

# AES and The Cryptonite Crypto Processor

Dino Oliva  
Agere Systems  
101 Crawfords Corner Rd  
Holmdel, NJ 07733  
oliva@agere.com

Rainer Buchty  
Agere Systems  
101 Crawfords Corner Rd  
Holmdel, NJ 07733  
buchty@agere.com

Nevin Heintze  
Agere Systems  
101 Crawfords Corner Rd  
Holmdel, NJ 07733  
nch@agere.com

## ABSTRACT

CRYPTONITE is a programmable processor tailored to the needs of crypto algorithms. The design of CRYPTONITE was based on an in-depth application analysis in which standard crypto algorithms (AES, DES, MD5, SHA-1, etc) were distilled down to their core functionality. We describe this methodology and use AES as a central example. Starting with a functional description of AES, we give a high level account of how to implement AES efficiently in hardware, and present several novel optimizations (which are independent of CRYPTONITE).

We then describe the CRYPTONITE architecture, highlighting how AES implementation issues influenced the design of the processor and its instruction set. CRYPTONITE is designed to run at high clock rates and be easy to implement in silicon while providing a significantly better performance/area/power tradeoff than general purpose processors.

## Categories and Subject Descriptors

C.1.1 [Single Data Stream Architectures]: RISC/CISC, VLIW Architectures; E.3 [Data Encryption]: Standards

## General Terms

Algorithms, Security, Performance, Design

## Keywords

AES, Round Key Generation, Software Implementation, Cryptography, Processor, Architecture, High-Bandwidth, High-Speed

## 1. INTRODUCTION & MOTIVATION

Our goal is to support high-bandwidth cryptography in the context of an embedded network processing element, such as will be necessary for high-end routers/gateways or SAN (storage area network) elements in the near future. The requirements here are high bandwidth (2Gb/sec-10Gb/sec), low cost and low power. In addition, it is critical that round-key generation (a central component of both AES and DES) be performed on the fly because the storage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'03, Oct. 30–Nov. 1, 2003, San Jose, California, USA.  
Copyright 2003 ACM 1-58113-676-5/03/0010 ...\$5.00.

requirements for precomputation of round keys is not feasible in an embedded processing element handling up tens of thousands of simultaneous ongoing security sessions.

One way to meet these requirements is to build hardware ASIC blocks. Such blocks can easily meet the functionality and performance requirements at comparably low costs and low power consumption. However, they have fixed functionality: they are limited to the algorithm(s) for which they have been designed. Moreover, implementing all known algorithms and configurations can lead to a proliferation of ASIC blocks.

Another approach is to use general purpose processors. For example, a high-end Pentium can support encryption rates approaching 1 Gb/sec. However, a high-end Pentium is more than 100x larger and consumes 100x more power than a dedicated hardware solution with comparable performance. FPGAs could also be used to provide a flexible crypto solution. However, they are costly and are not well suited to high volume products.

We take a middle approach: we design a programmable processor for crypto applications. Our design was based on an in-depth application analysis of a number of standard crypto algorithms, including AES, DES, MD5 and SHA-1. CRYPTONITE features a two-bank architecture where each bank contains an ALU with crypto functionality and a dedicated memory unit with a novel vector memory addressing mode enabling true parallel memory access for table-based encryption functions. In this paper we focus on AES application requirements and how these requirements influenced the CRYPTONITE architecture.

Starting with a functional description of AES, we give a high level account of how to implement AES efficiently in hardware, and present several novel optimizations. We describe the CRYPTONITE architecture, highlighting how AES implementation issues influenced the design of the processor and its instruction set. We present our AES implementation as realized on CRYPTONITE. We then give performance comparisons of CRYPTONITE with other solutions based on hardware encryption blocks or programmable crypto elements.

## 2. APPLICATION ANALYSIS: AES

In this section we give an overview of the AES algorithm. AES is the NIST-appointed successor to the existing DES encryption standard, selected during an open competition of several algorithms. The competition rules included the requirement that the algorithm can be easily implemented in hardware and on a broad range of processor architectures.

The AES standard[8] sets data block sizes at 128 bits while key sizes can be 128, 192, or 256 bits. For clarity, we will first consider only the 128 bit key case and will later discuss extending the work to the other key sizes.

In AES, the primary data structure is a 4 x 4 matrix of 1 byte quantities:

$$matrix_{4,4} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

Both the data and the key are copied into such matrices for use in the encryption/decryption calculation. The matrix that the data is copied into is known specifically as the state. The data block is copied into the state horizontally, row by row:

$$state = \begin{pmatrix} d_0 & d_1 & d_2 & d_3 \\ d_4 & d_5 & d_6 & d_7 \\ d_8 & d_9 & d_{10} & d_{11} \\ d_{12} & d_{13} & d_{14} & d_{15} \end{pmatrix}$$

where  $d_i$  refers to the  $i$ th byte in the input data. The initial key is copied into the key matrix similarly:

$$key = \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{pmatrix}$$

where  $k_i$  refers to the  $i$ th byte of the initial key.

**Notation:** In the discussion that follows, we use  $row_i(m)$  to denote the extraction of the  $i$ th row from matrix  $m$  and  $col_i(m)$  to denote the extraction of the  $i$ th column from matrix  $m$ . Additionally, the arithmetic in AES is defined over the  $GF\ 2^8$  algebra<sup>1</sup> and we use  $\oplus$ ,  $\ominus$ , and  $\otimes$  to explicitly denote addition, subtraction, and multiplication, respectively, in  $GF\ 2^8$ . We overload these operators to matrices by applying them point-wise in the obvious way. Finally, we use  $t[i]$  to denote 0-based  $i$ th element of table  $t$ .

## 2.1 Encryption

The AES encryption algorithm for a 128-bit key consists of 10 rounds. Each round is the same except for the final round which performs a subset of the transformations performed in the other rounds. The state, which is initialized to the input data block, holds the intermediate results during encryption and ultimately holds the encrypted data when the process is completed. Each round has its own key, the initial round uses the initial key and subsequent rounds use a key calculated from the key used in the previous round; we discuss key generation in Section 2.3.

The definitions of the various transformations used during encryption are given Figure 1 and the pseudo-code for the encryption algorithm is shown in Figure 2. Each round calculation starts with a key addition that adds the key into the state. The transformation, *AddMatrix*, is a simple point-wise matrix addition.

Next, each byte is replaced by the sbox of that byte. The sbox is a bijective function on byte quantities that is defined in the AES standard[8]. The transformation, *SubstMatrix*, applies the sbox function to each byte in the state.

The rows in the state are then permuted by rotating each by a specific amount. The transformation, *ShiftRows*, cyclically shifts each row by the specified amount which are 0, 1, 2, and 3 bytes, respectively, in the case of encryption.

<sup>1</sup> $GF\ 2^8$  is a standard field over which many cryptographic algorithms are defined. Addition, subtraction, and multiplication in  $GF\ 2^8$  are very different than their counterparts in the integers. For more information on  $GF\ 2^8$ , see the AES standard[8].

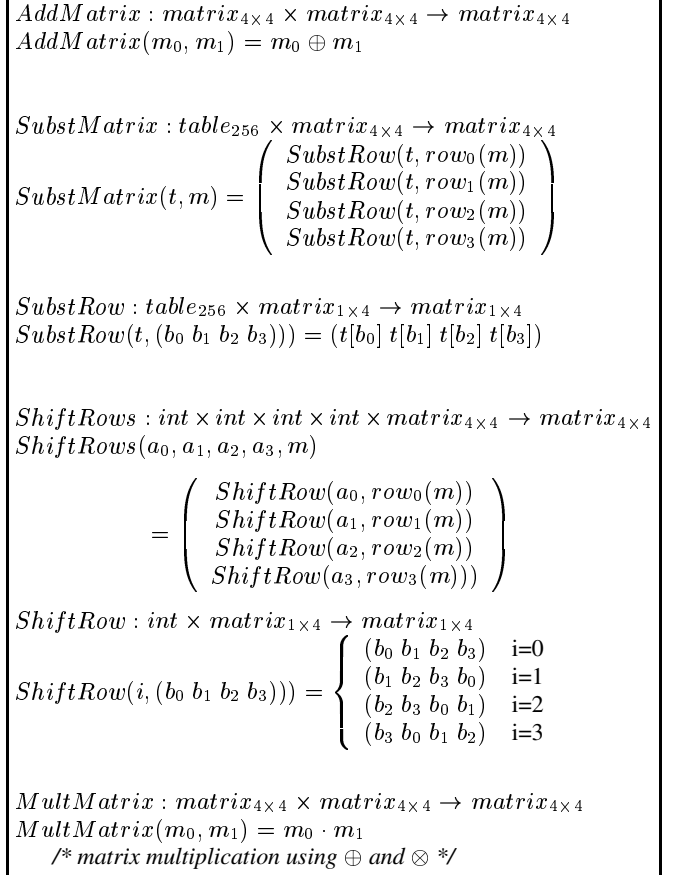


Figure 1: AES Transformations

Finally, the data in the state is intermixed across columns by the use of a variation on matrix multiplication where the scalar arithmetic operations,  $*$  and  $+$ , are those defined in the  $GF\ 2^8$  algebra,  $\otimes$  and  $\oplus$ . The state is transformed by multiplying it with the constant matrix:

$$encrypt\_matrix = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

The transformation, *MultMatrix*, is simply matrix multiplication using  $\otimes$  and  $\oplus$ .

## 2.2 Decryption

The decryption algorithm is similar to the encryption algorithm and uses the same basic transformations. As it is undoing encryption, it starts with the last key and essentially runs the encryption algorithm in reverse such that the transformations are performed in reverse order. Also, for each of the transformations, the parameters are basically inverses of the parameters used for encryption. Specifically, the table for doing substitution is the inverse of the table used in encryption, the cyclic shifts are the inverses of the cyclical shifts in encryption, and the matrix multiplied with the state is:

```

ROUNDS = 10
state: matrix[4][4]
key:   matrix[4][4]

encrypt ()
{
  for (i=0;i<ROUNDS-1;i++)
  {
    state = AddMatrix(state,key)
    state = SubstMatrix(sbox,state)
    state = ShiftRows(0,1,2,3,state)
    state = MultMatrix(encrypt_matrix,state)
    key   = NextKey(i,key)
  }

  // final round
  state = AddMatrix(state,key)
  state = SubstMatrix(sbox,state)
  state = ShiftRows(0,1,2,3,state)

  key   = NextKey(ROUNDS-1,key)
  state = AddMatrix(state,key)
}

```

Figure 2: AES Encryption

```

ROUNDS = 10
state: matrix[4][4]
key:   matrix[4][4]

decrypt ()
{
  // initial round
  state = AddMatrix (state,key)
  state = ShiftRows (0, -1%4, -2%4, -3%4,
                    state)
  state = SubstMatrix (invSbox,state)

  for (i=ROUNDS-2,i>0;i--)
  {
    key = prevKey (i,key)
    state = AddMatrix (state,key)
    state = MultMatrix(decrypt_matrix,state)
    state = ShiftRows (0, -1%4, -2%4, -3%4,
                      state)
    state = SubstMatrix(invSbox,state)
  }
  key = prevKey (0,key)
  state = AddMatrix (state,key)
}

```

Figure 3: AES Decryption

$$decrypt\_matrix = \begin{pmatrix} 14 & 11 & 13 & 09 \\ 09 & 14 & 11 & 13 \\ 13 & 09 & 14 & 11 \\ 11 & 13 & 09 & 14 \end{pmatrix}$$

The decryption algorithm is shown in pseudo-code in Figure 3. We note that the constant matrices used in *MatrixMult* for encryption and decryption have the same overall form:

$$\begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & c_0 & c_1 & c_2 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & c_2 & c_3 & c_0 \end{pmatrix}$$

where  $c_i$  are constants. This basic structure will be important in the efficient implementation of the algorithms.

### 2.3 Key Generation

The AES standard[8] denotes a round key as a lookup into a pre-computed array of round keys and then separately describes how to generate this array from the initial key. As described in the introduction, due to the memory and bandwidth requirements, storing all of the round keys associated with a session is not feasible so we must be able to dynamically generate round keys while simultaneously performing encryption/decryption. For encryption, we only store the initial round key for a session and compute the other round keys on-the-fly using the *NextKey* transformation, derived from the AES key schedule specification. For decryption, we only store the final round key for a session and compute the other round keys on-the-fly in reverse order using *PrevKey*. Both transformations are shown in Figure 4.

The calculation of round keys is a column based calculation. Given a round key, the calculation of the next round key starts with the calculation of a temporary column value via the *Temp* function. This function takes the last column in the key and first permutes each of the individual bytes in the column using the sbox table described previously. It then permutes the column by cycli-

cally rotating it up one row. Finally, a round constant is added to the topmost byte of the column. Once this temporary value has been calculated, it is added cumulatively across the columns of the old key to get the new key. Explicitly, the temporary column is added to the first column of the old key to get the first column of the new key. This new first column is then added to the second column of the old key to get the second column of the new key and so on. The table of round constants for the *Temp* function, denoted  $rc$ , is defined in the AES standard and indexed by the current round number.

Calculating keys in reverse order is not discussed in the AES standard but the operation, *PrevKey*, is easy to derive given the definition of *NextKey*. The only tricky part is that we must first calculate the third column of the previous key before we can apply the *Temp* function (shown in *PrevKey*'s definition on Figure 4).

## 3. OVERVIEW OF AES IMPLEMENTATION

We now give an overview of the basic insights necessary to implement AES efficiently. Of particular importance is the calculation of the *MultMatrix* and *NextKey* transformations. As these are the most compute intensive transformations, they must be implemented efficiently in order to have an overall efficient implementation of AES. Doing so is not straightforward.

In this discussion, we will represent the state as two 64-bit quantities (a  $1 \times 8$  matrix of byte values). In an actual implementation, the intention is to map these 64 bit quantities onto the architectures registers (which could be 128-bit, 64-bit, 32 bit, and so on).

Rows 0 and 1 will mapped onto one 64-bit variable denoted  $S_0$  while rows 2 and 3 will be mapped onto variable  $S_1$ . The initial data block to be encrypted is laid out as:

$$S_0 = (d_0 \ d_1 \ d_2 \ d_3 \ d_4 \ d_5 \ d_6 \ d_7) \\ S_1 = (d_8 \ d_9 \ d_{10} \ d_{11} \ d_{12} \ d_{13} \ d_{14} \ d_{15})$$

Similarly the key matrix will have its first 8 bytes held in a 64-bit variable,  $K_0$ , and its second 8 bytes held in a 64-bit variable  $K_1$  (recall that our presentation will focus on 128-bit keys). The initial key value is laid out as:

$$\begin{aligned}
& \text{NextKey} : \text{int} \times \text{matrix}_{4 \times 4} \rightarrow \text{matrix}_{4 \times 4} \\
& \text{NextKey}(\text{rnd}, (\text{col}_0 \text{ col}_1 \text{ col}_2 \text{ col}_3)) = (\text{col}'_0 \text{ col}'_1 \text{ col}'_2 \text{ col}'_3) \\
& \text{where } t_{\text{rnd}} = \text{Temp}(\text{rnd}, \text{col}_3) \\
& \quad \text{col}'_0 = t_{\text{rnd}} \oplus \text{col}_0 \\
& \quad \text{col}'_1 = \text{col}'_0 \oplus \text{col}_1 \\
& \quad \text{col}'_2 = \text{col}'_1 \oplus \text{col}_2 \\
& \quad \text{col}'_3 = \text{col}'_2 \oplus \text{col}_3 \\
\\
& \text{Temp} : \text{int} \times \text{matrix}_{4 \times 1} \rightarrow \text{matrix}_{4 \times 1} \\
& \text{Temp} \left( \text{rnd}, \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \right) = \begin{pmatrix} \text{rc}[\text{rnd}] \oplus \text{sbox}[b_1] \\ \text{sbox}[b_2] \\ \text{sbox}[b_3] \\ \text{sbox}[b_0] \end{pmatrix} \\
\\
& \text{PrevKey} : \text{int} \times \text{matrix}_{4 \times 4} \rightarrow \text{matrix}_{4 \times 4} \\
& \text{PrevKey}(\text{rnd}, (\text{col}'_0 \text{ col}'_1 \text{ col}'_2 \text{ col}'_3)) = (\text{col}_0 \text{ col}_1 \text{ col}_2 \text{ col}_3) \\
& \text{where } t_{\text{rnd}} = \text{Temp}(\text{rnd}, \text{col}'_3) \\
& \quad \text{col}_0 = \text{col}'_0 \ominus t_{\text{rnd}} \\
& \quad \text{col}_1 = \text{col}'_1 \ominus \text{col}'_0 \\
& \quad \text{col}_2 = \text{col}'_2 \ominus \text{col}'_1 \\
& \quad \text{col}_3 = \text{col}'_3 \ominus \text{col}'_2
\end{aligned}$$

Figure 4: AES Key Generation

$$\begin{aligned}
K_0 &= (k_0 \ k_1 \ k_2 \ k_3 \ k_4 \ k_5 \ k_6 \ k_7) \\
K_1 &= (k_8 \ k_9 \ k_{10} \ k_{11} \ k_{12} \ k_{13} \ k_{14} \ k_{15})
\end{aligned}$$

### 3.1 AES Transformations

We now derive the operations on 64-bit quantities necessary to implement the AES transformations efficiently for our representation of the state and key. Implementing *AddMatrix* requires an operation that xors 64-bit values. With this operation, the transformation can be defined as:

$$S'_0 = \text{xor}(S_0, K_0) \quad S'_1 = \text{xor}(S_1, K_1)$$

Here  $S'_0$  and  $S'_1$  denote the new values of the variables  $S_0$  and  $S_1$ . We will use this prime notation throughout the section. Implementing the *SubstMatrix* transformation requires an operation that applies the *sbox* function to the eight bytes in a 64-bit value simultaneously. With this operation, *SubstMatrix* can also be defined as:

$$S'_0 = \text{subst}(\text{sbox}, S_0) \quad S'_1 = \text{subst}(\text{sbox}, S_1)$$

The implementation of *ShiftRows* requires an operation that cyclically shifts the high and low-order 32 bits of 64-bit values by specified amounts and can be defined as:

$$S'_0 = \text{rbl}_{0,1}(S_0) \quad S'_1 = \text{rbl}_{2,3}(S_1)$$

The final transformation in the round is *MultMatrix*. Unlike the other transformations in the round calculation, this one works on columns in the state rather than rows. Given the standard calculation of matrix multiplication, results are defined a column at a time. In particular, the calculation of the  $i$ th column is:

$$\begin{aligned}
& \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & c_0 & c_1 & c_2 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & c_2 & c_3 & c_0 \end{pmatrix} \begin{pmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{pmatrix} \\
&= \begin{pmatrix} (c_0 \otimes b_{0,i}) \oplus (c_1 \otimes b_{1,i}) \oplus (c_2 \otimes b_{2,i}) \oplus (c_3 \otimes b_{3,i}) \\ (c_3 \otimes b_{0,i}) \oplus (c_0 \otimes b_{1,i}) \oplus (c_1 \otimes b_{2,i}) \oplus (c_2 \otimes b_{3,i}) \\ (c_2 \otimes b_{0,i}) \oplus (c_2 \otimes b_{1,i}) \oplus (c_0 \otimes b_{2,i}) \oplus (c_1 \otimes b_{3,i}) \\ (c_1 \otimes b_{0,i}) \oplus (c_3 \otimes b_{1,i}) \oplus (c_3 \otimes b_{2,i}) \oplus (c_0 \otimes b_{3,i}) \end{pmatrix}
\end{aligned}$$

where  $0 \leq i, j \leq 3$ ,  $b_{i,j}$  refers to the byte in the  $i$ th row and  $j$ th column of the state, and the  $c_i$  are constants. The problem with this approach is that, as it deals with columns, a straightforward implementation calculates only one byte at a time.

We now consider how to optimize the transformation in order to come up with a more efficient way to calculate it. We'd like to be able to reformulate it such that it can be calculated row-wise, like the other transformations, so that we can do the entire transformation of two rows simultaneously. We start by looking at the calculation for all the bytes in the 0th row of the state:

$$\begin{aligned}
b'_{0,0} &= (c_0 \otimes b_{0,0}) \oplus (c_1 \otimes b_{0,1}) \oplus (c_2 \otimes b_{0,2}) \oplus (c_3 \otimes b_{0,3}) \\
b'_{0,1} &= (c_0 \otimes b_{1,0}) \oplus (c_1 \otimes b_{1,1}) \oplus (c_2 \otimes b_{1,2}) \oplus (c_3 \otimes b_{1,3}) \\
b'_{0,2} &= (c_0 \otimes b_{2,0}) \oplus (c_1 \otimes b_{2,1}) \oplus (c_2 \otimes b_{2,2}) \oplus (c_3 \otimes b_{2,3}) \\
b'_{0,3} &= (c_0 \otimes b_{3,0}) \oplus (c_1 \otimes b_{3,1}) \oplus (c_2 \otimes b_{3,2}) \oplus (c_3 \otimes b_{3,3})
\end{aligned}$$

Notice that in the calculation of row 0, we multiply the 0th byte of each row by  $c_0$ , the first byte of each row by  $c_1$ , the second byte of each row by  $c_2$ , and the final byte of each row by  $c_3$ . Using  $c^n$  to denote a  $1 \times n$  matrix where each of the elements are the constant  $c$  and  $r_i$  to denote row  $i$  of the state, the calculation for row 0 can be rewritten as:

$$r'_0 = (c_0^4 \otimes r_0) \oplus (c_1^4 \otimes r_1) \oplus (c_2^4 \otimes r_2) \oplus (c_3^4 \otimes r_3)$$

The rest of the rows can be calculated similarly:

$$\begin{aligned}
r'_1 &= (c_3^4 \otimes r_0) \oplus (c_0^4 \otimes r_1) \oplus (c_1^4 \otimes r_2) \oplus (c_2^4 \otimes r_3) \\
r'_2 &= (c_2^4 \otimes r_0) \oplus (c_3^4 \otimes r_1) \oplus (c_0^4 \otimes r_2) \oplus (c_1^4 \otimes r_3) \\
r'_3 &= (c_1^4 \otimes r_0) \oplus (c_2^4 \otimes r_1) \oplus (c_3^4 \otimes r_2) \oplus (c_0^4 \otimes r_3)
\end{aligned}$$

This gives us a hint on how matrix multiplication can be implemented efficiently. In what follows we use  $S.h$  to denote the upper 4 bytes of 64-bit quantity  $S$ ,  $S.l$  to denote the lower 4 bytes of  $S$ , and  $|$  to denote the concatenation of two quantities such that  $S = S.h|S.l$ . Noting that  $r_0 = S_0.h$ ,  $r_1 = S_0.l$ ,  $r_2 = S_1.h$ , and  $r_3 = S_1.l$ , the above calculation can be rewritten as:

$$\begin{aligned}
S'_0.h &= (c_0^4 \otimes S_0.h) \oplus (c_1^4 \otimes S_0.l) \oplus (c_2^4 \otimes S_1.h) \oplus (c_3^4 \otimes S_1.l) \\
S'_0.l &= (c_3^4 \otimes S_0.h) \oplus (c_0^4 \otimes S_0.l) \oplus (c_1^4 \otimes S_1.h) \oplus (c_2^4 \otimes S_1.l) \\
S'_1.h &= (c_2^4 \otimes S_0.h) \oplus (c_3^4 \otimes S_0.l) \oplus (c_0^4 \otimes S_1.h) \oplus (c_1^4 \otimes S_1.l) \\
S'_1.l &= (c_1^4 \otimes S_0.h) \oplus (c_2^4 \otimes S_0.l) \oplus (c_3^4 \otimes S_1.h) \oplus (c_0^4 \otimes S_1.l)
\end{aligned}$$

This formulation is much better because it is defined on 32-bit quantities rather than on 8-bit quantities as shown previously. Even so, we still cannot define the calculation in terms of 64-bit quantities because of the data dependencies. In particular, the higher order bytes of  $S_0$  depend upon the lower order bytes of both  $S_0$  and  $S_1$  and similarly for the other rows. We would like to reformulate the equations such that higher order bytes only rely upon higher

order bytes and the lower order bytes only depend upon lower order bytes so that we could define the entire transformation in terms of 64-bit operations. To accomplish this, we first reorder the equations, grouping the higher-order and lower-order terms together:

$$\begin{aligned} S'_0.h &= (c_0^4 \otimes S_0.h) \oplus (c_2^4 \otimes S_1.h) \oplus (c_1^4 \otimes S_0.l) \oplus (c_3^4 \otimes S_1.l) \\ S'_0.l &= (c_0^4 \otimes S_0.l) \oplus (c_2^4 \otimes S_1.l) \oplus (c_3^4 \otimes S_0.h) \oplus (c_1^4 \otimes S_1.h) \\ S'_1.h &= (c_2^4 \otimes S_0.h) \oplus (c_0^4 \otimes S_1.h) \oplus (c_3^4 \otimes S_0.l) \oplus (c_1^4 \otimes S_1.l) \\ S'_1.l &= (c_2^4 \otimes S_0.l) \oplus (c_0^4 \otimes S_1.l) \oplus (c_1^4 \otimes S_0.h) \oplus (c_3^4 \otimes S_1.h) \end{aligned}$$

Now postulate that we have  $S_2 = S_0.l|S_0.h$  and  $S_3 = S_1.l|S_1.h$ , we can then reformulate the computation as:

$$\begin{aligned} S'_0.h &= (c_0^4 \otimes S_0.h) \oplus (c_2^4 \otimes S_1.h) \oplus (c_1^4 \otimes S_2.h) \oplus (c_3^4 \otimes S_3.h) \\ S'_0.l &= (c_0^4 \otimes S_0.l) \oplus (c_2^4 \otimes S_1.l) \oplus (c_3^4 \otimes S_2.l) \oplus (c_1^4 \otimes S_3.l) \\ S'_1.h &= (c_2^4 \otimes S_0.h) \oplus (c_0^4 \otimes S_1.h) \oplus (c_3^4 \otimes S_2.h) \oplus (c_1^4 \otimes S_3.h) \\ S'_1.l &= (c_2^4 \otimes S_0.l) \oplus (c_0^4 \otimes S_1.l) \oplus (c_1^4 \otimes S_2.l) \oplus (c_3^4 \otimes S_3.l) \end{aligned}$$

and now the calculation of higher-order bytes only depend upon higher-order bytes and the calculation of lower-order bytes only depend upon lower-order bytes. The upshot is that we can now define *MultMatrix* in terms of 64-bit quantities:

$$\begin{aligned} S'_0 &= ((c_0^4|c_0^4) \otimes S_0) \oplus ((c_2^4|c_2^4) \otimes S_1) \oplus ((c_1^4|c_3^4) \otimes S_2) \\ &\quad \oplus ((c_3^4|c_1^4) \otimes S_3) \\ S'_1 &= ((c_2^4|c_2^4) \otimes S_0) \oplus ((c_0^4|c_0^4) \otimes S_1) \oplus ((c_3^4|c_1^4) \otimes S_2) \\ &\quad \oplus ((c_1^4|c_3^4) \otimes S_3) \end{aligned}$$

In order to efficiently implement this definition of *MultMatrix*, we need an operation that can multiply (in  $GF\ 2^8$ ) point-wise the 8 bytes of a 64-bit quantity by 8 8-byte constants, an operation that can produce the swapped forms of  $S_0$  and  $S_1$ , and an operation that can xor's 4 values:

$$\begin{aligned} \text{mul}_8((c_0 \dots c_7), R_0) &= ((c_0 \dots c_7) \otimes R_0) \\ \text{swap}(R_0, R_1) &= R_0.l|R_1.h \\ \text{xor}(R_0, R_1, R_2, R_3) &= R_0 \oplus R_1 \oplus R_2 \oplus R_3 \end{aligned}$$

We can now implement the transformation as:

$$\begin{aligned} S_2 &= \text{swap}(S_0, S_0) & S_3 &= \text{swap}(S_1, S_1) \\ T_0 &= \text{mul}_8((c_0^4|c_0^4), S_0) & T'_0 &= \text{mul}_8((c_2^4|c_2^4), S_0) \\ T_1 &= \text{mul}_8((c_2^4|c_2^4), S_1) & T'_1 &= \text{mul}_8((c_0^4|c_0^4), S_1) \\ T_2 &= \text{mul}_8((c_1^4|c_3^4), S_2) & T'_2 &= \text{mul}_8((c_3^4|c_1^4), S_2) \\ T_3 &= \text{mul}_8((c_3^4|c_1^4), S_3) & T'_3 &= \text{mul}_8((c_1^4|c_3^4), S_3) \\ S'_0 &= \text{xor}(T_0, T_1, T_2, T_3) & S'_1 &= \text{xor}(T'_0, T'_1, T'_2, T'_3) \end{aligned}$$

We can also formulate matrix multiplication somewhat different way. Assuming we have  $S_2 = S_1.h|S_0.l$  and  $S_3 = S_0.h|S_1.l$ , the calculation then becomes:

$$\begin{aligned} S'_0.h &= (c_0^8 \otimes S_0.h) \oplus (c_2^8 \otimes S_1.h) \oplus (c_1^8 \otimes S_2.h) \oplus (c_3^8 \otimes S_3.h) \\ S'_0.l &= (c_0^8 \otimes S_0.l) \oplus (c_2^8 \otimes S_1.l) \oplus (c_1^8 \otimes S_2.l) \oplus (c_3^8 \otimes S_3.l) \\ S'_1.h &= (c_2^8 \otimes S_0.h) \oplus (c_0^8 \otimes S_1.h) \oplus (c_3^8 \otimes S_2.h) \oplus (c_1^8 \otimes S_3.h) \\ S'_1.l &= (c_2^8 \otimes S_0.l) \oplus (c_0^8 \otimes S_1.l) \oplus (c_3^8 \otimes S_2.l) \oplus (c_1^8 \otimes S_3.l) \end{aligned}$$

or:

$$\begin{aligned} S'_0 &= (c_0^8 \otimes S_0) \oplus (c_2^8 \otimes S_1) \oplus (c_1^8 \otimes S_2) \oplus (c_3^8 \otimes S_3) \\ S'_1 &= (c_2^8 \otimes S_0) \oplus (c_0^8 \otimes S_1) \oplus (c_3^8 \otimes S_2) \oplus (c_1^8 \otimes S_3) \end{aligned}$$

which means that for *encrypt\_matrix*, where  $c_2 = c_3 = 1$ , *MultMatrix* can be calculated as:

$$\begin{aligned} S'_0 &= (c_0^8 \otimes S_0) \oplus S_1 \oplus (c_1^8 \otimes S_2) \oplus S_3 \\ S'_1 &= S_0 \oplus (c_0^8 \otimes S_1) \oplus S_2 \oplus (c_1^8 \otimes S_3) \end{aligned}$$

This formulation also means that we can simplify our multiplication operation to multiply each byte of a 64-bit value by the same constant byte value.

In summation, we combine the transformations to get the code for encryption:

```

/* AddMatrix */
    S_0 = xor(S_0, K_0)           S_1 = xor(S_1, K_1)
/* SubstMatrix */
    S_0 = subst(sbox, S_0)       S_1 = subst(sbox, S_1)
/* ShiftRows */
    S_0 = rbl_0,1(S_0)           S_1 = rbl_2,3(S_1)
/* MultMatrix */
    S_2 = swap(S_0, S_0)         S_3 = swap(S_1, S_1)
    T_0 = mul_8(c_0^8, S_0)      T'_0 = mul_8(c_2^8, S_0)
    T_1 = mul_8(c_2^8, S_1)      T'_1 = mul_8(c_0^8, S_1)
    T_2 = mul_8((c_1^4|c_3^4), S_2) T'_2 = mul_8((c_3^4|c_1^4), S_2)
    T_3 = mul_8((c_3^4|c_1^4), S_3) T'_3 = mul_8((c_1^4|c_3^4), S_3)
    S_0 = xor(T_0, T_1, T_2, T_3) S_1 = xor(T'_0, T'_1, T'_2, T'_3)

```

### 3.2 Key Calculation

As previously noted, we represent the key in two 64-bit variables:

$$\begin{aligned} K_0 &= (k_0 \ k_1 \ k_2 \ k_3 \ k_4 \ k_5 \ k_6 \ k_7) \\ K_1 &= (k_8 \ k_9 \ k_{10} \ k_{11} \ k_{12} \ k_{13} \ k_{14} \ k_{15}) \end{aligned}$$

This allows the key addition to be done in with only two XOR operations. The problem with this representation appears when trying to calculate the next key (or previous key). As with the matrix multiplication transform, the next key calculation does its computation based on columns rather than rows. However, consider what happens if we transpose the rows and columns of the key so that the key is striped vertically across the key matrix:

$$\text{key}^T = \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Then the *Temp* operation is now performed on the last row in the matrix rather than the last column. This value is then xored across the rows of the transposed key. The transposed view of the key is represented as:

$$\begin{aligned} K_2 &= (k_0 \ k_4 \ k_8 \ k_{12} \ k_1 \ k_5 \ k_9 \ k_{13}) \\ K_3 &= (k_2 \ k_6 \ k_{10} \ k_{14} \ k_3 \ k_7 \ k_{11} \ k_{15}) \end{aligned}$$

Calculation of the *Temp* transformation proceeds in the following steps:

$$\begin{aligned} T_0 &= \text{subst}(sbox, K_3) \\ T_1 &= \text{rbl}_{0,1}(T_0) \\ T_2 &= \text{read}(rc, rnd) \\ T_3 &= \text{rbl}_{0,3}(T_2) \\ t_{rnd} &= \text{xor}(T_1, T_3) \end{aligned}$$

where *rnd* represents the round we are currently calculating. After this sequence of steps, the *Temp* value is in the lower 32 bits of

$t_{rnd}$  (recall, the temp value is only 4 bytes) which we may now use to calculate *NextKey*:

$$\begin{aligned} K'_2.h &= xor(t_{rnd}.l, K_2.h) \\ K'_2.l &= xor(K'_2.h, K_2.l) \\ K'_3.h &= xor(K'_2.l, K_3.h) \\ K'_3.l &= xor(K'_3.h, K_3.l) \end{aligned}$$

In order to give a more efficient implementation of the *NextKey* function, we lay out the round constant table in memory such that it is already rotated which gets rid of our need for  $T_3$  (e.g. we can just use  $T_2$  directly). We also introduce a new 64-bit operation:

$$fold(R_0, R_1) = xor(R_0.l, R_1.h) | xor(xor(R_0.l, R_1.h), R_1.l)$$

which allows us to calculate the next key in two steps rather than four:

$$\begin{aligned} K'_2 &= fold(t_{rnd}, K_2) \\ K'_3 &= fold(K'_2, K_3) \end{aligned}$$

Once we have calculated the key using the transposed representation, we need to be able to transpose it back again in order to add the key value to the state. Defining the following two 64-bit operations:

$$\begin{aligned} upper((b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7), (b'_0 b'_1 b'_2 b'_3 b'_4 b'_5 b'_6 b'_7)) \\ &= (b_0 b_4 b'_0 b'_4 b_1 b_5 b'_1 b'_5) \\ lower((b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7), (b'_0 b'_1 b'_2 b'_3 b'_4 b'_5 b'_6 b'_7)) \\ &= (b_2 b_6 b'_2 b'_6 b_3 b_7 b'_3 b'_7) \end{aligned}$$

allow us to transpose the new key into the non-transposed representation as follows:

$$K'_0 = upper(K_2, K_3) \quad K'_1 = lower(K_2, K_3)$$

In summation, assuming that  $K_2$  and  $K_3$  hold the transposed representation of  $K_0$  and  $K_1$ , we combine our definitions of *Temp* and *NextKey* to get the code for key calculation:

```
/* Temp */
T0 = subst(sbox, K3)
T1 = rbl0,1(T0)
T2 = read(rc, rnd)
trnd = xor(T1, T2)
/* NextKey */
K2 = fold(trnd, K2)
K3 = fold(K2, K3)
K0 = upper(K2, K3)
K1 = lower(K2, K3)
```

### 3.3 192 & 256-Bit Keys

We have focused on 128-bit keys but AES is also defined for 192 and 256-bit keys. In terms of the encryption and decryption code, this only changes the number of rounds that are performed (12 rounds for 192-bit keys and 14 round for 256-bit keys). The *NextKey* and *PrevKey* calculations are somewhat different and require an extra 64-bit temporary and the following operations:

$$\begin{aligned} swap_h(R_0, R_1) &= R_0.h | R_1.h \\ swap_l(R_0, R_1) &= R_0.l | R_1.l \end{aligned}$$

to do key generation. However, using these operations, the same techniques defined to handle 128-bit keys can be applied to handle the 192 and 256-bit cases.

## 4. CRYPTONITE MAIN DESIGN IDEAS

CRYPTONITE was explicitly designed for high throughput, small core area (with minimal external memory requirements) and low implementation complexity. Our approach combines single-cycle instruction execution with a three-stage pipeline consisting of simple stages that can be clocked at a high rate. In addition, CRYPTONITE is designed to support integrated on-the-fly round key generation.

To achieve these goals, CRYPTONITE employs a number of architectural concepts which will be discussed in this section. These concepts arose from an in-depth analysis of several cryptographic algorithms, namely DES/3DES [4], AES/Rijndael [8][7], RC6 [18], IDEA [19], and several hash algorithms (MD4 [16][15], MD5 [17], and SHA-1 [5]). Besides the following concepts, CRYPTONITE also employs a special XOR unit. However, as the special features of the XOR unit are not used for AES computation, the description for this unit has been omitted from this paper for space reasons.

### 4.1 Two-cluster Architecture

Most other work on implementing cryptographic algorithms on a programmable processor focuses solely on the core encryption algorithm and does not include round key generation (i.e. the round keys have to be precomputed). For embedded system solutions, however, on-the-fly round key generation is vital because storing and retrieving the round keys for thousands or millions of connections is not feasible. Coarse-grain parallelism can be exploited; round key calculation is usually independent of the core cryptographic operation. For example, in DES [4], the round key generation is completely independent of encryption or decryption. Certain coarse-grained parallelism exists also within hash algorithms like MD5 [17] or SHA [5]: these algorithms consist of the application of a non-linear function (NLF) followed by further adding of table-based values. In particular, the hash function's NLF can be calculated in parallel with summing up the table-based values.

Our analysis revealed that many algorithms show similar structure and would benefit from an architecture providing two independent computing clusters. Algorithm analysis further indicated that two clusters are a reasonable compromise between algorithm support and chip complexity. Adding further clusters would rarely result in speeding up computation but rather increase silicon.

### 4.2 Parameterizable Permutation Engine

Another basic operation of cryptographic operations is permutation, commonly implemented as a table lookup accessing so-called S-Boxes. In typical hardware designs of DES, these lookup tables are hardwired. However, for a programmable architecture, hardwired permutation tables are not feasible as they would limit the architecture to the provided tables. Supporting several algorithms would require separate tables and hence increase die size.

Instead, a reconfigurable permutation engine is necessary. CRYPTONITE employs a novel vector memory unit as its reconfigurable permutation engine. Algorithm analysis showed that permutation lookups are mostly done on a per-byte or smaller basis (e.g. DES: 6-bit address, 4-bit output; AES: permutation based on an 8-bit multiplication table with 256 entries). Depending on the input data size and algorithm, up to 8 parallel lookups are performed. In CRYPTONITE, the vector memory unit receives a vector of indexes and a scalar base address. This is used to address a vector of memories (i.e. n independent memory lookups are performed).

The result is a data vector. This differs from a typical vector memory unit which, when given a scalar address, returns a data vector (i.e. the  $n$  data elements that are sequentially stored at the specified address).

In non-vector addressing mode, the memory address used is the sum of a base address (from a local address register) and an optional index value. Each memory in the vector of memories receives the same address, and the results are concatenated to return a 64-bit scalar. The vector addressing mode is a slight modification to this scheme: we mask out the lower 8 bits of the base address provided by a local address register (LAR) and the 64 bits of the index vector are interpreted as eight 8-bit offsets from the base address as illustrated by Figure 5. CRYPTONITE’s vector memory unit is built from eight 8-bit memory banks.

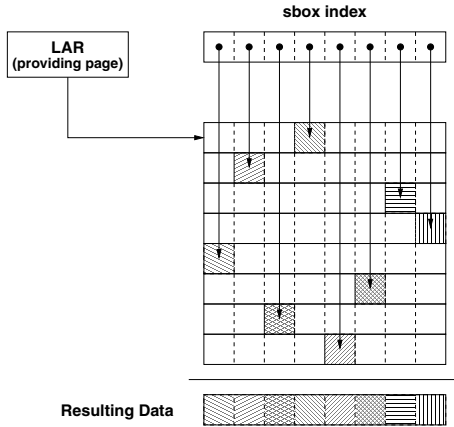


Figure 5: Vectored Memory Access

### 4.3 AES-supporting Functions

Motivated by the needs of AES (and in particular, the needs of AES round key generation), CRYPTONITE provides a family of generic fold, rotate and interleave instructions, as listed in Table 1. We expect these instructions to be of general utility.

With these functions, is it possible to implement an AES decryption routine with 81 cycles (8 cycles per round, 6 cycles setup, 3 cycles post-processing) and encryption with just 70 cycles (7 cycles per round, 1 cycle setup, 6 cycles post-processing)<sup>2</sup>. Both routines include on-the-fly round-key generation.

## 5. THE CRYPTONITE ARCHITECTURE

As previously mentioned, application analysis led to the two-cluster CRYPTONITE architecture which is pictured in Figure 6. Each cluster consists of an ALU and its accompanying data I/O unit (DIO) managing accesses to the cluster’s local data memory. A crosslinking mechanism enables data exchange between the ALUs of both clusters. The overall system is controlled by the control unit (CU) which parses the instruction stream and generates control signals for all other units. A simple external access unit (EAU) provides an easy method to access or update the contents stored in both local data memories: on external access, the CU puts all other units on hold and grants the EAU access to the internal data paths.

The CU also supplies a set of 16 registers for looping and conditional branching. 12 of these are 8-bit counter registers, the remaining four are virtual registers reflecting the two ALU’s states:

<sup>2</sup>The asymmetry arises from the fact that AES, although symmetric in terms of cryptography, is asymmetric in terms of computation. Decryption needs a higher number of table lookups.

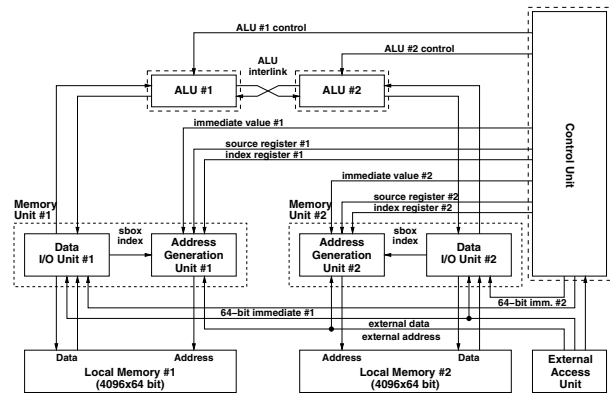


Figure 6: Overview of the CRYPTONITE architecture

we use these registers to realize conditional branches on ALU results such as zero result return (BEQ/BNE or JZ/JNZ) or carry overflow/borrow (BCC/BCS or JC/JNC). The CRYPTONITE CU is depicted in Figure 7. The use of special purpose looping registers reduces register port pressure, and routing issues. In addition, from our application analysis it was clear that most cryptographic algorithms have relatively small static loop bounds. In fact, data-dependent branching is rare (IDEA being one exception). Finally, the use of special purpose registers in conjunction with a post-decrement loop counter strategy allows us to reduce the branch penalty to 1 cycle.

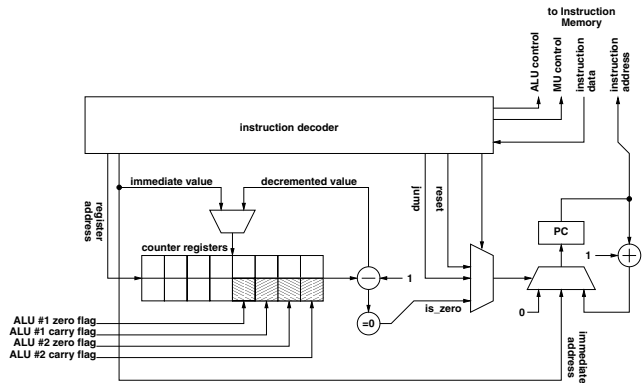


Figure 7: The CRYPTONITE Control Unit

### 5.1 The Cryptonite ALU

Much effort was put into the development of the CRYPTONITE ALU. Our target clock frequency was 400 MHz in TSMC’s 0.13  $\mu\text{m}$  process. To reach this goal, we had to carefully balance the ALU’s features (as required for the crypto algorithms) and its complexity (as dictated by technology constraints).

One result of this tradeoff is that the number of 64-bit ALU registers in each cluster was limited to four. Based on our application analysis, this was judged sufficient. To compensate for the low register count, each register can be either used as one 64-bit or two individually addressable 32-bit quantities. The use of a 64-bit architecture was motivated by both the requirements of DES and AES as well as parameterizable algorithms like RC6. To enable data exchange between both blocks the first register of each ALU is crosslinked with the first register in the other cluster. This crosslink eases register pressure as it allows cooperative computa-

Function	Description
$\text{swap}(x_{32}, y_{32})$	$f(x, y) = y \mid x$
$\text{upper}(x_{64}, y_{64})$	$* f(x, y) = x_7 \mid x_3 \mid y_7 \mid y_3 \mid x_6 \mid x_2 \mid y_6 \mid y_2$
$\text{lower}(x_{64}, y_{64})$	$* f(x, y) = x_5 \mid x_1 \mid y_5 \mid y_1 \mid x_4 \mid x_0 \mid y_4 \mid y_0$
$\text{rbl}_m(x_{64})$	$\star f(x) = (x_{63..32} \lll (m * 8) \mid (x_{31..0} \lll ((m + 1) * 8))$
$\text{rbr}_m(x_{64})$	$\star f(x) = (x_{63..32} \lll (m * 8) \mid (x_{31..0} \ggg ((m + 1) * 8))$
$\text{xor\_rbl}_m(x_{64}, y_{64})$	$f(x, y) = \text{rbl}_m(x \oplus y)$
$\text{fold}(x_{64}, y_{64})$	$\diamond f(x, y) = (x_1 \oplus y_0) \mid (x_0 \oplus y_1 \oplus y_0)$
$\text{ifold}(x_{64}, y_{64})$	$\diamond f(x, y) = (x_0 \oplus y_1) \mid (y_1 \oplus y_0)$

\* indices denote bytes    \* indices denote bits    ◊ indices denote 32-bit words

Table 1: AES-supporting ALU functions

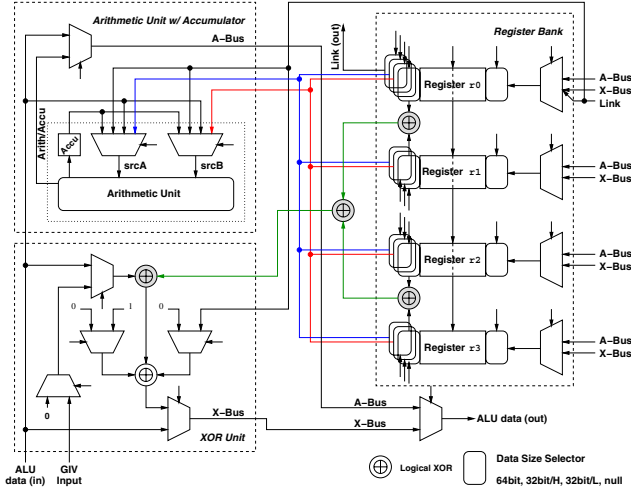


Figure 8: Overview of CRYPTONITE's ALU

tion on both ALUs (this is critical for AES and MD5). To further reduce register pressure, each ALU employs an accumulator for storing intermediate results.

The ALU itself consists of the arithmetic unit (AU) and a dedicated XOR unit (XU). As the special modes of the XU are not used for the AES implementation being discussed in this paper, the interested reader is referred to [6] for a detailed description of this unit. The AU provides conventional arithmetic and boolean operations but also specialized functions supporting certain algorithms. It follows the common 3-address model with two source registers and one destination register. These registers are limited to the ALU's accumulator, the four ALU registers, data input and output registers of the associated memory unit, and an immediate value provided by the CU.

The 64-bit results of AU and XU operations are placed on separate result buses. From these buses, either the upper 32-bits, lower 32-bits or the entire 64-bit value can be selected and stored in the assigned register (or register half). It is not possible to combine two individual 32-bit results from both result buses into one 64-bit register. Results may also be forwarded to the data unit. Figure 8 illustrates the CRYPTONITE ALU with its sub-units.

## 5.2 The Cryptonite Memory Unit

Access to local data memory is handled by the memory unit. It is composed of an address generation unit (AGU) and a data I/O unit (DIO). The address generation unit depicted in Figure 9. It generates the address for local memory access using the local address registers (LAR). The AGU contains a small add/sub/and ALU for

address arithmetic. This supports a number of addressing modes such as indexed, auto-increment and wrap-around table access as listed in Table 2.

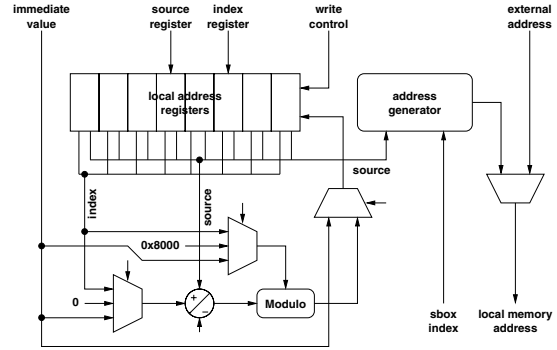


Figure 9: The Address Generation Unit

Furthermore, the sbx addressing mode performs eight parallel lookups to the 64-bit memory with 8-bit indices individual to each lookup. For a detailed description of this addressing mode please refer to Section 4.2.

The DIO, shown in Figure 10, contains two buffer registers which are the data input and data output registers (DIR and DOR). They buffer data from and to local memory. The DOR can also be used as an auxiliary register by the ALU. The DIR also serves as the sbx index to the AGU.

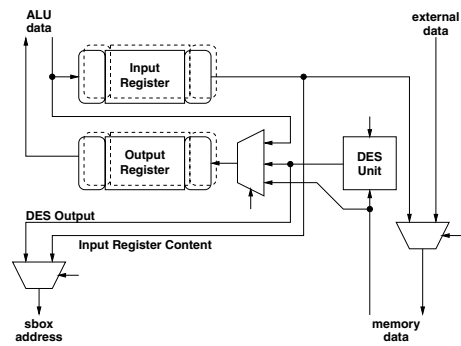


Figure 10: Data I/O from embedded SRAM

The DIO also contains a specialized DES unit. Fast DES execution not only requires highly specialized operations but also sbx access to memory. Hence the DES support instructions are integrated into the memory unit rather than the ALU.

### 5.3 AES Influence

A number of the design choices of CRYPTONITE were influenced by the needs of AES. We summarize them here:

- **two-cluster architecture:** one motivation for this choice was the loosely coupled structure of encryption/decryption and round key generation within AES.
- **parameterizable permutation engine:** while the need for a parameterized permutation engine was identified early on as an important generic crypto building block, the details of this block were specifically designed to support table-based multiplication and sbox lookups of AES.
- **AES-supporting instructions:** the inclusion of a family of generic fold, rotate and interleave instructions were directly motivated by AES needs.
- **64-bit architecture:** this was compatible with efficient AES implementation.

## 6. AES IMPLEMENTATION

In Section 3, we discussed how to efficiently implement the AES algorithm. We now describe how to map that implementation onto the Cryptonite Architecture.

In order to map the implementation onto Cryptonite, we need to map 64-bit quantities onto the architecture and map the 64-bit instructions: *xor*, *rbl<sub>i,j</sub>*, *swap*, *fold*, *upper*, *lower*, *subst*, and *mul<sub>s</sub>*.

We map the 64-bit quantities directly onto Cryptonite’s 64-bit registers. Cryptonite also directly supports all but the last two operations, *subst* and *mul<sub>s</sub>*, which will use Cryptonite’s multiway-lookup to implement. In order to do a *subst* based on the *sbox* function, we simply lay out eight copies of the *sbox* function represented in (256 entry) table form and perform the multi-way lookup on that table. Each byte in the register will then be replaced by it’s corresponding *sbox* value. We take a similar approach for the *mul<sub>s</sub>*. All of the *mul<sub>s</sub>*’s appearing in the AES calculation are the multiplications by a constant. To implement this, we first calculate the table representations of the multiplications for all of the constants (2,3,9,11,13,14) and lay them out as necessary (e.g. in encryption, we need to multiply all 8 bytes by 2, so we lay out eight copies of a table that represents the mul-by-2 function).

### 6.1 Encryption

Implementation of the encryption algorithm entails combining the various implementations of the transformations described in Section 3 with their cryptonite representation. We will first discuss the encryption process, followed by the dynamic key calculation, and finish off by showing the code which combines the two pieces. We will use symbolic names for the registers in describing the code and show how they are mapped to actual registers in the end.

Assuming the register  $S_0$  and  $S_1$  contain the state and  $K_0$  and  $K_1$  contain the round key, putting together the pieces results in the following calculation for a round:

```

/* AddMatrix & ShiftRows */
  S0 = xor-rbl0,1(S0, K0)    S1 = xor-rbl2,3(S1, K1)
/* SubstMatrix */
  S0 = m-read(sbox8, S0)    S1 = m-read(sbox8, S1)
/* MultMatrix */
  S2 = swap(S0, S0)        S3 = swap(S1, S1)
  T0 = m-read(mul28, S0)    T1 = m-read(mul28, S2)
  T2 = xor(S3, S1)          T3 = xor(S2, S0)
  T4 = m-read(mul38, S2)    T5 = m-read(mul38, S3)
  T6 = xor(T0, T2)          T7 = xor(T1, T3)
  S0 = xor(T4, T6)          S1 = xor(T5, T7)

```

Note that there is a lot of symmetry in the calculation of  $S_0$  and  $S_1$  and the actual implementation we will map the calculation of  $S_0$  to Bank A and the calculation  $S_1$  to Bank B.

There are number of modifications that we have made in order to map the transformations efficiently onto Cryptonite. In particular, we have swapped the order of *ShiftRows* and *SubstMatrix*, which we can do because *SubstMatrix* is independent of row location and *ShiftRows* does not effect the underlying byte values. We then combine *AddMatrix* and *ShiftRows* using Cryptonite’s xor followed by rotate instruction. We have also split the final xor of four intermediate results into three separate xors (which is due to register pressure). Finally, the *mul<sub>s</sub>*’s have been replaced by multi-lookups where we use the notation *table<sup>n</sup>* to denote *n* copies of *table*.

Assuming that  $K_2$  and  $K_3$  contain the transposed representation of the key in  $K_0$  and  $K_1$ , we can put together the pieces from Section 3 to the next key calculation:

```

/* Temp */
  K4 = read(rc, rnd)
  K5 = m-read(sbox8, K3)
  K6 = xor-rbl0,1(K4, K5)
/* NextKey */
  K2 = fold(K6, K2)
  K3 = fold(K2, K3)
  K0 = upper(K2, K3)
  K1 = lower(K2, K3)

```

Note that we first calculate the next values of  $K_2$  and  $K_3$  and then use the special instructions *lower* and *upper* to transpose them to get the next values of  $K_1$  and  $K_2$ . We also note that we have optimized this code by 1) using our xor followed by rotation function described previously and 2) assuming that the round key table, *rc*, has the round constants statically rotated by 3 so that the values are laid out appropriately to be *xored* in the calculation of  $K_6$ .

The complete code for the AES encryption algorithm is given in Figure 11. Similar to a vliw machine, each instruction packet is made up of several instructions, each controlling the different functional units of the machine. Each instruction packet executes simultaneously. We format the instructions such that the first column contains the instruction packet number, the second column contains the instructions to be executed on Bank A, and the third column contains the instructions to be executed on Bank B. For each bank, we also designate which functional unit the instruction is executed on. In the code shown, we use symbolic names for registers to make it easier to understand the algorithm with respect to our previous discussion. The mapping of these symbolic registers to the actual registers is given by the register map in column 4. The register map specifies where symbolic registers live after the instruction has executed and the initial mapping is given on its own<sup>3</sup>. Finally, there are a certain number of control instructions that set and increment counters and perform branches. For those instructions that use them, the control instructions are written next to the instruction packet number.

The loop kernel, which implements the round calculation, is composed of instructions 1 through 7.  $S_0$  is mapped to Bank A and  $S_1$  is mapped to Bank B. Instruction 1 performs both *AddMatrix* and *ShiftRows*, instruction 2 performs the *SubstMatrix* transformations using the multi-way lookup, and instructions 4-7 performs the *MultMatrix* calculation.

<sup>3</sup>Symbolic register may actually live in several physical registers. This is necessary because some instructions require values to be in particular registers

Addressing Mode	Address Computation	LAR Update
direct	$addr = LAR$	
<i>"</i> , w/ register modulo	$addr = LAR_x$	$LAR_x = LAR_x \% LAR_y$
<i>"</i> , w/ immediate modulo	$addr = LAR_x$	$LAR_x = LAR_x \% idx$
S-Box	$\forall 0 \leq i \leq 7 : addr_i = (LAR \wedge 0x7f00) \vee idx_i$ (LAR unchanged)	
immediate-indexed	$addr = LAR$	$LAR = LAR + idx$
ditto, w/ register modulo	$addr = LAR_x$	$LAR_x = (LAR_x + idx) \% LAR_y$
register-indexed	$addr = LAR_x$	$LAR_x = LAR_x + LAR_y$
ditto, w/ immediate modulo	$addr = LAR_x$	$LAR_x = (LAR_x + LAR_y) \% idx$

*Addressing modes written in italics are based on architectural side-effects and have not been designed in by purpose.*

**Table 2: Addressing modes supported by CRYPTONITE's AGU**

#	Bank A	Bank B	Register Map
			S0=A2 S1=B2 K0=A0,A1 K1=B0,B1
0	ctrl: i=1, c=8 aluA: K2 = upper(K0,K1)  xorA: S0 = xor(S0)	aluB: K3 = lower(K0,K1) memB: K4 = load(rcon3,0) xorB: S1 = xor(S1)	S0=A0 S1=B0 K0=A1 K1=B1 K2=A2 K3=B3,Bi K4=Bo
LOOP:			
1	aluA: S0 = xor_rbl(0,1,S0,K0)	aluB: S1 = xor_rbl(2,3,S1,K1) memB: K5 = multi-read(sbox^8,K3) xorB: K4 = xor(K4)	S0=Ai S1=Bi K5=Bo K4=B1 K2=A2 K3=B3
2	memA: S0 = multi-read(sbox^8,S0)	memB: S1 = multi-read(sbox^8,S1) aluB: k6 = xor_rbl(0,1,K4,K5)	S0=Ao S1=Bo K6=B0 K2=A2 K3=B3
3	aluA: K2 = fold(k6.l,K2) xorA: S0 = xor(S0)	xorB: S1 = xor(S1)	K2=A2 S0=A0,A1,Ai S1=B0,B1,Bi K3=B3
4	aluA: S2 = swapWords(S0,S1) memA: t0 = multi-read(2^8,S0) xorA: K2 = xor(K2)	aluB: S3 = swapWords(S1,S0) memB: t1 = multi-read(2^8,S1)	S2=A3,Ai S3=B2,Bi T0=Ao T1=Bo K2=A0,A2 S0=A1 S1=B1 K3=B3
5	xorA: t3 = xor(S2,S0)	xorB: t2 = xor(S3,S1) aluB: K3 = fold(K2.l,K3)	T3=A0 T2=B0 K3=B3 S2=A3,Ai S3=B2,Bi T0=Ao T1=Bo K2=A2 S0=A1 S1=B1
6	memA: t4 = multi-read(3^8,S2) xorA: t6 = xor(t0,t2) aluA: K2 = move(K2)	memB: t5 = multi-read(3^8,S3) xorB: t7 = xor(t1,t3) aluB: K3 = move(K3)	T4=Ao T5=Bo T6=A1 T7=B1 K2=A0,A2 K3=B0,B3,Bi
7	ctrl: brnz c LOOP, i += 1, c -= 1 aluA: K0 = upper(K2,K3)  xorA: S0 = xor(t4,t6)	aluB: K1 = lower(K2,K3) memB: K4 = load(rcon3,i) xorB: S1 = xor(t5,t7)	S0=A0 S1=B0 K0=A1 K1=B1 K2=A2 K3=B3,Bi K4=Bo
LOOP_END:			
8	aluA: S0 = xor_rbl(0,1,S0,K0)	aluB: S1 = xor_rbl(2,3,S1,K1) memB: K5 = multi-read(sbox^8,K3) xorB: K4 = xor(K4)	S0=Ai S1=Bi K5=Bo K4=B1 K2=A2 K3=B3
9	memA: S0 = multi-read(sbox^8,S0)	memB: S1 = multi-read(sbox^8,S1) aluB: k6 = xor_rbl(0,1,K4,K5)	S0=Ao S1=Bo K6=B0 K2=A2 K3=B3
10	aluA: K2 = fold(k6.l,K2)		K2=A0 S0=Ao S1=Bo K3=B3
11		aluB: K3 = fold(K2.l,K3)	K3=B0 K2=A0 S0=Ao S1=Bo
12	aluA: K0 = upper(K2,K3)	aluB: K1 = lower(K2,K3)	K0=A0 K1=B0 S0=Ao S1=Bo
13	xorA: S0 = xor(S0,K0)	xorB: S1 = xor(S1,K1)	S0=A2 S1=B2 K0=A0 K1=B0
END:			

**Figure 11: AES Encryption Code**

For the key calculation,  $K_0$  is on Bank A and  $K_1$  is on Bank B. Instruction 0 places the transform of  $K_0$  in  $K_2$  on Bank A and the transform of  $K_1$  in  $K_3$  on Bank B as well as reading the round constant into  $K_4$ . This is the invariant for the top of the loop,  $K_2$  and  $K_3$  hold the transpose of the key and  $K_4$  holds the round constant. In the loop, we intermix in the rest of the key calculation around the AES round calculation, re-establishing the loop invariant at the bottom of the loop.

## 6.2 Decryption

The mapping of decryption onto the architecture proceeds the same as the mapping of encryption. The only difference is that there are more multiplications (i.e. multi-way lookups) needed for the *MultiMatrix* transformation due to the constants in the decryption matrix (i.e. none of them are 1's, so all multiplications must occur). This adds an extra instruction in the loop kernel for round calculation which means that the decryption algorithm takes 10 cycles longer (1 cycle per round, 10 rounds per decryption) than the encryption one.

## 7. PERFORMANCE COMPARISON

Several algorithms were investigated and implemented on a custom architecture simulator. Based on the simulation results, the architecture was fine-tuned to provide minimum cycle count while maintaining maximum flexibility. In particular, the decision to incorporate the DES support instructions within the memory unit instead of the ALU (see Section 5.2) was directly motivated by simulation results.

### 7.1 DES and 3DES

As CRYPTONITE employs a dedicated DES unit, the results for the DES [4] and 3DES implementations were not surprising. CRYPTONITE reaches throughput of 732 MBit/s for DES and 244 MBit/s for 3DES. In contrast, the programmable CryptoManiac processor [25] achieves performance of 68 MBit/s for 3DES.

To quantify the tradeoffs of programmability versus performance, we give some performance numbers for DES hardware implementations. Hifn's range of cores ([14], 7711 [9], 7751 [10], 7811 [11] and 790x [12][13]) achieve performance of 143-245 MBit/s for DES and 78-252 MBit/s for 3DES. The OpenCore implementation of DES [24] achieves performance of 629 MBit/s. Arguably the state-of-the-art DES hardware implementation is by SecuCore [23]. SecuCore's high-performance DES hardware implementation (SecuCore DES/3DES Core [21]) achieves 2 GBit/s, just a factor of 2.73 better than CRYPTONITE. These results are summarized in Figures 12 and 13.

### 7.2 Advanced Encryption Standard (AES)

Figure 14 compares the AES performance of CRYPTONITE against a set of hardware implementations from Amphion [1], Hifn, and SecuCore as well as the programmable CryptoManiac. CRYPTONITE running at 400 MHz outperforms a number of hardware implementations by a factor of 1.25 to 2.6. Compared with CryptoManiac, CRYPTONITE shows an almost two times better performance.<sup>4</sup> This

<sup>4</sup>We remark that the CryptoManiac results appear to exclude round key generation whereas CRYPTONITE includes round key generation. In the RC4 discussion, [25] mentions impact from writing back into the key table. A similar note is missing for the AES implementation which suggests that only the main encryption algorithm (i.e. excluding round key generation) was coded. The cycle count of just 9 cycles per round without significant AES instruction support seems consistent with this assumption.

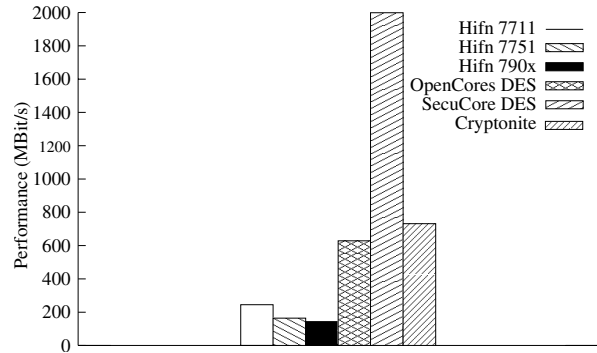


Figure 12: DES Performance Comparison

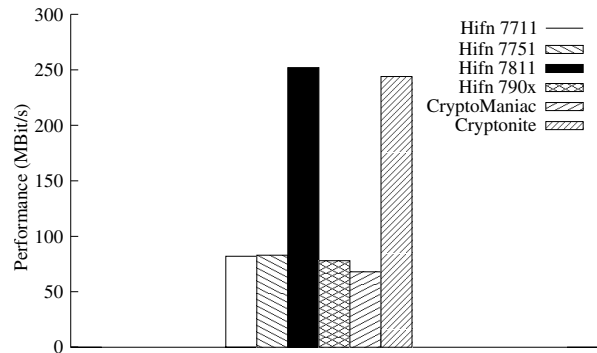


Figure 13: 3DES Performance Comparison

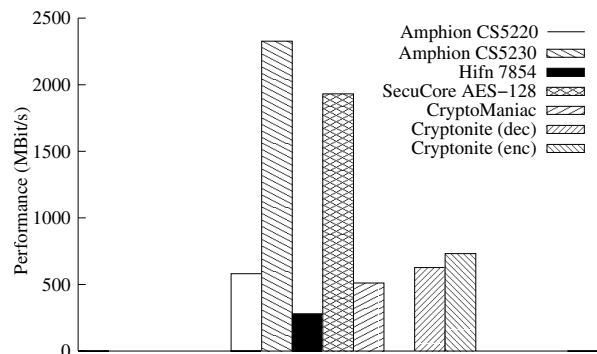


Figure 14: AES-128/128 Performance Comparison

result justifies our decision to go for simple ALUs providing more specialized functionality.

High-performance hardware AES implementations provided by Amphion CS5210-40 High Performance AES Encryption Cores [2] and CS5250-80 High Performance AES Decryption Cores [3] and SecuCore (SecuCore AES/Rijndael Core [20]) are able to outperform CRYPTONITE by a factor of 2.64. In addition, an extremely fast implementation from Amphion is even able to reach 25.6 GBit/s performance. This performance, however, is paid with an enormous gate count (10x bigger than other hardware solutions) which is why this version has not been included in the comparison chart shown in Figure 14.

### 7.3 MD5 Hashing Algorithm

CRYPTONITE performance on MD5 was 421 MBit/s at 400 MHz clock speed. It outperforms the Hifn hardware cores (7711 [9], 7751 [10], 7811 [11], and 790x [12][13]) by factors of 1.12 to 7.02. SecuCore's high-performance MD5 (SecuCore SHA-1/MD5/HMAC Core [22]), is a factor of 2.97 faster than CRYPTONITE, highlighting the programmability tradeoff. A comparison with CryptoManiac is omitted because the performance of MD5 is not reported in [25]. Figure 15 summarizes the results for MD5.

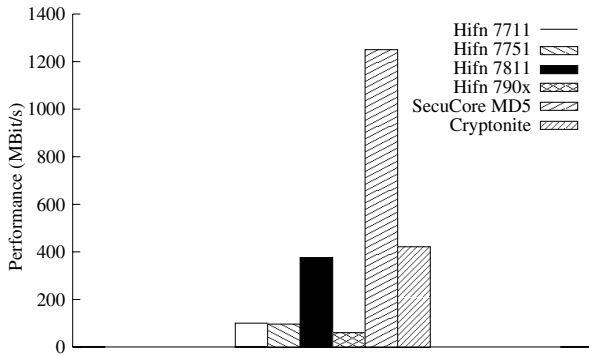


Figure 15: MD5 Performance Comparison

## 8. CONCLUDING REMARKS

The CRYPTONITE architecture was driven by an in-depth analysis of standard crypto algorithms. This analysis informed not only the topology and structure of the architecture, but also many aspects of the instruction set. Our focus in this paper has been the AES algorithm. We presented a high-level implementation strategy for AES, including a number of optimizations, which we believe will be of independent interest. We then presented an overview of the CRYPTONITE architecture and described how AES influenced various design choices of the architecture.

## 9. REFERENCES

- [1] Amphion Semiconductor Ltd. Corporate Web Site. 2001. <http://www.amphion.com>.
- [2] Amphion Semiconductor Ltd. CS5210-40 High Performance AES Encryption Cores Product Information. 2001. <http://www.amphion.com/acrobat/DS5210-40.pdf>.
- [3] Amphion Semiconductor Ltd. CS5210-40 High Performance AES Decryption Cores Product Information. 2002. <http://www.amphion.com/acrobat/DS5250-80.pdf>.
- [4] Ronald H. Brown, Mary L. Good, and Arati Prabhakar. Data Encryption Standard (DES) (FIPS 46-2). *Federal*

*Information Processing Standards Publication (FIPS)*, Dec 1993. <http://www.itl.nist.gov/fipspubs/fip46-2.html> (initial version from Jan 15, 1977).

- [5] Ronald H. Brown and Arati Prabhakar. FIPS180-1: Secure Hash Standard (SHA). *Federal Information Processing Standards Publication (FIPS)*, May 1993. <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.
- [6] Rainer Buchty. *Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications*. PhD thesis, Technische Universität München, LRR, September 2002. <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/buchty.pdf>.
- [7] J. Daemen and V. Rijmen. The block cipher Rijndael, 2000. LNCS1820, Eds: J.-J. Quisquater and B. Schneier.
- [8] J. Daemen and V. Rijmen. Advanced Encryption Standard (AES) (FIPS 197). Technical report, Katholieke Universiteit Leuven / ESAT, Nov 2001. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
- [9] Hifn Inc. 7711 Encryption Processor Data Sheet. 2002. <http://www.hifn.com/docs/a/DS-0001-04-7711.pdf>.
- [10] Hifn Inc. 7751 Encryption Processor Data Sheet. 2002. <http://www.hifn.com/docs/a/DS-0013-03-7751.pdf>.
- [11] Hifn Inc. 7811 Network Security Processor Data Sheet. 2002. <http://www.hifn.com/docs/a/DS-0018-02-7811.pdf>.
- [12] Hifn Inc. 7901 Network Security Processor Data Sheet. 2002. <http://www.hifn.com/docs/a/DS-0023-01-7901.pdf>.
- [13] Hifn Inc. 7902 Network Security Processor Data Sheet. 2002. <http://www.hifn.com/docs/a/DS-0040-00-7902.pdf>.
- [14] Hifn Inc. Corporate Web Site. 2002. <http://www.hifn.com>.
- [15] R. Rivest. RFC1186: The MD4 Message-Digest Algorithm. October 1990. <http://www.ietf.org/rfc/rfc1186.txt>.
- [16] R. Rivest. The MD4 message digest algorithm. *Advances in Cryptology - CRYPTO '90 Proceedings*, pages 303–311, 1991.
- [17] R. Rivest. RFC1312: The MD5 Message-Digest Algorithm, April 1992. <http://www.ietf.org/rfc/rfc1321.txt>.
- [18] Ronald R. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6<sup>TM</sup> Block Cipher. August 1998. <http://www.rsasecurity.com/rsalabs/rc6/>.
- [19] Bruce Schneier. 13.9: IDEA. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*, pages 370–377, 1996. ISBN 3-89319-854-7.
- [20] SecuCore Consulting Services. SecuCore AES/Rijndael Core. 2001. [http://www.secucore.com/secucore\\_aes.pdf](http://www.secucore.com/secucore_aes.pdf).
- [21] SecuCore Consulting Services. SecuCore DES/3DES Core. 2001. [http://www.secucore.com/secucore\\_des.pdf](http://www.secucore.com/secucore_des.pdf).
- [22] SecuCore Consulting Services. SecuCore SHA-1/MD5/HMAC Core. 2001. [http://www.secucore.com/secucore\\_hmac.pdf](http://www.secucore.com/secucore_hmac.pdf).
- [23] SecuCore Consulting Services. Corporate Web Site. 2002. <http://www.secucore.com/>.
- [24] Rudolf Usselman. OpenCores DES Core. Sep 2001. <http://www.opencores.org/projects/des/>.
- [25] Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *28th Annual International Symposium on Computer Architecture (ISCA 2001)*, June 2001.