A Key Management Architecture for Securing Off-Chip Data Transfers on an FPGA

Jonathan Peter Graf

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Peter M. Athanas, Co-Chair
Dr. Mark T. Jones, Co-Chair
Dr. Joseph G. Tront

June 18, 2004
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, Key Management, Security, Encryption, Amanuet

A Key Management Architecture for Securing Off-Chip Data Transfers on an FPGA

Jonathan Peter Graf

Abstract

Data security is becoming ever more important in embedded and portable electronic devices. The sophistication of the analysis techniques used by attackers is amazingly advanced. Digital devices' external interfaces to memory and communications interfaces to other digital devices are vulnerable to malicious probing and examination. A hostile observer might be able to glean important details of a device's design from such an interface analysis. Defensive measures for protecting a device must therefore be even more sophisticated and robust.

This thesis presents an architecture that acts as a secure wrapper around an embedded application on a Field Programmable Gate Array (FPGA). The architecture includes functional units that serve to authenticate a user over a secure serial interface, create a key with multiple layers of security, and encrypt an external memory interface using that key. In this way, the wrapper protects all of the digital interfaces of the embedded application from external analysis. Cryptographic methods built into the system include an RSA-related secure key exchange, the Secure Hash Algorithm, a certificate storage system, and the Data Encryption Standard algorithm in counter mode. The principles behind the encrypted external memory interface and the secure authentication interface can be adjusted as needed to form a secure wrapper for a wide variety of embedded FPGA applications.

This work is dedicated to my mother, who invested ten years of her life to educate me at home. This thesis represents a culmination of her efforts as much as mine.

Acknowledgements

Thanks to my fiancée, Leanna, for patiently loving me throughout the trials and rigors of graduate study. Without her support and love, I would certainly be nothing more than a stressed-out, data-crunching lab zombie. With it, I am a data-crunching lab zombie who is looking forward to marrying the woman of his dreams.

Thanks to Dr. Athanas, Dr. Jones, Dr. Tront, and Dr. Patterson for giving advice and guidance and for always believing I was capable of things I wasn't entirely convinced I could do.

Thanks to my father, mother, and brother, who are all graduating with master's degrees in the same weekend with me, for unfailingly supporting of my aspirations and for providing a bit of friendly academic competition.

Thanks to all my friends at the Configurable Computing Laboratory for creating such an incredible, fun research environment.

Thanks to God, the giver of all good gifts, including FPGAs.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In today's world of advanced security cracking techniques, it is difficult to secure a digital device against unauthorized use or tampering. Companies and governments wishing to deploy digital hardware into situations where the device may be subject to malicious analysis risk losing secret algorithmic and functional information to competitive or hostile entities. There is a growing need for devices that are capable not only of authenticating a user to the device but also of masking the device function through cryptographic techniques. For example, a military unit may want to deploy a portable digital device running a secret algorithm into hostile territory without revealing the nature of the algorithm in the event the device is captured. Since the loss of a cryptographic key could compromise even the best of these devices, key management is integral to and arguably the most important aspect of any cryptographically secured system.

Attacks used to gain information from a digital device are commonly carried out on the device's external data interfaces. For example, analysis of a device's authentication interface could lead to unauthorized system use by revealing secret authentication information to a malicious user. If a logic analyzer were placed on the address and data busses of an embedded device's external memory, the algorithmic function of the device could easily be compromised. Similar information about the device function could be gleaned from other

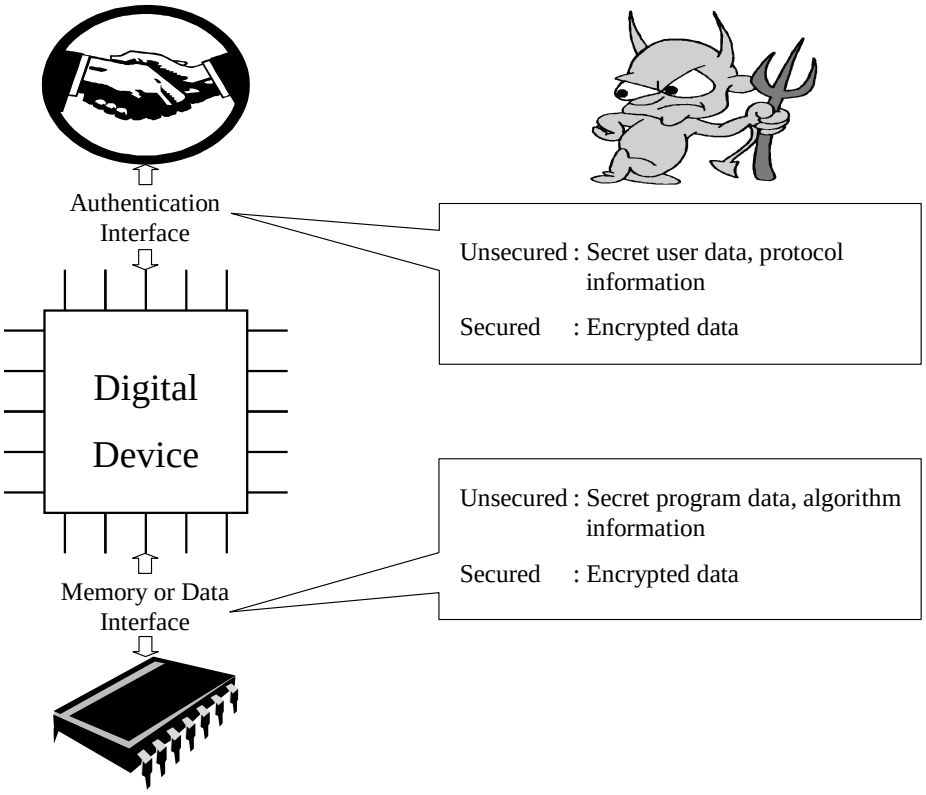

Figure 1.1: Exposed interfaces and the information they reveal

insecure data interfaces. A vulnerability of this nature exposes the digital device to the possibility of complete reverse engineering. To avoid this risk, a device is needed that enables a key management scheme encompassing both secure authentication and a cryptographically secure data transfer system.

Figure 1.1 depicts a digital device with an authentication interface and an interface to an external memory or other data device. The malicious user, represented by the demon, can learn secret user authentication data and the authentication protocol from an exposed authentication interface on an unsecured device. The unsecured device will also reveal secret

program data and information about the internal device algorithm through the external memory or data interface. In each case, the malicious user would only see encrypted data on a device with a secured authentication and memory system.

1.2 The Amanuet Architecture

There are compelling reasons for choosing a Field Programmable Gate Array (FPGA) as the platform for implementing an architecture to address these security concerns. Within a single FPGA, the user authentication system, the external data transfer controller, and every necessary cryptographic algorithm can be realized. Whereas an ASIC-based implementation would leave vulnerable the physical details of the secure architecture, the only physical architecture that can be analyzed on an FPGA is the standard layout of slices, RAMs, and interconnects that remains the same for any design realized in it. FPGAs are physically just generic arrays of logic that are programmed to a specific function by a bitstream. The bitstream defines the functions that the logic array will form and the interconnect between the functions. Those functions can range from simple decoders and bit-shifting operations to complex embedded processors and encryption algorithms.

Using an FPGA also makes this secure architecture easily applicable to existing insecure digital applications. The architecture can be envisioned as a wrapper around an embedded FPGA-based application or CPU. This wrapper creates a secure user authentication interface and cryptographically secures all of the embedded application's data transfer interfaces, effectively rendering the FPGA a black box capable of performing the task for which it was designed without betraying its internal methods or software to a hostile entity. The wrapper renders logic analysis of the authentication or data transfer interfaces ineffective as they would reveal only encrypted information.

This thesis presents an FPGA-based key management architecture that implements cryptographically secured interfaces for user authentication and external memory. This architecture has been named *Amanuet* – meaning "hidden one" – after the Egyptian goddess whose shadow was looked upon as a symbol of protection [1]. This architecture hides the true

nature of the embedded application or CPU that is protected within its shadow (wrapper)[1].

A proof-of-concept design for the Amanuet architecture has been implemented using the Celoxica RC1000 development platform [2]. This prototype utilizes the key management system to secure a memory port on the FPGA, protecting the contents from discovery. Additionally, a Dallas Semiconductor Java-Powered iButton [3] is used as a secure portable token that can authenticate a user to the system by establishing a secure channel for conveying user identification data from the iButton's memory to the FPGA. On the FPGA, an Authentication Control Unit (ACU) establishes the other side of the secure channel. The ACU accepts the incoming user ID, checks its signature against a known list of authorized users, and either accepts or rejects the user's session on the device. The user ID is then used to create a unique session key. This key is applied to the Data Encryption Standard (DES) algorithm [4] to secure the external memory interface. Further methods are described for uniquely applying the DES algorithm and accelerating its use in this particular encrypted memory controller (EMC).

1.3 Application Example

An immediate application for the methods presented here would be in intelligence equipment utilized throughout the world by the United States military. Surveillance equipment and secret devices are currently deployed in or near foreign nations whose governments would benefit technologically from the opportunity to analyze U.S. designs. This equipment is at risk of accidental loss or capture.

For example, on April 1, 2001, a Chinese F-8 fighter jet sent on an intercept mission collided with a U.S. Navy EP-3E [2] surveillance plane that was performing routine operations in the South China Sea [6]. As the F-8 slowed to match the 180 knot speed of the EP-3, it rapidly decelerated too close to its stall speed, causing the plane to become almost unmaneuverable. After performing two close-pass maneuvers within 3-5 feet of the autopiloting EP-3, the F-8

[1] It's quite difficult to come up with a technology name that has not been used before and that has a meaning that encompasses the function of the device.

[2] Image in Figure 1.2 made public by U.S. Navy [5]

Figure 1.2: U.S. Navy EP-3E

came in for a third pass. This time, the F-8's right wing clipped one of the EP-3's four propellers, which then forced the F-8's tailfin into the EP-3's port aileron. The result was to plunge the F-8 into a fatal dive and to throw the EP-3 into a near-inverted rapid-descent snap roll towards the ocean 22,000 feet below. The EP-3's pilot was able to arrest the descent at 8,000 feet. The plane was 600 nautical miles from the nearest allied airbase, the destroyed propeller was causing violent vibrations in the plane, and ditching in the ocean would have certainly killed some of the crew.

The decision was made to land the plane at a Chinese airfield on Hainan Island. The U.S. crew of 24 was held for 11 days [7] while the Chinese sent technology experts throughout the plane to examine U.S. equipment and learn U.S. surveillance methods [8]. This particular EP-3 was outfitted with the latest U.S. Navy surveillance upgrades [9]. Prior to landing,

the crew of the EP-3 was apparently able to follow standard procedures and drop portable electronic equipment into the ocean in weighted bags and flip the switches that are present at each operator's position to degauss software stored on any magnetic storage devices [9] [10]. Even with these safeguards, all of the hardware details of the plane's equipment were left exposed.

Consider, now, the situation if the proposed FPGA-based secure memory architecture were integrated into the EP-3's electronic systems. An intelligent safeguard for any kind of FPGA system onboard would be to enable the crew's software destruction switches to also deconfigure any running FPGAs and destroy any bitstream storage mechanisms. This would leave only an unprogrammed FPGA for examination. The case must be considered, though, wherein the crew does not have time to flip switches and dispose of portable electronic equipment. Thanks to expert piloting, the EP-3 crew had 20-minutes to perform these procedures. However, if they had been forced to ditch the plane, such technology destruction may have by necessity been ignored, possibly leaving intact secret digital devices available for recovery near China. In this instance, the Amanuet architecture would shine.

Upon examination, the authentication scheme would first have to be cracked. If the iButtons were not present upon the capture of the device, the secret user ID required to start the device would not be available. To even start the device, a secret ID and the correct encryption keys would have to be guessed - a solution of immense mathematical improbability. If the iButtons could not be ditched during standard destruction procedures and were recovered by those examining the hardware, the Amanuet architecture would prevent any information about the embedded application from being revealed through its digital I/O interfaces. Logic analysis of it would only reveal encrypted information flowing to and from its memory and I/O interfaces. If this key management scheme and secured data transfer system was used in surveillance aircraft and other secret digital military applications that are deployed near inquisitive foreign governments, the security of digital intellectual property and device function would be greatly improved.

1.4 Thesis Organization

The Amanuet architecture is presented in this thesis from the perspective of its cryptographic key management scheme. Chapter 2 provides a context for the system by discussing current cryptographic methods for device and FPGA security. Chapter 3 presents the general architecture of the FPGA functional units. Chapter 4 introduces the components used in the prototype. Chapter 5 discusses the cryptographic algorithms used to securely manage the keys as they pass from the iButton through the authentication control unit to the encrypted memory controller. Chapter 6 details the experiments that were performed to prove the viability of the Amanuet architecture.

Chapter 2

Background

2.1 Overview

Cryptographic techniques designed to secure digital information are numerous. While the combination of concepts that make up the Amanuet architecture represents a unique approach to device security, the cryptographic principles and algorithms used in the architecture are tried and true methods. The system is based on concepts from RSA encryption [11], the public key interface (PKI) [12], the secure hash algorithm (SHA) [13], and keyed block cyphers such as the DES [4]. Section 2.2 introduces these proven algorithms and discusses their cryptographic characteristics.

The world of FPGA intellectual property core security is becoming well-established, with the major FPGA manufacturers – Xilinx and Altera – investing a great deal of brain power to the problems of secure configuration and bitstream encryption. Section 2.3 explains in detail current state-of-the-industry FPGA security ideas and contrasts them to the concepts the make up the Amanuet architecture. Section 2.4 delves into the efforts others have done in the field of secure architectures. Finally, Section 2.5 introduces the field of secure tokens and relates it to the iButtons used in the Amanuet architecture.

2.2 Algorithms

As mentioned in the introduction, the algorithms used in this architecture are implementations or slight variations of proven cryptographic methods. A modified RSA algorithm is used to establish an encrypted channel for user authentication. The SHA hash is used to create a certificate matching and storage system. The DES algorithm is used to secure the external memory interface. These algorithms and methods are explained along with a few suggested alternative methods in the following sections.

2.2.1 RSA

Pondering the problem of enabling secure communications between two parties across an insecure channel, Rivest, Shamir, and Adleman came up with an algorithmic solution they named after the initials of their last names: RSA. Their research was an extension of existing ideas, such as the Diffie-Hellman key exchange [14], but with a novel mathematical basis. RSA is dependent on the principles of modular exponentiation and the fact that a modulus that is the product of two large prime numbers is very difficult for even the most advanced computer hardware to factor.

The classic discussion of RSA involves a secure message exchange between parties A and B, or, as we'll name them, Alice and Bob [15] [16]. Alice would like to send a message to Bob across a public channel without compromising the contents of the message. Bob starts the process by choosing two distinct large prime numbers, p and q that he keeps secret. He multiplies p and q together to create the modulus n.

$$n = pq \tag{2.1}$$

Bob then chooses a random number, e, that is relatively prime to $(p-1)(q-1)$. Two numbers are relatively prime when their greatest common denominator is 1, thus

$$\gcd(e, (p-1)(q-1)) = 1. \tag{2.2}$$

This number, e, will serve as the encryption key. To create a corresponding decryption key, d, Bob calculates

$$d = e^{-1} \bmod ((p-1)(q-1)). \tag{2.3}$$

Thus, by equivalence,

$$ed \equiv 1 \bmod (p-1)(q-1). \tag{2.4}$$

The pair (n, e) is now the public key, which is sent to Alice. The decryption key, d, is Bob's private key. Alice can now send Bob a message, m, by representing it as a number and encrypting with e. Note that m must be smaller than the modulus n. If it is not, it must be broken up into blocks that are smaller than n and transmitted in sections. With m less than n, the process that Alice uses to create the encrypted message, c is

$$c \equiv m^e \bmod n. \tag{2.5}$$

Alice sends c to Bob. Bob can then decrypt Alice's message, m, by calculating

$$m \equiv c^d \bmod n. \tag{2.6}$$

To understand this modular equivalence, Euler's theorem must be explained. Euler's theorem states that for integer a, $1 \leq a \leq n$, if $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \bmod n$. Euler's ϕ-function, $\phi(n)$, is defined as

$$\phi(n) = n \prod_{x|n} (1 - \frac{1}{x}). \tag{2.7}$$

where the product is over the distinct primes x that divide n. Euler's ϕ-function represents the number of integers $1 \leq a \leq n$ such that $\gcd(a, n) = 1$. In our case, $n = pq$, and Euler's ϕ-function reduces to

$$\phi(n) = \phi(pq) = pq(1 - \frac{1}{p})(1 - \frac{1}{q}) = (p - 1)(q - 1). \tag{2.8}$$

Therefore $ed \equiv 1 \pmod{\phi(n)}$, which, by the definition of modular arithmetic, can be written

$$ed = k\phi(n) + 1 \tag{2.9}$$

for some integer k. Going back to Alice's message, m, we can assume that $\gcd(m, n) = 1$, since p and q are very large and are the only factors of n. Thus, by Euler's theorem

$$m^{\phi(n)} \equiv 1 \bmod n. \tag{2.10}$$

Now we can see why the decryption process works:

$$c^d \equiv (m^e)^d \equiv m^{ed} \equiv m^{k\phi(n)+1} \equiv m \cdot (m^{\phi(n)})^k \equiv m \cdot 1^k \equiv m \bmod n. \tag{2.11}$$

The process of sending a message using RSA encryption is more simply stated in Table 2.1.

The strength of RSA is based on the fact that it mathematically difficult to factor large numbers. There are only a few attacks that have been proven to work against RSA [15]. The most straight forward is to factor[1] n to calculate the message, m, using the publicly available encrypted message, c, and the encryption key, e. Factoring a product of large prime numbers is far from a trivial problem. The large brains behind today's cryptanalysis field are constantly inventing and improving upon elegant factoring algorithms, such as the multiple variants of the quadratic sieve, the number field sieve, and elliptic curve-based factoring

[1]While factoring n is not the only way to break RSA, other known means of cracking the algorithm have been proven to be just as mathematically complex[15] [17].

Table 2.1: RSA Encryption

Step	*Action*	*Bob Knows*	*Alice Knows*
1	Bob creates n by choosing p and q	n,p,q	Nothing
2	Bob chooses e and computes d	n,p,q,e,d	Nothing
3	Bob sends public key pair, (n, e), to Alice	n,p,q,e,d	n,e
4	Alice creates message m and encrypted message c	n,p,q,e,d	n,e,m,c
5	Alice sends c to Bob	n,p,q,e,d,c	n,e,m,c
6	Bob decrypts c with d and n to retrieve m	n,p,q,e,d,c, m	n,e,m,c

systems [16]. However, a cryptographer wishing to secure an RSA system needs only to keep upgrading the size of the large primes used to create the moduli to keep ahead of current factoring technology.

Getting into actual numbers, RSA Laboratories suggests that an RSA key size of 1024 bits will be secure against attack until the year 2010 [18], at which point cryptographers will want to start using 2048-bit keys. When making recommendations for government security levels, the National Institute of Standards (NIST) claims that a 1024-bit RSA key will be secure until 2015 [19]. Shamir and Tromer, two leading cryptographers, claimed in 2003 that their TWIRL factoring device could break 1024-bit RSA keys in a year using the number field sieve if realized in a $10 million custom VLSI implementation [20]. This may sound like a lot of time and money to spend just to factor a number, but when it might be a government spending the money to steal encrypted secrets from a rival government, $10 million is very little and a year is quite short. Such a machine may actually already exist as part of a secret government code-breaking program.

2.2.2 Secure Hash Algorithm

The Secure Hash Algorithm (SHA) was a joint development project between the National Security Agency (NSA) and NIST [15]. It is a one-way hash, meaning that it is mathematically straightforward to calculate the output of the hash using the input, but it is nearly

impossible to determine the input of the hash knowing only the output. Commonly, the output of a one-way hash function is used as a certificate to authenticate a user. Passwords are stored on personal computers this way. The hash of the password, not the password itself, is saved on the hard disk. This makes it possible to authenticate a user who can produce a correct password – the input password is run through the one-way hash, and the result is compared with the stored hash of the authorized password – without actually storing the password. Since no passwords are actually stored on the PC's hard drive, it highly unlikely that an attacker could steal user passwords just by accessing the hashed password table.

A cracking strategy known as a dictionary attack is the most common means of breaking a PC password knowing only the hashed value of the password. Dictionary attacks are highly effective for passwords that are based on words that can be stored in quickly traversable databases. In the Amanuet architecture, the "password" that is hashed is a number – not based on any language that could limit the possible password combinations – rendering dictionary attacks on the hashed password table ineffective. For more on dictionary attacks see [15].

SHA is a particularly strong hash, and it is currently approved for use by NIST [13]. The input to SHA is called the message and the output is the message digest. The message is processed 512 bits at a time. If the message is significantly shorter than 512 bits, it is padded by appending a one, then enough zeros to make it 448 bits long, then a 64-bit value representing the length of the original message prior to the padding process. If the original message is longer than 512 bits, it is broken up into 512-bit blocks (short blocks padded when necessary), and run through the hash a block at a time. As will be seen, the way SHA works, the digest is always 160 bits.

SHA is based on a non-linear function function f_t, defined as

$$f_t(x,y,z) = \left\{ \begin{array}{ll} (x \wedge y) \vee ((\neg x) \wedge z) & 0 \leq t \leq 19 \\ x \oplus y \oplus z & 20 \leq t \leq 39 \\ (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) & 40 \leq t \leq 59 \\ x \oplus y \oplus z & 60 \leq t \leq 79 \end{array} \right\} . \tag{2.12}$$

The constant value K_t can also be defined non-linearly as

$$K_t = \left\{ \begin{array}{ll} \text{0x5A827999} & 0 \leq t \leq 19 \\ \text{0x6ED9EBA1} & 20 \leq t \leq 39 \\ \text{0x8F1BBCDC} & 40 \leq t \leq 59 \\ \text{0xCA62C1D6} & 60 \leq t \leq 79 \end{array} \right\}. \tag{2.13}$$

Five SHA variables, A, B, C, D, and E, are initialized to the values

$A = \text{0x67452301},$
$B = \text{0xEFCDAB89},$
$C = \text{0x98BADCFE},$
$D = \text{0x10325476},$
$E = \text{0xC3D2E1F0}.$

The 512-bit input message block is broken up into 16 32-bit sections, M_0 to M_{15}. These sections are then used to produce 80 32-bit words, W_0 to W_{79}, defined by the formula

$$W_t = \left\{ \begin{array}{ll} M_t & 0 \leq t \leq 15 \\ (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1 & 16 \leq t \leq 79 \end{array} \right\}, \tag{2.14}$$

where the operation $<<< n$ used above represents a circular shift n bits to the left.

Finally, SHA has five operation variables, a, b, c, d, and e, which get initialized to $a = A$, $b = B$, $c = C$, $d = D$, and $e = E$. SHA has 80 total operations, and is said to have four rounds, each 20-operation round represented by a different part of f_t. SHA's algorithm follows [15]:

FOR $t = 0$ to 79
$TEMP = (a <<< 5) + f_t(b, c, d) + e + W_t + K_t$
$e = d$
$d = c$
$c = b <<< 30$

$$b = a$$
$$a = TEMP$$

Once this process completes, the final calculated values of a, b, c, d, and e are added to A, B, C, D, and E, respectively. If there are more 512-bit message blocks in the message, the next block is processed using these updated values. After the last message block is processed, the output, or message digest, is the 160-bit concatenation of the final values of A, B, C, D, and E:

$$digest = ABCDE. \tag{2.15}$$

NIST considers SHA-1, the version of SHA described above, to be secure against cryptanalysis until 2015 [19]. There are several newer versions of the SHA hash, including SHA-256, SHA-384, and SHA-512. The numbering here corresponds to the size in bits of the digest each hash produces. NIST considers these versions secure well beyond 2015.

2.2.3 Keyed Block Ciphers

A block cipher is a cipher that operates on its input one block at a time, producing a ciphertext output block for each plaintext input block. A keyed block cipher is a block cipher that uses a secret key to alter the block cipher in such a way that the encryption process for the cipher becomes unique to that key. A property of keyed block ciphers is that using the same key, it will always encrypt a given plaintext input block to the same ciphertext output block. Since they do not contain modular arithmetic functions and can be easily pipelined, keyed block ciphers are thousands of times faster than RSA encryption [15]. They do not account for a secure key exchange, making them unsuited for authentication tasks. However, their speed makes them perfect candidates for high-bandwidth digital I/O interfaces.

While any secure keyed block cipher could be reasonably used in the architecture presented in this thesis, there are several algorithms that NIST has standardized that are well suited to this design. The DES algorithm serves as a proof-of-concept algorithm in the prototype

system, whereas the more secure Advanced Encryption Standard (AES) is suggested for commercial implementation [21]. DES is described here in detail, while AES is introduced briefly.

Data Encryption Standard

DES is a product of IBM and NIST, although at the time of DES's adoption in 1975 NIST was known as the National Bureau of Standards [15]. DES uses a 56-bit[2] key to encrypt data in 64-bit blocks. DES encryption consists of 16 rounds of identical substitution and permutation operations, with a unique permutation of the key applied to each round. DES decryption consists simply of applying those key permutations in reverse order to the 16 rounds and running the same substitution and permutation operations in each round.

For a cursory understanding of DES, it can be thought of as a three stage process, as illustrated in Figure 2.1 [15] [16]. Starting with a 64-bit plaintext message block, m, the first stage permutes m with initial permutation $IP(m)$ to give m_0. For future stages, m_0 is best thought of in first and last 32-bit halves, or L_0 and R_0, respectively. Mathematically,

$$m_0 = IP(m) = L_0 R_0. \tag{2.16}$$

The next stage consists of the sixteen rounds of substitution and permutation operations. The key is used here, but it is permuted differently for each of the i rounds. The key for each round is written K_i. There is also a function, $f(R, K)$ that is applied in each round and will be explained later. The second stage, mathematically, is for $1 \leq i \leq 16$,

$$L_i = R_{i-1} \tag{2.17}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i). \tag{2.18}$$

[2]The key is sometimes referred to in a 64-bit form. In this form, every 8th bit is a parity bit, so the effective key size is still 56 bits.

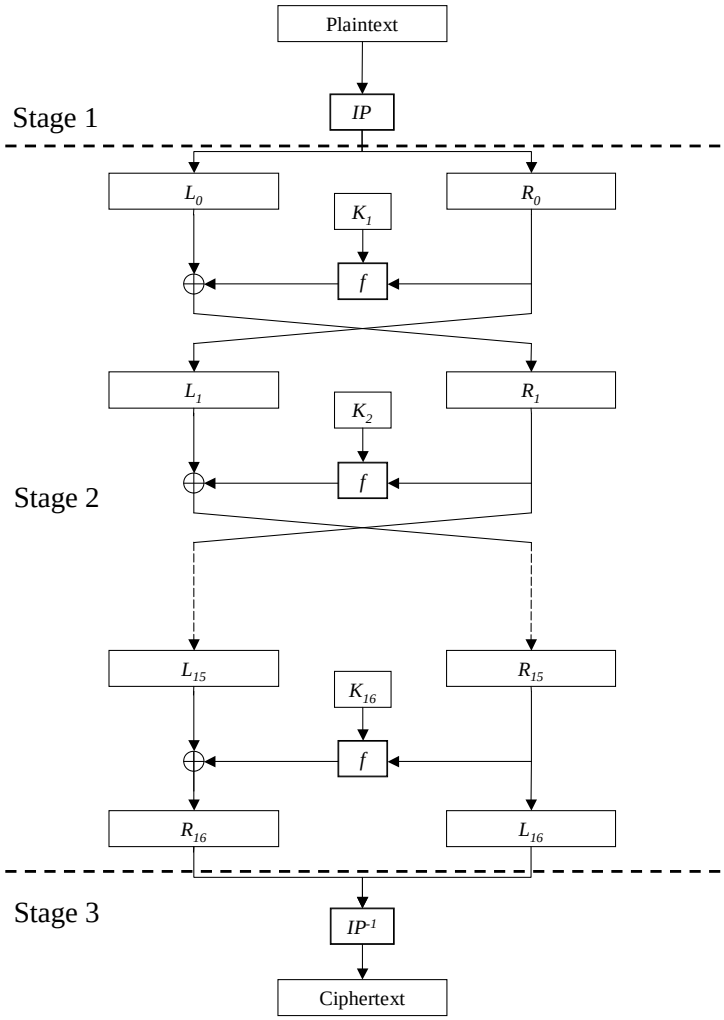

Figure 2.1: The DES algorithm

The final stage requires switching the left and right halves of the 16th stage and applying the inverse of the initial permutation to formulate the cyphertext, c, as follows:

$$c = IP^{-1}(R_{16}L_{16}) \tag{2.19}$$

The decryption process follows the same three stages, except that the key permutations are applied in the reverse order (i.e., K_{16} to K_1 instead of K_1 to K_{16}).

Getting into more detail, the initial permutation, IP, consists of taking the 64-bit message input, m and moving the bits into different positions to create m_0. Table 2.2 describes the initial permutation. This table is interpreted as saying that the first bit of m_0 is the 58th bit of m, the second bit of m_0 is the 50th bit of m, and so on. All other tables that describe bit permutations in this thesis follow the same convention.

Table 2.2: Initial Permutation

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

As mentioned previously, each round of DES algorithm – the 16 rounds from Stage 2 in Figure 2.1 – sees a different version of the key. Remember that the key is 64-bits in length, but every 8th bit is a parity bit, making the effective length 56 bits. The permutations that follow refer to the 64-bit key, with the parity bits excluded. Starting with initial key, K, it is first run through the key permutation shown in Table 2.3.

Table 2.3: Key Permutation

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	20
14	6	61	53	45	37	29	21	13	5	28	20	12	4

The result of this permutation is the 56-bit key K_0, which is divided into left and right 28-bit halves X_0 and Y_0, respectively. To begin the creation of keys used in the 16 DES rounds, K_1 to K_{16}, these halves are shifted left 1 or 2 times per round. Mathematically, we can say that for $1 \leq i \leq 16$,

$$X_i = S_i(X_{i-1}), \tag{2.20}$$
$$Y_i = S_i(Y_{i-1}), \tag{2.21}$$

$$K_i = CP(X_i Y_i), \tag{2.22}$$

where S_i is a left circular shift by the number of bits described by Table 2.4 and CP is a compression permutation that reduces the key size from 56 to 48 bits. The compression permutation is described in Table 2.5.

Table 2.4: Number of Key Bits Left Shifted Per Round

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 2.5: Key Compression Permutation

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

With the keys established, we can now examine the details of the function $f(R_{i-1}, K_i)$, illustrated in Figure 2.2. Remember that the input to f is the right half of the previous round, R_{i-1}, and the current round's key, K_i. The first step in f is to expand R from 32 bits to 48 bits with an expansion permutation, shown in Table 2.6.

Table 2.6: Expansion Permutation

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

The result, R_{exp}, of this expansion is then XORed with the current key, K_i, to form a 48-bit intermediate value I. This value is divided into eight 6-bit segments, which are written

$$I = I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8. \tag{2.23}$$

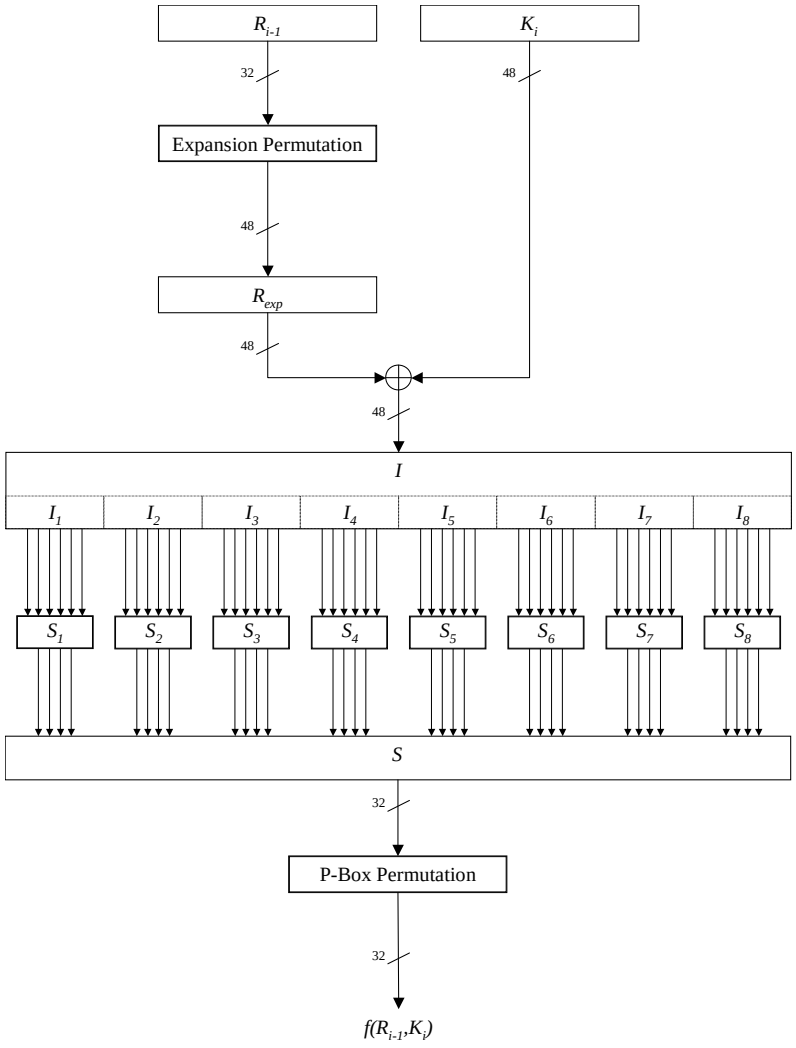

Figure 2.2: The DES function

The values I_1 through I_8 are the inputs into the S-Boxes S_1 through S_8, respectively. Each S-Box is a 4 by 16 matrix of numbers. The input, I_k for $1 \leq k \leq 8$, for S-Box S_k describes the row and column from which to select the output of each S-Box. The first and sixth bits of I_k determine which row, binary 00 through 11, and the middle four bits determine which column, 0000 through 1111. For example, if I_k were 101010, the output of the S-Box would

be selected from row 10 and column 0101. The S-Box matrices are described in Table 2.8. In this table, the binary values by each row of the S-boxes represent the value of the first and sixth bits of I_k, and the binary values above each column represent the middle four bits of I_k.

The output of each S-Box is a 4-bit value; all S-Box outputs together form a 32-bit number S. The final output of the function f is the result of a P-Box permutation on S, as described in Table 2.7

Table 2.7: P-Box Permutation

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Glancing at the structure of DES, a hardware designer can easily see how it can be pipelined. Each instance of the function f could become a stage in the pipeline, and a straightforward hardware implementation would only require one clock per pipeline stage. The DES implementation used in the prototype architecture presented in this thesis is designed in this way.

A differential cryptanalysis of the DES algorithm was first presented in 1993, though it was at the time only possible to break the first few rounds [22]. As processors become faster and memory becomes less expensive, even exhaustive searches of the DES key space have proven highly effective threats to systems secured by DES [16]. It is for this reason that NIST sought to phase DES out and replace it with the Advanced Encryption Standard.

Advanced Encryption Standard

The Advanced Encryption Standard is the most recent keyed block cipher to be approved by NIST [21]. It operates on input blocks of 128 bits, and uses keys that are either 128, 192, or 256 bits in length. AES is the result of NIST requesting in 1997 that the cryptographic community present several candidates to replace DES. Many algorithms were presented, but only one, an algorithm called Rijndael (pronounced “rain doll”), was selected to become AES [16].

Table 2.8: S-Boxes

S_1	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
01	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
10	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
11	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
01	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
10	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
11	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
01	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
11	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
01	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
10	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
01	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
10	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
11	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
01	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
10	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
11	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8	*0000*	*0001*	*0010*	*0011*	*0100*	*0101*	*0110*	*0111*	*1000*	*1001*	*1010*	*1011*	*1100*	*1101*	*1110*	*1111*
00	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
01	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
10	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
11	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

AES is based on three transformation steps and a round key addition step. The three transformations are byte substitution, row shifting, and column mixing. These are performed once in each of ten rounds, with each round having a unique round key derived from the original key. While it will not be explained in detail in this thesis, AES is cryptographically much stronger than DES, and it is also easy to create a pipelined hardware implementation of it. It is recommended that if the Amanuet architecture were implemented in a system other than the proof-of-concept prototype developed here, the AES cipher should be used instead of the DES cipher. Remember that in the Amanuet architecture, any keyed block cipher could be easily modified to replace the DES cipher, making the system updateable when necessary to meet new security threats.

2.3 FPGA Security

Many prior efforts have been made across the industry to secure device intellectual property on an FPGA. Rather than focusing on authorization and the prevention of bus snooping, most of these endeavors have focused on ensuring that the FPGA bitstream is safe from reverse engineering. The fear is that a system designed in which the bitstream is stored near the FPGA is vulnerable to a hostile user taking the bitstream and either applying it to other FPGAs or reverse engineering it to determine the design. Reverse engineering a bitstream would be quite a difficult task. While it may be straightforward to take a bitstream and extract a netlist describing the function of each FPGA slice, RAM, and interconnect element, interpreting that netlist would be an effort on the scale of trying to determine the C++ code behind a large Windows application starting only with the executable code. Nonetheless, FPGA designers still want a better level of security.

To address these concerns, the large FPGA manufacturers like Xilinx and Altera have produced systems that encrypt the bitstream in the bitstream generator and decrypt it in hardware at the time the FPGA is being programmed [23]. Such a system is available in all Xilinx Virtex II devices [24]. The Virtex II uses a variant of the DES algorithm called Triple DES, or 3DES, which essentially encrypts the bitstream with the DES algorithm three times using three keys, a vast improvement over the security of DES. Altera's current flagship

FPGA, the Stratix II, utilizes a 128-bit AES algorithm to secure its bitstreams [25]. With these efforts already providing robust bitstream security, this thesis does not attempt to improve upon them. The approach taken here is to prevent reverse engineering through the examination of an already configured device.

2.4 Other Secure Architectures

There are several architectures either in development or on the market today that propose to provide protection against unauthorized use and digital interface analysis. While many platforms, such as the IBM 4758 coprocessor [26] and the Microsoft Next-Generation Secure Computing Base[3] (NGSCB)[28], are referred to as secure computing platforms, the design that most closely resembles the Amanuet architecture is the AEGIS architecture from the Massachusetts Institute of Technology [29]. AEGIS takes a similar approach in that it trusts its processor, but does not trust external memory or peripherals. AEGIS has thus far only been demonstrated in high-level simulations. If it is ever built, AEGIS will require an application specific integrated circuit (ASIC) implementation, which will prevent it from being nearly as configurable as the Amanuet architecture. Additionally, all of the architectural details of AEGIS will be visible to physical examination. The FPGA-based approach will allow designers to apply the architecture to any embedded FPGA design and perform in-the-field upgrades when necessary, advantages not shared by the AEGIS system.

2.5 Secure Tokens

In the Amanuet architecture, the Java iButton is referred to as a secure token. Tokens, much like smart cards, are usually small devices with memory and possibly some processing capability. They are used most often as user identification devices, but they have also found usefulness recording temperatures using on-chip sensors, storing electronic cash for use at

[3]During the writing of this thesis, Microsoft was apparently reconsidering their plans for NGSCB. The project has suffered setbacks, but the company insists that there will be a version of the NGSCB included in the upcoming Longhorn release of Windows [27].

casinos, storing records for gaming systems, and tracking people as they move through a secure facility [30]. The Java iButton from Dallas Semiconductor was chosen for this design because of its cryptographic co-processor and Java programmability that allow it to securely communicate with other devices by enabling an RSA-like secure channel. It also has proven to have robust resistance to physical tampering through its ability to destroy its non-volatile memory (NVRAM) if its case is ever opened.

There is a wide variety of other tokens and smart cards available that could conceivably play the role of the iButton in the Amanuet architecture. Among the most popular options are the smart cards and USB tokens made by RSA Security [31]. These devices can support complex cryptographic algorithms such as DES, 3DES, RSA, RSA signatures, and SHA-1 while supporting Java Card applications on their embedded 8-bit CPUs. Another company, ActivCard, provides larger portable secure tokens that include built-in keypads for inputting personal identification numbers [32]. The Java iButton was chosen from among these options for its comparable secure capabilities, its small size, its physical security, and the simplicity of creating a serial interface between it and an FPGA. The serial interface will be discussed in detail in the next chapter.

Previous work has been done at Virginia Tech towards creating an authentication system that utilizes iButtons [33]. These efforts resulted in a system that used a Dallas Semiconductor TINI processor board to facilitate the communication between the Java iButton and the FPGA. The unfortunate security consequence of the decision to include the TINI is that the interface between the TINI and the FPGA is in clear text, leaving it vulnerable to probing and analysis. The system proposed for the Amanuet architecture eliminates the need for the TINI processor by implementing a direct secured communications path between the iButton and the FPGA. In the Amanuet architecture, no external element of the authentication interface ever communicates in clear text.

Chapter 3

Architecture

3.1 Overview

The Amanuet architecture can be conceptualized as a wrapper that is placed around any embedded FPGA application or CPU that requires the use of external I/O interfaces. The particular I/O interface used in the proof-of-concept system is a memory interface. Amanuet is depicted as a wrapper in Figure 3.1. The embedded application can essentially be "plugged in" to the Amanuet architecture to add the protection of user authentication and memory encryption.

3.2 Details

A more detailed block diagram of the Amanuet architecture is shown in Figure 3.2. Inside the Xilinx Virtex XCV2000E FPGA [34] are three major functional units. The first is the embedded application provided by the FPGA designer. The second two are components of the Amanuet architecture: the Authentication Control Unit (ACU) and the Encrypted Memory Controller (EMC). Devices connected to the FPGA are the external RAM and the iButton (*iB* in the figure). The external RAM is connected to the FPGA through external

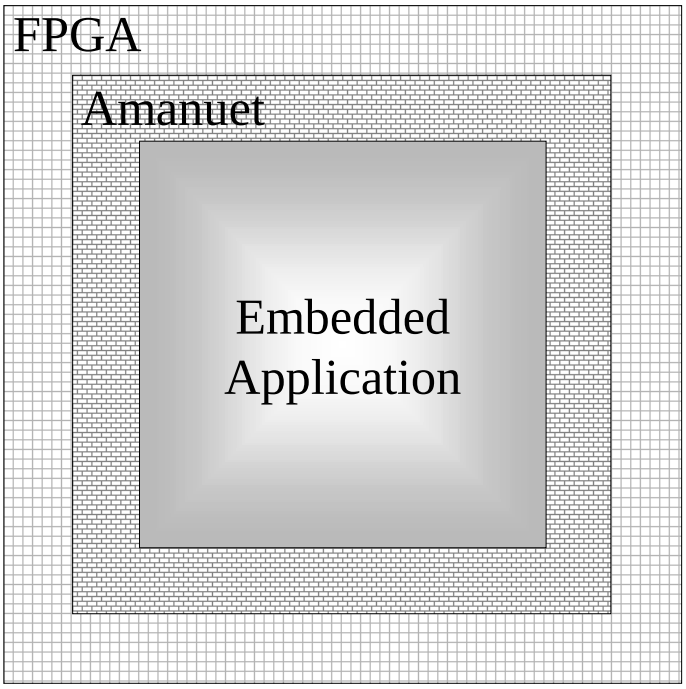

Figure 3.1: Amanuet as a wrapper for an embedded application

I/O pins.

The iButton interface communicates with the FGPA through a Dallas Semiconductor DS9097U Universal COM Port Adapter [35]. The DS9097U is necessary because the iButton communicates using a protocol developed by Maxim Integrated Products called the 1-Wire protocol [36]. The 1-Wire protocol provides both a communications channel and a power interface to 1-Wire compatible devices (such as an iButton) over a single-wire connection. The connection being only a single wire necessarily means that 1-Wire is a serial protocol. The 1-Wire interface on the DS9097U consists of an RJ-11 jack that connects inside the DS9097U to a DS2480B Serial 1-Wire Line Driver Chip [37]. It is the DS2480B that performs the protocol translation between the 1-Wire protocol and a standard serial protocol. A simplified block diagram of the DS9097U is shown in Figure 3.3. The RS-232 interface of the DS9097U uses levels of $\pm$12V. A MAX232 level translator [38] (not shown in any figure) on a custom board

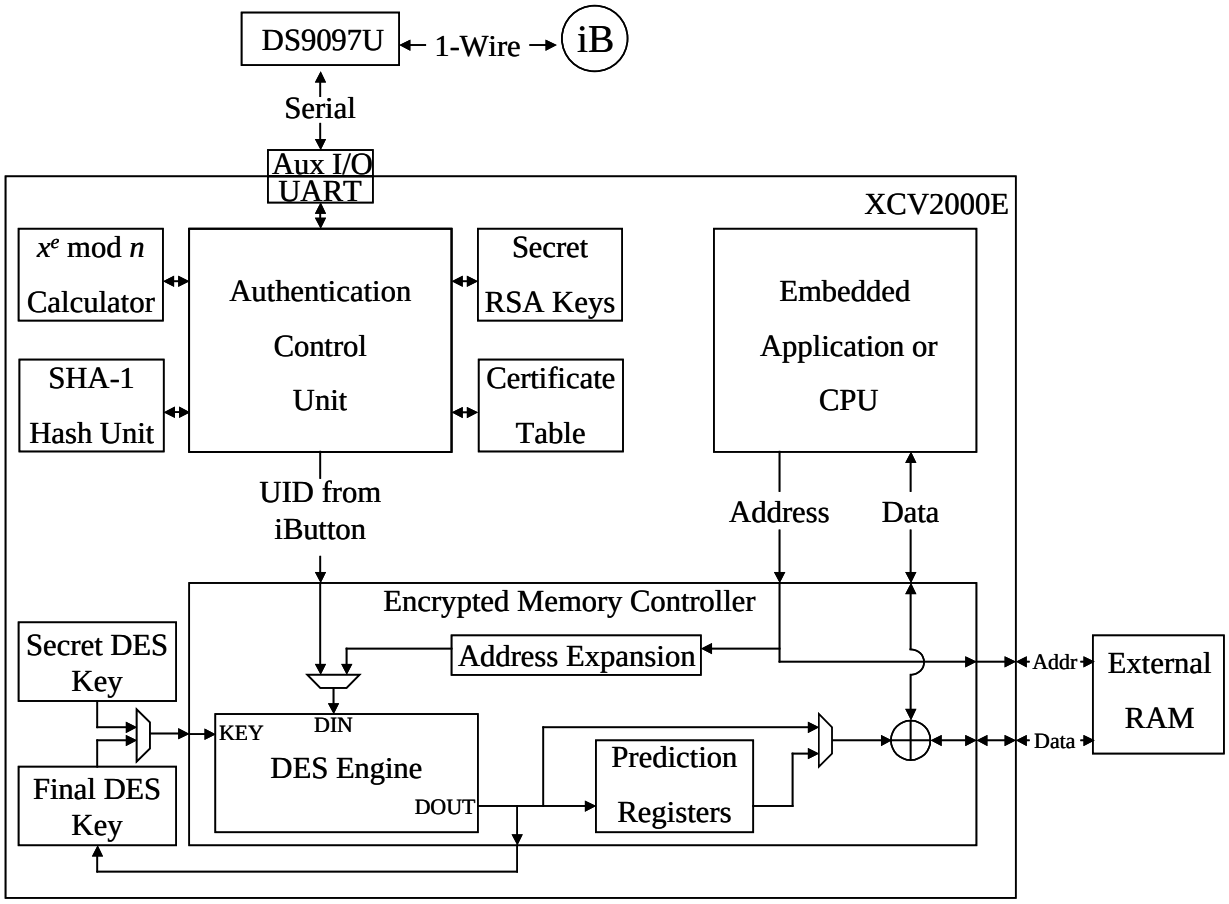

Figure 3.2: Block diagram of the Amanuet architecture

is used to translate from these levels to the 0V/5V levels required by the auxiliary I/O pins on the FPGA.

Inside the FPGA, a UART connected to those auxiliary I/O pins enables the ACU to communicate through the DS9097U to any device on the DS9097U's 1-Wire network. The ACU will wait until it detects an iButton the 1-Wire network to begin the authentication process. Prior to authentication, the embedded application or CPU is held in a wait state to prevent

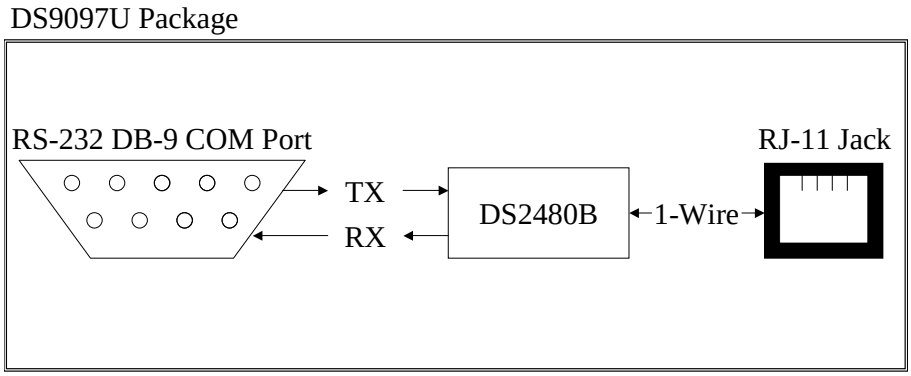

Figure 3.3: Simplified block diagram of DS9097U

it from running without an authorized user. When a user wishes to be authenticated and use the device, they will plug an authorized iButton into a receptacle connected to the 1-Wire network. The ACU will detect the presence of the iButton and begin a communication session with it. The link between the FPGA and the iButton is an exposed and potentially vulnerable interface. Measures, which will be described fully in Chapter 5, must be taken to protect the presumably exposed data transferring across this link.

As illustrated in Figure 3.4, the purpose of the session between the ACU and the iButton is threefold. First, the iButton must check to ensure that it is communicating with a valid FPGA host. Next, the FPGA must, in turn, determine that it is talking to a valid iButton. Since both parties – the FPGA and the iButton – hold secrets that could compromise the system's security, they must not communicate with devices outside the Amanuet system. Finally, the iButton must securely send the FPGA a user identification number (UID). The

ACU side of the secure connection is enabled by the secret RSA keys and the $x^e \bmod n$ calculator, as seen in Figure 3.2. The SHA-1 hash unit and the certificate table are used to calculate the hash of the UID and check it against the valid certificates in the certificate table to determine if the user is valid. If the UID is not valid, the user is not allowed to use the system, and the embedded application remains inactive. If the UID is valid, it is passed to the EMC to trigger it to prepare to start the application. Again, the details of this process are listed in Chapter 5.

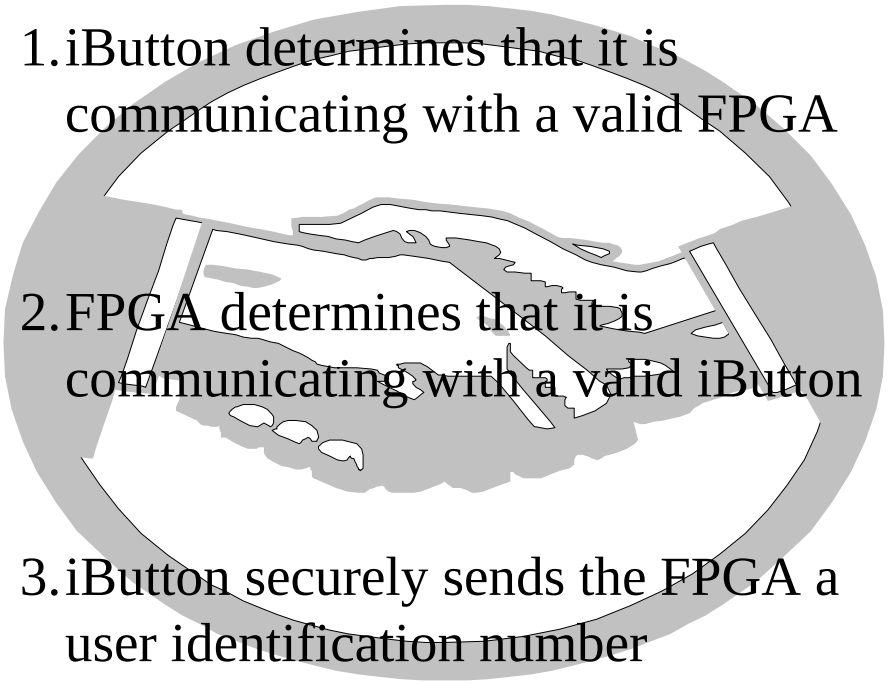

Figure 3.4: Communications session between ACU and iButton as a three-step process.

Once the EMC receives the UID, it applies it to the input of the DES engine using a secret DES key that is compiled into the FPGA bitstream. The 56 bits of the 64-bit encrypted result of this operation become the final DES key. The secret key is intended to be unique to each FPGA bitstream that is distributed. In this manner, when the secret key is used to create the final key, the final key will be unique to each user/FPGA combination, making it more difficult to break the security of the overall architecture.

With the final key in place, all of the security mechanisms in the Amanuet architecture have been implemented. The embedded application is triggered to begin operation. When the embedded application makes a memory reference – whether read or write – the EMC always takes control of it. The address of the reference is sent both to external RAM and to an address expansion unit inside the EMC. The address expansion unit takes as input the address and outputs a 64-bit number that is a linear function of the address. This 64-bit output becomes the input to the DES engine. The DES result becomes an XOR key for that particular address. On a write operation, the XOR key is XORed with the data from the embedded application and is written to memory. On a read, the XOR key is XORed with the incoming data from memory and is written to the application. The EMC also contains a set of prediction registers that hold XOR keys for addresses that are predicted to be accessed next. If the prescribed prediction function is correct, an XOR key might be available in the prediction registers sooner than it could be calculated by the DES engine, saving valuable time in the memory reference. This prediction register system, along with the whole key management framework, is explained in all of its cryptographic detail in Chapter 5.

Chapter 4

Platform

4.1 Overview

The Celoxica RC1000 was chosen as the platform for the proof-of-concept prototype of the Amanuet architecture. Built on a PCI card, the RC1000 was selected as a prototyping platform because of its ability to interface easily with both Windows and Linux PCs and because of the ease of writing C code using Celoxica-provided libraries that allow a PC-based program to communicate with the card. Section 4.2 discusses the RC1000 platform in detail.

As mentioned in prior chapters, the Dallas Semiconductor iButton is used as a secure token in the Amanuet architecture. The iButton interface and the process of creating an iButton host are detailed in Section 4.3.

4.2 RC1000

A simplified block diagram of the PCI-based Celoxica RC1000 platform is shown in Figure 4.1. The main feature of the RC1000 is the Xilinx Virtex XCV2000E FPGA. It also includes four 512k×32 SRAM banks each with their own address, data, and control interfaces. The RC1000 contains arbitration logic that allows the PCI bus and the FPGA to both

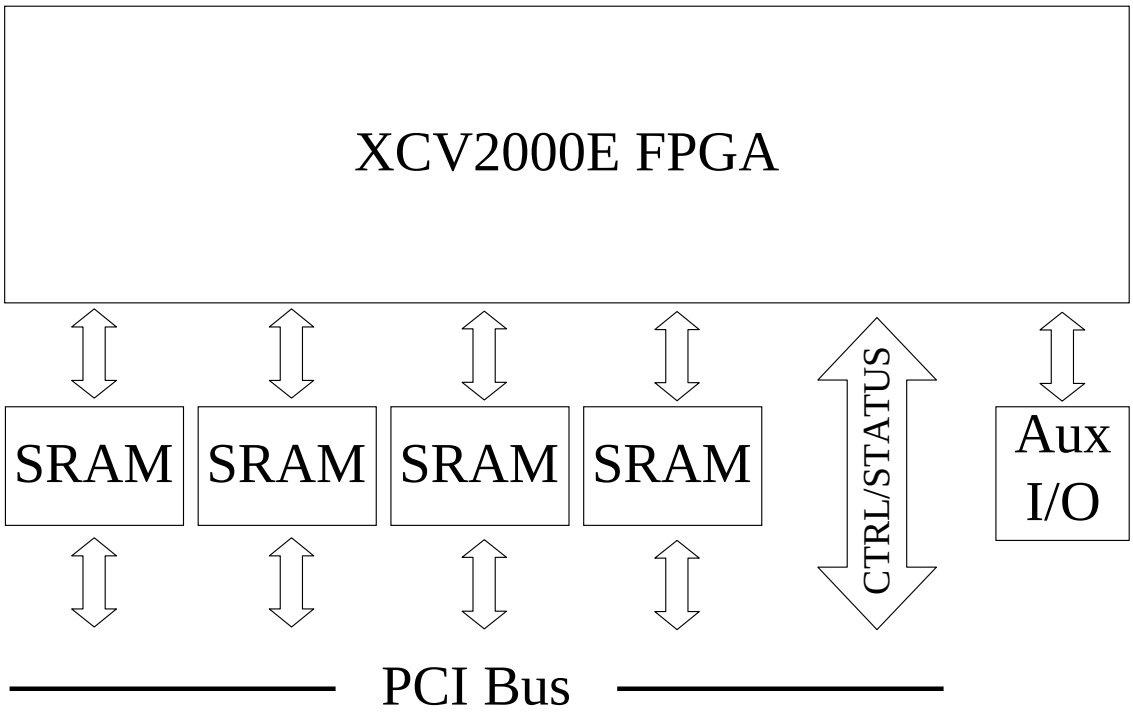

Figure 4.1: Simplified RC1000 block diagram

access the SRAM banks. Ownership of each bank is independent and is controlled by the program running on the PC (which is communicating with the banks through the PCI bus). These SRAM banks allow the PC program to pass data and commands to and from the FPGA through direct memory access (DMA) transfers. The RC1000 also has control and status signals and registers to further facilitate communication between the FPGA and the PC program through the PCI bus. The PC host program can control the FPGA clock and communicate directly with the FPGA through the control and status signals. The RC1000 also includes fifty auxiliary I/O pins that allow a direct connection to the FPGA's external I/O pins. In the Amanuet architecture, the DS9097U COM port adapter for the iButton is

interfaced directly to these auxiliary I/O pins.

Celoxica provides a robust C/C++ library to support the host PC programs that communicate with the FPGA. The library can be used from any C/C++ compiler, such as Microsoft Visual C++. It supplies simple functions to support the following operations:

- Get, open, and close available RC1000 cards
- Request, receive, and release memory banks
- Setup and perform DMA transfers
- Get and set status and control registers
- Load bitstream files to the FPGA
- Set the clock speed of the FPGA
- Control error handling specific to the RC1000 platform

An example C++ program written using this library would start by including the library at the beginning of the file.

```
#include <PP1000.h>
```

This example program is going to setup a connection with an RC1000 card, pass an array of data to it, and start the FPGA working on the data. All the functions and types created in the PP1000 library start with the "`PP1000`" prefix. For a detailed explanation of the functions used in this example program, see the RC1000 Functional Reference Manual [39]. An error handling function is required for errors specific to the RC1000 platform. It could be written as follows:

```
void Handler(char *FnName, PP1000_STATUS Status) {
    char Buffer[1024];
    PP1000StatusToString(Status,Buffer,sizeof(Buffer));
```

```
    printf("\n%s - %s\n", FnName, Buffer);
    exit(1);
}
```

In the `main` function, the program would then mount the error handler and open the first available RC1000 card.

```
    PP1000InstallErrorHandler(Handler);
    printf("Getting number of cards.\n");
    Status = PP1000GetCards(&NumCards, &Present);
    printf("%d cards in the system\n", NumCards);
    Card = new PP1000_HANDLE;
    Status = PP1000OpenCard(Present[0].CardID, &Card[0]);
    Status = PP1000GetCardInfo(Card[0], &CardInfo);
    printf("Card's serial number is %d or 0x%08lx\n\n",
            CardInfo.SerialNum, CardInfo.SerialNum);
```

Note that there can be multiple RC1000 cards installed on a single PC. The program could then configure the FPGA from a bitstream in a bitfile, set the clock frequency of the FPGA, and reset the FPGA. Acceptable clock frequencies are between 1 MHz and 100 MHz for the Virtex XCV2000E.

```
    printf("Configuring FPGA...\n");
    Status = PP1000ConfigureFromFile(Card[0], "bitstream.bit");
    printf("\nSet clock frequency\n");
    PP1000SetClockRate(Card[0], PP1000_MCLK, 50000000);
    PP1000ResetFPGA(Card[0]);
```

At this point, the program is free to either start the FPGA or load one of the four SRAM banks with data. This example program first sets up and runs a DMA transfer to load the first memory bank with `Buffer`, an array of integers.

```
Status = PP1000RequestMemoryBank(Card[0], 1);
Status = PP1000SetupDMAChannel  (Card[0], Buffer, 0,
                                 buffersize * 4,
                                 PP1000_PCI2LOCAL, &Channel);
printf("Initializing transfer\n");
PP1000DoDMA(Channel);
```

The PC host then wants the FPGA to have control of the memory bank to allow the FPGA to operate on the data transferred to it. The host accomplishes this by closing the DMA channel and releasing the memory bank.

```
PP1000CloseDMAChannel(Channel);
PP1000ReleaseMemoryBank(Card[0], 1);
```

With the data transferred to the FPGA, the program then writes to the control register to let the FPGA-based application know that it can start operation. In the Amanuet prototype system, this is when the FPGA would begin polling for the iButton.

```
Status = PP1000WriteControl(Card[0], 1);
```

The FPGA application would then perform whatever function it was designed to perform. Depending on how complex the host and FPGA applications are, multiple negotiations could occur between the host and the FPGA at this point via DMA transfers and communication through the control and status registers. A certain status value could indicate to the host that the FPGA is finished with its function.

```
PP1000ReadStatus(Card[0], &fpga_status);
if (fpga_status == DONE) end_program();
```

When the FPGA application is finished, more DMA transfers might retrieve data the application was working on. The Celoxica libraries are simple to use, but robust enough to support complex hardware/software co-designed applications.

4.3 iButton

The Dallas Semiconductor Java-Powered iButton model DS1957B serves as a secure token in the Amanuet architecture. The DS1957B is packaged in a cylindrical stainless steel canister 5 mm high and 16 mm in diameter, and it resembles a watch battery. Internally, it consists of a Java processor, a 1024-bit cryptographic math accelerator, a random number generator, 134-kilobytes of non-volatile RAM, and a serial communications interface for communicating using its proprietary 1-Wire interface protocol. Communications signals on this 1-Wire interface also serve to power the iButton whenever it is plugged into a 1-Wire compatible receptacle. The iButton's construction is physically tamper resistant due to a mechanism that will destroy the contents of its memory if its case is opened.

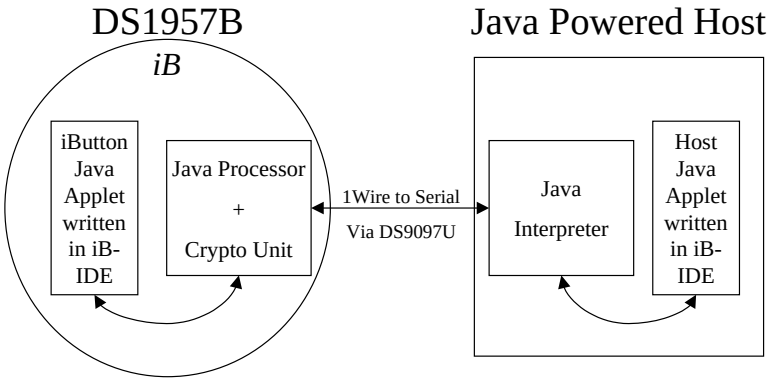

Figure 4.2: Communications model between the Java host and the Java iButton

Figure 4.2 illustrates the way the Java iButton is designed to communicate with a Java host. In this model, the iButton's Java processor and cryptographic acceleration unit run a Java applet that was developed using using Dallas Semiconductor's iButton Integrated

Development Environment (iB-IDE) [40]. A screenshot of the iB-IDE is shown in Figure 4.3.

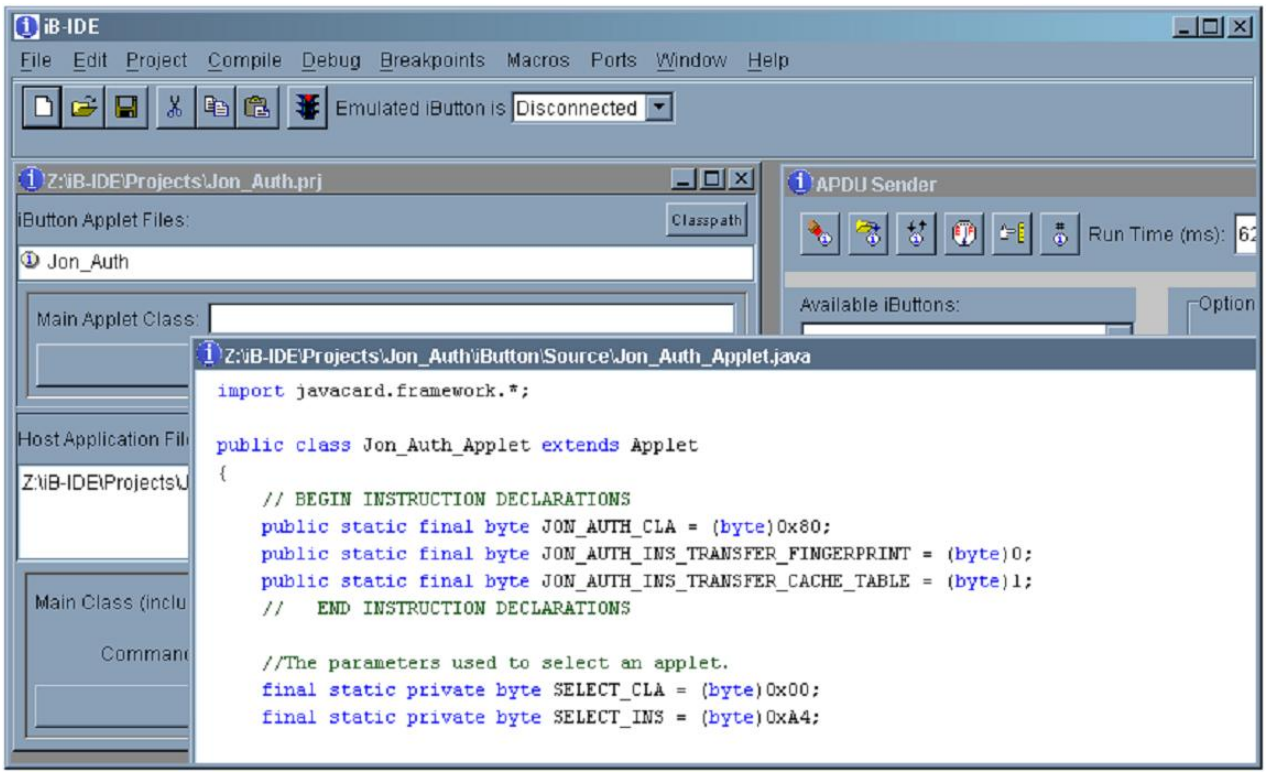

Figure 4.3: Screenshot of the iButton Integrated Development Environment

The DS9097U Universal COM Port Adapter provides the translation between the 1-Wire protocol of the iButton and the serial protocol on the host. The host consists of a host Java applet that was written using iB-IDE running on another Java interpreter. For most iButton uses, this host is a PC. The iButton and host applets are capable of creating a cryptographically secure communications channel across the DS9097U.

In the Amanuet architecture, the communications model between the host and the iButton is not quite this simple, since the host is an FPGA. Creating a Java virtual machine to run on the FPGA was not only beyond the scope of this project but also would have taken up far too much space on the FPGA. Thus, a development flow was created to facilitate the implementation of an FPGA host that emulates the behavior of a Java host over the 1-Wire interface.

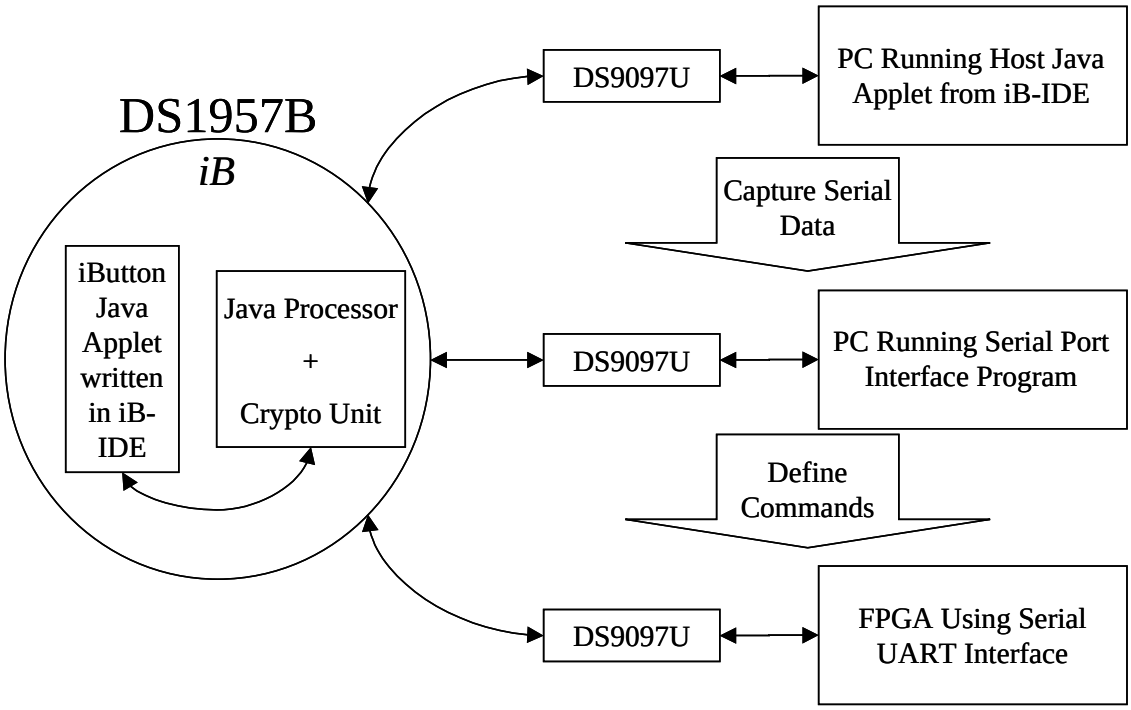

Figure 4.4: Java-to-VHDL host development flow

In the Java-to-VHDL host development flow, illustrated in Figure 4.4, the first step is to create an iButton Java applet to run on the DS1957B and a Java host to run on the PC. Communications between the two is facilitated by the DS9097U that interfaces the iButton to the PC's serial port. All of the serial data passing between the iButton and the host is then captured an analyzed to give clues as to the protocol running between the two parties.

To assist in defining this protocol, a C program was created to run on a PC and send serial commands to the iButton through the PC's serial port and across the DS9097U to the iButton. The second step in the development flow is to use this C program to define

the minimal set of serial commands necessary to fully communicate with the iButton and run through all steps of the authorization algorithm. Once this program is perfected and minimized, the final step is to mimic its function with a VHDL state machine that will be run on the FPGA. As mentioned in previous chapters, this final host also utilizes the DS9097U.

The most complex step in this process is defining the minimal set of serial commands from the captured serial data. Multiple thousands of individual serial bytes of data are captured during a single authorization session, and these bytes have been classified "by hand" (i.e., by human) and simplified into a several key command blocks of a few hundred bytes each. This process is repeated every time a change is made to the PC host or the iButton applet, making this an arduous, repetitive task. If the Java iButton commands were better documented as a serial protocol, this job would be greatly simplified. The result of this effort is to finally realize a truly secure communications interface between the Java iButton and the FPGA host.

Chapter 5

Key Management

5.1 Overview

The term *key management* is used in the Amanuet architecture to describe the entire process of retrieving the user identification information (UID) from the iButton through the secure user interface, modifying the UID to form a DES key, and using that DES key as the key source for the encrypted memory controller (EMC). The design of the key management process is the essential element that makes the Amanuet architecture effective in its stated goal to protect the details of the design around which it is wrapped. A poorly-designed key management scheme could render the Amanuet architecture ineffectual under the scrutiny of a well-trained attacker. A correctly-designed key management scheme could lengthen the time required to analyze the device to decades, making it effectively unbreakable.

This chapter discusses in extensive mathematical and architectural detail the key management scheme used in the Amanuet architecture. Drawing on the background material from Chapter 2 and the architectural discussion from Chapter 3, this chapter explains the ACU in Section 5.2 and the EMC in Section 5.3. Section 5.4 discusses the vulnerability of this system to analysis. A major theme in the proposed key management scheme is keeping the secret information on the iButton separated from the secret information on the FPGA to ensure that if either the iButton's or the FPGA's individual security methods are broken,

the entire system will not be violated. This rule is enforced throughout the following key management units.

5.2 Authentication Control Unit

The ACU, illustrated in Figure 5.1, is responsible for establishing a secure communications channel for the UID transfer from the iButton. At the ACU's disposal are an $x^e \bmod n$ calculator, a set of secret RSA keys, an SHA-1 hash calculator unit, and a table of authorized user certificates. The ACU and the iButton negotiate a secure channel and establish each others identities using an encryption, authentication, and certificate-checking system similar to the RSA-based [11] Public Key Infrastructure (PKI) [12].

There are a few significant differences, however. First, the public encryption keys for the iButton and the FPGA are never transmitted over an insecure channel. Since it is essential to our system that all keys remain secret, they will not be referred to as public and private keys. Those familiar with the RSA encryption methods described in Chapter 2 will recognize the Amanuet architecture's encryption key as RSA's public key and the decryption key as RSA's private key.

The second difference between the Amanuet authentication method and RSA is that certificates in the Amanuet architecture are Secure Hash Algorithm (SHA) [13] hashes of valid iButton UIDs. Recall that SHA is a one-way hash, meaning that it is mathematically difficult to calculate the input to the hash using only the result of the hash. This makes it safe to store the hash results of the UIDs in a certificate table outside the iButton. This table could reside directly on the FPGA, which would require modifying the FPGA bitstream every time a new certificate was issued to an authorized user, or it could be retrieved by some other means from a trusted certificate authority. The Amanuet architecture includes the table within the FPGA.

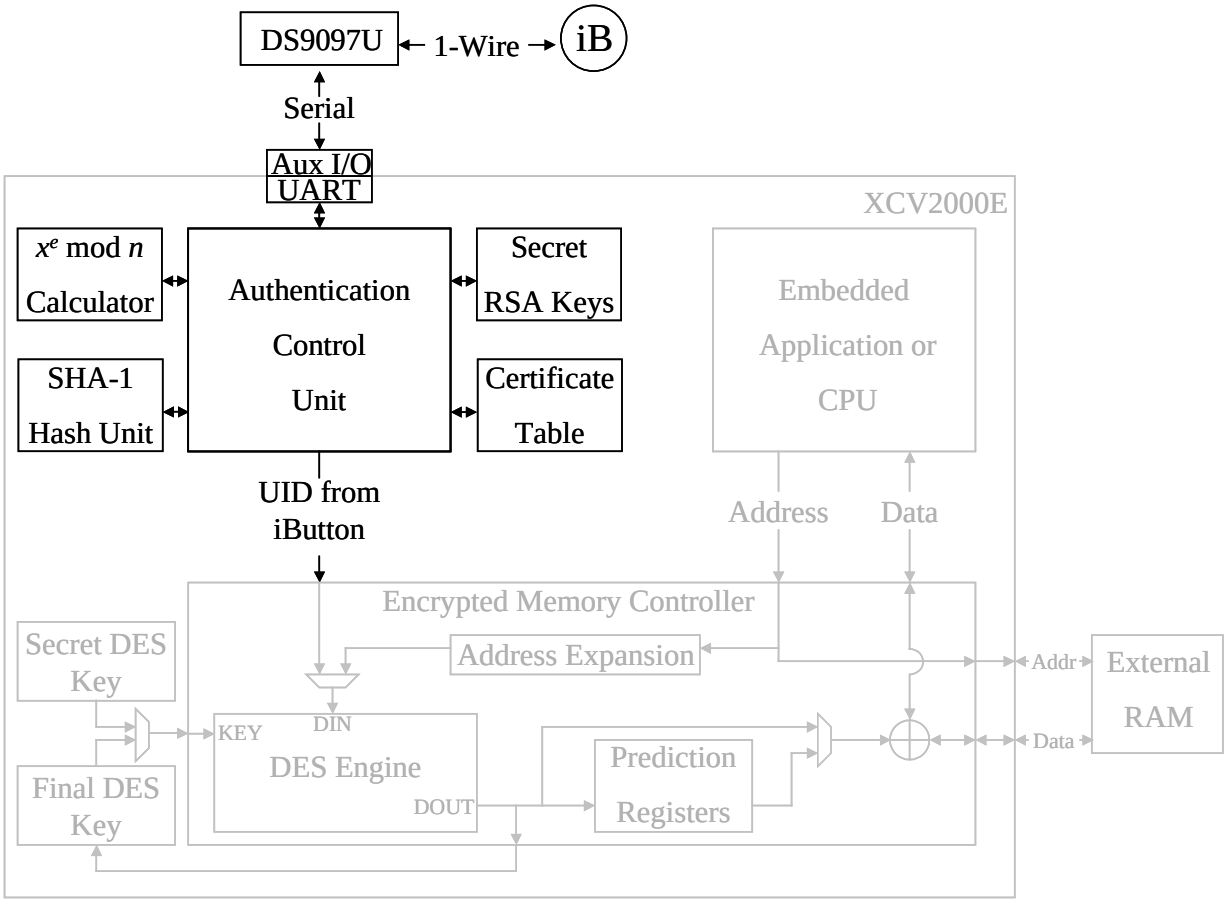

Figure 5.1: Block diagram of the authentication control unit and its connections

5.2.1 Authentication Mathematics

There are two pairs of encryption and decryption keys and two moduli used in Amanuet's RSA-related authentication scheme. Prior to the programming of the iButton and the creation of the FPGA bitstream, four large prime numbers are chosen by a key administrator

to create these key pairs and moduli. These primes[1], p_i, p_f, q_i, and q_f, are combined to form the moduli, n_i and n_f, through multiplication.

$$n_i = p_i q_i \tag{5.1}$$
$$n_f = p_f q_f \tag{5.2}$$

The encryption keys, e_i and e_f, are then chosen to be relatively prime to $(p_i - 1)(q_i - 1)$ and $(p_f - 1)(q_f - 1)$, respectively. Using e_i and e_f, the decryption keys, d_i and d_f, can be calculated.

$$d_i = e_i^{-1} \bmod ((p_i - 1)(q_i - 1)) \tag{5.3}$$
$$d_f = e_f^{-1} \bmod ((p_f - 1)(q_f - 1)) \tag{5.4}$$

In this way, a key pair and a modulus have been established for both the iButton – n_i, e_i, and d_i – and the FPGA – n_f, e_f, and d_f. Remember that *all* of these values are considered secret. They are calculated and stored on the FPGA and the iButton before the technology is deployed, and they are never transmitted over a clear channel. These secret keys and moduli are stored on the FPGA and iButton as shown in Table 5.1.

Table 5.1: Storage of Secret Keys and Moduli

FPGA	*iButton*
n_i, n_f, e_i, d_f	n_i, n_f, e_f, d_i

Recall that with the encryption key, e, the decryption key, d, and the modulus, n, chosen in the same way, the general RSA encryption algorithm computes an encrypted message, c, from a clear-text message, m, using e, through a modular exponentiation.

$$c \equiv m^e \bmod n \tag{5.5}$$

[1]The subscripts i and f indicate an association with either the iButton or the FPGA, respectively.

The RSA algorithm decrypts that message using d.

$$m \equiv c^d \bmod n \tag{5.6}$$

The Amanuet architecture's user authentication method uses this same relationship between the keys to encrypt communications between the iButton and the FPGA. By using the same thoroughly examined and proven mathematical key relationships that are used in RSA, this user authentication method is known to be mathematically secure.

5.2.2 Authentication Process

In this system, it is not only important for the iButton to authenticate itself as an authorized user to the FPGA, but the FPGA must also authenticate itself as an authorized host to the iButton. This prevents a man-in-the-middle attack wherein a device would pretend to be the the FPGA to the iButton while simultaneously pretending to be the iButton to the FPGA in an effort to steal the iButton's UID and falsely authenticate itself to the FPGA. Expanding on the simplified three-step description given in Chapter 3, the complete ACU-iButton authentication method goes through the following steps:

1. iButton is plugged in to the 1-Wire interface.

2. FPGA recognizes presence of iButton.

3. iButton generates a random number, m_i, encrypts it using e_f and n_f, and sends the encrypted result, c_i, to the FPGA.

4. FPGA uses d_f and n_f to decrypt c_i and retrieve m_i.

5. FPGA uses e_i and n_i to encrypt m_i and sends the encrypted result, c_{f1}, to the iButton.

6. iButton decrypts c_{f1} using d_i and n_i and confirms that the FPGA has correctly returned m_i. If so, the FPGA has authenticated itself to the iButton, and the process

continues. If not, the iButton does not respond to the FPGA and waits to be disconnected.

7. FPGA generates a random numbner, m_f, encrypts it using e_i and n_i, and sends the encrypted result, c_{f2}, to the iButton.

8. iButton uses d_i and n_i to decrypt c_{f2} and retrieve m_f.

9. iButton encrypts the value of (UID+$m_i + m_f$) using e_f and n_f and sends the encrypted result, c_{UID}, to the FPGA. The UID is not encrypted directly, or the UID packet would be the same for every authorization session with that iButton.

10. FPGA uses d_f and n_f to decrypt c_{UID} and retrieves UID by subtracting m_i and m_f from the decryption result.

11. FPGA runs UID through the SHA hash and treats the result, h_{UID}, as the iButton's certificate.

12. FPGA checks h_{UID} against the authorized certificates in its certificate table. If it finds a match, the iButton has been authenticated to the FPGA. If not, the iButton represents an invalid user, and the FPGA waits for the iButton to be disconnected.

When the process is complete for a valid user, the iButton has been authenticated to the FPGA and the FPGA to the iButton. The FPGA now knows the UID of the user, and it sends that UID to the EMC to create the final key for the DES engine.

5.3 Encrypted Memory Controller

The EMC, illustrated in Figure 5.2, sits in between the embedded application within the FPGA and the external memory outside it. It encrypts and decrypts every transaction between them. Upon startup, the EMC uses a secret DES key that is unique to and known only to the FPGA. This key is used only in the creation of the final DES key — never for

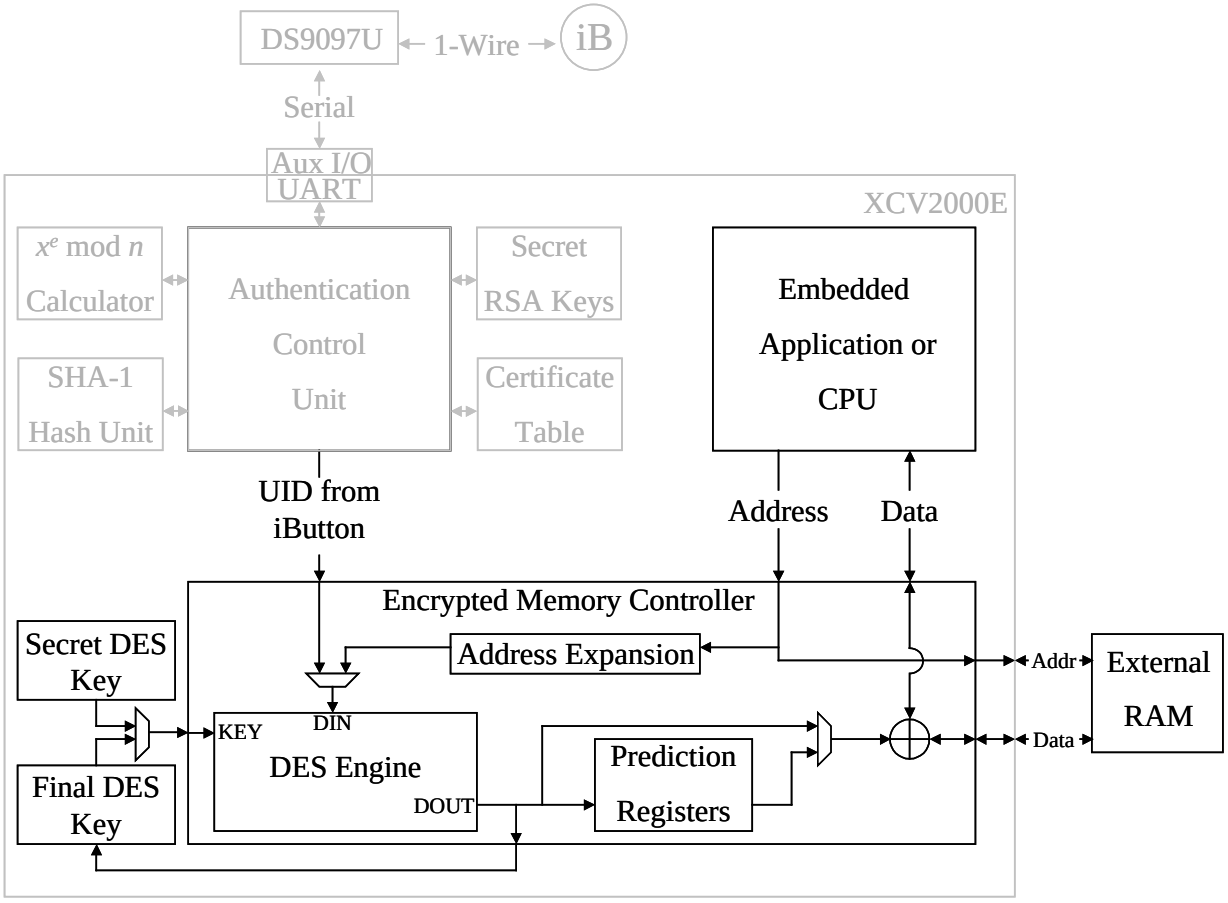

Figure 5.2: Block diagram of the encrypted memory controller and its connections

actually encrypting the external memory interface. The final DES key used for encrypting the external memory interface is formed by passing the UID from the iButton through the DES engine using the secret DES key. The first 56 bits of the 64-bit result of this operation are used as the final DES key. This is an example of the adopted philosophy of keeping the information necessary to crack the entire system separated between the iButton and the FPGA. Secret pieces of information from both the iButton and the FPGA are required to calculate the actual key used to encrypt the memory interface that protects the embedded

application data.

A cryptographer may question here whether the DES result of the UID and the secret key will always create a strong DES key as the final key. Differential cryptanalysis of the DES algorithm could be aided significantly if the final key is in the set of known weak and semi-weak DES keys [15]. The reality is that it is a matter of probability. There are 4 known weak keys and 12 known semi-weak keys in the overall set of 2^{56}, or 72,057,594,037,927,936, total DES keys. If we think of the final key as a random combination from the UID and the secret key, the probability of randomly choosing a weak key is

$$\frac{16}{2^{56}} = \frac{2^4}{2^{56}} = 2^{-52}. \tag{5.7}$$

Because it's fun to see big numbers in print, that's one chance in 4,503,599,627,370,496. Since this makes it highly unlikely that a weak key will ever be the DES result of the UID and the secret key, weak keys are not checked for in the Amanuet architecture. If the key distributor wants to address the problem of weak keys, they should ensure that no UID and secret key combination produces a weak key. If, as is recommended, the DES algorithm is replaced by a stronger keyed block cipher, a weak key analysis will need to be completed for that cipher as well.

5.3.1 Encrypt and decrypt operations

Once the final DES key is calculated, the EMC is ready for encrypted read and write operations to and from the external memory, and the embedded application or CPU is signaled to begin operation. The encryption method used by the EMC is related to DES counter-mode encryption [41]. In DES counter-mode encryption, a counter is placed as the input data to the DES engine, and the DES-encrypted result is used in an XOR operation with the data to be encrypted. Decryption in counter mode is similar. To decrypt, the same counter value is encrypted by the DES engine using the same key, and this number is used in an XOR operation with the encrypted data to retrieve the original data. This makes use of the

following property of the XOR function:

$$A \oplus B = C \Rightarrow B \oplus C = A. \tag{5.8}$$

In the case of DES counter-mode encryption, A is the original value, B is the DES encrypted counter value, and C is the counter-mode encrypted result.

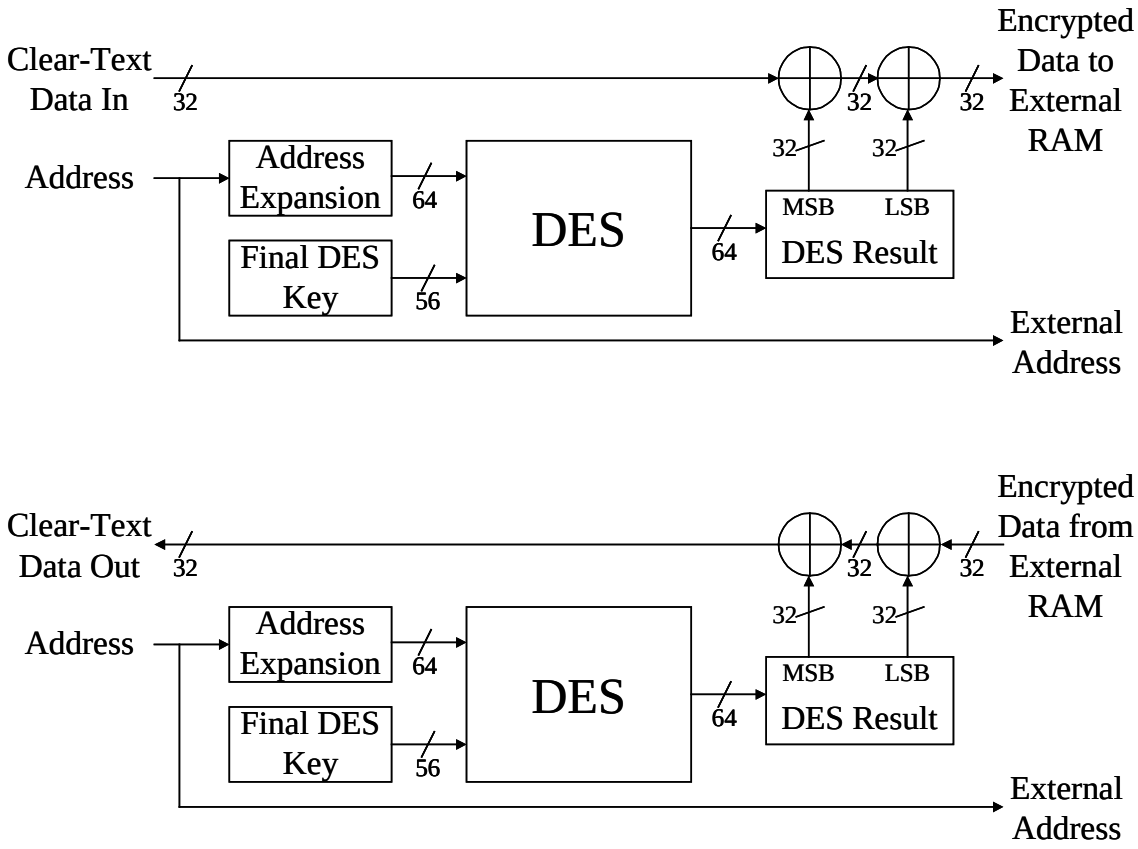

Figure 5.3: Encrypt-and-write (top) and read-and-decrypt (bottom) operations

The Amanuet encrypt-and-write and read-and-decrypt operations, illustrated in Figure 5.3, are similar. When an address and a 32-bit data word are written to the EMC from the

embedded application, the address is expanded using a 1-to-1 function to 64 bits and used as the input to the DES engine, using the final DES key as the key. The result of this is a 64-bit encrypted value. The 32-bit data word then goes through two XOR operations, one with the most significant 32 bits of the 64-bit DES result, the other with the least significant 32 bits. The result of these operations is an encrypted 32-bit number that is written to the original address value in the external memory. The original address bus width is not specified since the same method could be used for any size of address bus. The read-and-decrypt operation is exactly the same process for the DES engine, but the 32-bit input data going through the two XOR operations is the encrypted data from the external RAM. The output is the clear text 32-bit data value requested by the embedded application. Extending the XOR relationship of counter-mode encryption, the Amanuet architecture states for encryption and decryption that

$$A \oplus B \oplus C = D \Rightarrow B \oplus C \oplus D = A, \tag{5.9}$$

where A is the original clear text value, B represents the most-significant 32-bits of the DES result, C represents the least-significant 32-bits of the DES result, and D is the encrypted value that is written to the external memory.

Note that for both the encrypt-and-write and read-and-decrypt operations, only the DES encryption function is required, since the same DES result is used both for encryption and decryption of the values written to memory at any given address. The DES decryption function is never used. Because decryption and encryption in the DES algorithm use the same computational structures in the FPGA, the lack of need for the decryption portion does not result in any space savings on the FPGA.

5.3.2 Address Expansion

The address expansion unit in the Amanuet architecture can be designed with any 1-to-1 function that expands the original address to a 64-bit value. The extra bits beyond the original address size can be filled in with a replication of the address bits, parity bits for part or all of the original address, or some mathematical combination of the address bits.

To ensure the fullest possible range of DES outputs, the expansion function must be 1-to-1, meaning that each input has a unique expanded output. If the expansion algorithm were kept secret and included some cryptographic calculations, it could be used to increase the security of the overall system.

In the Amanuet architecture implemented on the prototype system, the address expansion unit was optimized for speed, meaning that no arithmetic computations were performed on the input bits. In this way, it becomes a permutation from the 21-bit original address to the 64-bit DES input, as illustrated in Table 5.2.

Table 5.2: Address Expansion Permutation

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	1	2	3	4	5	6	7	8	9	10	11

5.3.3 Prediction Registers

There is one layer of complexity in the Amanuet system that is not illustrated in Figure 5.3. As shown in Figure 5.2, there is a set of prediction registers in the EMC that hold DES result values for memory addresses predicted to be the next accessed. The reason for having the prediction registers is that the DES engine has a latency of sixteen clock cycles between the time the first value is clocked into the DES input and the time the result is present on the DES output. However, since the DES engine is a pipeline, it can output a new result on each clock cycle after the initial sixteen-cycle latency. Keeping in mind that a memory controller needs to run at the highest possible speed, the pipelined nature of the DES engine makes it expedient to continue running predicted addresses through the DES engine after the requested address has been processed. Mathematically, the time, t_{DES}, in clock cycles, that it takes to calculate a given number of pipelined address values, n_{ADDR}, can be described as

$$t_{DES} = 16 + (n_{ADDR} - 1) \tag{5.10}$$

In the Amanuet architecture, we expect that the behavior of the embedded application will follow the principle of locality of reference, meaning that the next address referenced will be very near the most recent address referenced. Following this guideline, when an address is run through the DES engine, the prediction registers are filled with DES results for the sixteen addresses above and the sixteen addresses below the currently requested address. Upon the next address reference, the address is checked to see if it is in the range of addresses whose DES results already reside in the prediction registers. If it does, the pre-calculated DES result is retrieved from the prediction register in one clock cycle, and the DES engine is not used. The sixteen-cycle latency is avoided. If the address does not fall in the range of the prediction registers, the DES engine is used to calculate the DES result, and the prediction registers are filled with the DES results for the addresses above and below this new address.

The prediction registers are designed to be filled without delaying the rest of the operation of the EMC. If a data word is read from or written to an address not in the prediction registers, once the DES result is calculated for the requested address, the prediction registers are filled independently while the rest of the read or write cycle takes place.

If the Amanuet architecture were to be modified for a more complex RAM system that includes a cache, the locality of reference principle is made slightly more complex. The prediction registers should hold the DES results for the data held in the cache lines. This new design would have to take into account more complex cache functions such as line flushes while strictly maintaining valid/invalid and clean/dirty information for each cache line. Similarly, in systems where the behavior of the embedded application could be easily predicted, the principle guiding the filling of the prediction registers should be modified accordingly.

5.4 Vulnerability Analysis

The Amanuet architecure's vulnerability to probing and cryptanalysis has been discussed throughout the text of this thesis, and the threats to its security are discussed generally in this section and more specifically in the next. An authorized user — that is, user with an authorized iButton — can use the iButton to start the embedded application on the FPGA

but can gather neither the value of their own authorized UID from the iButton nor the values of the data in RAM external to the FPGA without breaking RSA and DES, respectively. Recall that RSA's strength is in the difficulty of factoring its modulus, n, so a security analysis would need to be done prior to deploying the system to determine an appropriate size for n. It has been shown that DES is becoming increasingly vulnerable to differential cryptanalysis as computers become more adept at solving large numerical analysis problems [22]. As mentioned, a major advantage of the Amanuet architecture is that any stronger keyed block cipher, such as counter-mode Advanced Encryption Standard [21] or Triple DES [42], could replace the DES engine in this prototype. FPGAs add a higher degree of security by allowing cryptographic updates as needed to face new security threats.

If the physical security measures in the iButton, mentioned in Chapter 4, were broken and the contents of the iButton were known, an attacker would know an authorized UID, both RSA moduli, the iButton's decryption key, and the FPGA's encryption key. The attacker could then potentially negotiate an unauthorized session on the FPGA; however, the memory interface and the details of the embedded application data would still be secure due to the secret DES key's role in creating the final DES key from the UID. Only an attack that intercepts the contents of the FPGA would unravel the entire key management scheme. If the contents of the FPGA were known, the attacker would know the proper moduli and RSA keys to setup an unauthorized session with the iButton to steal the UID. The secret DES key would be known as well, allowing the attacker to construct the final DES key and analyze the memory interface. The data for all authorized iButtons would still be safe, however, since the certificate table only stores the SHA hash of the valid iButton UIDs. It is important to recognize that knowing the contents of the FPGA would also completely reveal the details of the embedded application. It is assumed in the scheme presented that it is not possible to know the contents of the FPGA, and, as discussed in Chapter 2, much work has been done by others in this area.

5.5 Specific Threat Analysis

When defining the security policy for the Amanuet architecture, it simplifies the discussion to visualize the architecture as consisting of nodes and interconnects. The FPGA and the iButton are the nodes, which are only trusted after authentication, while the authentication interface and the memory interface are the interconnects, which are never trusted. The RAM itself must also be thought of as an interconnect, since it is never authenticated like a node and its contents can be viewed. In the following discussion, the RAM and the memory interface are considered to be part of the same interconnect.

Attacks carried out on the interconnects include bus snooping attacks to either try to mathematically determine the contents of the communications over the interconnect or record the communications in order to mimic the behavior of a trusted node in the future in order to gain unauthorized access to another node. These attacks on the interconnect are discussed in Sections 5.5.1 and 5.5.2. Possible attacks on the nodes include malicious or poorly-designed embedded applications that reveal details of the Amanuet architecture or multitudinous varieties of physical attacks on the FPGA or iButton architectures. These are discussed in Sections 5.5.3 and 5.5.4.

5.5.1 Mathematical Attacks

Thus far in this thesis, the mathematical strength of the Amanuet architecture has been discussed in more detail than any other kind of protection against attack. The purpose of a mathematical attack on the authentication interface would be to determine the encryption and decryption keys and modulii. While the authentication scheme is protected by an algorithm that uses modular arithmetic in ways that mimic RSA encryption, the Amanuet algorithm differs in that it does not make public the encryption exponents or modulii of either party. Thus, the most effective mathematical attack against RSA – factoring the encryption modulus, which, as has been discussed, is incredibly difficult – cannot be used against the Amanuet architecture since the modulii are kept secret.

A mathematical attack against the memory interface would be carried out for the purpose

of learning the secret final DES key to decrypt the contents of the external RAM. As was mentioned in Chapter 2, determining the DES key by searching the entire DES key space is now possible in a matter of days through distributed and cluster computing approaches. This is why it is suggested throughout this thesis that the DES algorithm is only intended to prove the Amanuet architecture in concept while the AES algorithm is recommended for practice. The AES algorithm is not known to be vulnerable to differential cryptanalysis and has a much larger key space – up to 256 bits compared to DES's 56 bits – making it invulnerable, at least for now, to modern key space analysis algorithms [16].

5.5.2 Replay Attacks

A replay attack could be attempted on the authentication interface by recording a session between the FPGA and the iButton. The attacker could then in the future use the data recorded from the iButton to pretend to be an authorized iButton and attempt unauthorized access to the FPGA. This kind of attack is foiled, however, by steps 7–9 of the authentication algorithm described in Section 5.2.2, wherein the FPGA sends a random number challenge to the iButton. That random number becomes part of the UID package that the iButton sends to the FPGA. Without knowing the correct decryption key to determine the random number from the FPGA, the correct encryption keys to re-package the random number to be sent back, and the UID to gain access, an unauthorized user could not fake a session on the FPGA. Thus, a replay attack mimicking the iButton would be ineffective. A similar attack attempting to parody the FPGA to fake a session with a real iButton would also fail, due to the random number challenge the iButton presents to the FPGA in steps 3–6 of the authentication process.

Security experts such as Schneier [15] recommend that authentication algorithms include random values from both authenticating parties for the express purpose of preventing replay attacks. While the Amanuet authentication scheme is protected in this manner, it can be seen that in most circumstances, in order to record a session and perform a replay attack an authorized iButton and a fully-configured FPGA would both be required. If both of these components were in hostile hands already, a replay attack would hardly be necessary (device theft is discussed further in Section 5.5.4). The case of a clever replay attack must

be considered, however, wherein the FPGA is secretly probed and remotely monitored. This would not require the possession of an iButton. Also, the Amanuet architecture may be extended in the future to include remote authentication, where the FPGA and the iButton are distant physically but connected via some insecure data network. The authentication interconnect must be distrusted in this case, so the authentication algorithm is designed for this eventuality.

5.5.3 Malicious Application Attacks

A poorly or maliciously written embedded application may run the risk of betraying the contents of protected sections of memory. In one example attack, if an application is known to an attacker to write zero's to certain memory locations, the attacker would see directly the XOR key for that memory location, since any value XORed with zero is the original value. Once the key was known, any other value written to that memory location would also be known to the attacker. If enough of these memory locations were known, the secret algorithmic information that the Amanuet architecture purports to protect would be vulnerable. The logic of a malicious application attack is somewhat flawed when considering that the algorithmic information – specifically, which values are written to which memory locations – needs to be known in the first place to carry out an attack intended to steal algorithmic information. If the embedded application designer makes public their algorithmic information prior to protecting it with the Amanuet wrapper, the Amanuet architecture will not be effective. A prerequisite of using the Amanuet architecture to protect an embedded application is that the embedded application details be kept secret.

5.5.4 Physical Attacks

Physical attacks are carried out on the architecture of the iButton or FPGA in order to reveal keys or algorithmic secrets. Tampering attacks involve direct analysis of the architectures, while power attacks are classified as "side channel" attacks – attacks that analyze attributes of the architecture that are not typically thought of as possibly revealing data [43]. Finally, device theft is the simplest form of physical attack: just steal the devices necessary to gain

authorized access. Discussions on all of these attacks follow.

Device Tampering

Tampering with the iButton would simply consist of trying to open its case and analyze the NVRAM to extract the UID and secret key values. However, as was mentioned Section 4.3, the NVRAM within the iButton is connected to a device that will destroy the memory contents if the iButton's case is opened. Advances in token technology will likely make such tamper resistance even more robust in the future. For the purposes of the Amanuet architecture, the iButton is considered to be a tamper-proof token.

Tampering with the FPGA can be much more involved. For example, an advanced method such as a Focused Ion Beam (FIB) analysis could be used to visually inspect the architecture and programming of the FPGA. While incredibly painstaking, it is theoretically possible to determine the function of each programmable element in the FPGA with this method. However, there has not yet been a documented successful reverse engineering of an FPGA design using FIB analysis [43].

Power Attacks

Power analysis attacks – usually referred to as Simple or Differential Power Analysis (SPA or DPA) – can also be used to characterize the values stored in the programmable elements within an FPGA. The attacks are usually carried out on FPGA-based cryptographic algorithms to determine where the key value is stored on the FPGA. The power consumption of various locations on the FPGA are analyzed, and candidates are chosen as possible locations of the key. This is intended as a means of more quickly sifting through possible key values in the key space. It is possible that this could be used against both the authentication and memory interfaces of the Amanuet architecture; however, as advances in SPA and DPA are made, there are also advances in countermeasures. Structures have been designed for use at the key storage locations that de-correlate the key storage location from a pattern of power consumption that would reveal its location to SPA or DPA [44]. This field is still developing, and it is essential for any designer who wishes to encapsulate secret information within and

FPGA to follow the balance of power between power analysis and its countermeasures.

Device Theft

Finally, the theoretically simplest attack against the Amanuet architecture would be to simply steal an iButton to gain unauthorized access to an FPGA. The presence of the iButton in the authentication system makes it easy to disable access to the system. Remember the example of the EP-3 plane from Chapter 1. The Navy personnel could have ditched the iButtons in weighted bags prior to landing, as per Navy regulations. The problem with this simple approach is that the simple possession of an iButton makes a user authorized. Work has been done in related architectures to require both an iButton and a biometric authentication method, such as a fingerprint, prior to accessing a secured FPGA application [33] [45]. The biometric in these systems is tied to specific iButtons, meaning that an authorized user must be biometrically authenticated to the iButton before attempting an authentication session with the FPGA. This increases the complexity of the authorization algorithm and the iButton distribution process, but it ensures that users must be who their iButton claims they are before they can access the FPGA. If device theft is considered to be a great threat to the embedded application protected by Amanuet, a similar biometric authentication mechanism could be integrated into the Amanuet architecture.

Chapter 6

Experiments

6.1 Overview

To prove the viability of the concepts that make up the Amanuet architecture, a proof-of-concept architecture was developed on the RC1000 platform, and an array of experiments was performed on it. Section 6.2 explains this proof-of-concept architecture and discusses the embedded application that was used in the experiments. Section 6.3 discusses the performance of the system with and without encryption and with and without prediction registers.

6.2 Proof-of-Concept Architecture

A block diagram of the proof-of-concept architecture is shown in Figure 6.1. The system was realized using the RC1000 platform's Xilinx VirtexE XCV2000E FPGA, a DS9097U Serial to 1-Wire protocol adapter, and a DS1957B Java iButton. The architecture inside the FPGA was developed in VHDL, and several relevant sections of the code are included in Appendix A. The final architecture in the figure required 4,202 of the available 19,200 slices of the XCV2000E, leaving plenty of logic on the FPGA for larger embedded applications.

The proof-of-concept architecture represents a basic implementation of every major func-

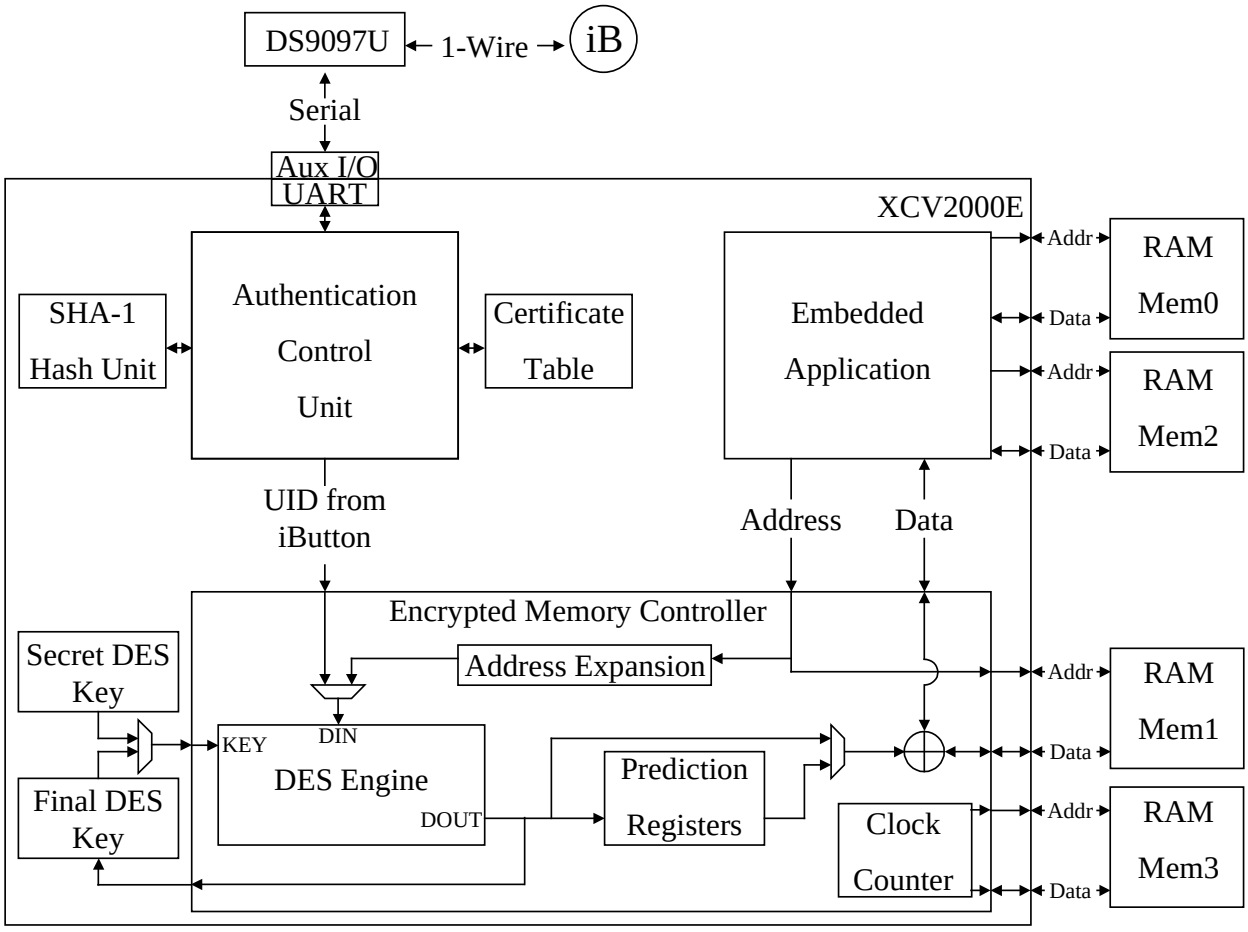

Figure 6.1: Block diagram of the proof-of-concept architecture

tional unit in the Amanuet architecture. The processes in the architecture are triggered by the authentication control unit. The ACU starts by negotiating the transfer of a user identification number from the Java iButton. In this architecture version, the UID is simply the 64-bit serial number of the Java iButton. To increase the security of the UID, it is recommended that a value larger than 64-bits should be used as the UID in full versions of the Amanuet architecture.

Once the UID has been received in the authentication control unit, it is passed through the SHA-1 hash unit. The SHA hash unit in this system was coded to accept and pad the the message – the 64-bit serial number – and produce a correct SHA digest of the message. The message digest is treated as the user's certificate, and the certificate table is checked to determine if the user has access to the embedded application. If a match is found in the certificate table, the user is authorized, and the UID is passed to the DES engine in the encrypted memory controller. If there is no match, the authentication fails, and the user is not allowed to run the embedded application. To produce the final DES key, the DES engine operates on the UID using the secret DES key. The result becomes the final DES key, which is used to encrypt the memory. Once the final DES key is in place, the embedded application is given permission to run.

The embedded application that was used to test the architecture is a simple memory transfer application. At startup, the unencrypted RAM bank Mem0 is loaded with 100,000 numbers by a DMA transfer from the PC. When the embedded application is triggered to start, it reads one number at a time from Mem0 and directs the EMC to encrypt each number and write it to Mem1. Once all 100,000 numbers have been written to Mem1, the embedded application instructs the EMC to read and decrypt each number from Mem1 one at a time. As they are decrypted, they are written to Mem2. Once the values are written to Mem2, the embedded application is finished.

The RAM banks on the RC1000 can be monitored from the PC using a diagnostic program provided by Celoxica. From the PC's perspective, Mem0 occupies the address space from 0x000000 to 0x1FFFFF, Mem1 occupies 0x200000 to 0x3FFFFF, Mem2 occupies 0x400000 to 0x5FFFF, and Mem3 occupies 0x600000 to 0x7FFFF. Using the diagnostic program, the values in Mem0 starting at address 0x000400 – a subset of the 100,000 numbers – can be examined as follows.

```
> dm 0x400

    0x00000400  :  00000100  00000101  00000102  00000103
    0x00000410  :  00000104  00000105  00000106  00000107
    0x00000420  :  00000108  00000109  0000010a  0000010b
    0x00000430  :  0000010c  0000010d  0000010e  0000010f
```

```
0x00000440  :  00000110  00000111  00000112  00000113
0x00000450  :  00000114  00000115  00000116  00000117
0x00000460  :  00000118  00000119  0000011a  0000011b
0x00000470  :  0000011c  0000011d  0000011e  0000011f
0x00000480  :  00000120  00000121  00000122  00000123
0x00000490  :  00000124  00000125  00000126  00000127
0x000004a0  :  00000128  00000129  0000012a  0000012b
0x000004b0  :  0000012c  0000012d  0000012e  0000012f
0x000004c0  :  00000130  00000131  00000132  00000133
0x000004d0  :  00000134  00000135  00000136  00000137
0x000004e0  :  00000138  00000139  0000013a  0000013b
0x000004f0  :  0000013c  0000013d  0000013e  0000013f
```

The corresponding address locations in Mem1 show these same values after they have been encrypted.

```
> dm 0x200400

    0x00200400  :  a03cccfc  8c375ea2  901ffea9  f56c2300
    0x00200410  :  8d138693  21d3acc6  8058f16b  8b95440c
    0x00200420  :  83d8a604  e1d675db  cf7b1c15  8810f985
    0x00200430  :  194daeb7  58304dd8  aeac82d9  32cbc3ce
    0x00200440  :  dbfb0a05  d4cc18c4  b81e8fd1  97666ef2
    0x00200450  :  edcda5d3  08d0c51b  0bcd219e  a3c89692
    0x00200460  :  78f6dcf2  a942ea38  d74b1f12  2178369b
    0x00200470  :  f046b966  cb6a2ba0  8bbc9c3e  9b29c4e9
    0x00200480  :  01de88a4  519c7f01  9cf09630  4a4e0f23
    0x00200490  :  257f9a81  42a94460  f65be41a  a4cd4a85
    0x002004a0  :  bce58982  1e9c706e  5519e1df  5a10dfef
    0x002004b0  :  9eda511d  92165ad7  3bfd37c1  241dda2b
    0x002004c0  :  7a394e14  2557c36c  5e4d2d54  c5ecacee
    0x002004d0  :  9ae3aef1  b1187e6f  a98db86f  39d3a86a
    0x002004e0  :  09efa6f9  36516a7e  6dd31506  c5e4916e
    0x002004f0  :  14785a55  e8f3ca6f  546ec3e9  cf32fe84
```

Since these values are all XORed with different DES results, note that no two are similar. Despite the fact that the numbers in this embedded application are stored in sequential order

and encrypted with sequential address values, the value that is actually stored at any given address location is still obfuscated. Finally, Mem2 shows the corresponding decrypted result of reading these values from Mem1 and writing them to Mem2.

```
> dm 0x400400

    0x00400400  :  00000100  00000101  00000102  00000103
    0x00400410  :  00000104  00000105  00000106  00000107
    0x00400420  :  00000108  00000109  0000010a  0000010b
    0x00400430  :  0000010c  0000010d  0000010e  0000010f
    0x00400440  :  00000110  00000111  00000112  00000113
    0x00400450  :  00000114  00000115  00000116  00000117
    0x00400460  :  00000118  00000119  0000011a  0000011b
    0x00400470  :  0000011c  0000011d  0000011e  0000011f
    0x00400480  :  00000120  00000121  00000122  00000123
    0x00400490  :  00000124  00000125  00000126  00000127
    0x004004a0  :  00000128  00000129  0000012a  0000012b
    0x004004b0  :  0000012c  0000012d  0000012e  0000012f
    0x004004c0  :  00000130  00000131  00000132  00000133
    0x004004d0  :  00000134  00000135  00000136  00000137
    0x004004e0  :  00000138  00000139  0000013a  0000013b
    0x004004f0  :  0000013c  0000013d  0000013e  0000013f
```

6.3 Performance

With the correct operation of the authentication control unit and the encrypted memory controller established, experiments were constructed to determine the degradation in memory performance that results from encrypting and decrypting before each write and read cycle. To monitor the performance, a clock counter unit was installed in the encrypted memory controller to determine the summation of all the clock cycles spent in the memory controller when it was actively encrypting, decrypting, reading, and writing. This clock counter and its interface to Mem3, where the final clock count is written, can be seen in Figure 6.1. Several versions of the encrypted memory controller were created to test the its performance using different EMC configurations. Figure 6.2 summarizes the results of these experiments.

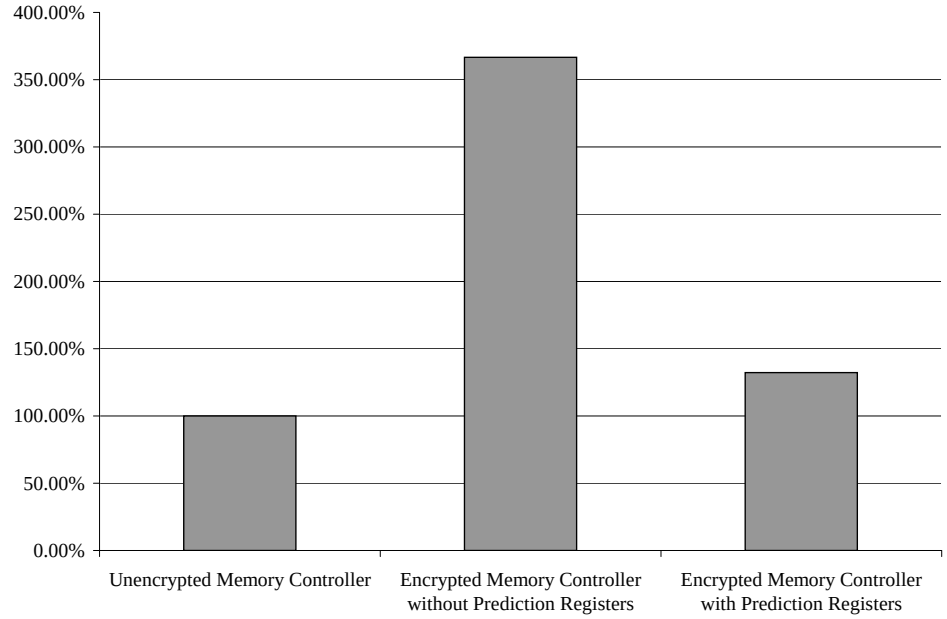

Figure 6.2: Time spent with controller active relative to unencrypted memory controller

When looking at the logic and considering clock speed instead of clock cycle quantity, the delay related to the XOR operations – required to encrypt and decrypt on writes and reads, respectively – must be considered. These XOR operations will be realized in look-up tables (LUTs) on the FPGA. For high-speed memory architectures, the impact of the logic delay through the LUT on the maximum clock speed of the memory controller will need to be considered. In the following experiments, performance is measured solely in terms of the number of required clock cycles.

The first EMC configuration that was tested this way was a stripped down version that actually lacks both the prediction registers and the DES engine – it is effectively an unencrypted memory controller. During the process of running the embedded application, this controller

was active for 1,200,018 clock cycles[1]. This number is proportional to the time needed by the read and write processes during the operation of the embedded application.

The second configuration that was constructed was an EMC that includes the DES engine but lacks the prediction registers. Every time a new address and data value is written to this EMC configuration, the entire 16-cycle pipeline penalty of the DES engine is incurred, making it quite slow. The entire pipeline penalty is incurred again with each read. During the operation of embedded application, this EMC was active for 4,400,051 clocks – 3,200,033 of which can be attributed to DES encryption- and decryption-related delays. Over the course of the 200,000 combined read and write cycles required by the embedded application, this works out to almost exactly 16 cycles per read/write operation.

The final[2] configuration that was tested was the EMC that includes both the DES engine and the prediction registers as pictured in Figure 6.1. The prediction registers in the proof-of-concept architecture were tuned specifically to enhance the performance of this particular embedded application. Sixteen registers were used. The first register holds the DES encryption result of the address that was requested; the remaining hold the DES encryption results for the 15 address locations above the currently requested address. Recall from the discussion in Chapter 5 that calculating the next addresses allows the EMC to incur the 16-cycle pipeline penalty once, then receive a new DES result on each clock. Succeeding memory references to address values whose DES results are already stored in the prediction registers only incur the single clock cycle penalty required to access that address's DES result from the register.

This advantage is clearly seen in the reduction of clocks spent with the EMC active during the operation of the embedded application. This version required only 1,587,581 clocks – only 387,563 more than the unencrypted controller – meaning it incurred an average encryption- and decryption-related penalty of about 2 clocks per read/write cycle. The penalty is this low due to the fact that the prediction registers perfectly predict the next addresses to be accessed by this specific embedded application. In more of a real-world application, the per

[1]Since the values are read one at a time from Mem0 and written one at a time to Mem1, no burst mode memory writes were used during this process.

[2]This final version is the one that produced the correct encryption results presented in the previous section.

read/write cycle penalty would be higher; however, to counter this, the prediction mechanism in the prediction registers can be adjusted to match the application.

Chapter 7

Conclusion

This thesis has presented the Amanuet architecture as a key management framework for creating a secure user authentication system and an encrypted off-chip data transfer method on an FPGA. It has been shown that the architecture can behave like a secure wrapper to hide the algorithmic details of the device it is protecting by preventing the analysis of all external device interfaces. While the framework is secured using modified versions of known cryptographic algorithms, the Amanuet architecture has introduced a secure token-based authentication scheme and an FPGA-based encrypted memory controller that includes the application of counter-based block cipher encryption. The strengths of the Amanuet architecture lie in its ability to secure a wide variety of embedded FPGA applications. Any embedded FPGA application with address and data memory buses can be dropped into the Amanuet architecture without modifying the application. Since the Amanuet architecture takes up very little room in the FPGA, the embedded application can be complex. Memory performance degradation is a drawback to the architecture, but the performance has been shown to improve with proper prediction register design.

The experiments that have been performed on the proof-of-concept system have shown the system to be viable and ready for application to real-world embedded projects. A complete realization of the encrypted memory controller in an FPGA proved to have a negative impact on memory performance that was reduced due to a specifically-targeted prediction system. When future embedded applications are paired with the Amanuet architecture, the prediction

mechanism will be a crucial factor in application performance. This system is currently well-suited to protect embedded applications and algorithms that will be deployed publicly under the scrutiny of competitive or hostile entities, and it is extensible to encompass both future advances in block cipher algorithms to keep up with new security threats and various digital I/0 interface structures for use in more advanced FPGA architectures.

Ongoing work with this architecture includes implementing it on a Virtex-II Pro FPGA for use with an embedded system running Linux. This will require the prediction register system to be updated to integrate with a much more advanced memory cache system, a topic breached in Chapter 5. Additionally, further research will be performed on possible replacements for the DES algorithm in the encrypted memory controller to better secure the data in external memory. The Amanuet architecture is currently in its infancy, but the strength of its concepts and its adaptability to new encryption algorithms gives it a bright future with many potential applications.

Appendix A

VHDL Samples

Since the body of VHDL code that makes up the proof-of-concept architecture is around 1500 lines long, only two samples are included in this appendix. The first is the SHA-1 hash code, presented in its entirety in Section A.1. The second is the process that makes up the the encrypted memory controller, shown in Section A.2. The DES engine used is the Free DES engine provided by The Free IP Project and David Kessner at http://www.free-ip.com.

A.1 VHDL SHA-1 Hash

```
-- SHA.VHD

-- A limited version of the SHA-1 algorithm
-- Only accepts 64-bit messages, but very easy to adapt for larger messages
-- By Jonathan Graf
-- Start/Finish June 10, 2004

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
```

```
entity sha is
    port(
        clk         :   in std_logic;
        reset       :   in std_logic;

        message     :   in std_logic_vector(63 downto 0);
        message_vld :   in std_logic;

        digest      :   out std_logic_vector(159 downto 0);
        digest_vld  :   out std_logic
        );
end sha;

architecture limited_sha of sha is

    signal  run_hash                : std_logic;
    signal  w0,w1,w2,w3,w4,w5,w6, w7, w8, w9, w10,w11,
            w12,w13,w14,w15,w       : std_logic_vector (31 downto 0);
    signal  a, b, c, d, e, Kt, temp : std_logic_vector (31 downto 0);
    signal  count                   : std_logic_vector (4 downto 0);
    signal  stage                   : std_logic_vector (1 downto 0);
    type    sha_state_type is (setup, calc, done);
    signal  sha_state               : sha_state_type;

    signal   m0, m1 : std_logic_vector(31 downto 0);
    constant m2     : std_logic_vector(31 downto 0)
                    := "10000000000000000000000000000000";
                    -- appended a 1, then zeroes
    constant m3     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m4     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m5     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m6     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m7     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m8     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
```

```
    constant m9     : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m10    : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m11    : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m12    : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m13    : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m14    : std_logic_vector(31 downto 0)
                    := "00000000000000000000000000000000";
    constant m15    : std_logic_vector(31 downto 0)
                    := "00000000000000000000001000000000";
                    -- 64-bit representation of the length of the string

    constant Ka     : std_logic_vector(31 downto 0)
                    := "01100111010001010010001100000001"; --X"67452301";
    constant Kb     : std_logic_vector(31 downto 0)
                    := "11101111110011011010101110001001"; --X"efcdab89";
    constant Kc     : std_logic_vector(31 downto 0)
                    := "10011000101110101101110011111110"; --X"98badcfe";
    constant Kd     : std_logic_vector(31 downto 0)
                    := "00010000001100100101010001110110"; --X"10325476";
    constant Ke     : std_logic_vector(31 downto 0)
                    := "11000011110100101110000111110000"; --X"c3d2e1f0";

    constant Kt0    : std_logic_vector(31 downto 0)
                    := "01011010100000100111100110011001"; --X"5a827999";
    constant Kt20   : std_logic_vector(31 downto 0)
                    := "01101110110110011110101110100001"; --X"6ed9eba1";
    constant Kt40   : std_logic_vector(31 downto 0)
                    := "10001111000110111011110011011100"; --X"8f1bbcdc";
    constant Kt60   : std_logic_vector(31 downto 0)
                    := "11001010011000101100000111010110"; --X"ca62c1d6";

begin

    start_signal : process (clk, reset)
    begin
```

```
    if (reset = '1') then
        m0 <= (others => '0');
        m1 <= (others => '0');
        run_hash <= '0';
    elsif (clk'event and clk = '1') then
        if (message_vld = '1') then
            m0 <= message(63 downto 32);
            m1 <= message(31 downto  0);
            run_hash <= '1';
        end if;
    end if;
end process;

HASH : process (clk, reset)
begin
    if (reset = '1') then
        a <= Ka; b <= Kb; c <= Kc; d <= Kd; e <= Ke;
        w0 <= (others => '0');
        w1 <= (others => '0');
        w2 <= m2; w3 <= m3; w3 <= m3; w4 <= m4; w5 <= m5; w6 <= m6; w7 <= m7;
        w8 <= m8; w9 <= m9; w10 <= m10; w11 <= m11; w12 <= m12; w13 <= m13;
        w14 <= m14; w15 <= m15;
        w  <= (others => '0');
        Kt   <= Kt0;
        temp <= (others => '0');
        count <= "00000";
        stage <= "00";
        sha_state <= setup;
        digest <= (others => '0');
        digest_vld <= '0';

    elsif(clk'event and clk = '1') then
        if (run_hash = '1') then
            case sha_state is
            when setup =>
                if (count < "10000" and stage = "00") then
                    case count is
                        when "00000" => w <= m0; w0 <= m0; w1 <= m1;
                        when "00001" => w <= w1;
                        when "00010" => w <= w2;
```

```
                when "00011" => w <= w3;
                when "00100" => w <= w4;
                when "00101" => w <= w5;
                when "00110" => w <= w6;
                when "00111" => w <= w7;
                when "01000" => w <= w8;
                when "01001" => w <= w9;
                when "01010" => w <= w10;
                when "01011" => w <= w11;
                when "01100" => w <= w12;
                when "01101" => w <= w13;
                when "01110" => w <= w14;
                when "01111" => w <= w15;
                when others  => w <= (others => '1');
            end case;
        else
            w <= w13 XOR w8 XOR w2 XOR w0;
        end if;

        case stage is
            when "00" =>    Kt <= Kt0;
            when "01" =>    Kt <= Kt20;
            when "10" =>    Kt <= Kt40;
            when "11" =>    Kt <= Kt60;
            when others =>  Kt <= Kt0;
        end case;

        sha_state <= calc;

    when calc =>
        e <= d;
        d <= c;
        c <= b(1 downto 0) & b(31 downto 2);
        b <= a;

        case stage is
            when "00" =>
                a <=    (   a(26 downto 0) & a(31 downto 27)    )   +
                        (   (b and c) or ((not b) and d)        )   +
                        (   e + (w(30 downto 0) & w(31)) + Kt   );
```

```
            if (count = "10011") then stage <= "01"; end if;
            sha_state <= setup;

        when "01" =>
            a <=    (   a(26 downto 0) & a(31 downto 27)    )   +
                    (   b xor c xor d                       )   +
                    (   e + (w(30 downto 0) & w(31)) + Kt   );
            if (count = "10011") then stage <= "10"; end if;
            sha_state <= setup;

        when "10" =>
            a <=    (   a(26 downto 0) & a(31 downto 27)    )   +
                    (   (b and c) or (b and d ) or (c and d))   +
                    (   e + (w(30 downto 0) & w(31)) + Kt   );
            if (count = "10011") then stage <= "11"; end if;
            sha_state <= setup;

        when "11" =>
            a <=    (   a(26 downto 0) & a(31 downto 27)    )   +
                    (   b xor c xor d                       )   +
                    (   e + (w(30 downto 0) & w(31)) + Kt   );
            if (count = "10011") then
                sha_state <= done;
            else
                sha_state <= setup;
            end if;

        when others =>
            a <= (others => '1');
    end case;

    if (count > "01111" or stage > "00") then
        w0 <= w1; w1 <= w2; w2 <= w3; w3 <= w4; w4 <= w5;
        w5 <= w6; w6 <= w7; w7 <= w8; w8 <= w9; w9 <= w10;
        w10 <= w11; w11 <= w12; w12 <= w13;
        w13 <= w14; w14 <= w15;
        w15 <= w(30 downto 0) & w(31);
    end if;

    if (count = "10011") then
```

```
                    count <= "00000";
                else
                    count <= count + 1;
                end if;

            when done =>
                digest <= a & b & c & d & e;
                digest_vld <= '1';

            end case;
        end if;
    end if;
  end process;
end limited_sha;
```

A.2 VHDL Encrypted Memory Controller

In the top-level RC1000 entity:

```
--** MEMORY BANK 1: Refer to AS7C4096 Datasheet for more info
mem1_req_l  : OUT std_logic;
mem1_gnt_l  : IN std_logic;                    -- Request memory lock (active low)
mem1_ce_l   : OUT std_logic_vector(3 DOWNTO 0);-- Memory lock granted (active low)
mem1_oe_l   : OUT std_logic;                   -- Byte chip enables (active low)
mem1_we_l   : OUT std_logic;                   -- Output Enable (active low)
mem1_addr   : OUT std_logic_vector(22 DOWNTO 2);--Write Enable (active low)
mem1_data   : INOUT std_logic_vector(31 DOWNTO 0);-- Memory address bus
```

In the architecture component and signal declarations section:

```
-- Encryption Engine
component des_fast
port (  clk           :in     std_logic;
        reset         :in     std_logic;
        stall         :in     std_logic;
```

```
        encrypt       :in     std_logic;  -- 1=encrypt, 0=decrypt
        key_in        :in     std_logic_vector (55 downto 0);
        din           :in     std_logic_vector (63 downto 0);
        din_valid     :in     std_logic;

        dout          :out    std_logic_vector (63 downto 0);
        dout_valid    :out    std_logic;
        key_out       :out    std_logic_vector (55 downto 0)
        );
end component;
for all : des_fast use entity work.des_fast(arch_des_fast);

-- address expansion unit
component address_expansion_unit
port(
    address_in      :   in  std_logic_vector (21 downto 1);
    address_out     :   out std_logic_vector (64 downto 1)
    );
end component;
for all :   address_expansion_unit
            use entity
            work.address_expansion_unit(address_expansion_unit_arch);


--read_write signals
type write_state_type is (set, write, done);
signal write1_state     : write_state_type;
type   read_state_type is (set, read, done);
signal read1_state      : read_state_type;
signal read1_active     : std_logic;
signal address1         : std_logic_vector(20 downto 0);
signal data1            : std_logic_vector(31 downto 0);

-- DES_ENGINE signals
signal des_din, des_dout                    :   std_logic_vector (63 downto 0);
signal des_key                              :   std_logic_vector (55 downto 0);
signal des_reset, des_din_rdy,
        des_dout_rdy, des_encrypt, des_state :   std_logic;
signal des_xor_key                          :   std_logic_vector (31 downto 0);
signal mem1_decrypted_data                  :   std_logic_vector (31 downto 0);
```

```
type   encrypt_state_type is (encrypt, operate);
signal encrypt_state : encrypt_state_type;

signal adjusted_address1    : std_logic_vector (21 downto 1);

type prediction_state_type is (  p0, p1, p2, p3, p4, p5, p6, p7, p8, p9,
                                 p10,p11,p12,p13,p14,p15);
signal pout_state, pin_state : prediction_state_type;

signal r0, r1, r2, r3, r4, r5, r6, r7, r8, r9,
       r10,r11,r12,r13,r14,r15                  : std_logic_vector(31 downto 0);

signal prediction_address_max                   : std_logic_vector(21 downto 1);
signal expansion_in, temp_address1              : std_logic_vector(21 downto 1);
signal expansion_out                            : std_logic_vector(63 downto 0);
signal address_difference                       : std_logic_vector(21 downto 1);
signal prepare_key_state                        : std_logic;
```

In the architecture component port map section:

```
-- DES Engine
DES_ENG: DES_FAST
    port map (
        reset       => des_reset,
        clk         => clk,
        stall       => '0',
        encrypt     => des_encrypt,
        key_in      => des_key,
        din         => des_din,
        din_valid   => des_din_rdy,
        dout        => des_dout,
        dout_valid  => des_dout_rdy,
        key_out     => open
        );

-- Address expansion unit
ADDR_EXP: ADDRESS_EXPANSION_UNIT
    port map (
```

```
        address_in  => expansion_in,
        address_out => expansion_out
        );
```

Finally, the encrypted memory controller process.

```
-- Complete DES encrypted read/write interface
-- to memory 1 with prediction registers and
-- secret and final keys
mem1_data   <= (data1 XOR des_xor_key) when write1_go = '1' else (others => 'Z');
mem1_ce_l   <= "0000";
mem1_addr   <= address1;
des_encrypt <= '1';
mem1_decrypted_data <= des_xor_key XOR mem1_data;
READWRITE1      : process (clk, start)
begin
    if (start = '0') then
        read1_active    <= '0';
        write1_active   <= '0';
        read1_state     <= set;
        write1_state    <= set;
        encrypt_state   <= encrypt;
        mem1_oe_l       <= '1';
        mem1_we_l       <= '1';
        des_reset       <= '1';
        des_din_rdy     <= '0';
        clock_counter   <= (others => '0');
        pout_state  <= p0;
        pin_state   <= p0;
        expansion_in    <= (others => '0');
        temp_address1   <= (others => '0');
        prediction_address_max <= (others => '1');
        r0  <= (others => '0');
        r1  <= (others => '0');
        r2  <= (others => '0');
        r3  <= (others => '0');
        r4  <= (others => '0');
        r5  <= (others => '0');
        r6  <= (others => '0');
```

```
    r7  <= (others => '0');
    r8  <= (others => '0');
    r9  <= (others => '0');
    r10 <= (others => '0');
    r11 <= (others => '0');
    r12 <= (others => '0');
    r13 <= (others => '0');
    r14 <= (others => '0');
    r15 <= (others => '0');
    prepare_key_state   <= '0';
    des_key             <= (others => '0');
    des_din             <= (others => '0');

elsif (clk'event and clk = '1') then
case prepare_key_state is
when '0' =>
    des_key <= X"AAAAAAAAAAAAAA";     -- secret des key
    if (certificate_match = '1') then
        des_din <= uid_string;
        des_reset <= '0';
        des_din_rdy <= '1';
        if (des_dout_rdy = '1') then
            des_key <= des_dout(55 downto 0);   -- final des key
            des_reset <= '1';
            des_din_rdy <= '0';
            prepare_key_state <= '1';
        end if;
    end if;
when '1' =>
    if (read1_go = '1' or write1_go = '1') then
        des_din <= expansion_out;
        clock_counter <= clock_counter + 1;
        case encrypt_state is
        when encrypt =>
            -- if the prediction registers hold the
            -- des_xor_key already
            if (address_difference < "00000000000000010000") then
                case address_difference(4 downto 1) is
                    when "1111" => des_xor_key <= r0;
                    when "1110" => des_xor_key <= r1;
```

```
            when "1101" => des_xor_key <= r2;
            when "1100" => des_xor_key <= r3;
            when "1011" => des_xor_key <= r4;
            when "1010" => des_xor_key <= r5;
            when "1001" => des_xor_key <= r6;
            when "1000" => des_xor_key <= r7;
            when "0111" => des_xor_key <= r8;
            when "0110" => des_xor_key <= r9;
            when "0101" => des_xor_key <= r10;
            when "0100" => des_xor_key <= r11;
            when "0011" => des_xor_key <= r12;
            when "0010" => des_xor_key <= r13;
            when "0001" => des_xor_key <= r14;
            when "0000" => des_xor_key <= r15;
            when others => des_xor_key <= r0; -- this is an error case
        end case;
        encrypt_state <= operate;

    -- if the prediction registers do not already
    -- hold the des_xor key
    else
        des_reset   <= '0';
        des_din_rdy <= '1';
        case pin_state is
            when p0  => expansion_in <= adjusted_address1;
                        temp_address1 <= adjusted_address1 + 1;
                        pin_state <= p1;
            when p1  => expansion_in <= temp_address1;
                        temp_address1 <= temp_address1 + 1;
                        pin_state <= p2;
            when p2  => expansion_in <= temp_address1;
                        temp_address1 <= temp_address1 + 1;
                        pin_state <= p3;
            when p3  => expansion_in <= temp_address1;
                        temp_address1 <= temp_address1 + 1;
                        pin_state <= p4;
            when p4  => expansion_in <= temp_address1;
                        temp_address1 <= temp_address1 + 1;
                        pin_state <= p5;
            when p5  => expansion_in <= temp_address1;
```

```
                temp_address1 <= temp_address1 + 1;
                pin_state <= p6;
    when p6  => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p7;
    when p7  => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p8;
    when p8  => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p9;
    when p9  => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p10;
    when p10 => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p11;
    when p11 => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p12;
    when p12 => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p13;
    when p13 => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p14;
    when p14 => expansion_in <= temp_address1;
                temp_address1 <= temp_address1 + 1;
                pin_state <= p15;
    when p15 => expansion_in <= temp_address1;
end case;

if (des_dout_rdy = '1') then
    case pout_state is
    when p0  => des_xor_key <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                r0          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p1;
    when p1  => r1          <=  des_dout(63 downto 32)
```

```
                                XOR des_dout(31 downto 0);
                                pout_state <= p2;
when p2  => r2          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p3;
when p3  => r3          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p4;
when p4  => r4          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p5;
when p5  => r5          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p6;
when p6  => r6          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p7;
when p7  => r7          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p8;
when p8  => r8          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p9;
when p9  => r9          <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p10;
when p10 => r10         <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p11;
when p11 => r11         <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p12;
when p12 => r12         <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p13;
when p13 => r13         <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
                                pout_state <= p14;
when p14 => r14         <=  des_dout(63 downto 32)
                                XOR des_dout(31 downto 0);
```

```
                                        pout_state <= p15;
            when p15 => r15         <=  des_dout(63 downto 32)
                                        XOR des_dout(31 downto 0);
                                        pout_state <= p0;
                                        pin_state <= p0;
                                        prediction_address_max
                                        <= address1 +
                                        "00000000000000001111";
                                        encrypt_state <= operate;
            end case;
        end if;
    end if;

when operate =>
    des_din_rdy <= '0';
    des_reset   <= '1';

    if (read1_go = '1') then
        read1_active <= '1';
        case read1_state is
        when set =>
            mem1_oe_l <= '0';
            read1_state <= read;
        when read =>
            read1_state <= done;
        when done =>
            mem1_oe_l <= '1';
            read1_active <= '0';
            read1_state <= set;
            encrypt_state <= encrypt;
        end case;

    elsif (write1_go = '1') then
        write1_active <= '1';
        case write1_state is
        when set =>
            mem1_we_l <= '0';
            write1_state <= write;
        when write =>
            write1_state <= done;
```

```
                    when done =>
                        mem1_we_l <= '1';
                        write1_active <= '0';
                        write1_state  <= set;
                        encrypt_state <= encrypt;
                    end case;
                end if;
            end case;
        end if;
    end case;
    end if;
end process READWRITE1;
```

Bibliography

[1] Egypt State Information Service, Cairo, Egypt, *Egypt, The Cradle of Monotheism*, 2004. http://www.sis.gov.eg/cradle/english/cradlea.htm.

[2] Celoxica Limited, Abingdon, Oxfordshire, United Kingdom, *RC1000 Hardware Reference Manual Version 2.3*, 2001.

[3] Maxim/Dallas Semiconductor Corporation, Dallas, Texas, *Java-Powered Cryptographic iButton*, 2003. http://www.ibutton.com/ibuttons/java.html.

[4] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *FIPS Publication 46-2: Data Encryption Standard*, 1993.

[5] Naval Air Warfare Center Weapons Division, Naval Air Weapons Station China Lake, CA, *NAVAIR Weapons Division PAO: Photo gallery*, March 2002. Image and information released to the public on November 12, 1998 at http://www.nawcwd.navy.mil/ pao/pg/Photo/air/ep3/EP3E.htm.

[6] P. Felstead, "Inside account further exonerates ep-3 pilot," *Jane's Defence Weekly*, May 2001. http://www.janes.com/regional_news/asia_pacific/news/misc/ep3_010518_1_n.shtml.

[7] CNN, "U.s. spy plane crew land in guam," *CNN.com*, April 2001. http://www.cnn.com/2001/WORLD/asiapcf/east/04/11/air.collision.09/.

[8] CNN, "China may be stripping spy plane, says u.s.," *CNN.com*, April 2001. http://www.cnn.com/2001/WORLD/asiapcf/east/04/10/plane.stripping/index.html.

[9] R. Wall, "New intelligence gear on china-held ep-3," *Aviation Week & Space Technology*, April 2001. http://www.aviationnow.com/content/publication/awst/20010409/ep3.htm.

[10] National Public Radio, Washington, D.C., *Technology used on the EP-3 and what the Chinese might learn about our spying capabilities from the plane itself*, April 2001. http://www.globalsecurity.org/org/news/2001/010413-aries1.htm.

[11] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in *Communications of the ACM*, vol. (2) 21, pp. 120–126, 1978.

[12] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *NIST PKI Program*, 2001. http://csrc.nist.gov/pki/.

[13] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *FIPS Publication 180: Secure Hash Standard*, 1993.

[14] W. Diffie and M. Hellman, "New directions in cryptography," in *IEEE Transactions in Information Theory*, vol. 22, pp. 644–654, 1976.

[15] B. Schneier, *Applied Cryptography*. New York, New York: John Wiley & Sons, Inc., 1996.

[16] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 2002.

[17] C. Wu and X. Wang, "Determination of the true value of the euler totient function in the rsa cryptosystem from a set of possibilities," in *Electronic Letters*, pp. 84–85, 1993.

[18] B. Kaliski, "Twirl and rsa key size," *RSASecurity.com*, May 2003. http://www.rsasecurity.com/rsalabs/technotes/twirl.html.

[19] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *Special Publication 800-57: Recommendation for Key Management. Part 1: General Guideline. Draft.*, 2003. http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html.

[20] A. Shamir and E. Tromer, "Factoring large numbers with the twirl device," in *LNCS 2729*, pp. 1–26, Crypto 2003, Springer-Verlag, 2003.

[21] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *FIPS Publication 197: Advanced Encryption Standard*, 2001.

[22] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. New York, New York: Springer-Verlag, 1993.

[23] T. Kean, "Secure configuration of field programmable gate arrays," in *LNCS 2147*, FPL 2001, Springer-Verlag, 2001.

[24] Xilinx, Incorporated, San Jose, California, *Virtex-II Platform FPGAs: Complete Data Sheet*, October 2003. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[25] Altera Corporation, Sunnyvale, California, *Stratix II Devices: The Biggest & Fastest FPGAs*, 2004. http://www.altera.com/products/devices/stratix2/st2-index.jsp.

[26] J. Dyer, M. Lindermann, R. Perez, R. Sailer, L. van Doorn, S. Smith, and S. Weingart, "Building the ibm 4758 secure coprocessor," in *IEEE Computer*, pp. 57–66, IEEE, 2001. Vol. 34.

[27] "Ars technica: Microsoft kills next-generation secure computing base," *Ars Technica: The PC Enthusiast's Resource*, May 2004. http://arstechnica.com/news/posts/1083785108.html.

[28] "Next-generation secure computing base," *Microsoft.com*, 2004. http://www.microsoft.com/resources/ngscb/default.mspx.

[29] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and temper-resistant processing," *Proceedings of the 17th Annual International Conference on Supercomputing*, June 2003.

[30] Maxim/Dallas Semiconductor Corporation, Dallas, Texas, *iButton Applications*, 2003. http://db.maxim-ic.com/ibutton/applications/index.cfm.

[31] RSA Security, Inc., Bedford, MA, *RSA Security - Smart Cards and USB Tokens*, 2004. http://www.rsasecurity.com/node.asp?id=1215.

[32] ActivCard, Inc., Fremont, California, *ActivCard Tokens*, 2004. http://www.activcard.com/en/products/4_3_3_tokens.php.

[33] A. Abraham, "It is I: An Authentication System for a Reconfigurable Radio," Master's thesis, Virginia Tech, August 2002. Describes an authentication system that uses iButtons and a biometric reader.

[34] Xilinx, Incorporated, San Jose, California, *Virtex-E 1.8V Field-Programmable Gate Arrays*, July 2002. http://www.xilinx.com/bvdocs/publications/ds022.pdf.

[35] Maxim Integrated Products, Dallas, Texas, *DS9097U Universal 1-Wire COM Port Adapter*, 2004. http://dbserv.maxim-ic.com/quick_view2.cfm?qv_pk=2983.

[36] Maxim Integrated Products, Dallas, Texas, *Maxim 1-Wire and iButton: Communications components for identification, sensor, control, and memory functions*, 2004. http://www.maxim-ic.com/1-Wire.cfm.

[37] Maxim Integrated Products, Dallas, Texas, *DS2480B Serial 1-Wire Line Driver with Load Sensor*, 2004. http://www.maxim-ic.com/quick_view2.cfm?qv_pk=2923.

[38] Maxim Integrated Products, Dallas, Texas, *MAX220, MAX222, MAX223, MAX225, MAX230, MAX231, MAX232, MAX232A, MAX233, MAX233A, MAX234, MAX235, MAX236, MAX237, MAX238, MAX239, MAX240 MAX241, MAX242, MAX243, MAX244, MAX245, MAX246, MAX247, MAX248, MAX249 +5V-Powered, Multichannel RS-232 Drivers/Receivers*, 2004. http://www.maxim-ic.com/quick_view2.cfm/qv_pk/1798.

[39] Celoxica Limited, Abingdon, Oxfordshire, United Kingdom, *RC1000 Functional Reference Manual Version 1.3*, 2001.

[40] Maxim/Dallas Semiconductor Corporation, Dallas, Texas, *iB-IDE – New IDE for the Java-powered iButton*, 2003. http://www.ibutton.com/iB-IDE/.

[41] H. Lipmaa, P. Rogaway, and D. Wagner, "Comments to nist concerning aes modes of operations: Ctr-mode encryption," in *Modes of Operation for Symmetric Key Block Ciphers*, 2000. http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf.

[42] National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, *FIPS Publication 46-2: Data Encryption Standard*, 1999.

[43] T. Wollinger and C. Paar, "How Secure Are FPGAs in Cryptographic Applications," in *International Conference on Field Programmable Logic and Applications*, FPL, September 2003.

[44] L. McDaniel, "An Investigation of Differential Power Analysis Attacks on FPGA-based Encryption Systems," Master's thesis, Virginia Tech, May 2003.

[45] S. Harper, *A Secure Adaptive Network Processor*. Ph.D. thesis, Virginia Tech, April 2003.

Vita

Jonathan Graf was born in Oakland, California in 1980. In 1984, after briefly living in Denver, Colorado, his family settled in Vienna, Virginia, where he grew up. Starting in the second grade, Jonathan was home-schooled by his parents. Home schooling led to many uncommon high school experiences, including owning an unnecessarily large supply of chemistry equipment and founding, at age 15, a small but successful computer retail business. Jonathan's first collegiate endeavors began at age 16 at Northern Virginia Community College, where he earned the school's top honors in mathematics and physics. Two years later, Virginia Tech accepted Jonathan into its computer engineering program, from which he earned a B.S. *magna cum laude* in 2002.

During his college years, Jonathan has enjoyed a variety of professional experiences. At Intel he learned about topics in 3D graphics architecture, while at the startup Vision Point Systems he ventured into embedded systems programming and project management. He also served as a web programmer for TREEV and a telecommunications network designer for Sprint. Rounding out his collegiate experiences, he spent the summer of 2001 at a college in Shizuoka, Japan working for the Navigators.

This year, 2004, is a big year for Jonathan. Not only does it include his earning an M.S. from Virginia Tech, but also every other member of his family – his father, mother, and brother – earned master's degrees in May with just about every available honor in their respective programs. In August, he will marry his best friend and reason for sanity, Leanna Harman, at a ceremony at Virginia Tech's chapel. In June, he will start work at Luna Innovations in Blacksburg, Virginia, where he will continue researching cryptographic FPGA applications. Jonathan is considering pursuing a Ph.D. related to his ongoing research in the future.

When taking a break from being a full-time geek, Jonathan occasionally has time to remember a period in his life when he had hobbies. Those hobbies include playing a wide variety of music on his classical, steel-string, electric, and bass guitars and recording the cacophony that results in his home recording studio. He also enjoys discussing at almost insufferable length the philosophies espoused by the teachers, politicians, theologians, music, movies, and books that shape the ideologies of our world.