School of Electrical & Electronic Engineering

EEE8097 : Dissertation

Physically Unclonable Functions for Networked Device Authentication

---

Author:

Michael Walker (130623935)

Supervisor:

Dr. Alex Bystrov

September 22, 2014

Word Count: 7996

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 22, 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

List of Tables

Chapter 1

Introduction

Physically unclonable functions (PUFs) are a relatively recent concept under much research. By harnessing physical properties that are hard to simulate or copy, yet are easy to sample from as a data stream source of for cryptographic key information, an identification and authentication package can be constructed for any device that contains or incorporates that physical property. 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 into the implementation of a PUF system 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. A naive setup for this would involve the extraction of data from a physical device with the properties of a PUF in response to a challenge specifiying the parameters used for extraction, all data sent in plaintext via a standard network protocol

This first step provides a simply challenge response protocol (CRP) by which the device can provide identity information, but as has been found in the course of the project, this is not enough for a complete authentication solution for two reasons. Firstly both the challenge and the reponse are in plaintext and thus 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 nature of PUFs means they provide their data somewhat unrealiably, this must be allowed for and mitigated. The concept of fuzzy extraction shall therefore be introduced and explored as a means to provide 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 device with the PUF authenticates itself over a network to some central 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 that the device requires is in some way withheld by the authenticator. By this means network security can be achieved without any human intervention. This would surely be a important and neccesary step in facilitating the likely future integration of innumerable inexpensive embeddded systems into wireless home and office networks. This is because, without burdening the user with any increased installation inconvience adaquate security provision can be provided. The use of PUFs in this provision by the estabilshment of a network security protocol was also investigated.

In the next chapter (chapter 2) I will report my background research into the subjects of PUFs and the use of static random-access memory (SRAM) as a PUF source. Then the 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 the third chapter 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 intital research. These designs broadly fit into three areas of design effort.

- An implementation of a RS-232 based CRP 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.

In the forth chapter the results of the practical implementations of the designs are presented, followed in the fifth chapter by a discussion of the implications of this study.

Chapter 2

Background Research

The intention of this project is to investigate and impliment in whole - or in part - a prototypical networed device that can be authenticated using a PUF. As explained in the Introduction there are three areas under investigation. Below are the research findings within each of these areas which supported system design that will be discussed in the next chapter.

2.1 PUFs & SRAM

The first thing required for a PUF based authentication system is obviously, a PUF. Introduced in 2002 [1], PUFs can be built from a wide variety of technologies, some have electronic construction, some not and new designs seems 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.

SRAM PUFs that use the meta-stability of the six transistor SRAM cell (see **??**) are of the former, were initially proposed by Guadardo [2] 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

[1]such as subtle variances in the transistors size, tiny changes in performance charac-

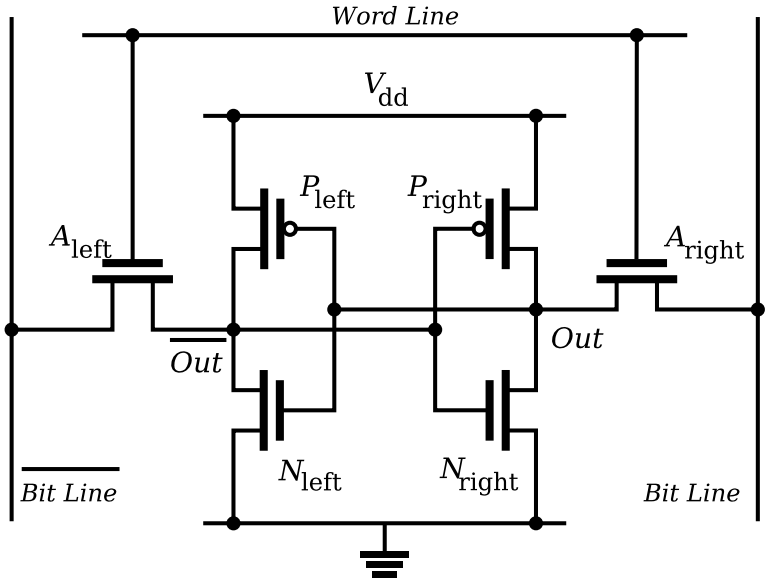

Figure 2.1: Six transistor SRAM diagram

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.

To construct a 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 access from outside. In a practical PUF core of an embedded device the former may be easier to implement, but the later 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 FPGA, the acquisition of a separate SRAM device is required as a basic FPGAs chip itself is unlikely to have the memory quantity required. A device that does not have any 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 to be desired.

teristics due to silicon impurities, or any other of minute variance in any of a thousand manufacturing steps

Unfortunately, in the case of many FPGA development boards, such as the Altera DE-1 [3] 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, however the simplicity of the interface and it easy availability make it a good choice for preliminary work such as that carried out in this project.

2.2 Fuzzy Extractors

2.2.1 Introduction

The way 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. This can be naively solved by accepting some threshold of errors in the response of the PUF with respect any particular challenge. However, this is not efficient and severely limits the cryptographic security of the device. It would be far preferable to be able to have some sort of *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.

As it turns out, such a wrapper exists. This is called a **Fuzzy Extractor** which was first proposed by Dodis et al [4] for handling biometric data for cryptographic situations. In laymans terms, this involves three separate processes. Firstly, and most essentially, a *secure sketch* which extracts and recovers stable cryptographic key data in a noise tolerant way. Secondly, a *privacy amplification* process is implemented through a *randomness extractor*. In more specific terms, this process 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 processes of a Fuzzy extractor, it is at least necessary to implement one further component; a source of randomness. This is needed by both others in the generation stage.

These components can be implemented in a variety of ways, but generally the secure sketch requires the implementation of some kind of ECC. The randomness extractor requires the implementation of a cryptographic hash function. and the source of randomness needs to be practically non-deterministic.

2.2.1.1 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, true random number generators (TRNGs) and pseudo-random number generators (PRNGs). The difference being that true randomness requires sampling from a noisy physical process and are (in theory) completely unpredictable. Pseudo random number generators are deterministic, and as such can be considered a mathematical function which can be replicated by an attacker. Thus, it is far more secure to use a true random number generator if possible.

On an FPGA, a relatively easy way to implement a TRNG is by the sampling of one internal/external clock of the device at periods controlled by an independent clock. Due to the slight variances in each clocks rate (based on their individual underlying physical implementations) it is undeterminable whether a clock will be high or low when it is sampled.

Implementing this on a FPGA was attempted but not completed, and so a MATLAB© simulation was employed for this purpose.

2.2.2 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, especially 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 convolutional 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, Implementation of complimentary BCH Encoder and Decoder in

FPGA was looked at, but for reasons of expediency this was not completed, and a MATLAB© simulation was used.

2.2.3 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 it 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 (collision resistant) and finally it is practically impossible to generate the required input data for 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. Rigorously assessing the cryptographic security of a hash deceptively difficult and beyond the scope of this project. Therefore, implementation an original hashing scheme would be ill-advised. It is far better to tailor an existing and well-understood and tested scheme for the purpose.

The standard cryptographic hash function set called SHA-2. secure has algorithm (version 2) (SHA-2) was designed by the national Security Agency (NSA) in 2001. 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, but for easy implementation in VHDL the

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

2.3 PUF Enhanced Network Authentication Protocols

In providing a PUF authentication mechanism thought should be first given to where in the protocol stack would be most effective. For an embedded device a high layer such as the applications or session layers of the OSI model would seem both inefficient and unnecessary. Going done the stack, it would also seem that establishment of the transport and network levels presupposes the device has already obtained access to a point-to-point connection or local area network (LAN). This would suggest 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 point to point protocol (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 reling on PUFs it would seem advisable to look a the multi-node protocols, specifically those for wireless networking. However in attempting to keep the project focused it would be preferable to investigate procotols in a simplier setting. Thus a compromise of investigation of modifications to the Ethernet protocol was reached.

Any modern link-layer network authentication system that is intended to replace or improve upon current standards in the most part necessitates adhering to much the same high security methods utilized in modern cryptographic protocols such as wi-fi protected access (WPA) and more recently wi-fi protected access II (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 glspppoe all support the use of .. for security

provision at the data link layer employ the extensible authentication protocol (EAP) framework. The underlying authentication methods is encapsulated by EAP which provides for the safe transfer of any parameters and secret key information for the method itself.

EAP can be utilised over Ethernet using the ISO/IEC/IEEE standard 802.1X [5] and 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 knwon as /emphsupplicants are not allowed to send Ethernet packets of any other EtherType, including general ones such as `0x0800` IPv4 Packets. Thus a attacking device can do no harm to any other device on the network - every attempt to communicate will be dropped until authenticated.

In the case of Ethernet

Paramount to the issue of running out of keying data for a protocol given the ultimately finite nature of SRAM memory and therefore the finite number of challenges that can be made without repeating the data - a fundemental issue in cryptpography.

In utilizing a PUF in

As the digest output of hash function is the ultimate response of the device to a challenge this means that the response size is 256-bits or 32 octets. This is half the minimum required packet payload size.

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, without which a Fuzzy Extractor could not be implemented, thus both the generator and reproduction implementations of the Fuzzy Extractor in it's entirety were not implemented in FPGA.

The Ethernet protocol is also comparatively complex and so modification of this for the purposes of devices authentication was also removed from the design. Instead a study of possible protocol schemes was made 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 is something that was felt possible for full implementation in FPGA, due to the easy availability of multiple DE-1 development boards, each with integrated SRAM chips. To extract 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 of 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 most simple 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 it's programs.

3.2 SRAM-PUF interface implementation on FPGA

3.2.1 SRAM

The DE-1 Development Board contains a IS61LV25616 chip. This is contains 2^{22} SRAM cells organised into 2^{18} words of 16-bits each. This should be a 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. To access the chip as a PUF we need only to concern ourselves with the read cycle of the device. 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 ($\overline{UB}$). While 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 utilised 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 be sent via the rs-232 link in two transmissions. In a similar way the 16-bit address will 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 [**?**], 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 signal active. After some delay greater than 10ns the upper byte can be 'forwarded' and the Upper and Lower Byte Access signals toggled. After at least another identical delay and if a strobe has been recieved from the communication module to say the previous byte has been sent the lower byte can be forwarded. Thus ew have a system for converting challenges in the form of memory addreses into responses in the form of memory data.

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

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

bit rates supported range from 75 to 115,200 bits/s.

The RS-232 protocol uses binary signalling. 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 transmittion 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 fix 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. ¡¡TODO ADD DIAGRAM OF RS-232 PROTOCOL¿¿ While RS-232 can utilise extra handshaking signals, for flow control, however these are not required and so are not implemented.

The design of a universal asyncronous receiver/transmitter (UART) is split into two distinct halves; the transmitter and the receiver. Each operates somewhat independantly.

3.2.2.1 Transmitter

The PISO Shift Register is a component of this half of the UART. Transmission

3.2.2.2 Receiver

3.3 Fuzzy Extractor in MATLAB

3.3.1 Introduction

The fuzzy extractor has two procedures, the Generation procedure and the Reproduction procedure. The generation procedure 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 manu-

facturing of the device at the factory. The Reproduction procedure 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 procedure. These are different because in the generation procedure, a response and helper data must be generated. Whereas, in the reproduction procedure, that previously generated helper data is now required to allow for later differences in the raw responses of the PUF. In both cases, a randomness extractor consisting of a cryptographic hash function is used to generate a Privacy Amplified response. In generation, raw input from the PUF and random data is captured from a true random number generator of the same length are combined through an XOR operator before being applied as the input to an implementation of the SHA-256 cryptographic hash function which outputs the response. In reproduction the input to the randomness extractor needs to be generated through the secure sketch first but is still applied to the hash function to output a response. The random data used must also be stored unaltered as the first of two pieces of helper data (let us call this x).

The core difference between the procedures 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.

More accurately, the sketch 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 true random number generators (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'll call n is passed into a BCH-encoder sub-component that therefore generates a redundant encoding of that random data well call m which can later be error-corrected during the recovery stage. This is then XORed with the PUF data to create the second of the two pieces of helper data. 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 true random number generators (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), this we call x and this is the same size as the first helper data. This process securely generates helper data from the raw PUF (and therefore noisy) data by which the second procedure, 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 which, to reiterate, is likely to be different from the first time. The second is the s helper data. By XORing them together, and then using that as the input to a BCH-Decoder, we can use the properties of forward error correction to duplicate the original random number generator used in the generation procedure. Thus, we have recovered (error-free) the data used in generating the response. 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. When XORed with 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. Given the same message, the hash function must produce the same digest, thus the response in the reproduction procedure will be the same as in the generation procedure even with differences in raw PUF data, yet the security of that PUF data is cryptographically assured and thus authentication of the PUF is performed securely. Even though helper data 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 message cannot be retrieved from it.

3.3.2 Components Required by Our Fuzzy Extractor

To implement our complete Fuzzy Extractor in VHDL the following sub-components must be created, correctly test benched 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.

Furthermore, assuming that the output of the PUF is a block of data of length w and the SHA-256 hash function is used the length w is somewhat determined, because it is used as an input of the hash in the generation procedure. Firstly, for reasons of simplicity and efficiency in the SHA-256 implementation a PUF output block size w that is a power of 2 should be used. For purposes of cryptographic strength (to be looked into further in the section on Cryptographic Hashing) the length of the message w should also be approximately similar in length to the hash. This limits the reasonable choices for the size of PUF data blocks to be either 128, 256 or 512 bits in length, in general however 512 bits is the operational size used for the message block in SHA-256, and so the smaller values would simply introduce 4 or 2 repeats of the message data to the hash functional element. The length of the PUF block w then, in turn, sets the required length of the

other inputs and outputs of the secure sketch. As mentioned above, the output of the BCH Encoder and the Hash-RNG need to be the same size as w. The Sketch-RNG is required to output a block of a length shorter than that of the BCH Encoder output. This is because extra data will be added for error-correction purposes. The exact size of the Sketch-RNG output is to be determined through the requirements of the BCH encoder implementation.

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 on the original message one block at a time, an alternate scheme is to use convolution codes such as the Viterbi which function continuously on streams of data, however for the purposes of the fuzzy extractor this simply adds extra complexity and 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. A 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 that 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 variste Galois. The order of the field is the number of elements is contain, 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 6 conditions:

Closure

For all operations on two operands that are elements of the finite field the result is also an element of the field (i.e. c=a+b where a,b,c ?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 * 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 * 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 which 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

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.

However, adaquate 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 underwhich a BCH encoded message could be generated. As the memory system employed consisted of 16-words and the coded output of the BCH encoder is XOR'ed together with that ouput, it would be prefereable to use BCH code lengths that are the same size. However,

n	k	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

efficient BCH code lengths are somewhat set by the properties of the cyclic codes used to distances for BCH code code length ('n') that have the form $2^m - 1$, where m is an integer and Matlab computations limitations limit this to;

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

Thus, considering anything below 16-bits is wasteful as that is the minimum memory data retrieved, 31, 63, 127, 255, 511 and 1023 would seem to be the choices available for efficient code length. as the corresponding outputs from the PUF will necessarily be multiples of 16, one bit of the puf will need to be 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 but 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 lengthd of 63, 127 and 255 are shown in table...

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

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. there is a guarantee 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 practise 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 practically - if not theoretically - 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. For the SHA-2 hash family - to be implemented in the project - a brute force attack can only be done in exponential time. 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 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 an second message is found with the same digest as a set first message. Again this form of attack seems 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. However 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 [6].

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 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 is 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 behind the design other than that it jumbles well.

The SHA-2 family use 6 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 main part of the function one at a time in a queued fashion, each feed occurring during a round of operation, but 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 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. Also, normally the function is used to hash large messages that are split into multiple blocks, but in our implementation only one 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 internal implementation of a hash function can take many forms, however in our case it must be through the use of linear feedback shift registers (LFSR) throughout the design.

3.6 True Random Number Generation on FPGA

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. We can utilise the clock drift as a source of randomness. 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 random number generators used need only to generate a few hundred bits of random data

Figure 3.1: SHA-256 Block Diagram

for each Challenge//Response Pair. 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.

3.7 Modified Ethernet Authentication Protocol Design

Chapter 4

Results

4.1 PUF

4.2 Fuzzy Extractor

4.3 SHA-256

4.4 Ethernet Authentication

4.5 Overall System & it's Implications

Chapter 5

Conclusion

This is the conclusion chapter, it should be approximately 750 Words

Chapter 6

References

[1] 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

[2] 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

[3] 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

[4] 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.

[5] 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.

[6] 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.

References generated using BibTeX

Appendix A

Appendix

A.1 Fuzzy Extractor Matlab Code

```
%% Fuzzy Extractor Demo
%----------------------
% Michael Walker 2014

%% Initialization

% this is the test data - small change is OK, big
   change unacceptable
original_PUF_txt1 = 'thisisgooddata';
authntic_PUF_txt1 = 'thisasgooddata';
attacker_PUF_txt1 = 'veryevildatais';

% alternative data - attacker passes
original_PUF_txt2 = 'ismoregooddata';
authntic_PUF_txt2 = 'ismoreguoddata';
attacker_PUF_txt2 = 'ismoreevildata';

% Create binary data vector to use in the fuzzy
   extractor
```

```
% organize data as a binary value row vector and pad
    to 127 bits
original_PUF_data1 = [str2num(reshape(dec2bin(
   original_PUF_txt1)',[],1))' zeros(1,30)];
authntic_PUF_data1 = [str2num(reshape(dec2bin(
   authntic_PUF_txt1)',[],1))' zeros(1,30)];
attacker_PUF_data1 = [str2num(reshape(dec2bin(
   attacker_PUF_txt1)',[],1))' zeros(1,30)];

original_PUF_data2 = [str2num(reshape(dec2bin(
   original_PUF_txt2)',[],1))' zeros(1,30)];
authntic_PUF_data2 = [str2num(reshape(dec2bin(
   authntic_PUF_txt2)',[],1))' zeros(1,30)];
attacker_PUF_data2 = [str2num(reshape(dec2bin(
   attacker_PUF_txt2)',[],1))' zeros(1,30)];

% Create BCH Encoder and Decoders for the fuzzy
   extractor process
GenEnc = comm.BCHEncoder('CodewordLength',127,'
   MessageLength',64);
RepEnc = comm.BCHEncoder('CodewordLength',127,'
   MessageLength',64);
RepDec = comm.BCHDecoder('CodewordLength',127,'
   MessageLength',64);

%% Generation Procedure
w = original_PUF_data1;
% Produce a random 64 bit binary number for secure
   sketch encoder
k = randi([0 1],1,64);
% Produce a random 127 bit binary number for
   randomness extractor
x = randi([0 1],1,127);
```

```
38
39 % Sketch part of secure sketch
40 r = step(GenEnc, k')';
41 s = xor(r,w);
42
43 % Randomness Extraction (get hash output and
   rearrange to row vector)
44 xw = xor(x,w);
45 hash_result = hex2dec(sha256(binaryVectorToHex(xw)))
   ;
46 original_Response = str2num(reshape(dec2bin(
   hash_result,256)',[],1))';
47
48 %% Reproduction Procedure On Authentic Device
49
50 % Get PUF value (assume slight change in data)
51 w_dash = authntic_PUF_data1;
52
53 % Recovery part of secure sketch
54 r_dash = xor(s,w_dash);
55 k_repro = step(RepDec, r_dash')';
56 r_repro = step(RepEnc, k_repro')';
57 w_repro = xor(s, r_repro);
58
59 % Randomness Extraction (get hash output and
   rearrange to row vector)
60 xw_repro = xor(x,w_repro);
61 hash_result_r = hex2dec(sha256(binaryVectorToHex(
   xw_repro)));
62 reproduced_Response = str2num(reshape(dec2bin(
   hash_result_r,256)',[],1))';
63
64 %% Reproduction Procedure On Atacking Device
```

```
65
66 % Get PUF value (assume slight change in data)
67 w_dash_a = attacker_PUF_data2;
68
69 % Recovery part of secure sketch
70 r_dash_a = xor(s,w_dash_a);
71 k_repro_a = step(RepDec, r_dash_a')';
72 r_repro_a = step(RepEnc, k_repro_a')';
73 w_repro_a = xor(s, r_repro_a);
74
75 % Randomness Extraction (get hash output and
      rearrange to row vector)
76 xw_repro_a = xor(x,w_repro_a);
77 hash_result_a = hex2dec(sha256(binaryVectorToHex(
      xw_repro_a)));
78 reproduced_Response_a = str2num(reshape(dec2bin(
      hash_result_a,256)',[],1))';
79
80 %% Testing Response
81
82 AuthDist = pdist([+original_Response;+
      reproduced_Response],'hamming');
83 AttkDist = pdist([+original_Response;+
      reproduced_Response_a],'hamming');
84
85 if AuthDist == 0
86     disp('Authentic Device Passed Verification');
87 else
88     disp('Authentic Device Failed Verification');
89 end
90
91 if AttkDist == 0
92     disp('Attackers Device Passed Verification');
```

```
93 else
94     disp('Attackers Device Failed Verification');
95 end
```

A.2 SHA-256 Matlab Implementation

```
% 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',...
    '243185be', '550c7dc3', '72be5d74', '80deb1fe', '9bdc06a7',...
    'c19bf174', 'e49b69c1', 'efbe4786', '0fc19dc6', '240ca1cc',...
```

```
    '2de92c6f', '4a7484aa', '5cb0a9dc', '76f988da',
      '983e5152',...
    'a831c66d', 'b00327c8', 'bf597fc7', 'c6e00bf3',
      'd5a79147',...
    '06ca6351', '14292967', '27b70a85', '2e1b2138',
      '4d2c6dfc',...
    '53380d13', '650a7354', '766a0abb', '81c2c92e',
      '92722c85',...
    'a2bfe8a1', 'a81a664b', 'c24b8b70', 'c76c51a3',
      'd192e819',...
    'd6990624', 'f40e3585', '106aa070', '19a4c116',
      '1e376c08',...
    '2748774c', '34b0bcb5', '391c0cb3', '4ed8aa4a',
      '5b9cca4f',...
    '682e6ff3', '748f82ee', '78a5636f', '84c87814',
      '8cc70208',...
    '90befffa', 'a4506ceb', 'bef9a3f7', 'c67178f2'};

% initialise registers - to initial hash values
a = H(1);
b = H(2);
c = H(3);
d = H(4);
e = H(5);
f = H(6);
g = H(7);
h = H(8);

% initialise the 16 message schedule registers
W = {...
    '0000000', '0000000', '0000000', '0000000',...
    '0000000', '0000000', '0000000', '0000000',...
    '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
```

```
1  % pad the message to 512-bits
2  % original message should be a string of hexadecimal
       characters
3  % it can be no longer than 110 hexadecimal digits
      long
4  % output will be message of 128 hexadecimal digits
      (4 bits per hex * 128).
5
6  % it makes assumption that the hexadecimal digits
      are in pairs to represent
7  % bytes so there must be an even number of them for
      function to work..
8
9  % Adds stop bit (2 hex characters) and 64 bit length
       code (16 chars)
10 % Hence 110 is maximum orignal size..
11 function paddedmessage = padmessage(message)
12     % work out the length of the message in binary
          digits
13     l = size(message,2)*4;
14
15     % add end bit - assume full bytes in, so just
          add a hex '80' (10000000)
16     message = strcat(message, '80');
17
18     % Create the zero padding, add 2 for the end bit
           we just added above
19     k = 112-(l/4+2);
20
```

```
    % 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
```

```
16        % remove the MSB 4 bits (like a crude overflow
              )
17        output = result(abs(diff_size)+1:end);
18    end
19  end
```

```
1   function output = t1(h,s1,ch,kt,wt)
2
3     i = hex2dec(h);
4     j = hex2dec(s1);
5     k = hex2dec(ch);
6     l = hex2dec(kt);
7     m = hex2dec(wt);
8
9     output = dec2hex(mod(i + j + k + l + m,2^32));
10  end
```

```
1   function output = t2(s0,maj)
2
3     i = hex2dec(s0);
4     j = hex2dec(maj);
5
6     output = dec2hex(mod(i + j,2^32));
7   end
```

```
1   function output = Maj(a,b,c)
2     x  = hex2dec(a);
3     y  = hex2dec(b);
4     z  = hex2dec(c);
5
6     i = bitand(x,y);
7     j = bitand(x,z);
8     k = bitand(y,z);
```

```
9
10    l = bitxor(i,j);
11    m = bitxor(l,k);
12
13    output = dec2hex(m);
14  end
```

```
1   function output = Ch(e,f,g)
2     x  = hex2dec(e);
3     nx = bitxor(x,hex2dec('ffffffff'));
4     y  = hex2dec(f);
5     z  = hex2dec(g);
6
7     i = bitand(x,y);
8     j = bitand(nx,z);
9
10    k = bitxor(i,j);
11
12    output = dec2hex(k);
13  end
```

```
1   function output = BigSigma0(a)
2     x  = hex2dec(a);
3
4     rotr2  = bitror(ufi(x,32,0), 2);
5     rotr13 = bitror(ufi(x,32,0),13);
6     rotr22 = bitror(ufi(x,32,0),22);
7
8     i = bitxor(rotr2,rotr13);
9     j = bitxor(i,rotr22);
10    output = j.hex;
11  end
```

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

  rotr6  = bitror(ufi(x,32,0), 6);
  rotr11 = bitror(ufi(x,32,0),11);
  rotr25 = bitror(ufi(x,32,0),25);

  i = bitxor(rotr6,rotr11);
  j = bitxor(i,rotr25);
  output = j.hex;
end
```

```
function output = LittleSigma0(w)
  x  = hex2dec(w);
  z  = fi(x,0,32,0);

  rotr7  = bitror(z, 7);
  rotr18 = bitror(z,18);
  srl3   = bitsrl(z, 3);

  i = bitxor(rotr7,rotr18);
  j = bitxor(i,srl3);
  output = j.hex;
end
```

```
function output = LittleSigma1(w)
  x  = hex2dec(w);

  rotr17  = bitror(ufi(x,32,0),17);
  rotr19  = bitror(ufi(x,32,0),19);
  srl10   = bitsrl(ufi(x,32,0),10);

  i = bitxor(rotr17,rotr19);
```

```
9   j = bitxor(i,srl10);
10  output = j.hex;
11 end
```

A.3 Appendix III

Third Appendix, probably some source code...

A.4 Appendix IV

Fourth Appendix, probably some source code...

A.5 Appendix V

Fifth Appendix, probably some source code...