Securing the AES Finalists Against Power Analysis Attacks

Thomas S. Messerges

Motorola Labs, Motorola
1301 E. Algonquin Road, Room 2712, Schaumburg, IL 60196
tomas@ccrl.mot.com

Abstract. Techniques to protect software implementations of the AES candidate algorithms from power analysis attacks are investigated. New countermeasures that employ random masks are developed and the performance characteristics of these countermeasures are analyzed. Implementations in a 32-bit, ARM-based smartcard are considered.

1 Introduction

The field of candidates for the final round of the Advanced Encryption Standard (AES) selection process has been narrowed from fifteen down to five finalists: Mars [1], RC6 [2], Rijndael [3], Serpent [4], and Twofish [5]. The cryptographic strength of the remaining AES candidates is currently being evaluated and the winning algorithm will soon be selected. One would expect that a cryptosystem using the AES winning algorithm would be unbreakable. However, history has proven that otherwise secure cryptographic algorithms can often succumb to weaknesses in their implementations [6]. Attackers of the AES algorithm may try to exploit such weaknesses.

Attacks on implementations are of particular concern to issuers and users of smartcards. Smartcards are becoming a preferred way of securely managing applications in industries such as telecommunications [7], health care [8], transportation [9], pay-TV [10] and internet commerce [11]. Smartcards have also been suggested for use in security applications such as network access [12] and physical access to locations such as automobiles, homes, and businesses [13]. Smartcards, however are potentially vulnerable to implementation attacks. These attacks include power analysis attacks [14,15], timing attacks [16,17], fault insertion attacks [18,19], and electromagnetic emission attacks [20]. All of these attacks exploit the fact that a hardware device can sometimes leak information when running a cryptographic algorithm. Kelsey et al. [21] use the term "side-channel" to describe this unintended leakage of information.

In a power analysis attack the side-channel information is the device's power consumption. The power consumption of a vulnerable device, such as a smartcard, can leak information about the secrets contained inside the device. Kocher et al. first described power analysis attacks against the DES algorithm in a 1998 technical report [22] and later followed up with a paper presented at CRYPTO '99 [14]. Researchers have also begun to study the vulnerabilities of public-key cryptosystems to these attacks [23, 24]. Power analysis attacks against the AES algorithms have also been studied [25-27]. The purpose of this paper is to introduce and describe new implementations of the AES candidate algorithms that are secure against power analysis attacks. I present masking tech-

B. Schneier (Ed.): FSE 2000, LNCS 1978, pp. 150–164, 2001.

niques that are used to randomize the fundamental operations used by the AES algorithms. I also provide results showing the performance and memory requirements for these implementations in a 32-bit, ARM-based smartcard.

1.1 Research Motivation

Tamper-resistant devices, such as smartcards, can be used to store and apply secret keys in symmetric-key cryptosystems. In a simple transaction, the smartcard might prove its authenticity through a basic challenge-and-response protocol. In such a protocol, an external device, called a reader, will challenge the smartcard to encrypt a random nonce. The smartcard will then use its secret key to encrypt this nonce and produce a response. Since the reader and smartcard share the same secret key, the reader can examine the response and verify whether the smartcard is authentic.

Honest readers possess a copy of the smartcard's secret key, thus these readers will be able to verify the result of a challenge. However, dishonest readers will not know the value of the smartcard's secret key. Nevertheless, when the power consumption leaks information from a smartcard, a dishonest reader might be able to ascertain the value of the secret key. Successful attacks against a smartcard's secret key might enable fraudulent behavior such as the counterfeiting of smartcards. Thus, smartcard issuers and users will want to ensure that the power consumption information cannot reveal the value of a secret key.

Future smartcard cryptosystems will likely use the AES algorithm. Thus, it is vital to begin understanding the issues involved in protecting smartcard AES implementations from attack.

1.2 Previous Work

The topic of power analysis attacks against the AES algorithms was discussed in March 1999 during the second AES Candidate Conference. At this conference, Biham and Shamir [25] submitted a paper that describes ways to attack many of the AES algorithms' key scheduling routines. In their paper, Biham and Shamir use the fact that a power consumption signal may leak the Hamming weight of the data being processed. They show how knowledge of the Hamming weights can enable attacks. In another paper, Chari et al. [26] look at attacks against the encrypt and decrypt routines of the AES candidate algorithms. Chari et al. assess each of the AES algorithm's vulnerabilities and, as an example, give results from an actual power analysis attack against a naive implementation of the Twofish algorithm. Daemen and Rijmen [27] also look at power analysis attacks against the AES algorithms. In their paper, Daemen and Rijmen examine the fundamental operations used by the AES algorithms and comment on possible vulnerabilities. Daemen and Rijmen also propose some possible countermeasures against these attacks.

All three of these papers recommend that comparisons of smartcard AES implementations consider the performance of versions that are secured against power analysis attacks rather than merely naive implementations.

1.3 Paper Overview

A smartcard microprocessor has a minimal amount of computing power and memory. Unfortunately, as pointed out by Chari et al. [26], software countermeasures against power analysis attacks can result in significant memory and execution time overhead. The amount of overhead seems to depend on the type and arrangement of the fundamental operations used by an algorithm. In this paper I examine the fundamental operations used by each of the AES finalist algorithms. I then develop techniques that use random masks to make software implementations of these operations resistant to power analysis attacks. Finally, I use these new countermeasures to implement masked versions of each of the remaining AES algorithms. The performance and implementation characteristics of these countermeasures in a 32-bit, ARM-based smartcard are analyzed.

The organization of this paper is as follows; first in Section 2, the fundamental operations used by each of the AES finalist algorithms are described. Next in Section 3, the basic principles of power analysis attacks are reviewed and previously suggested countermeasures are discussed. Then in Sections 4 and 5, my specific countermeasures for the AES finalist algorithms are described and implementation details are provided. Finally, the results for secure implementations of each of the algorithms are reported in Section 6.

2 Fundamental Operations in the AES Algorithms

The fundamental operations used by the AES algorithms were previously summarized by Daemen and Rijmen [27]. I review these fundamental operations and make the cautious assumption that each of these operations is potentially vulnerable to some form of power analysis attack. I then convert these vulnerable operations into secured operations using a strategy that employs random masks. Finally, I use these secured operations as the building blocks for the AES algorithms. Daemen and Rijmen [27] also proposed countermeasures, however their countermeasures are different from the masking strategies described in this paper. Daemen and Rijmen suggest software countermeasures such as data balancing and instruction sequence scrambling, whereas my countermeasures involve masking all intermediate data with random masks.

The AES candidate algorithms share many of the same fundamental operations. These operations include table lookups, bitwise AND, OR and XOR functions, shift and rotate operations, multiplication and addition modulo 2^{32} operations, permutations, polynomial multiplications over GF(2^8), and other various types of linear transformations. In this paper, I consider only the fundamental operations that are needed for implementations in a smartcard. Smartcards have a severely limited amount of memory, especially RAM, so some operations that could make the algorithms run more efficiently in a memory-rich environment are not considered in this paper.

A detailed description of the fundamental operations used in smartcard implementations of the AES algorithms is now given. A summary of these fundamental operations is also provided in Table 1.

2.1 Table Lookup Operations

All of the AES finalist algorithms, except the RC6 algorithm and a bitslice implementation of the Serpent algorithm, require table lookup operations. A table lookup operation operates on n input bits and produces m output bits. The n input bits specify an address into a table and the data at this address is the m-bit output. The size of the table grows exponentially with n, thus for smartcard implementations n must be kept fairly small. A table lookup operation T that has x as input and y as output is symbolically denoted as $y = T[x]$. The countermeasures that I will propose in Section 4 require the table be randomly masked prior to running the algorithm. This means that the table will need to be copied into RAM. Thus, table size is critical when considering secure implementations in smartcards with a minimal amount of RAM.

Mars. The Mars algorithm requires the largest table size out of all the AES algorithms. For Mars, $n = 9$ and $m = 32$ resulting in a table size of 2,048 bytes. During the forward and backwards mixing stages of Mars, this large table is viewed as two smaller tables each with $n = 8$ and $m = 32$.

Rijndael. The Rijndael algorithm actually does not need a table lookup operation because the table can be described arithmetically. However, this approach would result in a very inefficient implementation. Thus, for efficiency, Rijndael uses a table lookup operation where $n = 8$ and $m = 8$. The resulting table size is 256 bytes. More tables could be used for more efficient implementations, but these implementations are less suitable for low-memory smartcards.

Serpent. The Serpent algorithm can be implemented in either standard mode or in a more efficient bitslice mode. Only the standard mode requires table lookups. In this case, there are eight tables where $n = m = 4$. Thus, for standard mode Serpent, there is a need for 64 bytes of table memory.

Twofish. The Twofish algorithm can have its tables represented in a variety of forms. The tables are all originally derived from eight small permutation tables that have a combined memory requirement of 64 bytes. Implementations using these small tables would be inefficient, so more optimized implementations represent these small tables using two larger tables requiring a total of 512 bytes. More efficient implementations have been described that use key dependent tables [5], but these implementations are not suitable for low-memory smartcards.

2.2 Bitwise Boolean Functions

Bitwise Boolean functions include the AND, OR, and XOR functions. All of the AES algorithms use the XOR function. A bitslice implementation of the Serpent lookup tables and a routine to "fix" Mars subkeys are the only two places where the AND and OR functions are used. The logical operators $\oplus$, $\wedge$, and $\vee$ are used to denote the bitwise XOR, AND and OR functions, respectively. Smartcards can efficiently perform these operations, but if countermeasures are not taken, information regarding the operands and results may leak.

2.3 Shift and Rotate Operations

There are two types of shift and rotate operations, fixed and data dependent. All of the AES candidates use a fixed rotate or shift operation. Mars and RC6 also use data dependent rotate operations. Data dependent rotate operations can be very difficult to mask in smartcard implementations, especially if the smartcard microprocessor can only rotate by one bit at a time. Shift operations are denoted using the $\gg$ or the $\ll$ operator and rotate operations are denoted using the $\lll$ or the $\ggg$ operator.

2.4 Addition and Multiplication Modulo 2^{32}

Mars, RC6, and Twofish extensively use addition modulo 2^{32}. The Mars and RC6 algorithms also require a modular multiplication operation. In RC6, multiplication is used twice during each round. In Mars, multiplication is used once during each of the sixteen keyed transformation rounds.

2.5 Bitwise Permutations

A bitwise permutation is a rearrangement of the bits within a sequence of bits. The only AES algorithm that uses a bitwise permutation is the Serpent algorithm, and Serpent only uses the bitwise permutation if it is implemented in standard mode. A permutation operation P that has x as input and y as output is symbolically denoted as $y = \pi_P[x]$.

2.6 Polynomial Multiplications over GF(2^8)

The Rijndael and Twofish algorithm are the only AES finalists that use polynomial multiplications over GF(2^8). Polynomial multiplication can be implemented either directly or through the use of table lookup operations. When implemented directly, the multiplication decomposes into a series of conditional bitwise XOR operations and shifts. Conditional operations, however may allow for timing attacks. Therefore, smartcard implementations may instead use a combination of table lookups, XOR operations and shifts.

2.7 Linear Transformations

Serpent uses a linear transformation during each round and a recursive linear transformation during the key scheduling. Mars also uses a recursive linear transformation during the key schedule. Linear transformations can be implemented using shift and XOR operations, so techniques to protect these operations can also apply to linear transformations. A linear transform is symbolically denoted as $y = \mathrm{LT}[x]$, where x is the input to the transform and y is the output.

3 Review of Power Analysis Attacks and Countermeasures

Before looking at ways to securely implement the AES fundamental operations, it is useful to review the basic concepts of power analysis attacks. Kocher et al. [14] have described two types of attacks, a Simple Power Analysis (SPA) attack and a Differential Power Analysis (DPA) attack. An SPA attack is described as an attack where the adver-

sary can directly use a single power consumption signal to break a cryptosystem. Attacks where an adversary can learn the Hamming weight of data that was processed or can learn how branch instructions were executed are examples of SPA attacks. The information in the power signal is usually quite small; thus steps such as executing random dummy code or avoiding memory accesses by processing data in registers can often help to protect against SPA attacks.

DPA attacks, on the other hand, can be much harder to protect against. A DPA attack uses statistical analysis to extract information from a power signal. Information that might be imperceptible by using SPA can often be extracted using DPA. In its minimal form, DPA reduces to the analysis of the probability distributions of points in the power consumption signal. For example, in the original DPA attack described by Kocher et al., the means of the probability distributions are analyzed.

Let $f(p)$ be the probability distribution function of a point in the power consumption signal that is vulnerable to attack. The underlying mechanism that enables a DPA attack is the fact that $f(p)$ can be dependent on the data input to the algorithm, the data output from the algorithm, and the secret key used by the algorithm. Most operations performed by an algorithm have this property, thus most operations are potentially vulnerable to a DPA attack.

Daemen and Rijmen [27] suggested software countermeasures against DPA attacks. These countermeasures include the insertion of dummy code, power consumption randomization, and the balancing of data. These methods will degrade the strength of a DPA attack, but may not be enough to prevent an attack. Chari et al. [28] suggest that ad hoc countermeasures will not suffice since attackers can theoretically use signal processing techniques to remove dummy code and can analyze more data to overcome the effects of randomization and data balancing. They suggest a better approach is to split all intermediate data results using a secret sharing scheme, thereby forcing attackers to analyze joint distribution functions on multiple points in the power signal. Goubin et al. [29] proposed a similar strategy, called the duplication method, to protect the DES algorithm from DPA

	Mars	**RC6**	**Rijndael**	**Serpent**	**Twofish**
Table-Lookup	two 8 to 32, or one 9 to 32	none	one 8 to 8	none, or eight 4 to 4	eight 4 to 4, or two 8 to 8
(Table Size)	(2,048 bytes)	(0 bytes)	(256 bytes)	(64 bytes)	(64 or 512 bytes)
Bitwise Boolean	XOR	XOR	XOR	XOR, AND, OR	XOR
Shift or Rotate Operation	Variable	Variable	Fixed		Fixed
Multiplication mod 2^{32}	X	X			
Addition mod 2^{32}	X	X			X
Multiplication GF(2^8)			X		X
Bitwise Permutation				standard mode	
Linear Transformation	X			X	

Table 1. Summary of the Fundamental Operations in the AES Finalist Algorithms
The fundamental operations used by the AES finalist algorithms are given in the above table. The memory requirements are also provided for the table lookup operations. These memory requirements are given assuming a typical smartcard implementation.

attacks. The countermeasures I propose for the AES algorithms mask all intermediate data results and are similar to those suggested by Chari et al. and Goubin et al.

Chari et al. [28] suggested that not all intermediate data in all rounds of an algorithm need to be masked. For example, they suggest that only the first and last four rounds of DES need to use their scheme. On the other hand, Fahn et al. [30] developed an Inferential Power Analysis (IPA) method that can even attack the middle rounds of an algorithm. My implementations take a conservative approach and mask all data for all rounds.

4 Secure Implementations of the AES Fundamental Operations

My implementations resist SPA attacks by avoiding branch instructions. Other steps to prevent SPA attacks can be taken as deemed necessary, but the main focus of my attention was on resisting DPA attacks. The DPA countermeasures that I implement use random masks to obscure the calculations made by the fundamental operations. The random masks force the power consumption signals to be uncorrelated with the secret key and the input and output data; thus DPA attacks will require analysis of joint probability distributions.

In this paper I work exclusively with 32-bit words and use two types of masking operations. One type, that I refer to as *Boolean* masking, uses the bitwise XOR operation as the mask operator. The other type, that I refer to as *arithmetic* masking, uses addition and subtraction modulo 2^{32} as the mask operator. As an example of each type of masking strategy consider the masking of a word x with a random mask r_x. The results of masking x using each strategy give the following masked values x':

$$\text{Boolean mask: } x' = x \oplus r_x \quad \text{or} \quad \text{arithmetic mask: } x' = (x - r_x) \bmod 2^{32}$$

My overall strategy is to randomly mask the input data and key data prior to executing the algorithm. The algorithm is then executed using the masked data so all intermediate results of the algorithm are also masked. Since new masks are randomly chosen for each new run of the algorithm, simple statistical analysis of the algorithm's power consumption is inadequate for a successful attack. The only way attackers will be able to mount a statistical attack will be to look at joint probability distributions of multiple points in the power signal. Such an attack is referred to as a higher-order DPA [14] attack and is much more formidable to execute than a normal DPA attack.

The above masking strategy is possible if all of the fundamental operations of the AES algorithms can work with masked input data and produce masked output data. All operations, except addition and multiplication, can readily work with Boolean masked data. For addition and multiplication, arithmetic masking will be used. Many of the AES algorithms combine Boolean and arithmetic functions, thus a way to convert back and forth between Boolean masking and arithmetic masking is needed.

The conversion from one type of masking to another needs to be done in such a way to avoid vulnerabilities to DPA attacks. The algorithm shown in Fig. 1 gives one possible approach. In this approach, the unmasked data is x, the masked data is x', and the mask is r_x. The algorithm works by unmasking x using the XOR operation and then by arithmetically remasking x using modular subtraction. Of course, an unmasked x may

be vulnerable to power analysis attack. Thus, a random value C is used to randomly select whether x or $\bar{x}$ is unmasked. A DPA attack using statistical analysis of the means will not work against this algorithm because the attacker does not know whether D or $\overline{D}$ is being processed at the circled statement in the algorithm. Attackers that can determine the value of C will be able to run a DPA attack, but finding out the value of C is an SPA attack and SPA attacks are protected against using other means. A similar algorithm to that given in Fig. 1 can be used to convert from arithmetic masking back to Boolean masking.

The following sections now describe how each of the fundamental operations can work with masked data.

4.1 Table Lookup Operations

Recall that a table lookup operation takes an input x and produces an output y such that $y = T[x]$. In order to mask a table lookup operation, the table itself needs to be masked. The easiest way to mask a table is to use an input mask r_{in} and an output mask r_{out}. A Boolean masked table T' can then be defined in terms of T, r_{in}, and r_{out} where

$$T'[x] = T[x \oplus r_{\text{in}}] \oplus r_{\text{out}}$$

The masked table T' takes inputs that are masked with r_{in} and produces outputs that are masked with r_{out}. Thus, the table lookup operation has been converted to an operation that takes masked data for inputs and produces masked data as outputs.

For practical implementations in a smartcard, the random values for r_{in} and r_{out} can be chosen at the beginning of the algorithm. These values can then be used to construct the masked table that will be stored in RAM. Now, anytime a table lookup operation needs to be performed, the input data can be masked with r_{in} and the masked table can be used. The output of the masked table will be masked with r_{out}, so it can either be unmasked to reveal the true output or reused in another secure operation.

Fortunately, most of the AES algorithms use tables that are small enough to fit into the RAM available in a smartcard. Mars is the only algorithm where this solution is likely to pose a problem.

```
BooleanToArithmetic(x', r_x){
   randomly select: C = 0 or C = -1
   B = C ⊕ r_x;          /* B = r_x  or  B = r̄_x                    */
   A = B ⊕ x';           /* A = x  or  A = x̄                        */
   A = A - B;            /* A = x − r_x  or  A = x̄ − r̄_x            */
   A = A + C;            /* A = x − r_x  or  A = x̄ − r̄_x − 1        */
   A = A ⊕ C;            /* A = x − r_x                             */
   return(A);}
```

Fig. 1. Algorithm to convert from Boolean to Arithmetic Mask
This algorithm takes masked data x' and mask r_x as input, and returns a masked value A such that $(A + r_x)$ is equal to $(x' \oplus r_x)$. The circled statement is where x or $\bar{x}$ is unmasked, depending on the value of C.

4.2 Bitwise Boolean Functions

The bitwise XOR operation trivially works with Boolean masked data. To compute the XOR of two masked operands, simply compute the XOR of the masked operands and then the XOR of their corresponding masks. Thus, if the masked operands x' and y' are masked with r_x and r_y, respectively, then the masked output is $z' = x' \oplus y'$ and the new mask is $r_z = r_x \oplus r_y$.

The bitwise AND operator can also be masked, but the calculation of the mask is a little more complicated. Again, the operands x' and y' are assumed to be Boolean masked with r_x and r_y, respectively. The masked output of the AND operation is $z' = x' \wedge y'$ and the mask can be shown to be $r_z = (r_x \wedge y') \oplus (r_y \wedge x') \oplus (r_x \wedge r_y)$. A similar expression can be derived for the bitwise OR operator.

A straightforward implementation of the above expression for r_z will first calculate $r_x \wedge y'$ and then use the XOR operation to combine this with $r_y \wedge x'$ or $r_x \wedge r_y$. Unfortunately, the intermediate result of this operation will cause some data of x or y to become unmasked. A simple fix is to use an intermediate random mask during these calculations.

4.3 Shift and Rotate Operations

Fixed rotate or shift operations can easily be performed on Boolean masked data. The masks simply rotate or shift along with the data. Thus, if the masked operand is x' and the mask is r_x, then the output of a right rotate by n is $x' \ggg n$ and the new mask is $r_x \ggg n$.

For data dependent rotate operations, the rotation amount also needs to be masked. This rotation amount, however, needs to be masked with an arithmetic mask rather than a Boolean mask. Thus, the data to be rotated is still represented as masked operand x' and Boolean mask r_x, but the rotation amount is now represented as masked operand n' and arithmetic mask r_n. A masked data dependent rotate operation can now be performed using two double rotations. The masked output of the right rotate operation is $(x' \ggg n') \ggg r_n$ and the corresponding mask is $(r_x \ggg n') \ggg r_n$.

4.4 Addition and Multiplication Modulo 2^{32}

The arithmetic operations of addition and multiplication are more compatible with arithmetic masking than with Boolean masking. The addition operation trivially works with arithmetic masked data. Given masked operands x' and y', which are masked with r_x and r_y, respectively, the masked output of the addition operation is simply $z' = (x' + y') \bmod 2^{32}$ and the new mask is $r_z = (r_x + r_y) \bmod 2^{32}$.

Multiplication of masked data is more involved. The masked result for multiplying masked operands x' and y' is $z' = (x'y') \bmod 2^{32}$ and the corresponding mask can be shown to be $r_z = (r_x y' + r_y x' + r_x r_y) \bmod 2^{32}$.

4.5 Bitwise Permutations

Bitwise permutations work very nicely with Boolean masked data. If the masked operand is x' and the mask is r_x, then the masked output is $z' = \pi_P[x']$ and the corresponding mask is $r_z = \pi_P[r_x]$.

4.6 Polynomial Multiplications over GF(2^8)

There are various ways that polynomial multiplications can work with Boolean masked data. If the multiplications are performed using table lookups, shift and XOR operations, then the corresponding methods to protect these operations can be used. Also, a data byte g that is Boolean masked with r_g using the XOR operation is equivalent to polynomial $g(x)$ being arithmetically masked with polynomial $r_g(x)$ using polynomial addition. Therefore, polynomial multiplication can be secured using an approach similar to that used for multiplication modulo 2^{32}.

4.7 Linear Transformations

Non-recursive linear transformations work nicely with masked data. Given a masked operand x' which is Boolean masked with r_x, the masked output is $z' = \mathrm{LT}[x']$ and the corresponding mask is $r_z = \mathrm{LT}[r_x]$. Recursive linear transformations can be represented as a series of shift and XOR operations, so the corresponding methods to protect shift and XOR operations can be used.

5 Implementation Details

The previously described masking techniques were used to implement secure versions of the five remaining AES candidates. Naive versions of the algorithms, without the use of masking, were also implemented as a baseline. It was very difficult to determine the best implementation methods from some of the algorithm specifications, so the details provided by the algorithm authors to NIST [31] proved to be useful references.

Each of the secured algorithms begin with an initialization step where random masks are generated and used to mask the input and key data. If needed, randomized tables are also constructed. The algorithms are then executed normally, except the masked data is processed with secure versions of the fundamental operations. The efficiency is reduced because the number of computations is increased and extra memory is required for the masks and the masked table data. Many operations need to be computed twice, once for the masked data and once for the masks. The table lookup operations also require extra overhead because the input data needs to be remasked using the mask that was originally used to construct the table. Some algorithms also require a significant amount of overhead to convert between Boolean and arithmetic masking.

5.1 Implementations for a Specific Processor

I chose to use a 32-bit, ARM-based processor as an evaluation platform for my AES implementations. The ARM processor, manufactured by ARM Ltd., is a RISC machine with fourteen general use registers. A smartcard containing an ARM processor typically has 4K bytes of RAM and 48K bytes of ROM. The ARM processor also has a barrel shifter and a 32-bit multiply instruction, both of which proved useful for my AES implementations. I wrote the code for the ARM processor completely in C and compiled the code using the C compiler that comes with the “ARM Software Development Kit” available from ARM Ltd. The code was compiled using the “-Otime” compiler flag, so the resulting machine code is optimized for time rather than size. More optimal

code would likely be possible if some of the routines were written in assembly, but the intent of my experiment was to determine the relative costs of implementing secure code rather than producing the most efficient code possible.

I chose to implement my masking technique in a newer 32-bit smartcard processor, but could also have chosen an 8-bit processor. My software implementations required more than 256 bytes of RAM so low-memory 8-bit processors with only 256 bytes of RAM would not be suitable. However, newer 8-bit processors, such as the ST19 microprocessor manufactured by ST Microelectronics, would be sufficient for my implementations. An ST19 smartcard processor typically has 1K bytes of RAM and 32K bytes of ROM.

For information on 8-bit versus 32-bit implementations one could look at the work by Hachez et al. [32]. They examine implementations in both types of processors, so their results are useful for comparing 32-bit to 8-bit implementations. Comparisons of the Hachez et al. implementations to my implementations, however, should consider that the implementations by Hachez et al. were optimized to maximize performance, rather than to prevent power analysis attacks.

5.2 Algorithm Specific Issues

Masking the operations of the AES algorithms can be a very costly undertaking. The simplest approach to masking assigns each variable in the algorithm its own unique mask. Both the masks and the masked data need to be processed, thus both the amount of processing and the memory requirements double. In addition, there is also the cost associated with initializing the masks and storing and initializing masked lookup tables. Fortunately, there are a few techniques that can be used to reduce these costs.

One technique to save memory is to reuse the masks. In my implementations, two variables that are never directly combined are allowed to share the same mask, thus conserving valuable memory. Masks are also sometimes reused between rounds. For example, all the subkeys can often be masked with the same mask.

One technique to save processing is to start each round of the algorithm with a fixed set of masks. As the data for a round is manipulated, the masks will also change. In some cases, the changes to the masks are independent of the data being masked. Thus, in these cases the changes to the masks are predictable. A preprocessing step can calculate the intermediate mask values based on the initial masks and these values can be reused for every round. This technique reduces the need to continuously recalculate new masks during each round.

These techniques and other previously described masking techniques were used to implement the AES candidates. The main goal of these implementations was to keep the RAM memory requirements relatively low (less than 1K bytes) and the processing speed as fast as possible. The performance and memory requirements depend on the type of operations used in an algorithm and also the order of these operations. Comments on each of the AES implementations are now provided.

Mars. Mars was the most difficult algorithm to mask. The lookup table for Mars is 2K bytes, thus is unreasonably large to be masked and stored in RAM. As an alternative, two versions of the table were store in ROM. One table contained the normal

unmasked data and the other table contained the complement data in reverse order. A random bit was used to determine which table to use. Other issues with Mars are that the multiplications are time consuming to mask and converting between Boolean and arithmetic masking is often required.

RC6. The RC6 algorithm, due to its simple design, was very easy to mask, but again multiplications and the need for repeatedly changing from Boolean to arithmetic masking caused a good deal of overhead. In general, the calculation of a mask for a multiplication operation requires three multiplies and two add operations, thus causing much overhead. Also, masks used in multiplications are dependent on the data being masked, thus these mask values cannot be precalculated. This prevents the use of the speedup technique based on precalculating the mask values.

Rijndael. The structure of the Rijndael algorithm made masking very efficient. There was no extra overhead for arithmetic operations and the lookup table was small enough to be masked and stored in RAM. During a Rijndael round, all operations on the masks proved to be independent of the data. Thus, the technique of precalculating mask values was used extensively in this implementation.

Serpent. The Serpent algorithm was only implemented in bitslice mode. An implementation in standard mode seemed too inefficient and potentially more vulnerable to a power analysis attack because individual bits would need to be processed during the permutation operation. The main source of overhead in the Serpent implementation is due to the bitwise AND and OR operations. Calculating the masks for one AND operation requires three AND operations and four XOR operations. The OR operation also requires two additional complement operations. Also, my technique to precalculate the mask values could not be used with the AND and OR operations. Thus, even though Serpent does not use costly arithmetic operations, the overhead was still relatively high.

Twofish. The Twofish algorithm uses arithmetic operations, but does not use multiplies. Thus, the masks can be precalculated. Also, the order of operations allowed for many masks to be shared. Overall, these properties led to a more efficient implementation. The main source of overhead in Twofish is the 512 bytes of RAM that is needed to store the masked lookup table.

6 Performance Measurements

Implementations of the encrypt mode of the AES algorithms were tested. The cycle counts and memory requirements for masked and unmasked implementations on the 32-bit ARM processor are given in Table 2. The security cost for each algorithm is also given in Table 2. The security cost was calculated by dividing the masked implementation result by the unmasked result.

It is clear that when the countermeasures described in this paper were used, some of the algorithms fared better than others. As expected, algorithms that used multiplication operations, such as Mars and RC6, showed the worst degradation in performance. Serpent is also not ideally suited for masking countermeasures, but its performance is a little more acceptable. Rijndael and Twofish are the best suited algorithms for random

Cycle Count	Mars	RC6	Rijndael	Serpent	Twofish
Unmasked	9,425	5,964	7,086	15,687	19,274
Masked	72,327	46,282	13,867	49,495	36,694
Security Cost	7.67	7.76	1.96	3.16	1.90

RAM (bytes)	Mars	RC6	Rijndael	Serpent	Twofish
Unmasked	116	232	52	176	60
Masked	232	284	326	340	696
Security Cost	2.00	1.22	6.27	1.93	11.60

ROM (bytes)	Mars	RC6	Rijndael	Serpent	Twofish
Unmasked	2,984	464	1,756	2,676	1,544
Masked	7,404	1,376	2,393	9,572	2,656
Security Cost	2.48	2.97	1.36	3.58	1.72

Table 2. Implementation Results for an ARM-based Processor
The implementation results above show the cycle count and memory requirements for the masked and unmasked versions of the AES algorithms. The security cost is calculated by dividing the masked implementation requirement by the unmasked requirement.

masking. The overall results show that masking countermeasures can be implemented in smartcards. The performance is degraded, but in some applications, security against power analysis attacks is more important than efficiency.

7 Conclusions

This paper has introduced various strategies for randomly masking the operations used by the AES finalists. These strategies were used in implementations of the AES algorithms and the performance of these implementations was reported. The results provide a useful means for comparing the efficiency of secure smartcard AES implementations. Perhaps future researchers can continue searching for more efficient secure implementation techniques. The efficiency of masking arithmetic operations especially needs to be addressed and secure implementations in hardware also need to be studied. Another approach may be to mask only critical operations, such as the first and last few rounds of an algorithm. Hopefully, the results of this paper can provide some initial guidance towards the selection of the winning AES algorithm and also assist in the future development of more secure software cryptosystems.

Acknowledgments

I would like to thank my colleagues Ezzy Dabbish and Louis Finkelstein of Motorola Labs for their helpful editing. Ezzy also provided many useful discussions regarding the AES implementations. I would also like to thank my Ph.D. advisor Professor Robert H. Sloan of the University of Illinois at Chicago. Professor Sloan provided encouragement and advice for writing this paper and also suggested many edits that improved the quality of the paper.

References

1. Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford and Nevenko Zunic, "MARS – a candidate cipher for AES," IBM Corporation, AES submission available at: http://www.nist.gov/aes.
2. Ronald L. Rivest, M.J.B. Robshaw, R. Sidney and Y.L. Yin, "The RC6 Block Cipher," AES submission available at: http://www.nist.gov/aes.
3. Joan Daemen and Vincent Rijmen, "The Rijndael Block Cipher," AES submission available at: http://www.nist.gov/aes.
4. Ross Anderson, Eli Biham and Lars Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," AES submission available at: http://www.nist.gov/aes.
5. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall and Niels Ferguson, "Twofish: A 128-Bit Block Cipher," AES submission available at: http://www.nist.gov/aes.
6. Ross Anderson, "Why Cryptosystems Fail," in Proceedings of *1st ACM Conference on Computer and Communications Security*, ACM Press, November 1993, pp. 215-227.
7. R. Mitchell, "The Smart Money is on Smart Cards: Digital Cash for Use in Pay Phones," *Business Week*, no. 3437, August 14, 1995, p. 68.
8. D. Maloney, "Progress of Card Technologies in Health Care," *CardTech/SecurTech 1998 Conference Proceedings*, Vol. 2, April 1998, pp. 333-351.
9. D. Fleishman, "Transit Cooperative Research Program Study: Potential of Multipurpose Fare Media," *CardTech/SecurTech 1998 Conference Proceedings*, Vol. 2, April 1998, pp. 755-769.
10. David M. Goldschlag and David W. Kravitz, "Beyond Cryptographic Conditional Access," *Proceedings of USENIX Workshop on Smartcard Technology*, May 1999, pp. 87-91.
11. R. J. Merkert, Sr., "Using Smartcards to Control Internet Security," *CardTech/SecurTech 1999 Conference Proceedings*, May 1999, pp. 815-824.
12. N. Itoi and P. Honeyman, "Smartcard Integration with Kerberos V5," *Proceedings of USENIX Workshop on Smartcard Technology*, May 1999, pp. 51-61.
13. F. J. Valente, "Tracking Visitors in the Brazilian Coffee Palace Using Contactless Smartcards," *CardTech/SecurTech 1998 Conference Proceedings*, Vol. 2, April 1998, pp. 307-313.
14. Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential Power Analysis," *Proceedings of Advances in Cryptology–CRYPTO '99*, Springer-Verlag, 1999, pp. 388-397.
15. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, "Investigations of Power Analysis Attacks on Smartcards," *Proceedings of USENIX Workshop on Smartcard Technology*, May 1999, pp. 151-161.
16. Paul Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of Advances in Cryptology–CRYPTO '96*, Springer-Verlag, 1996, pp. 104-113.
17. J. F. Dhem, F. Koeune, P. A. Leroux, P. Mestré, J-J. Quisquater and J. L. Willems, "A Practical Implementation of the Timing Attack," in *Proceedings of CARDIS 1998*, Sept. 1998.
18. D. Boneh and R. A. Demillo and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Proceedings of Advances in Cryptology–Eurocrypt '97*, Springer-Verlag, 1997, pp. 37-51.
19. Eli Biham and Adi Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Proceedings of Advances in Cryptology–CRYPTO '97*, Springer-Verlag, 1997, pp. 513-525.

20. W. van Eck, "Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk," *Computers and Security*, v. 4, 1985, pp. 269-286.
21. J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," in *Proceedings of ESORICS '98*, Springer-Verlag, September 1998, pp. 97-110.
22. Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Introduction to Differential Power Analysis and Related Attacks," http://www.cryptography.com/dpa/technical, 1998.
23. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 144-157.
24. Jean-Sébastien Coron, "Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 292-302.
25. Eli Biham, Adi Shamir, "Power Analysis of the Key Scheduling of the AES Candidates," *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, March 1999.
26. S. Chari, C. Jutla, J.R. Rao, P. Rohatgi, "A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards," *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, March 1999.
27. Joan Daemen and Vincent Rijmen, "Resistance Against Implementation Attacks: A Comparative Study of the AES Proposals," *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, March 1999.
28. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao and Pankaj J. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks," *Proceedings of Advances in Cryptology–CRYPTO '99*, Springer-Verlag, 1999, pp. 398-412.
29. Louis Goubin and Jacques Patarin, "DES and Differential Power Analysis – The Duplication Method," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 158-172.
30. Paul N. Fahn and Peter K. Pearson, "IPA: A New Class of Power Attacks," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer-Verlag, August 1999, pp. 173-186.
31. NIST, "CD-3: AES Finalists," http://csrc.nist.gov/encryption/aes/round2/aescdrom.htm, October 1999.
32. G. Hachez, F. Koeune, J-J. Quisquater, "cAESar Results: Implementation of Four AES Candidates on Two Smart Cards," *Second Advanced Encryption Standard (AES) Candidate Conference*, http://csrc.nist.gov/encryption/aes/round1/conf2/aes2conf.htm, March 1999.