security*. Springer, 2012, pp. 374–389.

[15] N. Sklavos and O. Koufopavlou, "On the hardware implementations of the sha-2 (256, 384, 512) hash functions," in *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, vol. 5. IEEE, 2003, pp. V–153.

[16] ISO, *Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Part 1X: Port-based network access control*, International Organization for Standardization ISO, 2013.

[17] Altera. (2011) Is61wv25616edbll: 256k x 16 high speed asynchronous cmos static ram with ecc. [Online]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE1/DE1_Datasheets.zip

[18] H. Feistel, "Cryptography and computer privacy," *Scientific american*, vol. 228, pp. 15–23, 1973.

[19] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260–269, 1967.

[20] S. K. Sanadhya and P. Sarkar, "A combinatorial analysis of recent attacks on step reduced sha-2 family," *Cryptography and Communications*, vol. 1, no. 2, pp. 135–173, 2009.

[21] IANA. (2014) Extensible authentication protocol (eap) registry. [Online]. Available: http://www.iana.org/assignments/eap-numbers/eap-numbers.xhtml

[22] N. FIPS, "180-2: Secure hash standard (shs)," Technical report, National Institute of Standards and Technology (NIST), Tech. Rep., 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[23] ——, "Descriptions of sha-256, sha-384, and sha-512," Technical report, National Institute of Standards and Technology (NIST), Tech. Rep., 2001. [Online]. Available: http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

[24] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

References generated using BibTeX

Appendix A

Appendix

A.1 SRAM-PUF Implementation in VHDL

A.1.1 puf.vhd - Top Level Design Entity

```
1  --
     ----------------------------------------------------------------
2  -- PUF  - Core Physically Unclonable Function
     Implementation
3  --        For EEE8097 Masters Project
4  --
5  -- Links both the SRAM wrapper and UART modules
     together and
6  -- to the pins connected to the on-board hardware of
      the DE-1
7  --
8  -- Copyright (C) 2014 Michael Walker
9  --
     ----------------------------------------------------------------
10
```

```
11 LIBRARY ieee;
12 USE ieee.std_logic_1164.all;
13
14 -- This is the Top-Level Entity of the PUF system
      implemented in FPGA
15 -- all signals in port list of entity are linked to
      physical parts of DE-1
16 -- with necessary names for pins as specified in the
       user manual.
17 ENTITY PUF IS
18   PORT
19   (
20     CLOCK_50   : IN  STD_LOGIC;
         --Master Clock (50MHz)
21     KEY        : IN  STD_LOGIC_VECTOR(0 TO 0);
         --KEY0 button for reset
22
23     UART_RXD   : IN  STD_LOGIC;
         --RS-232 Receive wire
24     UART_TXD   : OUT STD_LOGIC;
         --RS-232 Transmit wire
25
26     SRAM_DQ    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
         --16-bit SRAM Data bus
27     SRAM_ADDR  : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
         --18-bit SRAM Address bus
28
29     LEDG       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         --Green LEDs - RXD Byte
30     LEDR       : OUT STD_LOGIC_VECTOR(0 TO 0);
         --Red LED - not(reset)
31
```

```
32      SRAM_WE_N : OUT STD_LOGIC;
           -- SRAM Write Enable
33      SRAM_OE_N : OUT STD_LOGIC
           -- SRAM Output Enable
34    );
35  END PUF;
36
37  ARCHITECTURE rtl OF PUF IS
38
39  -- UART communicates via RS-232 with rest of PUF
       system simulated in Matlab
40  -- It handles both converting the serial stream
       received over RS-232 into
41  -- parallel bytes to be passed to SRAM component and
        the return path back.
42  COMPONENT UART1
43    PORT(
44      CLK : IN STD_LOGIC;
45      RST : IN STD_LOGIC;
46      RXD : IN STD_LOGIC;
47      TXD : OUT STD_LOGIC;
48      DRX : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
49      DTX : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
50      STR : OUT STD_LOGIC;
51      STT : IN STD_LOGIC
52    );
53  END COMPONENT;
54
55  -- SRAM receives in a 16-bit address as 4 bytes of
       ASCII hexadecimal
56  -- characters and sets the address bus of the SRAM
       memory to that value.
```

```
57 -- It then retrieves the resulting 16-bit data from
       the SRAM memory and
58 -- transmits it back in same hexadecimal format.
59 COMPONENT SRAM
60   PORT(
61     CLK : IN  STD_LOGIC;
62     RST : IN  STD_LOGIC;
63     AIN : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
64     AOT : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
65     DIN : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
66     DOT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
67     STR : IN  STD_LOGIC;
68     STT : OUT STD_LOGIC;
69     WEN : OUT STD_LOGIC;
70     OEN : OUT STD_LOGIC
71   );
72 END COMPONENT;
73
74 -- Connecting signals linking the two components
75 SIGNAL ADDR_BUS : STD_LOGIC_VECTOR(7 DOWNTO 0);
76 SIGNAL DATA_BUS : STD_LOGIC_VECTOR(7 DOWNTO 0);
77 SIGNAL STR_WIRE : STD_LOGIC;
78 SIGNAL STT_WIRE : STD_LOGIC;
79
80 BEGIN
81
82 -- Link up the LEDs to the Reset Button (KEY0) and
       the most recently
83 -- received byte in the UART (DRX)
84 LEDR <= KEY;
85 LEDG <= ADDR_BUS;
86
87 uart_inst : UART1
```

```
88  PORT MAP(
89    CLK => CLOCK_50,
90    RST => KEY(0),
91    RXD => UART_RXD,
92    DRX => ADDR_BUS,
93    DTX => DATA_BUS,
94    TXD => UART_TXD,
95    STR => STR_WIRE,
96    STT => STT_WIRE
97  );
98
99  sram_inst : SRAM
100 PORT MAP(
101   CLK => CLOCK_50,
102   RST => KEY(0),
103   AIN => ADDR_BUS,
104   AOT => SRAM_ADDR,
105   DIN => SRAM_DQ,
106   DOT => DATA_BUS,
107   STR => STR_WIRE,
108   STT => STT_WIRE,
109   WEN => SRAM_WE_N,
110   OEN => SRAM_OE_N
111 );
112
113 END rtl;
```

A.1.2 uart.vhd - Implementation of UART module

```
1  --
   ----------------------------------------------------------------------------
2  -- UART - Universal Asynchronous Receiver/
   Transceiver
3  --        For EEE8097 Masters Project
4  --
5  -- Converts RS-232 serial input into parallel 8-bit
   (Byte) output
6  -- and parallel 8-bit input into serial RS-232
   output
7  --
8  -- Copyright (C) 2014 Michael Walker
9  --
   ----------------------------------------------------------------------------
10
11 LIBRARY ieee;
12 USE ieee.std_logic_1164.all;
13 USE ieee.numeric_std.all;
14 USE ieee.math_real.all;
15
16 ENTITY UART IS
17   GENERIC(
18     F_CLK : INTEGER := 50000000;  -- Master clock
         frequency: default = 50MHz
19     BAUD  : INTEGER := 115200;    -- Baud for RS
         -232: default = 115,200
20     BITS  : INTEGER := 8          -- Number of Data
         Bits
21   );
```

```
22    PORT
23    (
24      CLK : IN  STD_LOGIC;                      --
          Master Clock (50MHz)
25      RST : IN  STD_LOGIC;                      --
          Reset (Active Low)
26      RXD : IN  STD_LOGIC; -- UART Serial In
27      TXD : OUT STD_LOGIC; -- UART Serial Out
28      DRX : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- SIPO
          Register for Receiver
29      DTX : IN  STD_LOGIC_VECTOR(7 DOWNTO 0); -- PISO
          Register for Transmitter
30      STR : OUT STD_LOGIC; -- Strobe out when a byte
          completely received
31      STT : IN  STD_LOGIC  -- Strobe in when a byte is
           ready to be transmitted
32    );
33  END ENTITY;
34
35  ARCHITECTURE rtl OF UART IS
36
37    -- Defaults 50M/115200 = 434 clock cycles per bit
38    CONSTANT FRQ_BIT  : INTEGER := F_CLK / BAUD;
39    -- Need width to count down from 1.5 times the
        cycles per bit constant
40    CONSTANT MAX_FRQ  : INTEGER := FRQ_BIT + (FRQ_BIT
        / 2);
41    CONSTANT FRQ_WIDTH: INTEGER := INTEGER(ceil(log2(
        REAL(MAX_FRQ))));
42    -- Need width to count all data bits and include a
         'no more bits' state too
43    CONSTANT BIT_WIDTH: INTEGER := INTEGER(ceil(log2(
        REAL(BITS + 1))));
```

```
44    -- Counters for timing length of one bit based on
         cycles of the master clock
45    SIGNAL FRQ_CNT_RX : UNSIGNED(FRQ_WIDTH - 1 DOWNTO
         0);
46    SIGNAL FRQ_CNT_TX : UNSIGNED(FRQ_WIDTH - 1 DOWNTO
         0);
47    -- Counters for counting the data bits in one RS
         -232 'packet'
48    SIGNAL BIT_CNT_RX : UNSIGNED(BIT_WIDTH - 1  DOWNTO
          0);
49    SIGNAL BIT_CNT_TX : UNSIGNED(BIT_WIDTH - 1  DOWNTO
          0);
50    -- Readable copy of unreadable STR output
51    SIGNAL STR_INT    : STD_LOGIC;
52
53  BEGIN
54
55    PROCESS (CLK, RST)
56    BEGIN
57      -- Reset Condition
58      IF (RST ='0') THEN
59        FRQ_CNT_RX <= (OTHERS=>'0');
60        FRQ_CNT_TX <= (OTHERS=>'0');
61        BIT_CNT_RX <= (OTHERS=>'0');
62        BIT_CNT_TX <= (OTHERS=>'0');
63        STR_INT    <= '0';
64      -- Normal Operating Conditions
65      ELSIF (RISING_EDGE(CLK)) THEN
66
67        -- RECEIVER SECTION
68        --------------------
69        -- If both counters 'finished' counting down
             then just go idle and
```

```
70          -- Wait for a start bit, this is when signal
               on RXD goes low
71          IF ((BIT_CNT_RX = 0) AND (FRQ_CNT_RX = 0))
               THEN
72            -- The Start bit just arrived - get ready to
                  read data bits
73            IF (RXD = '0') THEN
74              -- Wait for start bit to finish, plus wait
                    another half bit
75              -- to allow data have settled when read
76              FRQ_CNT_RX <= TO_UNSIGNED(MAX_FRQ,
                   FRQ_WIDTH);
77              BIT_CNT_RX <= TO_UNSIGNED(BITS, BIT_WIDTH)
                   ;
78            END IF;
79          -- If end of frequency count, but not bit
               count, then it is time
80          -- to receive a bit and store it in the
               register
81          ELSIF ((BIT_CNT_RX < BITS) AND (BIT_CNT_RX >
               0) AND
82              (FRQ_CNT_RX = 0)) THEN
83            -- Store the received bit value
84            -- Nb. Serial data is Little-Endian; LSB
                  comes first
85            DRX(BITS - TO_INTEGER(BIT_CNT_RX)) <= RXD;
86            -- Reset the clocks per bit counter; wait
                  for middle of next bit
87            FRQ_CNT_RX <= TO_UNSIGNED(FRQ_BIT, FRQ_WIDTH
                  );
88            -- If this is the last bit to be received
                  then trigger the
```

```
89          -- strobe to indicate parallel data now
               ready to be read.
90          IF BIT_CNT_RX = 1 THEN
91            STR_INT<='1';
92          END IF;
93          -- Decrement the bit counter; because this
               bit now processed
94          BIT_CNT_RX <= BIT_CNT_RX - 1;
95        -- If not at the end of the frequency count,
             then nothing to do
96        -- but continue counting down until it is at
             the end...
97        ELSE
98          FRQ_CNT_RX <= FRQ_CNT_RX - 1;
99        END IF;
100
101       -- Always reset the strobe on the next clock
             cycle
102       IF (STR_INT='1') THEN
103         STR_INT <= '0';
104       END IF;
105
106       -- TRANSMITTER SECTION
107       ----------------------
108       -- If a strobe to transmit is received then
             start up the
109       -- transmission process
110       IF (STT = '1') THEN
111         -- First bit to send is the Start bit, which
               is always '0'
112         TXD        <= '0';
113         -- Start up the Counters for transmitting
```

```
114         FRQ_CNT_TX <= TO_UNSIGNED(FRQ_BIT - 1,
              FRQ_WIDTH);
115         BIT_CNT_TX <= TO_UNSIGNED(BITS, BIT_WIDTH);
116       -- If Data bits to send, send them a end of
              frequency count
117       ELSIF ((FRQ_CNT_TX = 0) AND (BIT_CNT_TX > 0))
              THEN
118         -- Send the current Data bit (nb. LSB First)
119         TXD         <= DTX(BITS - TO_INTEGER(
              BIT_CNT_TX));
120         -- Update counters
121         BIT_CNT_TX <= BIT_CNT_TX - 1;
122         FRQ_CNT_TX <= TO_UNSIGNED(FRQ_BIT -1,
              FRQ_WIDTH);
123       -- If sent all the data bits, send stop bit
              (1) and go idle
124       ELSIF FRQ_CNT_TX=0 AND BIT_CNT_TX=0 THEN
125         TXD <= '1';
126       -- not the end of the frequency count yet, so
              keep counting
127       -- down until it is at an end
128       ELSE
129         FRQ_CNT_TX <= FRQ_CNT_TX - 1;
130       END IF;
131
132     END IF;
133   END PROCESS;
134
135   -- Keep track of the strobe output which cannot be
        read in the process
136   -- by maintaining an internal signal copy that can
        be read.
137   STR <= STR_INT;
```

```
138
139 END rtl;
```

A.1.3 sram.vhd - Wrapper function for SRAM

```
1  --
     ----------------------------------------------------------------
2  -- SRAM - Wrapper around the SRAM chip on-board the
     DE1
3  --        For EEE8097 Masters Project
4  --
5  -- Converts received 4 character ASCII hexadecimal
     into 16-bit Address
6  -- and sets the 18-bit address bus to the same (with
      2 MSB set to '00').
7  -- then it converts 16-bit data bus values to
     transmit as ASCII hexadecimal.
8  --
9  -- Copyright (C) 2014 Michael Walker
10 --
     ----------------------------------------------------------------
11
12 LIBRARY ieee;
13 USE ieee.std_logic_1164.all;
14 USE ieee.numeric_std.all;
15
16 ENTITY SRAM IS
17   PORT(
18     CLK : IN  STD_LOGIC;                    -- MAIN
         Clock
19     RST : IN  STD_LOGIC;                    --
        Reset (Active Low)
20
```

```
21      AIN : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);  --
          Hexadecimal Nibble In
22      AOT : OUT STD_LOGIC_VECTOR(17 DOWNTO 0); --
          Address Bus to SRAM Memory
23
24      DIN : IN  STD_LOGIC_VECTOR(15 DOWNTO 0); -- Data
           Bus to SRAM Memory
25      DOT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  --
          Hexadecimal Nibble Out
26
27      STR : IN  STD_LOGIC; -- Strobe from UART when it
           receives a byte
28      STT : OUT STD_LOGIC; -- Strobe to UART when it
          should transmit a byte
29
30      WEN : OUT STD_LOGIC;                      --
          Write Enable  (Active Low)
31      OEN : OUT STD_LOGIC                       --
          Output Enable (Active Low)
32    );
33  END ENTITY;
34
35  ARCHITECTURE rtl OF SRAM IS
36
37    -- There are 4 HEX characters per 16-bit address
        to receive or data to send
38    -- 4 'modes': upper '11' for highest nibble or
        lower '00' for lowest nibble
39    SIGNAL TMODE: UNSIGNED( 1 DOWNTO 0); -- keep track
         of nibble to transmit
40    SIGNAL TPREV: UNSIGNED( 1 DOWNTO 0); -- previous
        TMODE, watched for change
```

```
41    SIGNAL RMODE: UNSIGNED( 1 DOWNTO 0); -- keep track
          of nibble to be received
42    -- Counter to wait for current transmission to
          finish before sending another
43    SIGNAL TWAIT: UNSIGNED(13 DOWNTO 0); -- Wait for
          10 bits * 443 clocks = 4430
44    -- Counter to wait for 64 characters sent, then
          need to send a linefeed char
45    SIGNAL LFCNT: UNSIGNED( 5 DOWNTO 0) := (OTHERS =>
          '1');
46
47  BEGIN
48
49    WEN <= '1'; -- Prevent ability to write data to
          SRAM (Inactive high)
50    OEN <= '0'; -- Ensure ability to output data from
          SRAM (Active low)
51
52    PROCESS (CLK, RST)
53      -- temporary stores for conversion of ASCII to
            binary nibbles
54      VARIABLE RNIB: UNSIGNED(3 DOWNTO 0);
55      VARIABLE TNIB: UNSIGNED(3 DOWNTO 0);
56      VARIABLE VALIDHEX: BOOLEAN;
57    BEGIN
58      VALIDHEX := FALSE;
59      -- Reset Condition
60      IF (RST = '0') THEN
61        RMODE <= "11"; -- Start receive  at upper
              nibble first
62        TMODE <= "11"; -- Start transmit at upper
              nibble first
```

```
63        LFCNT <= (OTHERS => '1'); -- Reset linefeed
              count (2^6 = 64)
64        AOT(17 DOWNTO 16) <= "00"; -- Leave the 2 MSBs
              unchanged
65        AOT(15 DOWNTO 0)  <= (OTHERS => '0'); -- Reset
              memory address
66      -- Normal Operating Conditions
67      ELSIF RISING_EDGE(CLK) THEN
68        -- Prevent inferred latches for unchanged 2
              MSB in address
69        AOT(17 DOWNTO 16) <= "00"; -- Leave 2 MSB
              unchanged
70
71        -- RECEIVER SECTION
72        --------------------
73        -- Check if strobe received , if so, set
              received address
74        IF (STR = '1') THEN
75
76          -- Calculate 4-bit binary value of
                hexadecimal character,
77          -- ensuring ASCII character is a valid hex -
                (0-9 or a-f).
78          -- First - Numbers (in ASCII upper 4 bits
                will be '0011')
79          -- note Lower nibble of the ASCII will map
                correctly for number
80          IF ((AIN(7 DOWNTO 4) = "0011") AND
81            (AIN(3 DOWNTO 0) < "1010")) THEN
82            -- Set nibble value ready to place on
                  address bus
83            RNIB   := UNSIGNED(AIN(3 DOWNTO 0));
84            VALIDHEX := TRUE;
```

```
85          -- Second - Letters (either lower or upper
               case: a-f OR A-F)
86          -- (in ASCII upper four bits will be 0110
               for lower case
87          --  or 0100 for upper case. lower bit will
               be 0001 for 'a/A'
88          --  up to 0110 for 'f/F'. this means they
               are offset 9
89          --  lower than their correct binary/
               hexadecimal value..
90          --  eg. HEX A = BIN 1010, but lower nibble
               of ASCII A is '0001'
91          --  to convert ADD '9'... this means 0001 +
               1001 = 1010 = HEX A)
92          ELSIF ((AIN(7 DOWNTO 4) = "0110") OR
93               (AIN(7 DOWNTO 4) = "0100")) AND
94              (AIN(3 DOWNTO 0) < "0111") THEN
95            -- Set nibble value ready to place on
                 address bus
96            RNIB     := UNSIGNED(AIN(3 DOWNTO 0)) +
                 TO_UNSIGNED(9,4);
97            VALIDHEX := TRUE;
98          END IF;
99
100         IF (VALIDHEX) THEN
101           IF (RMODE = "11") THEN
102             AOT(15 DOWNTO 12) <= STD_LOGIC_VECTOR(
                   RNIB);
103             RMODE <= "01";
104           ELSIF RMODE = "10" THEN
105             AOT(11 DOWNTO  8) <= STD_LOGIC_VECTOR(
                   RNIB);
106             RMODE <= "01";
```

```
107           ELSIF RMODE = "01" THEN
108             AOT( 7 DOWNTO  4) <= STD_LOGIC_VECTOR(
                  RNIB);
109             RMODE <= "00";
110           ELSIF RMODE = "00" THEN
111             AOT( 3 DOWNTO  0) <= STD_LOGIC_VECTOR(
                  RNIB);
112             RMODE <= "11";
113             -- Got full address word, so transmit
                  data after small
114             -- delay to allow SRAM output to settle
115             TWAIT <= TWAIT + TO_UNSIGNED(3,14);
116             -- Just ensure upper bit sent first...
117             TMODE <= "11";
118           -- now check fOR letters a-f OR A-F
119           END IF;
120         END IF;
121       END IF;
122
123       -- TRANSMITTER SECTION
124       ----------------------
125       -- If something ready to send and UART ready
            then send it
126       -- use penultimate count (1) to set up
            transmission, so that the
127       -- final count (0) can be used to return
            system to an idle state.
128
129       -- First - if we are waiting to be ready, then
             just keep waiting
130       IF (TWAIT >  TO_UNSIGNED(1,14)) THEN
131         TWAIT <= TWAIT - TO_UNSIGNED(1,14);
```

```
132         -- Penultimate Count - setup for transmission
               of memory contents
133         ELSIF (TWAIT = TO_UNSIGNED(1,14)) THEN
134           -- Set strobe for transmission of a byte by
                 UART high
135           STT   <= '1';
136           -- ensure countdown stops next cycle - hard
                 set count to zero
137           TWAIT <= TO_UNSIGNED(0,14);
138
139           -- Extract nibble to send as hex from SRAM
                 data bus
140           CASE TMODE IS
141             WHEN "11" => TNIB := UNSIGNED(DIN(15
                    DOWNTO 12));
142             WHEN "10" => TNIB := UNSIGNED(DIN(11
                    DOWNTO  8));
143             WHEN "01" => TNIB := UNSIGNED(DIN( 7
                    DOWNTO  4));
144             WHEN "00" => TNIB := UNSIGNED(DIN( 3
                    DOWNTO  0));
145             WHEN OTHERS => TNIB := "0000";
146           END CASE;
147
148           -- Set ASCII character to be sent on this
                 transmission --
149           -- Set upper nibble to '0011' for ASCII
                 numerals if binary < 10
150           -- otherwise set it to '0100' for ASCII
                 upper case letters (A-F)
151           -- Set lower nibble directly from data bus
                 if numeral to be used
```

```
152         -- for letters, need to subtract 9 from bus
               data to convert.
153         IF TNIB < TO_UNSIGNED(10,4) THEN
154           DOT(7 DOWNTO 4) <= "0011";
155           DOT(3 DOWNTO 0) <= STD_LOGIC_VECTOR(TNIB);
156         ELSE
157           DOT(7 DOWNTO 4) <= "0100";
158           DOT(3 DOWNTO 0) <= STD_LOGIC_VECTOR(TNIB -
                 TO_UNSIGNED(9,4));
159         END IF;
160
161         -- Update State
162         -- Special case first - might need to send
               linefeed character
163         -- override output if this is the case
164         IF (LFCNT = TO_UNSIGNED(0,6)) THEN
165           LFCNT <= TO_UNSIGNED(63,6); -- Reset line
                 counter
166           DOT <= "00001010"; -- ASCII CODE for a
                 linefeed (0A in HEX)
167           TMODE <= "11"; -- next nibble should be
                 start of new data
168         -- First Nibble to be sent
169         ELSIF TMODE = "11" THEN
170           TMODE <= "10";
171           LFCNT <= LFCNT - TO_UNSIGNED(1,6);
172         ELSIF TMODE = "10" THEN
173           TMODE <= "01";
174           LFCNT <= LFCNT - TO_UNSIGNED(1,6);
175         ELSIF TMODE = "01" THEN
176           TMODE <= "00";
177           LFCNT <= LFCNT - TO_UNSIGNED(1,6);
178         ELSIF TMODE = "00" THEN
```

```
179             TMODE <= "11";
180             LFCNT <= LFCNT - TO_UNSIGNED(1,6);
181           END IF;
182         -- If character just sent is NOT last of 4
              then more to transmit
183         ELSIF ((TMODE /= "11") AND (TPREV /= "00"))
              THEN
184           --Reset strobe and wait for transmission of
                the current data
185           -- before sending the next character.
186           STT   <= '0';
187           TWAIT <= TO_UNSIGNED(4500,14);
188         ELSE
189           -- Last character of data just sent, so just
                go idle
190           -- reset strobe and await transmission of
                first character
191           STT   <= '0';
192           TMODE <= "11";
193         END IF;
194         -- Keep track of last process cycle
              Transmission mode, so that
195         -- we can spot when the last character is sent
196         TPREV <= TMODE;
197       END IF;
198     END PROCESS;
199 END;
```

A.2 Fuzzy Extractor implementation in Matlab

A.2.1 FuzzyGenerator.m - Implementation of Generation process of Fuzzy Extractor

```
1  %% Fuzzy Extractor Generation Procedure
2  % w - Raw PUF data as input, will be 1 bit longer
       than Code length (n),
3  %     by default this will be 127 - so w will be 128
        bits. (requried)
4  %     the first bit will not be used in the fuzzy
       extractor, as it requires
5  %     a length that is 2^m - 1, whereas the PUF data
        comes as multiples of
6  %     16-bit memory contents, and as such will not
       fit. exactly. Instead
7  %     the Fuzzy Extractor expects PUF data to equal
       2^m (i.e. 1-bit larger)
8  % k - Messagte length (optional smaller will make
       fuzzy extractor more
9  %     robust - defaults to 64, which allows 10
       errors to be corrected)
10 % m - Code length will be (2^m)-1, (optional -
       defaults to 7 - so n = 127)
11 % R - Original Response of the fuzzy extractor - to
       compare with the
12 %     later response of the reproduction procedure
       to validate it.
13 %     (256-bits - SHA-256 output digest)
14 % s - Helper Data 1 - Secure Sketch output of PUF &
       random 'rng1' data
```

```
15 %   - this will be
16 % x - Helper Data 2 - Random 'rng2' value used in
       Randomness Extractor
17 function [R, s, x] = FuzzyGenerator(w,k,m)
18     switch nargin
19         case 3
20             n = 2^m - 1; % variable set code length
21         case 2
22             n = 2^7 - 1; % default code length    =
                   127 bits
23         otherwise
24             k = 64;      % default message length =
                   64 bits
25             n = 2^7 - 1; % default code length    =
                   127 bits
26     end
27
28     % convert hexadecimal input to binary vector
           format
29     w = hexToBinaryVector(w,n+1);
30
31     % setup the BCH Encoder used in the Secure
           Sketch
32     GenEnc = comm.BCHEncoder('CodewordLength',n,'
           MessageLength',k);
33     % Produce a random binary number of length 'k'
           for secure sketch, used
34     % as the message for the bch encoder to encode
           and in some sense this
35     % is what is ultimately the job of the
           FuzzyReproducer to recreate.
36     rng1 = randi([0 1],1,k);
```

```
37      % Produce a random  binary number of length 'n',
            it will be combined
38      % with the (almost all) of the PUF data for use
           in randomness extractor
39      rng2 = randi([0 1],1,n);
40
41      % Perfore the Sketch part of secure sketch by
           running the BCH encoder
42      % on the rng1 data and xoring the result with (
           almost all) the PUF data
43      % then output the result as Hexadecimal helper
           data 's'.
44      ss = step(GenEnc, rng1')';
45      sbin = xor(ss,w(2:end));
46      s = char(binaryVectorToHex(sbin));
47
48      % Peform the Randomness Extraction by running
           the SHA-256 hash function
49      % on the xor of (almost all) the puf data and a
           complementary random
50      % number to get the digest output (rearranged to
            row vector)
51      xw = xor(rng2,w(2:end));
52      R = sha256(char(binaryVectorToHex(xw)));
53      % output the random number used in the
           rondomness extractor
54      % to be used as helper data 'x'
55      x = char(binaryVectorToHex(rng2));
56  end
```

A.2.2 FuzzyReproducer.m - Implementation of Reproduction process of Fuzzy Extractor

```
%% Fuzzy Extractor Reproduction Procedure
% w - Raw PUF data as input, will be 1 bit longer than Code length (n),
%     by default this will be 127 - so w will be 128 bits. (requried)
% s - Helper Data 1 - Used in Secure Sketch Recovery (required)
% x - Helper Data 2 - Used in Randomness Extractor ( required)
% k - Message Length used in BCH Encoder - defaults to 64 bits (optional
%     but should be the same as that used in the FUzzyGenerator function.
% R - Response of the fuzzy extractor - to compare with the
%     original response of the generation procedure to validate it.
%     (256-bits - SHA-256 output digest)
function R = FuzzyReproducer(w,s,x,k)
    % setup parameters to defaults if not passed in
    switch nargin
        case 4
            kk = k;  % veriable set message length
        otherwise
            kk = 64; % default message length = 64 bits
    end

    % convert hexadecimal inputs to binary vector format
```

```
21      w = hexToBinaryVector(w,size(s,2)*4);
22      s = hexToBinaryVector(s,size(s,2)*4);
23      x = hexToBinaryVector(x,size(x,2)*4);
24
25      % Trim s and x down to 2^m-1, instead of 2^m -
            remove MSB...
26      s = s(2:end);
27      x = x(2:end);
28
29      % setup the BCH Encoder and Decoder used in
            Secure Sketch Recovery
30      RepEnc=comm.BCHEncoder('CodewordLength',size(s
            ,2),'MessageLength',kk);
31      RepDec=comm.BCHDecoder('CodewordLength',size(s
            ,2),'MessageLength',kk);
32
33      % Recovery part of secure sketch
34      % mod 2 add of secure sketch helper and puf data
35      r = xor(s,w(2:end));
36      k = step(RepDec, r')';
37      r = step(RepEnc, k')';
38      w = xor(s, r);
39
40      % Randomness Extraction (get hash output and
            rearrange to row vector)
41      xw = xor(x,w);
42      R = sha256(char(binaryVectorToHex(xw)));
43  end
```

A.3 EAP-PUF Protocol Simulation in Matlab

A.3.1 GenerateRequest.m - Generates a full EAP-PUF Ethernet Packet

```
1  %% GenerateRequest
2  % This takes the data required to issue a PUF
       challenge in hexadecimal and
3  % generates an ethernet packet containing EAPoL
       EAP_PUF request packet
4  %
5  % C      - The Challenge (will be 16 bytes long in
       demo) (input)
6  % x      - Helper Data 1 (will be 16 Bytes long in
       demo) (input)
7  % s      - Helper Data 2 (will be 16 Bytes long in
       demo) (input)
8  % packet - full ethernet packet represented in
       Hexadecimal (output)
9  function packet = GenerateRequest(C,s,x)
10     % ensure challenge data are strings (not cell
          arrays) for processing
11     C = char(C);
12     x = char(x);
13     s = char(s);
14
15     % Start with the Ethernet packet header
16     destinationMAC = '00005E005301'; % reserved MAC
          addr for documentation
17     sourceMAC      = '00005E0053FF'; % reserved MAC
          addr for documentation
```

```
18      etherType       = '888E';          % type for
           IEEE802.1x (EAPoL packets)
19      packet = strcat(destinationMAC, sourceMAC,
           etherType);
20      % Append EAPoL Header
21      eapolVersion    = '02';  % always version 2
22      eapolCode       = '00';  % EAP authentication
           data is always 0
23      % nb. for length EAP_PUF Packet body should be
           75 bytes long:
24      %       5 Byte EAP Header
25      %    + 16 Byte Challenge
26      %    + 32 Bytes of Helper Data
27      %   =  53 Bytes - (which is 35 in Hex)
28      length          = '0035';     % for demo this can
           be set
29      packet = strcat(packet, eapolVersion, eapolCode,
            length);
30      % Append EAP header
31      eapCode         = '01';  % This is a EAP request
           packet; code = '1'
32      eapIdentity     = '55';  % Assumed for demo
           purposes
33      eapType         = 'B0';  % EAP_PUF type (decimal
           192)
34      packet = strcat(packet, eapCode, eapIdentity,
           length, eapType);
35      % Append challenge and helpder data
36      packet = strcat(packet, C, x, s);
37      % Finally Append the CRC and ensure output is
           string not cellarray.
38      packet = strcat(packet, crc(packet));
39      packet = char(packet);
```

```
40 end
```

A.3.2 ProcessRequest.m - Extracts Challenge from EAP-PUF Packet

```
1  %% ProcessRequest
2  % This takes a hexadecimal ethernet packet and
       extracts the required
3  % PUF challenge data from it, to be used to generate
       reponse
4  %
5  % packet - full ethernet packet represented in
       Hexadecimal
6  % C      - The Challenge (will be 16 bytes long)
7  % x      - Helper Data 1 (will be 16 Bytes long)
8  % s      - Helper Data 2 (will be 16 Bytes long)
9  function [ C,s,x ] = ProcessRequest( packet )
10     % ensure packet is a string (not cell array) for
           processing
11     packet = char(packet);
12     % Packet should be 75 bytes long:
13     %      14 Byte Ethernet Header
14     %    +  4 Byte EAPoL Header
15     %    +  5 Byte EAP Header
16     %    + 16 Byte Challenge
17     %    + 32 Bytes of Helper Data
18     %    +  4 Byte CRC32 FCS (Frame Check Sequence)
19     %   =  75 Bytes
20     % required challenge should start at the 24th
          byte and be 16 bytes long
21     % however this number needs to be doubled to
          '32' as the hex digits
```

```
22      % represent nibbles of 4-bits, not Bytes of 8-
           bits. so 32 hex digits
23      % for 128 bits of data.
24      C = packet(47:78);
25      % Helper data comes straight after and are both
           16 bytes long too
26      x = packet(79:110);
27      s = packet(111:142);
28  end
```

A.3.3 crc.m - Generates FCS of Ethernet packet

```
1   %% shift register 'like' implementation of CRC-32
       encoder for ethernet FCS
2   function output = crc(packet)
3       % generator polynomial for CRC-32
4       poly = [1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0
           1 1 1 0 1 1 0 1 1 0 1 1 1]';
5       % you can use binaryVectorToHex to get 0
           x104c11db7) to check correct
6
7       % convert the hex input into a binary vector
8       % also, it needs to be > 32 bits for this to
           work, so if input is
9       % 8 hex digits (32 bits) or less then pad with
           leading zeros
10      if (size(char(packet),2) < 9)
11          bits = hexToBinaryVector(packet,36)';
12      else
13          bits = hexToBinaryVector(packet)';
14      end
```

```
15      % Flip the first 32 bits (this is required by
            the spec...
16      bits(1:32) = 1 - bits(1:32);
17      % Add 32 zeros at the back for easier processing
18      bits = [bits; zeros(32,1)];
19      % Initialize the remainder calculator to all
            zeros
20      rem = zeros(32,1);
21
22      % Main compution loop
23      for i = 1:length(bits)
24          % like a lfsr we keep appending bits to the
                end
25          rem = [rem; bits(i)];
26          % modulo 2 addition, only need to add when
                '1' in remainder bit
27          if rem(1) == 1
28              rem = mod(rem + poly, 2);
29          end
30          % shift the register along before adding
                next bit
31          rem = rem(2:33);
32      end
33
34      % output the compliment of the remainder as the
            CRC
35      output = char(binaryVectorToHex((1 - rem)'));
36  end
```

A.4 SHA-256 hash function implementation in Matlab

A.4.1 sha256.m - Main SHA-256 Matlab Implementation code

```
% SHA-256 Implementation
% Only capable of one 512-bit block
% for purposes of integration into fuzzy extractor project

% very inefficient - works on string representations of hexadecimal numbers
% this is for the purposes of clarity of design, as speed is not an issue

function digest = sha256(message)
  % setup initial hash values
  % nb. obtained from the fractional parts of the square roots of the first
  % eight prime numbers
  H = {...
  '6a09e667', 'bb67ae85', '3c6ef372', 'a54ff53a',...
    '510e527f', '9b05688c', '1f83d9ab', '5be0cd19'};

  % the 64 constants used for K values..
  K = {...
  '428a2f98', '71374491', 'b5c0fbcf', 'e9b5dba5', '3956c25b',...
    '59f111f1', '923f82a4', 'ab1c5ed5', 'd807aa98', '12835b01',...
```

```
20      '243185be', '550c7dc3', '72be5d74', '80deb1fe',
          '9bdc06a7',...
21      'c19bf174', 'e49b69c1', 'efbe4786', '0fc19dc6',
          '240ca1cc',...
22      '2de92c6f', '4a7484aa', '5cb0a9dc', '76f988da',
          '983e5152',...
23      'a831c66d', 'b00327c8', 'bf597fc7', 'c6e00bf3',
          'd5a79147',...
24      '06ca6351', '14292967', '27b70a85', '2e1b2138',
          '4d2c6dfc',...
25      '53380d13', '650a7354', '766a0abb', '81c2c92e',
          '92722c85',...
26      'a2bfe8a1', 'a81a664b', 'c24b8b70', 'c76c51a3',
          'd192e819',...
27      'd6990624', 'f40e3585', '106aa070', '19a4c116',
          '1e376c08',...
28      '2748774c', '34b0bcb5', '391c0cb3', '4ed8aa4a',
          '5b9cca4f',...
29      '682e6ff3', '748f82ee', '78a5636f', '84c87814',
          '8cc70208',...
30      '90befffa', 'a4506ceb', 'bef9a3f7', 'c67178f2'};
31
32    % initialise registers - to initial hash values
33    a = H(1);
34    b = H(2);
35    c = H(3);
36    d = H(4);
37    e = H(5);
38    f = H(6);
39    g = H(7);
40    h = H(8);
41
42    % initialise the 16 message schedule registers
```

```
43  W = {...
44    '0000000', '0000000', '0000000', '0000000',...
45    '0000000', '0000000', '0000000', '0000000',...
46    '0000000', '0000000', '0000000', '0000000',...
47    '0000000', '0000000', '0000000', '0000000'};
48
49  % pad message to 512-bits
50  paddedmessage = padmessage(message);
51
52  % apply the compression function and update the
       registers
53  for j = 1:64
54
55    % compute message schedule
56    if j <= 16
57       % for first 16 iterations we just copy the
            message to the registers
58       % so move next 32bit chunk of message into
            the message scheduler
59       messagechunk = paddedmessage((j-1)*8+1:j*8);
60       W(17-j) = cellstr(messagechunk);
61       Wout = W(17-j);
62    else
63       % now message is in we can perform the
            scrambling functions
64
65       %get the next scrambled input to the message
            block
66       temp1 = hexadd(W(16), LittleSigma0(W(15)));
67       temp2 = hexadd(temp1, W(7));
68       temp3 = hexadd(temp2, LittleSigma1(W(2)));
69
```

```
70          % shift registers once writting in the new
               value to the vacant pos.
71          W(2:16) = W(1:15);
72          W(1)  = cellstr(temp3);
73          Wout = W(1);
74      end
75
76      % Compute compression function subfunctions
77
78      tt1 = t1(h, BigSigma1(e), Ch(e, f, g), K(j),
           Wout);
79      tt2 = t2(BigSigma0(a), Maj(a, b, c));
80
81      % perform compression function on registers
82      h = g;
83      g = f;
84      f = e;
85      e = cellstr(hexadd(d,tt1));
86      d = c;
87      c = b;
88      b = a;
89      a = cellstr(hexadd(tt1,tt2));
90  end
91
92  % compute the final hash value (only one block, so
        end here)
93
94  H(1) = cellstr(hexadd(a, H(1)));
95  H(2) = cellstr(hexadd(b, H(2)));
96  H(3) = cellstr(hexadd(c, H(3)));
97  H(4) = cellstr(hexadd(d, H(4)));
98  H(5) = cellstr(hexadd(e, H(5)));
99  H(6) = cellstr(hexadd(f, H(6)));
```

```
100   H(7) = cellstr(hexadd(g, H(7)));
101   H(8) = cellstr(hexadd(h, H(8)));
102
103
104   digest = char(strcat(H(1),H(2),H(3),H(4),H(5),H(6)
         ,H(7),H(8)));
105 end
```

A.4.2 Various files - SHA-256 Matlab Implementation Helper functions

```
% pad the message to 512-bits
% original message should be a string of hexadecimal characters
% it can be no longer than 110 hexadecimal digits long
% output will be message of 128 hexadecimal digits (4 bits per hex * 128).

% it makes assumption that the hexadecimal digits are in pairs to represent
% bytes so there must be an even number of them for function to work..

% Adds stop bit (2 hex characters) and 64 bit length code (16 chars)
% Hence 110 is maximum orignal size..
function paddedmessage = padmessage(message)
    % work out the length of the message in binary digits
    l = size(message,2)*4;

    % add end bit - assume full bytes in, so just add a hex '80' (10000000)
    message = strcat(message, '80');

    % Create the zero padding, add 2 for the end bit we just added above
    k = 112-(l/4+2);

    % create a string of zeros k digits long
```

```
    zm = blanks(k);
    zm = strrep(zm,' ','0');

    % need to append the length as a 64-bit number
       (64/4 = 16 hex digits)
    append = dec2hex(l);
    % make it long enough
    za = blanks(16 - size(append,2));
    za = strrep(za,' ','0');
    append = strcat(za,append);

    paddedmessage = strcat(message, zm, append);
end
```

```
function output = hexadd(x,y)
  x = char(x);
  y = char(y);
  result = dec2hex(mod(hex2dec(x) + hex2dec(y),2^32)
     );
  % keep the output the same size as the input
  input_size = max([size(x,2);size(y,2)]);
  diff_size = input_size - size(result,2);
  if diff_size == 0
      output = result;
  elseif diff_size > 0
      % create a string of zeros diff_size digits
         long
      pad = blanks(diff_size);
      pad = strrep(pad,' ','0');
      output = strcat(pad,result);
  else
      % remove the MSB 4 bits (like a crude overflow
         )
```

```
      output = result(abs(diff_size)+1:end);
  end
end
```

```
function output = t1(h,s1,ch,kt,wt)

  i = hex2dec(h);
  j = hex2dec(s1);
  k = hex2dec(ch);
  l = hex2dec(kt);
  m = hex2dec(wt);

  output = dec2hex(mod(i + j + k + l + m,2^32));
end
```

```
function output = t2(s0,maj)

  i = hex2dec(s0);
  j = hex2dec(maj);

  output = dec2hex(mod(i + j,2^32));
end
```

```
function output = Maj(a,b,c)
  x  = hex2dec(a);
  y  = hex2dec(b);
  z  = hex2dec(c);

  i = bitand(x,y);
  j = bitand(x,z);
  k = bitand(y,z);

  l = bitxor(i,j);
```

```
  m = bitxor(l,k);

  output = dec2hex(m);
end
```

```
function output = Ch(e,f,g)
  x  = hex2dec(e);
  nx = bitxor(x,hex2dec('ffffffff'));
  y  = hex2dec(f);
  z  = hex2dec(g);

  i = bitand(x,y);
  j = bitand(nx,z);

  k = bitxor(i,j);

  output = dec2hex(k);
end
```

```
function output = BigSigma0(a)
  x  = hex2dec(a);

  rotr2  = bitror(ufi(x,32,0), 2);
  rotr13 = bitror(ufi(x,32,0),13);
  rotr22 = bitror(ufi(x,32,0),22);

  i = bitxor(rotr2,rotr13);
  j = bitxor(i,rotr22);
  output = j.hex;
end
```

```
function output = BigSigma1(e)
  x  = hex2dec(e);
```

```
3
4   rotr6  = bitror(ufi(x,32,0), 6);
5   rotr11 = bitror(ufi(x,32,0),11);
6   rotr25 = bitror(ufi(x,32,0),25);
7
8   i = bitxor(rotr6,rotr11);
9   j = bitxor(i,rotr25);
10  output = j.hex;
11 end
```

```
1  function output = LittleSigma0(w)
2    x  = hex2dec(w);
3    z  = fi(x,0,32,0);
4
5    rotr7  = bitror(z, 7);
6    rotr18 = bitror(z,18);
7    srl3   = bitsrl(z, 3);
8
9    i = bitxor(rotr7,rotr18);
10   j = bitxor(i,srl3);
11   output = j.hex;
12 end
```

```
1  function output = LittleSigma1(w)
2    x  = hex2dec(w);
3
4    rotr17  = bitror(ufi(x,32,0),17);
5    rotr19  = bitror(ufi(x,32,0),19);
6    srl10   = bitsrl(ufi(x,32,0),10);
7
8    i = bitxor(rotr17,rotr19);
9    j = bitxor(i,srl10);
10   output = j.hex;
```

```
11 end
```

A.5 Full Project Demonstration code

A.5.1 pufdemo.m - Fuzzy Extractor Demonstration code

```
%% Demonstration Script
clear all;
clc;
fprintf('0. Ready To Start Demo\n');
pause;

%% 1. CHALLENGE GENERATION
% First Generate the challenge this is a list of 16-bit memory addresses,
% in this case 8, which amounts to just a random 128-bit number formatted
% as 32 digits of hexadecimal for the purposes of this demo.
% In practical usage care should be taken to weight the areas of memory
% queried to avoid areas of both highest and lowest metastability on the
% SRAM circuit and areas that degrade through age or temperature the worst.
C = challengegen();
% Display Current State
fprintf('1. Challenge Generated:\n');
cprintf('*blue','   %s\n',C);
pause;

%% 2. RAW PUF INTERACTION 1 (enrolment)
% Send the challenge to the FPGA over a serial communication and
```

```
22 % retrieve the PUF response (memory contents at
     addresses specified)
23 prompt = 'What is the PUF Output? ';
24 w = input(prompt,'s');
25 % Display Current State
26 fprintf('2. PUF Raw Response extracted:\n');
27 cprintf('*blue','   %s\n',w);
28 pause;
29
30 %% 3. FUZZY EXTRACTOR - GENERATION
31 % enrolment of the PUF - look into the code for it's
     design...
32 [R,s,x] = FuzzyGenerator(w);
33 % store the Response for later in the demo
34 original_response = R;
35 % Display Current State
36 fprintf('3. Fuzzy Extractor Generation Process:\n')
37 fprintf(' a) Original Response to verify against:\n'
     );
38 cprintf('*blue','   %s\n',R);
39 fprintf(' b) Helper Data to include in challenge:\n
     s=');
40 cprintf('*blue','%s\n',s);
41 fprintf(' x=');
42 cprintf('*blue','%s\n',x);
43 pause;
44
45 %% 4. SEND EAP_PUF CHALLENGE
46 % send the Challenge bits, and the two piceces of
     helper data
47 p = GenerateRequest(C,s,x);
48 % Display Current State
```

```
49 fprintf('4. Ethernet EAP_PUF Request Packet created
      :\n');
50 % long string so split it in it's component parts
      before printing out
51 cprintf('*blue','   %s...\n',p(1:28));
52 cprintf('*blue','    %s... ',p(29:36));
53 cprintf('*blue','%s...\n',p(37:46));
54 cprintf('*blue','     %s...\n',p(47:78));
55 cprintf('*blue','     %s...\n',p(79:110));
56 cprintf('*blue','     %s...\n',p(111:142));
57 cprintf('*blue','   %s\n',p(143:end));
58 pause;
59
60 %% 5. RECEIVE EAP_PUF CHALLENGE
61 % extract the CHallenge and the helper data
62 [C,s,x] = ProcessRequest(p);
63 % Display Current State
64 fprintf('5. Ethernet EAP_PUF Request Packet received
      :\n C=');
65 cprintf('*blue','%s\n',C);
66 fprintf(' s=');
67 cprintf('*blue','%s\n',s);
68 fprintf(' x=');
69 cprintf('*blue','%s\n',x);
70 pause;
71
72 %% 6. RAW PUF INTERACTION 2 (verification)
73 prompt = 'What is the PUF Output? ';
74 w = input(prompt,'s');
75 % Display Current State
76 fprintf('6. PUF Raw Response extracted:\n');
77 cprintf('*blue','   %s\n',w);
78 pause;
```

```
79
80 %% 7. FUZZY EXTRACTOR - REPRODUCTION
81 % validation of the PUF - look into the code for it'
   s design...
82 R = FuzzyReproducer(w,s,x);
83 % in practice an ethernet packet would be sent back
   ... but skip this as
84 % we have already explored and it's the same
   technique
85 reproduced_response = R;
86 % Display Current State
87 fprintf('7. Fuzzy Extractor Reproduction Process:\n'
   )
88 cprintf('*blue','   %s\n',R);
89 pause;
90
91 %% 8. VALIDATE
92 fprintf('8. ');
93 validator('demo', original_response,
   reproduced_response)
```

A.5.2 FuzzySim.m - Fuzzy Extractor error correction testing

```
%% EAP-PUF Demo (Without FPGA PUF)
%---------------------------------
% Michael Walker 2014

%% Initialization
clear all;
clc;

pufsize = 128;

% Challenge Generation - for now use the challenge only - not the raw puf
% response
w = challengegen(pufsize)

% generate some responses from an 'authentic' but degraded puf by adding a
% few random errors - default error correction should deal with 10 errors
% so try 0 5, 8, 9, 10, 11, 12 and 15 errors to test response
temp = xor(hexToBinaryVector(w,pufsize),randerr(1, pufsize,  5));
wa05 = char(binaryVectorToHex(temp))
temp = xor(hexToBinaryVector(w,pufsize),randerr(1, pufsize,  8));
wa08 = char(binaryVectorToHex(temp))
temp = xor(hexToBinaryVector(w,pufsize),randerr(1, pufsize,  9));
wa09 = char(binaryVectorToHex(temp))
```

```
temp = xor(hexToBinaryVector(w,pufsize),randerr(1,
   pufsize, 10));
wa10 = char(binaryVectorToHex(temp))
temp = xor(hexToBinaryVector(w,pufsize),randerr(1,
   pufsize, 11));
wa11 = char(binaryVectorToHex(temp))
temp = xor(hexToBinaryVector(w,pufsize),randerr(1,
   pufsize, 12));
wa12 = char(binaryVectorToHex(temp))
temp = xor(hexToBinaryVector(w,pufsize),randerr(1,
   pufsize, 15));
wa15 = char(binaryVectorToHex(temp))
% generate some responses from
wr1 = challengegen(pufsize)
wr2 = challengegen(pufsize)
wr3 = challengegen(pufsize)

%% Generation Procedure

[R,s,x] = FuzzyGenerator(w)

%% Reproduction Procedures On Authentic Device

R00 = FuzzyReproducer(w, s, x)
R05 = FuzzyReproducer(wa05, s, x)
R08 = FuzzyReproducer(wa08, s, x)
R09 = FuzzyReproducer(wa09, s, x)
R10 = FuzzyReproducer(wa10, s, x)
R11 = FuzzyReproducer(wa11, s, x)
R12 = FuzzyReproducer(wa12, s, x)
R15 = FuzzyReproducer(wa15, s, x)

```

```
52 %% Reproduction Procedure On Different Devices with
      random responses
53
54 RR1 = FuzzyReproducer(wr1, s, x)
55 RR2 = FuzzyReproducer(wr2, s, x)
56 RR3 = FuzzyReproducer(wr3, s, x)
57
58 %% Testing Response
59
60 validator('R00', R, R00);
61 validator('R05', R, R05);
62 validator('R08', R, R08);
63 validator('R09', R, R09);
64 validator('R10', R, R10);
65 validator('R11', R, R11);
66 validator('R12', R, R12);
67 validator('R15', R, R15);
68 validator('RR1', R, RR1);
69 validator('RR2', R, RR2);
70 validator('RR3', R, RR3);
```

A.5.3 FuzzyLengthTest.m - Fuzzy Extractor length parameter testing

```
%% EAP-PUF Width Tests
%---------------------
% Michael Walker 2014

%% Initialization
clear all;
clc;


% try different sizes
%--------------------
% generate integer for the code length for BCH
   encoder (allows 3..16)
m = 6;
% calc codelength
n = 2^m - 1;
% generate the possible combinations of message
   lenth for give codelength
minnums = bchnumerr(n);
% Select the middlle of the options for the actual
   message length
k = minnums(round(size(minnums,1)/2),2);
% keep note of the number of errors that will be
   corrected by the code
d = minnums(round(size(minnums,1)/2),3);
pufsize = n + 1;

% Challenge Generation - for now use the challenge
   only - not the raw puf
% response
```

```
26 w = challengegen(n + 1)
27
28 % generate some responses from an 'authentic' but
      degraded puf by adding a
29 % few random errors
30 temp = xor(hexToBinaryVector(w,pufsize),randerr(1, n
      +1,  4));
31 wa04 = char(binaryVectorToHex(temp))
32 temp = xor(hexToBinaryVector(w,pufsize),randerr(1, n
      +1,  5));
33 wa05 = char(binaryVectorToHex(temp))
34 temp = xor(hexToBinaryVector(w,pufsize),randerr(1, n
      +1,  6));
35 wa06 = char(binaryVectorToHex(temp))
36 temp = xor(hexToBinaryVector(w,pufsize),randerr(1, n
      +1,  7));
37 wa07 = char(binaryVectorToHex(temp))
38
39 % generate some responses from
40 wr1 = challengegen(n+1)
41 wr2 = challengegen(n+1)
42 wr3 = challengegen(n+1)
43
44 %% Generation Procedure
45
46 [R,s,x] = FuzzyGenerator(w, k, m)
47
48 %% Reproduction Procedures On Authentic Device
49
50 R00 = FuzzyReproducer(   w, s, x, k)
51 R04 = FuzzyReproducer(wa04, s, x, k)
52 R05 = FuzzyReproducer(wa05, s, x, k)
53 R06 = FuzzyReproducer(wa06, s, x, k)
```

```
54 R07 = FuzzyReproducer(wa07, s, x, k)
55
56 %% Reproduction Procedure On Different Devices with
      random responses
57
58 RR1 = FuzzyReproducer(wr1, s, x, k)
59 RR2 = FuzzyReproducer(wr2, s, x, k)
60 RR3 = FuzzyReproducer(wr3, s, x, k)
61
62 %% Testing Response
63
64 validator('R00', R, R00);
65 validator('R04', R, R04);
66 validator('R05', R, R05);
67 validator('R06', R, R06);
68 validator('R07', R, R07);
69 validator('RR1', R, RR1);
70 validator('RR2', R, RR2);
71 validator('RR3', R, RR3);
```

A.5.4 challengegen.m - Generates random hexadecimal challenge string

```
function hexoutput = challengegen(n)
% CHALLENGEGEN generates a random PUF challenge of hex characters
% optional parameter 'n' sets the bit size of the challenge and therefore
% the equally sized PUF reponse. it defaults to 128 bits.
    switch nargin
        case 1
            nn = n;
        otherwise
            nn = 128;
    end

    hexoutput = char(binaryVectorToHex(randi([0 1],1,nn)));
end
```

A.5.5 Validator.m - Validates two Responses

```
%% Validator
% Checks that two responses (SHA-256 digests in hex) are identical.
% - R_orig  : Response from the FuzzyGenerator
% - R_repro : Response from the FuzzyReproducer
% - name    : Device name under test (i.e. where the response of the
%             FuzzyReproducer comes from)
function validator(name, R_orig, R_repro)
```

```
8       RH1 = hexToBinaryVector(R_orig,size(R_orig,2)*4)
            ;
9       RH2 = hexToBinaryVector(R_repro,size(R_orig,2)
            *4);
10
11    Dist = pdist([+RH1; +RH2],'hamming');
12
13      if Dist == 0
14          message = sprintf('The %s device passed
                verification',name);
15          disp(message);
16      else
17          message = sprintf('The %s device failed
                verification, distance: %.1f%%',name,Dist
                *100);
18          disp(message);
19      end
20  end
```