HIGH-PERFORMANCE ADVANCED ENCRYPTION STANDARD (AES) SECURITY CO-PROCESSOR DESIGN

A Thesis

Presented to

The Academic Faculty

By

Prateek Tandon

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science in the School of Electrical and Computer Engineering

Georgia Institute of Technology

November 2003

High-Performance Advanced Encryption Standard (AES)

Security Co-Processor Design

Approved by:

Dr. Hsien-Hsin S. Lee

Dr. Sung Kyu Lim

Dr. David Schimmel

Date Approved: 11/20/2003

ACKNOWLEDGEMENTS

I would like to thank Dr. Hsien-Hsin S. Lee for being my advisor, and Dr. Sung-Kyu Lim and Dr. David Schimmel for agreeing to be on my thesis reading committee.

I would also like to thank Taeweon Suh for his help with Verilog, Chris Clark for his help with the Celoxica setup, and Bruce Sheplan and Max Pozdin for their help in debugging the initial AES code.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES:	Advanced Encryption Standard
DES:	Data Encryption Standard
FIPS:	Federal Information Processing Standard
HDL:	Hardware Description Language
NIST:	National Institute of Standards and Technology
Gbps:	Gigabits per second
TSMC:	Taiwan Semiconductor Manufacturing Company
GF:	Galois Field
SHA:	Secure Hash Algorithm
FPGA:	Field Programmable Gate Array

SUMMARY

The objective of this research is to build a specialized co-processor to enable high performance Advanced Encryption Standard (AES) encryption and decryption. With the acceptance of Rijndael as the AES, whereby the Data Encryption Standard (DES) has been replaced as the Federal Information Processing Standard (FIPS), the demand for dedicated hardware that can efficiently perform AES encryption and decryption has increased. Keeping this increased demand in mind, this research aims to build a specialized co-processor or processor plug-in to enable high performance Rijndael encryption and decryption for real-time data transmission, particularly in the fields of image, speech, and video transmission.

The method of attacking this problem is to implement a specialized Rijndael hardware in Verilog HDL, while parallelizing and minimizing the various steps of the Rijndael algorithm without compromising its inherent security and integrity. The optimizations proposed in this thesis improve performance by approximately 16% over a non-optimized design, and the resulting co-processor achieves a throughput of 2.08 Gbps in a 0.35μm technology.

CHAPTER 1

INTRODUCTION

1.1 Background

With the acceptance of Rijndael as the Advanced Encryption Standard (AES), whereby the Data Encryption Standard (DES) has been replaced as the Federal Information Processing Standard (FIPS), the demand for dedicated hardware that can efficiently perform AES encryption and decryption has increased. Keeping this increased demand in mind, this research aims to build a specialized co-processor or processor plug-in to enable high performance Rijndael encryption and decryption for real-time data transmission, particularly in the fields of image, speech, and video transmission.

The method of attacking this problem is to implement a specialized Rijndael hardware in Verilog HDL, while parallelizing and minimizing the various steps of the Rijndael algorithm without compromising the inherent security and integrity of the cipher algorithm.

1.2 Origin and History of the Problem

In the year 2000, the block cipher Rijndael was chosen as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST) [1][2][3][4]. AES officially replaced the Data Encryption Standard (DES) [5][6][7][8] as the new Federal Information Processing Standard (FIPS).

Subsequently, there have been multiple attempts to create dedicated hardware to perform AES based encryption and decryption. One of the most recent open-core AES cipher chips with a maximum throughput of 2.97 Gbps was built by researchers at the Laboratory for Reliable Computing, National Tsing Hua University in Taiwan [9]. This full-custom chip was designed and manufactured using TSMC 0.25μm technology, and has a core area of 1,279 by 1,271 μm^2. The design uses 63,400 gates and can run at a maximum frequency of 250 MHz.

Other prominent designs are mentioned in [10], [11,12], [13], and [14]. In [11,12], the synthesized Rijndael core achieves a maximum throughput of 2.29Gbps for 256-bit keys and data, while utilizing 173,000 gates in a 0.18 micron technology. However, this design only includes an encryption device and does not perform decryption. Another synthesized core is presented in [15]. This device can perform encryption and decryption with a maximum throughput of 2.33 Gbps for 256-bit keys and data, while utilizing 28,626 gates. A pipelined FPGA implementation that can produce a throughput of 6.95 Gbps for a 128-bit key and 128-bit data block is described in [13]. When used in combination with a feedback mode of operation, the throughput of this implementation is 695 Mbps. The FPGA implementation in [10], describes both a pipelined and non-pipelined implementations, but only of the encryption core. It does not include decryption or key scheduling. The best pipelined implementation reaches 1.94 Gbps, and the best feedback mode implementation reaches 300 Mbps.

With the above designs in view, this research aims to achieve better encryption and decryption data rates, thereby speeding up the security processing of real-time applications like image, speech, and video transmission while maintaining the security offered by Rijndael. Aside from using the specialized co-processor for real-time data, it is also expected that this co-processor would also find uses in high performance servers such as web servers that process hundreds of thousands of requests per second. These servers are usually hard-pressed when it comes to performing encryption or decryption along with their regular duties. However, with the addition of a co-processor that can handle the encryption and decryption, a lot of the strain can be removed from the servers, thereby enabling them to perform their other duties more efficiently. Therefore, the co-processor will go a long way in reducing the loads on the servers mentioned above.

Chapter 2

FUNDAMENTALS OF DATA ENCRYPTION

2.1 Terminology

This section deals with the terminology used in the field of cryptography:

Plaintext: Plaintext refers to unencrypted data that needs to be protected from unauthorized entities.

Ciphertext: Encrypted version of plaintext.

Encryption: The transformation of plaintext into an apparently less readable form (called ciphertext) through a mathematical process.

Decryption: The process of obtaining plaintext from ciphertext is called decryption. Encryption and decryption usually make use of secret keys and the encryption/decryption can be performed only if one possesses the proper keys.

Block Cipher: A block cipher encrypts data in discrete units (called blocks). The most prominent block ciphers in use today are DES and AES.

Feistel Cipher: A special class of iterated block ciphers where the ciphertext is calculated from the plaintext by the repeated application of the same transformation called a round function.

Galois Field (GF): A field with a finite number of elements. The size of the finite field must be a power of prime number. [16]

2.2 AES Basics

Before getting into the details of the research, it would be germane to discuss the basics of the Rijndael/AES cipher. Rijndael consists of different transformations that operate on the intermediate result, which is called the State. The State can be pictured as a rectangular array of bytes with four rows. The number of columns is denoted by *Nb* and is equal to the block length divided by 32. Figure 1 shows a data block of 128 bits. $A_{i,j}$ represents an element in the *ith* row and *jth* column.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

Figure 1: Rijndael State

Similarly, the Cipher Key can be pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by *Nk* and is equal to the key length divided by 32. Further information about the Rijndael algorithm can be found in [4].

Various design criteria were taken into account during the design of AES. These criteria include resistance against all known attacks, speed and code compactness on a wide range of platforms, and design simplicity, among others. Rijndael is different from many other cipher algorithms (including DES and its various variants) in that it does not use the Feistel structure in its round transformation [4][5]. Instead, the round transformation is composed of three distinct invertible uniform transformations, called layers which are summarized below:

- The linear mixing layer – Guarantees high diffusion over multiple rounds.

- The non-linear layer – Parallel application of S-boxes that have optimum worst-case nonlinearity properties.

- The key addition layer – A simple XOR of the Round Key with the intermediate State.

Each round of encryption in Rijndael consists of four different transformations which are as follows:

1. Byte Substitution

The Byte Substitution transformation is a non-linear operation. Each of the State bytes is acted upon independently. The substitution table (or S-box) is invertible and is constructed by the composition of the following two transformation steps [4]:

1. First, the multiplicative inverse in GF(2^8) is taken.

2. Then, an affine transformation (over GF(2^8)) is applied as shown below.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 2: Affine Transform

Essentially, Byte Substitution maps one byte to exactly one other byte within the GF(2^8). The Byte Substitution process can be represented as follows in figure 3, where $A_{i,j}$ is an element of the state which undergoes byte substitution to become $B_{i,j}$ (*i* being the row number and *j* being the column number):

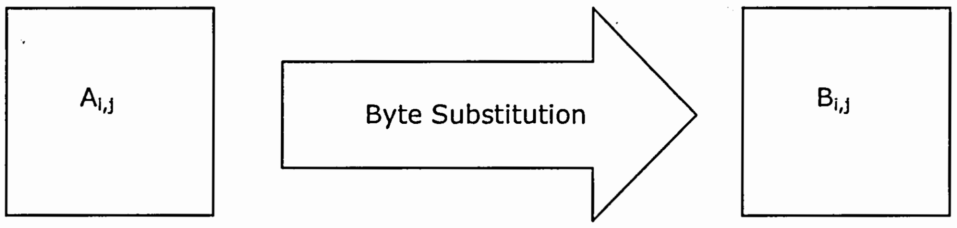

Figure 3: Byte Substitution

2. Shift Row

In Shift Row, all the rows of the State, except row 0 are cyclically right shifted over different offsets. The shift offsets C1, C2 and C3 for row 1, row 2 and row 3 respectively depend on *Nb*. The dependence of the shift offsets on the block length *Nb* is as shown in table 1:

Table 1: Row shifts based on *Nb*

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

3. Mix Column

In Mix Column, the columns of the State are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial c(x), which is given by c(x) =

'03' x^3 + '01' x^2 + '01' x + '02'. The Mix Column steps can be represented by figures 4 and 5 (it needs to be noted that Mix Column acts on a column at a time):

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figure 4: Mathematical representation of Mix Column

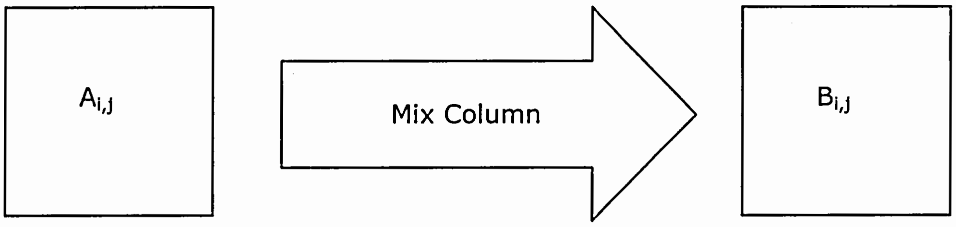

Figure 5: Mix Column

4. *Round Key Addition*

The Round Key Addition step consists of XORing the State with the Round Key. The Round Key (which is different for every round) is derived from the Cipher

Key by means of the key schedule. The Round Key length is equal to the block length *Nb*. The total number of Round Key bits is equal to the block length multiplied by the number of rounds plus 1.

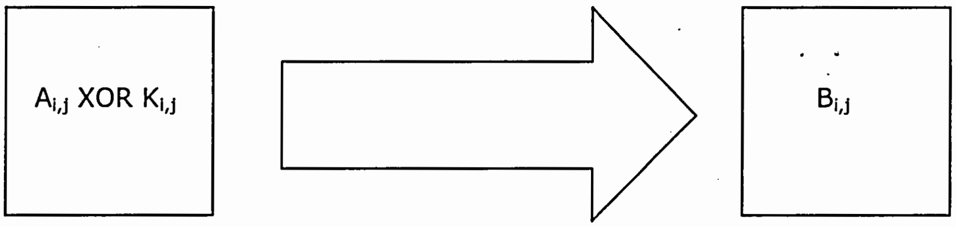

Figure 6: Round Key Addition

The key schedule consists of two components: Key Expansion and Round Key Selection. First, the Cipher Key is expanded into an Expanded Key. For *Nk* less than or equal to 6 the following algorithm is used for key expansion. *Nr* represents the number of rounds.

```
Key_Expansion(byte Key[4*Nk], word[Nb * (Nr+1)])
{
   for (x = 0; x < Nk; x++)
   {
      W[x] = (Key[4*x], Key[4*x+1], Key[4*x+2], Key[4*x+3]);
      for ( x = Nk; x < Nb*(Nr+1); x++ )
      {
         temp = W[x-1];
         if (x % Nk == 0)
      temp = SubByte(RotByte(temp))^Rcon[i/Nk];
         W[x] = W[x - Nk] ^ temp;
      }
   }
}
```

SubByte is a function that returns a 4-byte word in which each byte is transformed using Byte Substitution. RotByte performs a cyclic permutation such that the word (a, b, c, d) is converted into (b, c, d, a).

For *Nk* greater than 6, the algorithm is as follows:

```
Key_Expansion(byte Key[4*Nk], word[Nb * (Nr+1)])
{
   for (x = 0; x < Nk; x++)
   {
      W[x] = (Key[4*x], Key[4*x+1], Key[4*x+2], Key[4*x+3]);
      for (x = Nk; x < Nb*(Nr+1); x++)
      {
         temp = W[x-1];
         if (x % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i/Nk];
         else if (x  % Nk = = 4)
            temp = SubByte (temp);
         W[x] = W[x - Nk] ^ temp;
       }
      }
}
```

Rcon[x] is defined as follows:

Rcon[x] = (RC[x], '00', '00', '00') where RC[x] is an element in GF(2^8) with a value of $z^{(x-1)}$ such that:

RC[1] = 1 (i.e. '01')

RC[x] = z (i.e. '02') . (RC[x-1]) = $z^{(x-1)}$

After the Key Expansion has been performed, Round Keys are taken from the Expanded Key by choosing the first key to consist of the first *Nb* words, the second key of the next *Nb* words, and so on. [4]

In order to decrypt the data that has been encrypted using the above steps, only the inverse of the above steps needs to be performed.

Chapter 3

OPTIMIZATION TECHNIQUES

From the introduction to the various steps involved in Rijndael-based encryption and decryption, it can be seen that there is inherent parallelism within the Rijndael cipher. A considerable amount of coarse-level parallelism exists, in that data can be acted upon simultaneously at the level of bytes, rows, or even the State. Also, it can be noted that there is the potential for a few of the four steps outlined above to be performed simultaneously; that is, they can be combined into a single step to exploit the parallelism of operations.

With a view to exploit the inherent parallelism in Rijndael and to use the property of there being no strong ordering between some of the steps (for example, between Byte Substitution and Shift Row, and between Mix Column and Round Key Addition), the following exercise was carried out. The number of the various operations (AND, OR, XOR) carried out in the four steps was calculated, and using this number of operations as a guide, an estimate of the number of cycles that each step would take was made. This hand-based approximation yielded the following: The ratio of the execution cycles in terms of Byte Substitution: Shift Row: Mix Column: Round Key Addition is approximately 1: 2: 4: 1. Since these hand-based approximations are not highly accurate (hence approximations), Rijndael was implemented in C++ and the ratio of the cycles in terms of Byte Substitution: Shift Row: Mix Column: Round Key Addition was found to be 1: 2: 3: 1.

Based on the approximated execution time required for each step as profiled in the software implementation, Byte Substitution and Shift Row are combined into a single step since there is no data dependency between the two steps. In addition to the combination of the two steps, each byte of the State can be acted upon in parallel thereby enabling faster encryption or decryption. Similarly, Mix Column and Round Key Addition are combined into a single step since there is no strong ordering between the two steps. Also, the State in this newly combined step can be acted upon in sets of four bytes (one column) in parallel. The combination of the Mix Column and Round Key Addition raises an important issue that needs to be addressed. The Key Expansion needs to be done separately in order to enable the combination of Mix Column and Round Key Addition. Key

Expansion is either performed at the beginning of the encryption process since it needs to be done only once for a given key, or it is performed such that the required round key is available prior to Round Key Addition. Additionally, the Byte Substitution is implemented as a set of equations. The substitution calculation is made every time a substitution needs to be made. The rationale behind this implementation is described in chapter 4. In summary, Byte Substitution and Shift Row are combined into one single step. All the bytes of the State are acted upon in parallel. Also, Mix Column and Round Key Addition are collapsed into a single step. The State is acted upon at the level of columns (four bytes) in parallel. The Key Expansion needs to be done prior to the cipher rounds being executed. The Rijndael cipher in its original form can be represented as follows in figure 7:

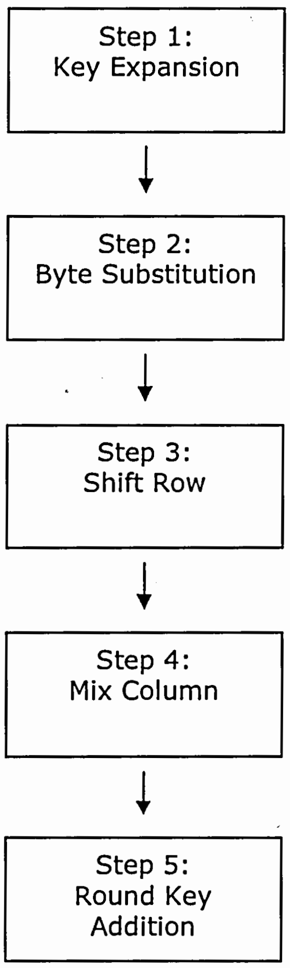

Figure 7: Original Rijndael Cipher

The Rijndael cipher after the optimizations (collapsing steps, and acting on multiple bytes at the same time) can be represented as follows in figure 8. It needs to be noted that in the Byte Substitution + Shift Row step all bytes of the state are acted upon in parallel. Also, in the Mix Column + Round Key Addition step, the State is acted upon the level of columns in parallel, that is, four bytes in parallel.

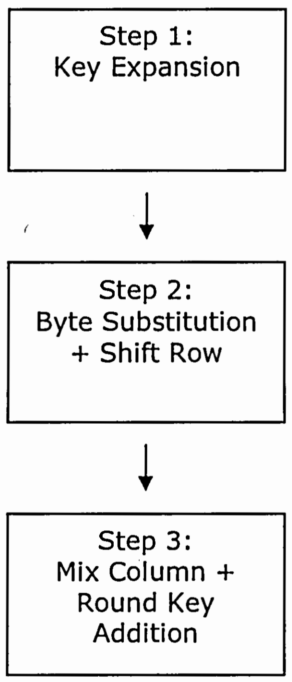

Figure 8: Optimized Rijndael Cipher

Chapter 4

HARDWARE IMPLEMENTATION

4.1 Overview

The Rijndael co-processor was designed using Verilog HDL, synthesized using Synopsys Design Analyzer and targeted towards the HP 0.35µm libraries. The design consists of three separate modules – one for key expansion, one for encryption and one for decryption. Since key expansion is required both for encryption and decryption, the key expansion module is shared by the encryption and decryption modules. The encryption and decryption circuits are synthesized as two separate modules because having two separate modules allows greater flexibility in placing the encryption and decryption modules on separate data paths if needed. Additionally, separate modules which do not share any circuitry can allow for encryption and decryption to be carried out simultaneously. This is a major advantage of the current implementation over prior schemes. The three basic modules also make use of a combination of the following sub-modules: byte substitution module, inverse byte substitution module, mix key module and the inverse mix key module. The byte substitution and inverse byte substitution modules perform the byte substitution and inverse byte substitution functions respectively. The mix key and inverse mix key modules perform the mix column and round key addition, and inverse mix column and key addition functions respectively. The mix key and inverse mix key modules perform operations on 16, 24, or 32 bytes simultaneously, depending on the data block length. The mechanism for shifting rows is discussed later.

All of the modules are constructed so that all different Rijndael combinations of data block and key block lengths are covered. The modules can thus operate under the following 9 modes as shown in table 2:

Table 2: Modes of operation of design

Mode #	Key Block Length	Data Block Length
0	128	128
1	128	192
2	128	256
3	192	128
4	192	192
5	192	256
6	256	128
7	256	192
8	256	256

From the above table it can be seen that the three modules perform AES encryption and decryption as a subset of the Rijndael algorithm. (Modes 0, 3, and 6 correspond to AES encryption and decryption).

The following sections explain each of the three modules in detail.

4.2 Key Expansion Module

The key expansion module is responsible for performing key expansion of the input 128, 192, or 256 bit key and transforming the same into an expanded key (also referred to in this text as output key). The key expansion module can perform either online or offline sub-key computation, a major advantage in terms of flexibility over other schemes. The lengths of the expanded keys for the 9 modes of operation, and the number of cycles in the hardware implementation required to complete the expansion are listed in table 3:

Table 3: Expanded key lengths and number of cycles required for key expansion

Mode #	Input Key Length	Output Key Length	Number of Cycles
0	128	1408	11
1	128	2496	20
2	128	3840	30
3	192	1664	9
4	192	2496	13
5	192	3840	20
6	256	1920	14
7	256	2880	22
8	256	3840	29

Even though the output key lengths may be the same for some modes, the number of cycles needed to generate the output key is different for each case. The reason for this difference in the number of clock cycles is that a different algorithm needs to be followed for a different input key length as mentioned earlier. This accounts for the difference in the number of clock cycles needed to perform the complete key expansion. Also, a trend seen in the table above is that as the block size increases, the length of the expanded key increases, thereby increasing the number of clock cycles required for the completion of the key expansion.

The key expansion module also makes use of four byte substitution modules to perform four parallel byte substitutions required in the key expansion process. As mentioned earlier, the byte substitution module is not implemented as a lookup table. Lookup tables generally suffer from increased latencies associated with the lookup. Since the byte substitution module would need to use 256 different possibilities, and since multiple byte substitution modules are used in this design, the case statement approach was not deemed feasible for coding. The approach taken to solve this problem was as follows: Each of the 256 inputs and corresponding outputs were coded in PLA format, and read using Design Analyzer. The compiled file was saved in Verilog format and added to the design.

Two clocks act as inputs to the key expansion module. While the main key expansion module operates on the rising clock edge of the first clock (referred to henceforth as clock_0), the byte substitution module operates on the rising edge of the second clock (referred to henceforth as clock_1). Clock_0 and clock_1 are synchronized such that clock_1 operates at twice the frequency of clock_0.

The key expansion module consists of 34,227 gates and has a worst case path delay of 4.18ns. Therefore, the maximum frequency at which this circuit can be clocked is 239.2MHz (for clock_1). This implies a frequency of 119.6MHz for clock_0. At this frequency, the following throughput can be achieved for the previously mentioned nine modes of operation (with the addition of four more byte substitution modules this throughput can be doubled; since clock_1 is twice the frequency of clock_0, this would enable the computation of two expanded keys simultaneously):

Table 4: Throughput of key expansion module

Mode #	Throughput (Gbps)
0	1.392
1	0.765
2	0.510
3	2.552
4	1.766
5	1.148
6	2.187
7	1.392
8	1.056

4.3 Encryption Module

The encryption module is responsible for encrypting the input data block with the expanded key. This module also operates in all nine modes. The encryption module contains the sub-modules for the byte substitution and mix key operations. There are a total of 72 byte substitution modules and 8 mix key modules. The reason for the high number of byte substitution modules is the following: For the shift row operation, instead of the shifting the bytes of the state by using assignment statements, the state is inputted to the byte substitution modules in a fixed way that effectively accomplishes the shift. To make the above point clear, consider the following example. Say there are 2 bytes [x,y] that need to undergo a shift to the state [y,x]. Using assignment statements, one of the ways that the the shift can be accomplish is as follows: temp=y; y=x; x=temp. Alternatively, x<=y and y<=x can be used. The new state [y,x] can now be used as an input to the byte substitution module. However, in the current design, the inputs to the byte substitution modules are defined such that they are of the form [y,x]. Therefore, using this optimization, the shift row step is effectively removed. While the savings may seem trivial for the case when there are two bytes, the savings are substantial when there are up to 32 bytes in a state and the shift row needs to be done up to 14 times for a single data block. However, the tradeoff involved with the lower operation count is that there is extra hardware involved. Since the shifts for each mode are not uniform, there need to be separate byte substitution modules for each length of data blocks, namely for 128, 192 and 256 bytes.

Once again the main encryption module is clocked with clock_0, while the byte substitution and mix key modules are clocked with clock_1. Another optimization made in the encryption module is that at a given point of time, the encryption module can handle two blocks of data, each up to 256 bits in length. In other words, the encryption module is pipelined. While one data block is undergoing byte substitution, the other is in the mix key module. Therefore, the hardware utilization and data throughput are increased as compared to the case when there is no pipelining.

Also, the current design can accommodate the use of separate keys for each data block thereby removing any issue of lowered security. In general, with pipelining the same key is used to encrypt multiple data blocks. This raises an issue where for the same plaintext and same key, the same ciphertext is produced. However, this is not really an issue because of the following: Even if the two data blocks are identical, the cost involved in breaking even a 128-bit AES key is very high. Furthermore, if the key is changed for the next set of data blocks, the security level is increased. In addition, for cases involving transmission of non-critical data, using the same key for multiple data blocks may not be a huge security risk. However, since the current design can accommodate separate keys for separate data blocks, the above argument does not hold.

Another reason why pipelining is sometimes not used is that if the encryption needs to be done in a feedback mode (the ciphertext is used in the next input), the effective throughput is reduced as a factor of the number of pipeline stages (in other words, the throughput becomes equal to the case when there is no pipelining). However, this argument also does not apply to the current design. Due to the nature of the design, the two data blocks can be considered parts of two different feedback loops. Therefore, the throughput remains the same. The reason why the current design operates on only two blocks of data at a time is that the latencies of the byte substitution module and of the mix key module are about the same. Therefore, there is no scope for adding more data blocks.

Table 5 lists the number of cycles needed to perform a complete Rijndael encryption for the nine different modes:

Table 5: Number of cycles required for encryption

Mode #	Number of Cycles
0	21
1	25
2	29
3	25
4	25
5	29
6	29
7	29
8	29

Another interesting feature of the current design is that encryption can begin as soon as the first round key is ready in the key expansion stage. Thus, the encryption module does not have to wait for the entire key to be expanded before the encryption can begin. In other words, a large part of the key expansion process can be performed in parallel with the encryption.

The encryption module consists of 65,580 gates and has a worst case path delay of 4.252ns. Therefore, the maximum frequency at which this circuit can be clocked is about 235MHz (for clock_1). For clock_0 this translates to a frequency of 117.5MHz. At this frequency, the following throughput as shown in table 6 can be achieved for the previously mentioned nine modes of operation:

Table 6: Throughput of encryption module

Mode #	Throughput (Gbps)
0	1.432
1	1.203
2	1.037
3	1.805
4	1.805
5	1.556
6	2.075
7	2.075
8	2.075

4.4 Decryption Module

The decryption module is responsible for decrypting the input data block with the expanded key. This module also operates in all nine modes. The decryption module contains the sub-modules for the inverse byte substitution and inverse mix key operations. There are a total of 72 inverse byte substitution modules and 8 mix key modules. The reason for the high number of inverse byte substitution modules is the same as specified for the encryption module. Once again the main decryption module is clocked with clock_0, while the inverse byte substitution and inverse mix key modules are clocked with clock_1.

The decryption can also handle two blocks of data up to 256 bits in length each. While one data block is undergoing inverse byte substitution, the other is in the inverse mix key module. Table 7 lists the number of cycles needed to perform a complete Rijndael decryption for the nine different modes:

Table 7: Number of cycles required for decryption

Mode #	Number of Cycles
0	21
1	25
2	29
3	25
4	25
5	29
6	29
7	29
8	29

However, the decryption module cannot begin decryption until the entire key has been expanded. The reason for this is that the rounds keys are used in the reverse order as encryption. Therefore, the number of cycles required for decryption is higher than that required for encryption. A way to overcome this overhead is to compute the expanded key offline and therefore save the cycles required for key expansion. However, the assumption that encryption is taking place in a manner that the key for each encryption or decryption is different leads to a very conservative estimate of the decryption/encryption latency. In most general purpose block-oriented data transmission a mode known as Cipher Block Chaining (CBC) is used. In CBC mode, the input to the encryption algorithm is the XOR of the next block of plaintext (128, 192, or 256 bits depending on the data block length) and the preceding block of ciphertext. In CBC mode the latency associated with having to wait for the key expansion to complete does not pose a major bottleneck, since the same key is used for multiple blocks. [8]

The decryption module consists of 75,180 gates and has a worst case path delay of 5ns. Therefore, the maximum frequency at which this circuit can be clocked is 200MHz (for clock_1). This translates to a frequency of 100MHz for clock_0. The reason for the higher latency in the decryption module is that the inverse mix column step requires extra steps when compared to mix column. At the above frequencies, the following throughput of can be achieved for the previously mentioned nine modes of operation:

Table 8: Throughput of decryption module

Mode #	Throughput (Gbps)
0	1.219
1	1.024
2	0.883
3	1.536
4	1.536
5	1.324
6	1.766
7	1.766
8	1.766

4.5 Performance Comparisons

Table 9 compares the encryption throughput of the current design in Gbps with previously proposed designs, namely, those by research groups at the University of California at Los Angeles [11,12], and the University of Michigan at Ann Arbor [15], since these are the synthesized designs with the highest throughputs reported. Table 9 shows only encryption throughput because the design in [11,12] does not implement a decryption module. Furthermore, [15] does not specify different throughputs for encryption and decryption. It also needs to be noted that the round keys cannot be generated in parallel with the decryption because the first round of decryption uses the last x bits (x =

128, 192, or 256) of the expanded key. It is not possible to generate the round keys in the reverse order as would be required for round key generation in parallel with decryption.

Table 9: Performance comparisons (Gbps)

Mode	UCLA (0.18µm)	UMich (0.18µm)	Current design (0.35µm)	Current design (0.18µm)
0	1.28	1.64	1.43	3
1	1.33	1.36	1.20	2.52
2	1.14	1.16	1.04	2.19
3	2	2.09	1.81	3.8
4	2	2.09	1.81	3.8
5	1.71	1.78	1.56	3.28
6	2.29	2.33	2.08	4.37
7	2.29	2.33	2.08	4.37
8	2.29	2.33	2.08	4.37

From table 9 it can be seen that the current design when implemented in 0.35µm does not perform as well as previously proposed designs. It needs to be noted though that the current design uses the HP 0.35µm technology, while the designs in [11,12] and [15] use 0.18µm technology. It is widely accepted that with a 0.7x scaling in technology, the clock frequency increases by 1.43x [17]. Therefore, if the current design were to be migrated to a

0.18µm process, the clock frequency would increase by a factor of 2.1. From the projections for the current design's throughput in a 0.18µm process as shown in table 9, it can be noted that this design can outperform the previous designs.

The design in [11,12] makes use of 173,000 gates to implement a key expansion and encryption mechanism. The design in [15] uses 28,626 gates to implement the key expansion, encryption and decryption modules. However, the full-custom design in [9] uses 63,400 gates. The current design uses a total of 174,987 gates for the key expansion, encryption and decryption modules.

Due to the differences in the technology files used for the above designs, a baseline model using the HP libraries was implemented. This model neither had the combination of steps outlined earlier, nor any pipelining. The baseline encryption module consists of 74,394 gates and has a worst case path delay of 4.87ns. Therefore, the maximum frequency at which this circuit can be clocked is 205MHz (for clock_1). For clock_0 this translates to a frequency of 102.5MHz.

The following table compares the throughputs (in Gbps) of the optimized model with the baseline model (speedup is defined as the ratio between throughput of optimized model and the throughput of the baseline model):

Table 10: Throughput comparisons between optimized and baseline models (Gbps)

Mode #	Baseline (non-pipelined)	Optimized	Speedup w.r.t. non-pipelined baseline	Speedup w.r.t. pipelined baseline
0	0.60	1.43	2.38	1.19
1	0.51	1.20	2.35	1.18
2	0.44	1.04	2.36	1.18
3	0.76	1.81	2.38	1.19
4	0.76	1.81	2.38	1.19
5	0.66	1.56	2.36	1.18
6	0.88	2.08	2.36	1.18
7	0.88	2.08	2.36	1.18
8	0.88	2.08	2.36	1.18

From the above table, it can be seen that the optimized version performs about 123% better than the baseline version (when the baseline is not pipelined; percentage expressed with respect to baseline values). If the baseline is pipelined, then the optimized version outperforms the baseline by about 16% on average.

CHAPTER 5

DESIGN VERIFICATION

The verification of the current design was performed in two steps. First, the synthesized model produced by Synopsys was simulated in ModelSim. Each of the nine modes of operation was tested. Since the key expansion, encryption and decryption modules were synthesized separately, they were simulated separately also. After the design had passed post-synthesis simulation in ModelSim, final verification was done using a Xilinx XCV1000 field-programmable gate array (FPGA). The FPGA used was housed on a Celoxica RC1000-PP board. The design was synthesized using Xilinx XST, and the bit-file created by XST was loaded onto the FPGA. Before the creation of the bit-file, constraints based on the pins of the RC1000-PP board were added. This was done in order to specify the locations of the various signals on the FPGA ports. The synthesis of the design with XST produced a model that had a maximum operating frequency of about 36.8MHz. For purposes of testing 10MHz and 20MHz were used as values of clock frequency for the two clocks clock_0 and clock_1. However, since the maximum clock frequency values had already been found using Synopsys, it can be said that the current design can run at clock frequencies much higher than the ones encountered on the FPGA board. Note that the XCV1000 FPGA is of the -6 speed grade. Faster FPGAs would allow higher clock frequency values. The method by which verification was done on the FPGA is illustrated in figure 9:

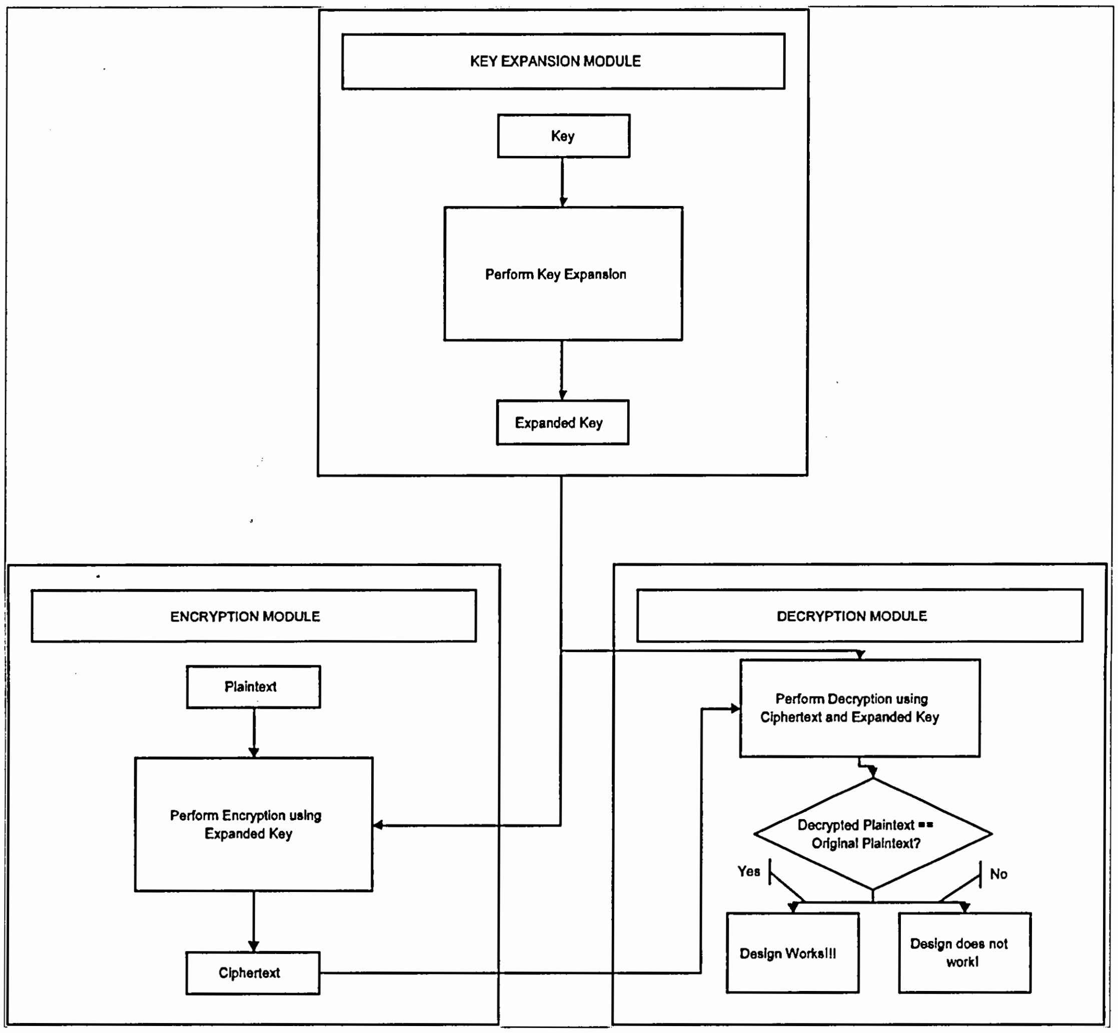

Figure 9: FPGA Verification

Essentially in the figure above, the key expansion module, the encryption module and the decryption module are all programmed onto the FPGA for 128 bits of data and key. Using a known key, key expansion is performed and an expanded key is generated. This expanded key is then inputted to the encryption module where encryption is performed using the expanded key and a known plaintext value. The ciphertext generated in the encryption module is forwarded to the decryption module where the expanded key is used to decrypt the ciphertext. The decrypted ciphertext is then compared with the original plaintext to ensure correct operation of the design on the FPGA. In order to observe the results of the comparison, the LEDs on the Celoxica RC1000-PP board are utilized. If the decrypted ciphertext and the known plaintext are the same, a certain pattern is outputted on the LEDs. Similarly, if the decrypted ciphertext and the known plaintext are not equal, another pattern is outputted on the LEDs. The results of the FPGA verification however showed that the design worked as expected and the decrypted ciphertext and the known plaintext were the same. Test vectors for the verification of the design were obtained from [18].

Additionally, to test the ability of the co-processor to process multiple data blocks and to perform encryption and decryption of large sets of data, the following method is used. The Celoxica RC1000-PP board comes equipped with four banks of SRAM which have capacities of 2MB each. Data are written into SRAM bank 0 from locations 0000h to 003Fh. Figure 10 shows the contents of the SRAM. The data from the locations 0000h to 0007h are first read into the FPGA. Using the first 128-bits (locations 0000h to 0003h) of the data as a key, an expanded key is generated. The 128-bits between locations 0000h and 0003h also act as the first data block, while data in locations 0004h through 0007h act as the second data block. Using the expanded key, the first data and second data blocks are encrypted and written to locations 0040h through 0043h, and 0044h through 0047h respectively. Using the same key and the same protocol for dividing the data into the two plaintext blocks, data in subsequent locations are encrypted and written to the SRAM. Using the encrypted values and the expanded key, the plaintext values are obtained using decryption. Once again, the calculated and original plaintext values are compared to ensure correctness of the design. In figure 10, locations 0000h through 003Fh contain the original plaintext, locations 0040h through 007Fh contain encrypted values, and locations 0080h through 00BFh contain the plaintext calculated from the encrypted values. As shown, the values of the original and calculated plaintext values are in agreement, thereby proving the correctness of the design's operation. Additionally, the encrypted values are decrypted with the program used to profile the operation of the various steps in Rijndael, and with the code from [18]. Also, the ciphertext obtained from the programs mentioned above, is compared with the ciphertext shown in figure 10. The aim of this exercise is to prove that the encryption and decryption are in fact in keeping with the Rijndael specifications, and that the results obtained are not merely obtained because encryption and decryption are the inverse of each other. To further illustrate the encryption and decryption processes more clearly, the picture in figure 11 is encrypted. The encrypted file should be found to have been changed such that it cannot be opened in a picture editor (since the headers of the file should chang during encryption). During testing, this was indeed seen to be true. Figure 12 shows the decrypted version of the encrypted file. It can be noted that figures 11 and 12 are the same on a per pixel basis, thereby proving the working of the design.

```
0[00000] 0ffa0000 0ffa0100 0ffa0200 0ffa0300
0[00004] 0ffa0400 0ffa0500 0ffa0600 0ffa0700
0[00008] 0ffa0800 0ffa0900 0ffa0a00 0ffa0b00
0[0000c] 0ffa0c00 0ffa0d00 0ffa0e00 0ffa0f00
0[00010] 0ffa1000 0ffa1100 0ffa1200 0ffa1300
0[00014] 0ffa1400 0ffa1500 0ffa1600 0ffa1700
0[00018] 0ffa1800 0ffa1900 0ffa1a00 0ffa1b00
0[0001c] 0ffa1c00 0ffa1d00 0ffa1e00 0ffa1f00
0[00020] 0ffa2000 0ffa2100 0ffa2200 0ffa2300
0[00024] 0ffa2400 0ffa2500 0ffa2600 0ffa2700
0[00028] 0ffa2800 0ffa2900 0ffa2a00 0ffa2b00
0[0002c] 0ffa2c00 0ffa2d00 0ffa2e00 0ffa2f00
0[00030] 0ffa3000 0ffa3100 0ffa3200 0ffa3300
0[00034] 0ffa3400 0ffa3500 0ffa3600 0ffa3700
0[00038] 0ffa3800 0ffa3900 0ffa3a00 0ffa3b00
0[0003c] 0ffa3c00 0ffa3d00 0ffa3e00 0ffa3f00

0[00040] cdb6bd99 439ded41 2b7077e0 f6ce58dd
0[00044] dd14c413 2863b5d9 d639329d e3a7efe6
0[00048] af149574 f7a455be 2b307d49 c3144fd8
0[0004c] 220e7724 b1fc3ed7 44c82b74 936b38a8
0[00050] 8c00aa12 4d6e49e7 8523c2bf 9fa56c73
0[00054] 5500bdb7 74d01570 21fa6130 cde2ea60
0[00058] 0dc6b3f1 e0c96d37 d0705053 968282d0
0[0005c] 3ecc6cd8 e9220195 522a97fc d04eea10
0[00060] 0f961527 31846420 2fe8ac4f 4cd29dde
0[00064] 4369e3c6 a77ff69f 31a30c24 e21f3947
0[00068] af37fa2e 0998e4cb 65a20cd6 87bce50d
0[0006c] df21e5ee 9649c7b1 d6b0ae30 681d2add
0[00070] e51795af fa17a294 7d4d09ed 47d03fa3
0[00074] 815f6601 8545b349 eab94252 b1d9c81e
0[00078] beb2d7c1 24daf5d2 e292a9f8 da0e00f6
0[0007c] d90cb56f 331b2007 4974a1d7 e57d3d01

0[00080] 0ffa0000 0ffa0100 0ffa0200 0ffa0300
0[00084] 0ffa0400 0ffa0500 0ffa0600 0ffa0700
0[00088] 0ffa0800 0ffa0900 0ffa0a00 0ffa0b00
0[0008c] 0ffa0c00 0ffa0d00 0ffa0e00 0ffa0f00
0[00090] 0ffa1000 0ffa1100 0ffa1200 0ffa1300
0[00094] 0ffa1400 0ffa1500 0ffa1600 0ffa1700
0[00098] 0ffa1800 0ffa1900 0ffa1a00 0ffa1b00
0[0009c] 0ffa1c00 0ffa1d00 0ffa1e00 0ffa1f00
0[000a0] 0ffa2000 0ffa2100 0ffa2200 0ffa2300
0[000a4] 0ffa2400 0ffa2500 0ffa2600 0ffa2700
0[000a8] 0ffa2800 0ffa2900 0ffa2a00 0ffa2b00
0[000ac] 0ffa2c00 0ffa2d00 0ffa2e00 0ffa2f00
0[000b0] 0ffa3000 0ffa3100 0ffa3200 0ffa3300
0[000b4] 0ffa3400 0ffa3500 0ffa3600 0ffa3700
0[000b8] 0ffa3800 0ffa3900 0ffa3a00 0ffa3b00
0[000bc] 0ffa3c00 0ffa3d00 0ffa3e00 0ffa3f00
```

Figure 10: SRAM Contents

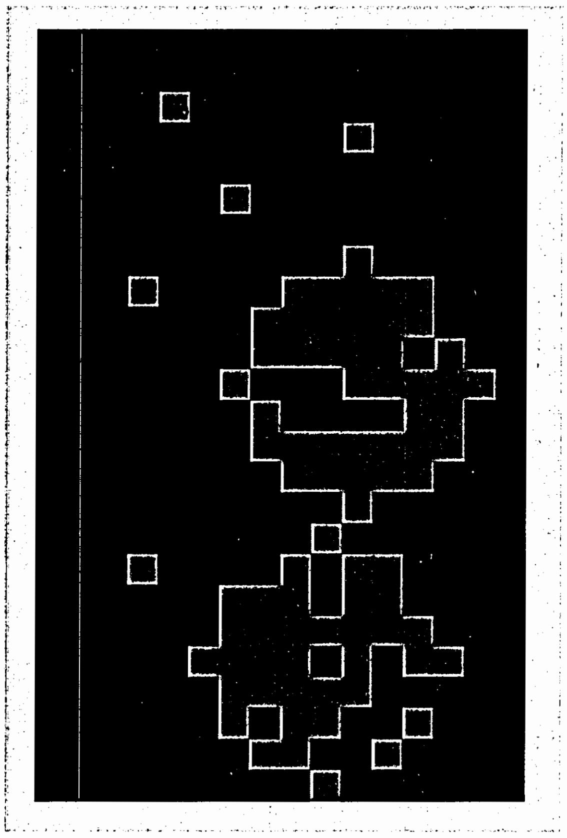

Figure 11: Original version of picture

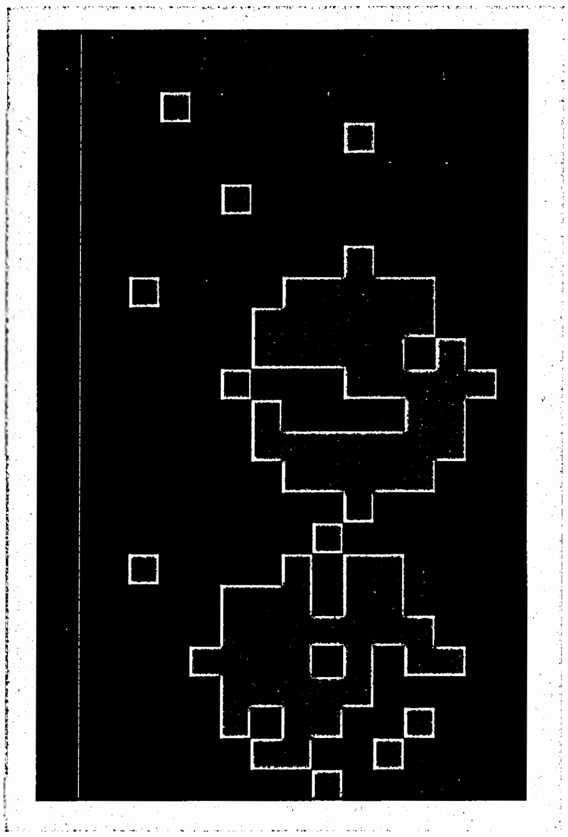

Figure 12: Decrypted version of encrypted picture

Chapter 6

CONCLUSIONS

The objective of this thesis was to build a specialized co-processor or processor plug-in to enable high performance Advanced Encryption Standard (AES) encryption and decryption.

The Rijndael/AES co-processor designed as a result of this research achieves high data-rates without compromising the inherent security of Rijndael. The optimizations proposed in this thesis have been shown to improve performance over a non-optimized model by approximately 16%. Additionally, the design consists of three different modules for encryption, decryption, and key expansion, thereby allowing for freedom of placement of these modules on the chip. Another salient feature of this design is that the three different modules can operate independently of each other.

The AES co-processor finds numerous applications in the area of data transmission. With an increase in the popularity of content-on-demand, a lot of data is being transmitted over the Internet. In order to maintain the integrity of content while being transmitted over the

Internet, AES encryption is ideal. Since encrypted data is difficult to copy, data rights can be managed very effectively. Another use of this co-processor is in the area of wireless networks. The new Wi-Fi standard 802.11i makes use of AES encryption to ensure integrity of messages [19]. The co-processor designed during this research can easily handle the up to 54Mbps data rates that the 802.11x standards provide. Additionally, since the encryption and decryption modules do not share any hardware, the design is very well suited to performing encryption and decryption at the same time.

The co-processor designed as a result of this research also finds uses in the area of secure computing and processing. A new protocol for achieving secure computing through the use of secure authenticated memory is described in Appendix A.

APPENDIX A

AUTHENTICATED SECURE MEMORY

Aside from the primary applications of the current design for encryption and decryption for real-time data transmission, particularly in the fields of image, speech, and video transmission, there is one very important application that the current design can be used for.

Over the past few years, secure processing and storage has become a major area of interest for researchers. While secure computing has been a concern for many years [20][21], Internet-based attacks which seem to be becoming increasingly destructive, have brought the seriousness of internet-based attacks to the forefront. While on the one hand, the Internet has enabled computers in one part of the world to be connected to computers in another distant location, on the other, it has also led to many attacks that exploit the security vulnerabilities of computers. For example, NIMDA an automated cyber attack which was a blend of a computer worm and a computer virus, wreaked havoc around the world in 2001. NIMDA propagated across the nation with enormous speed and tried several different ways to infect computer systems it invaded, until it got in and destroyed files. It went from nonexistent to nationwide in an hour, lasted for days, and attacked 86,000 computers. NIMDA caused significant damage in industries, forcing firms to go offline, shutting down customer access, and requiring some firms to rebuild systems entirely. Industry sources estimate that the overall financial impact of cyber attacks resulting from malicious code was around 13 billion dollars in 2001 [22]. Tamper resistant software [23], and tamper evident and tamper resistant processing have been proposed as goals of secure computing [24][25]. The currently proposed architectural schemes to aid in the construction of tamper resistant software, and to authenticate untrusted external memory using trusted on-chip storage, make use of various techniques to achieve their goals. A common feature is the use of encrypted instructions or data stored in the external memory along with a mechanism to ensure confidentiality. The encryption and decryption scheme used is usually a block cipher type of scheme. However, the schemes that are currently proposed suffer from increased latency due to the overhead that is associated with encrypting and decrypting data and instructions, and also due to the overhead associated with verifying the authenticity of these data and instructions. A major source of the increased latency is that encryption and authentication need to be done after the data or instructions have been fetched from external memory. The overhead associated with the decryption and authentication adds up with the overhead of the fetch from memory, and thus has an effect on performance. A stream cipher can be used to overcome the latency issues, since using a stream cipher allows for pre-computation of a key-stream in parallel with data-fetching, thereby hiding a major part of the latency associated with encryption/decryption. AES can be used in the CTR mode of operation to enable stream cipher based encryption [26]. However, use of a stream cipher also creates the problem of vulnerability to replay attacks. Use of hash trees to ensure authenticity of data is a well known technique [24] and can be used to prevent replay attacks. However, schemes that use hash trees can have memory overheads as high as 25% [24].

One solution to create secure and tamper proof memory could be the use of an authentication mechanism to authenticate the memory and CPU to each other. Only an authenticated CPU can write to or read from memory, and only authenticated memory can be written to or read from. This section deals with how this authentication mechanism can be realized.

Essentially, if memory does not allow a read or a write from an unauthenticated source, then it can be said that the memory is secure. This of course assumes that physical security of the memory is maintained. In other words, the memory would have to be constructed such that any physical tampering [27] of the memory would render the memory unusable by the attacker. One way to achieve this would be to make memory self-destruct upon physical tampering [28].

The proposed scheme to authenticate the memory and the CPU to each other is somewhat similar to mechanisms used to authenticate computers to each other. The memory and CPU would share a common secret that would be used to authenticate one to the other. This secret would be updated at frequent intervals in order to make sure that no replay attacks are possible using this secret if it were to fall into an attacker's hands.

To explain the above scheme it may be best to take the aid of an example. Consider a new computer purchased by a user. The memory and the CPU each contain one 256-bit secret value, S_0 and S_1 respectively, and a 256-bit key K_0 embedded in each of them (the secret value's length can be changed depending on the security level desired). Additionally, the CPU contains a value H_0 which is a one-way hash of S_0, while the memory contains H_1 which is a hash of S_1. SHA-256 or SHA-512 can be used to generate the hashes [29][30]. Note that neither the memory nor the CPU knows the value of each other's secret value S_i. When the system is first powered up, the CPU encrypts S_1 with K_0 and sends the encrypted value E_1 to the memory, while the memory encrypts S_0 with K_0 and sends the encrypted value E_0 to the CPU. The CPU decrypts E_0 and then performs a one-way hash on the decrypted value D_0. The calculated hash C_0 should match the stored hash H_0. Similarly, the memory decrypts E_1 and then performs a one-way hash on the decrypted value D_1. The calculated hash C_1 should match the stored hash H_1. If the calculated and stored hashes match at both the CPU and the memory, then the two devices are authenticated to each other. The CPU then sends a new session key K_1 to the memory under the protection of K_0. K_0 is now replaced both in the CPU and memory with K_1.

Now that the CPU and memory are authenticated to each other, the next round of communication can take place under the protection of K_1. As the round of communication draws to a close, the CPU sends a new key K_2 and a new shared secret

S_2 to the memory. The new key and the new secret can then be used in the next round of communication. To start the new round of communication, the CPU encrypts S_2 with K_1 and sends the encrypted value to memory. Since only a valid CPU and a valid memory know the value of K_1, only valid devices can encrypt or decrypt messages. Furthermore, the memory creates a hash of the secret S_2 and sends the hash to the CPU under the protection of K_1. The CPU also calculates a hash of S_2 and compares the calculated hash with the hash received from memory. If the memory and CPU are authenticated to each other, then communication can take place with a new encryption key.

To add more security to the above scheme, the CPU and memory can use a new encryption key at the end of every write-back. Basically, the CPU would generate a new key and attach the key to the encrypted write command and send the message to the memory. A new key at the end of each write-back is sufficient because first of all the CPU is authenticated. Secondly, even if an attacker were to manage to sniff the memory bus, he or she would not be able to achieve anything because all communication on the memory bus is encrypted.

It needs to be noted that data in memory resides in an unencrypted state. The reason why data needs to be unencrypted is that if the data is stored in the original encrypted state, the corresponding K_i would need to be stored also. However, since a write to or read from memory can only be performed by an authenticated CPU, there is no danger in storing plaintext values in memory. If additional security is desired, the data can be decrypted in memory on write-back, and then re-encrypted with a different key. The data can then be decrypted before it is read from memory again. This re-encryption in memory can be useful in making physical reading of memory impossible (of course it is assumed that the memory would self-destruct if it is physically tampered with, so this situation does not really arise).

It would seem that the cost this scheme would be exorbitant. In the case where the session key is changed after every write-back, every write-back to memory could involve the additional overhead of sending the next session key to the memory. However, it has been shown that write-backs are infrequent. Therefore, the overhead involved is tolerable. One aspect of the scheme that has not been touched so far is the generation of the secret values and session keys. These values can be generated using a white noise based random number generator [31]. Such random number generators generate values that are considered to be the closest yet to true random values.

REFERENCES

[1] AES Home Page; Dec 4, 2001; <http://csrc.nist.gov/encryption/aes/>

[2] National Institute of Standards and Technology; *AES Home Page*; 2002; <http://csrc.nist.gov/CryptoToolkit/aes/>

[3] *Rijndael Home Page*; <http://www.rijndael.com>

[4] V. Rijmen, J. Daemen; *AES Proposal Version2*, March 9, 2003

[5] FIPS 46-2 - (DES); *Data Encryption Standard*; 30 December, 1993; <http://www.itl.nist.gov/fipspubs/fip46-2.htm>

[6] A.J. Menezes, P. C. van Oorschot, S. A. Vanstone; *Handbook of Applied Cryptography*; CRC; 1996

[7] B. Schneier; *Applied Cryptography*; Addison-Wesley; 2001

[8] W. Stallings; *Cryptography and Network Security: Principles and Practice*; Prentice Hall; 1999

[9] C.P. Su, T.F. Lin, C.T. Huang, C.W. Wu; *A Highly Efficient AES Cipher Chip; ASP-DAC*, January 2003

[10] A. J. Elbirt, W. Yip, B. P. C. Chetwynd; *An FPGA Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*; IEEE Trans. VLSI Systems, Vol. 9, No. 4, August 2001

[11] H. Kuo, I. Verbauwhede, P. Schaumont; *A 2.29 Gb/s, 56 mW non-pipelined Rijndael AES Encryption IC in a 1.8 V, 0.18 um CMOS Technology*; Custom Integrated Circuits Conference 2002

[12] P. Schaumont, H. Kuo, I. Verbauwhede; *Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Core*; Design Automation Conference 2002

[13] M. McLoone, J. McCanny; *High Performance Single-Chip FPGA Rijndael Algorithm Implementations*; In Proceedings of the Cryptographic Hardware and Embedded Systems Workshop, CHES, Paris, May 2001

[14] M. Bean, C. Ficke, T. Rozylowicz, B. Weeks; *Hardware Performance simulations of Round 2 Advanced Encryption Standard Algorithms*; <http://csrc.nist.gov/encryption/aes/round2/NSAAESfinalreport.pdf>

[15] N. S. Kim, T. Mudge, R. Brown; *An 2.3Gb/s Integrated and Synthesizable AES Rijndael Core*; In Proceedings of Custom Integrated Circuits Conference; 2003

[16] Cryptography FAQ; RSA Laboratories; <http://www.rsasecurity.com/rsalabs/faq/B.html>

[17] K. Roy; *Low Power CMOS Design*; <http://www.ece.purdue.edu/~vlsi/one.pdf>

[18] B. Gladman; Cryptography Technology; <http://fp.gladman.plus.com/cryptography_technology/rijndael>

[19] National Institute of Standards And Technology; *IEEE 802.11 Overview*; <http://csrc.nist.gov/wireless/S10_802.11i%20Overview-jw1.pdf>

[20] C. J. Wilson; Aardvark: *A Highly Virus-Resistant Computer Architecture*; 1993; <http://noncorporeal.com/people/pathfinder/pdf/aardvark.pdf>

[21] W. A. Arbaugh, D. J. Farber, J. M. Smith; *A Secure and Reliable Bootstrap Architecture*; In Proceedings of IEEE Symposium on Security and Privacy; 1997

[22] National Institute of Standards And Technology; *The National Strategy to Secure Cyberspace (Draft)*; 2002; <http://www.csrc.nist.gov/policies/cyberstrategy-draft.pdf>

[23] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz; *Architectural Support For Copy and Tamper Resistant Software*; In Proceedings Of The 9th International Conference On Architectural Support For Programming Languages And Operating Systems; 2000

[24] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, S. Devadas; *Caches And Merkle Trees For Efficient Memory Integrity Verification*; In Proceedings of the 9th International Symposium on High Performance Computer Architecture; 2003

[25] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas; *AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing*; In Proceedings of the 17th International Conference on Supercomputing; 2003

[26] D. W. H. Lipmaa, P. Rogaway; Comments to NIST Concerning AES Modes of Operation: CTR-Mode Encryption; 2000; <http://csrc.nist.gov/encryption/modes/workshop1/papers/lipmaa-ctr.pdf>

[27] M. Kuhn; Design Principles for Tamper-Resistant Smartcard Processors; Usenix Workshop on Smartcard Technology; 1997; <http://www.cl.cam.ac.uk/~mgk25/sc99-tamper.pdf>

[28] B. Yee; Using Secure Coprocessors; 1994; <ftp://www.cs.ucsd.edu/pub/bsy/pub/th.psfonts.pdf>

[29] National Institute of Standards And Technology; *FIPS 180-2, Secure Hash Standard (SHS)*; 2002; <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>

[30] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, B. Schott; *Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512*; In Proceedings of the Information Security Conference; 2002

[31] Intel Corporation; Intel *Random Number Generator*; <http://www.intel.com/design/security/rng/rng.htm>