AES implementation on Smart Card

Pongnukit Juthamas, Tingthanathikul Witit

Abstract— **This paper focus on the cryptographic algorithm on smart card. Many algorithms are used to implement on smart card. The AES implementation is the one of many techniques and will be presented in this paper. The Advanced Encryption Standard (AES) is a symmetric-key block cipher algorithm, which is defined in the Federal Information Processing Standards Publication and uses keys with the lengths of 128, 192, and 256 bits and processes block lengths of 128 bit. However, recently discovered power and timing attacks. Then AES requires a very cautious approach to evaluate smart-card suitability. In this paper we will also study a method for secure implementation of the AES. Data masking technique is one method which is the most widely used countermeasure against power analysis and timing attacks at a software level.**

I. Introduction

The smart card becomes more important and widely use in our daily life. This is a credit card sized plastic card embedded with an integrated circuit chip. It is a card that can be used in many applications such as identification card, medical history card, and also debit and credit card that are using everywhere. Therefore, it is important smart card is secure and safe enough to store any information. Then it requires strong security protection and authentication. The Advanced Encryption Standard (AES) encryption algorithm for smart cards has been studied in this paper. In AES algorithm, most operations work on bytes. Therefore, every byte that appears as an intermediate result must look random to protect against side channel attacks. This paper will talk about brief description of the advanced Encryption Standard algorithm, and then the implementing AES encryption and decryption by using data masking techniques. Also, we will talk about attacks in AES implementation including zero attack, power attack, and timing attack.

II. Overview of AES algorithm

The Advanced Encryption Standard (AES) is a symmetric-key block cipher algorithm, and uses keys with the lengths of 128, 192, and 256 bits and processes block lengths of 128 bits. Internally the AES implements the Rijndael algorithm. Rijndael is an iterated block cipher, meaning that the initial input block and cipher key undergoes multiple transformation cycles before producing the output. Each intermediate cipher result is called a State. Rijndael can operate over a variable-length block using variable-length keys of a 128, 192, or 256 bits key to encrypt data blocks that are 128, 192, or 256 bits long, and all nine combinations of key and block length are possible, but AES contains only some of Rijndael's total capabilities. AES encryption and decryption are based on four different transformations that are performed repeatedly in a certain sequence; each transformation maps an input state into an output state. The transformations are grouped in rounds and are slightly different for encryption and decryption. The number of rounds depends on the key/block size. Figure 1 illustrates the general structure of the AES algorithm. Compared to encryption, decryption is simply an execution of the inverse transformations in the inverse order.

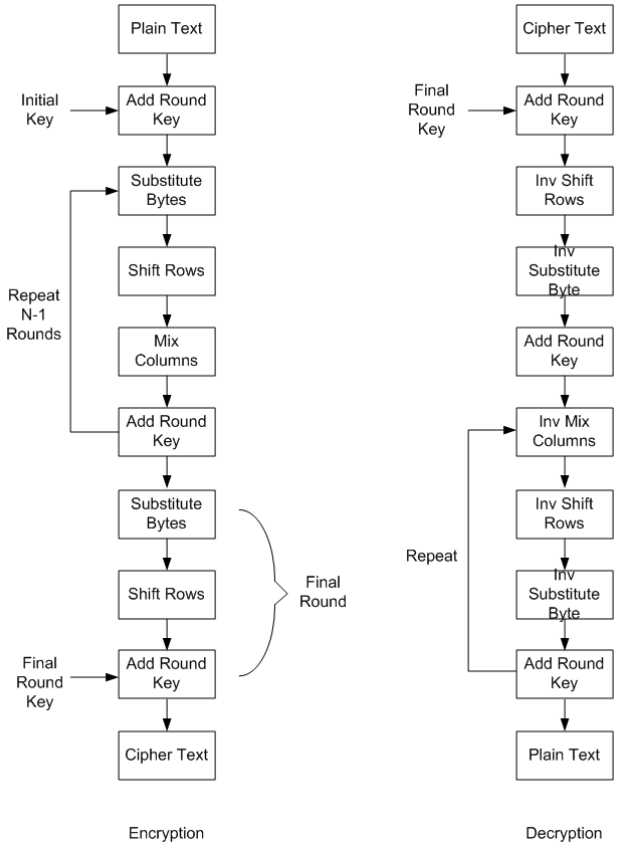

Fig. 1. The structure of the AES encryption and decryption algorithms.

III. AES implementation

The AES key changes frequently, maybe each block of data. One issue that arises in software implementations is the basic underlying architectures. The performance of AES or any other encryption algorithm depends on a particular high-level language used. In most cases, the software is strongly affects the performance. This increases the difficulty of measuring performance across a variety of platforms. It is found that Rijndael performed better on some platforms. For Rijndael, key setup or encryption/decryption is noticeably slower for 192 bits key than for 128 bits key, and slower still for 256 bits keys. Rijndael specifies more rounds for the larger key sizes, affecting the speed of both encryption/decryption and key setup.

A. AES encryption and decryption using log/alog tables

We suggest a method that combines a full protection against side channel attacks(including zero attack) with low memory requirements and low computational costs. This became possible due to a change of representation of field elements and using so called log and alog tables for arithmetic computations in Galois field on masked data. A polynomials $a_0 + a_1x + ... + a_{n-1}x^{n-1}$, where all a_i are all elements in $GF(2)$ represents elements in $GF(2)$ that is a standard basic to do calculation in a finite filed $GF(2^n)$ and addition is done modulo 2 ,However, new representation is used in this paper. It is based on the fact that all non-zero elements in a finite finite field $GF(2^n)$ can be obtained by exponentiation of a generator in this filed. So after choosing a basis for $GF(2^n)$, we look for a field generator γ and calculate all pairs (γ, i) such that $\alpha = \gamma^i,\ 0 \leq i \leq 2^n - 1,\ \alpha \in GF(2^n)$ Such representation of non-zero elements is $GF(2^n)$ is unique for a fixed primitive element γ; i is the discrete logarithm of α with respect to γ There are two table that store α, and i that is called a log table and an alog table, also each table takes $2^n - 1$ words of n bits.

The calculation in $GF(2^n)$ use log/alog table , so it can avoid all Mips-intensive operation. A sum of two file elements, α and β is calculated with three table lookups: $\alpha.\beta = \text{alog}[(\log[\alpha] + \log[\beta])\ mod(2^n - 1)]$. An inversion operation on a non zero element from $GF(2^n)$ can be calculated with two table lookups only: $\alpha^{-}1 = \text{alog}[\text{-log}[\alpha]\ mod(2^n - 1)]$ Inversion is defined only for non zero elements, but zero is always mapped into itself by conversion. Namely, $\log[0] = 2^n - 1$ and $\text{alog}[2^n - 1] = 0$.

B. Implementation of round operation using log/alog tables

Maintaining pre-computed tables to simplify operations, combining different operation of the round transformation in a single set, was suggested for AES. This approach basically combines the matrix multiplication required in the MixColumn operation with the S-box, and involves 4 tables with 256 4-byte entries.

The solution is to trade memory for speed, and use two 256-byte lookup tables for the SubByte and InvSubByte operations, while implementing the MixColumn/InvMixColumn operations separately. Each call to the MixColumn or InvMixColumn operations results in sixteen field multiplications. A straightforward implementation of the multiplication operation in the field is Mips-intensive.

Moreover, another is to implement field multiplication by repeated application of the xtime operation.This approach involves slight computational overhead and is less efficient than table lookups, but saves memory, and is often used for memory-constrained devices and 8-bit microprocessors.

The log/alog tables have been used to provide an efficient software method to compute the MixColumn and InvMixColumn operations. In comparison with a conventional table lookup for MixColumn/InvMixColumn, the new solution reduces memory requirements from 6×256 bytes to 2×256 bytes only, which is an important factor for smart cards.

Thus, the same log/alog tables can be used in both, the MixColumn and SubByte operations for encryption, and in the InvMixColumn and InvSubByte operations for decryption. Sharing log alog tables reduces the total memory requirements for complete AES implementation to 512 bytes only. At the same time, all Mips-intensive (and thus, power consuming) field arithmetic operations are replaced by table lookups, ensuring not only memory but also time and power efficient implementation. As an additional bonus, an overall program that realizes both, encryption and decryption, has a small footprint.

The combination of these three properties makes the described solution ideal for smart cards and related embedded devices.

IV. Protection against side channel attacks with data masking

Correlation between the physical measurements taken during computation and the internal state of the processing device, which itself is related to a secret key. For example, power consumption, EMF radiation, time of computations. The Differential Power Analysis(DPA)attack uses correlations between power consumption patterns and specific key-dependent bits which appear at known steps of the cryptographic computations. There are many way to combat side-channel attacks. The most powerful software countermeasure is *bit splittting* which in case when each bits is split into two shared can be reduced to *masking* data with random values. The idea how to apply data masking to AES is simple. The message, as well as the key, are marked with some random masks at the beginning of computations, and other things are almost the same as usual. However, the value of the mask at the end of some fixed step must be known in order to re-establish the expected value at the end of the execution, and it is called *mask correction*

A traditional XOR operation is used as a masking countermeasure.The computation is compatible with the AES structure except for SubByte, which is the only non-linear transformation since it uses inversion in the field. Also, it is easy to compute mask correction for all transformations in around, apart from the inversion step of the *SubByte*. If every byte A of the (initial or intermediate) state is masked with some random mask R, then $\text{OP}(A \oplus R) = \text{OP}(A) \oplus \text{OP}(R)$, where OP $\in$ MixColumn/invMixColumn, ShiftRow/InvShiftRow, AddRoundKey. Thus, given any random mask, it is easy either to pre compute the corresponding mask correction, or compute it on the fly in parallel with computations on *masked* data.

In Transformed masking method, first and additive mask is replaced by a multiplicative mask in a series of multiply and add operations in $GF(2^8)$,after which normal inversion takes place, and finally, a transformation of a multiplicative mask into an additive mask is carried out. Unfortunately, a multiplicative mask does not blind zero element

in $GF(2^n)$,enabling a so-called zero attack.

V. Inversion on masked data using log/alog-tables

We want to find value $(A^{-1}) \oplus (R)$ by the the most efficient method, and never revealing A and A^{-1} in a process. Also, we have information of $A \oplus R$, where A is a byte of a state and R is some uniformly distributed random value. Here R' can be either equal to R or any other(uniformly distributed) random value.

The solution to find the most efficient method is based on the following observation. First, a new representation allows us to infer how $((A) \oplus (R))^{-1}$ differs from A^{-1}.

(1) A field element $A \oplus R$ can be represented as $\gamma^y = \gamma^i \oplus \gamma^r$ where γ^i is a representation of the unknown byte A of the state, and γ^r is a representation of known random random mask R.

(2) By simple formulae manipulations, we obtain

$$\gamma^y = \gamma^i \oplus \gamma^r = \gamma^i.(\gamma^{r-i} \oplus 1) \ .$$

(3) Therefore, we can write

$$\gamma^{-y} = (\gamma^i \oplus \gamma^r)^{-1} = (\gamma^i)^{-1}.(\gamma^{r-i} \oplus 1)^{-1} \ .$$

Hence, $(\gamma^{r-i} \oplus 1)^{-1}$ is the mask correction for "masked inversion". Next, we show how to compute this mask correction without revealing A or A^{-1}

(1) we can consider $A \oplus R$ from a different view point, namely as

$$\gamma^y = \gamma^i \oplus \gamma^r = \gamma^r.(\gamma^{i-r} \oplus 1) \ .$$

(2) Hence, if we multiply γ^y by γ^{-r},we get

$$\gamma^y.\gamma^{-r} = \gamma^{-r}.\gamma^r.(\gamma^{i-r} \oplus 1) = \gamma^{i-r} \oplus 1 \ .$$

(3) Executing $(\gamma^{i-r} \oplus 1) \oplus 1$, we find γ^{i-r},after which using log-alog table, we easily compute $(\gamma^{i-r})^{-1} = \gamma^{r-i}$.

(4) Finally, after XOR-ing γ^{r-i} with 1 and inverting the result, we find the mask correction $(\gamma^{r-i} \oplus 1)^{-1}$.

A. MixColumn/InvMixcolumn on masked data with log/alog tables

It is easy to implement the *Mixcolumn* and *InMixColumn* operations on masked data using log/alog tables as well. Indeed, since one of the terms in each field multiplication involved in these operation is fixed, the operation is linear. Let I denotes a denotes a fixed term, then $(A \oplus R).I = (A.I) \oplus (R.I)$. The corresponding mask correction can be computed trivially as $R.I$.

Hence, each of the *Mixcolumn/InvMixColumn* operations on masked data is reduced to 2× (16× 3)table lookups using the same log/alog table that were used to compute inversion.

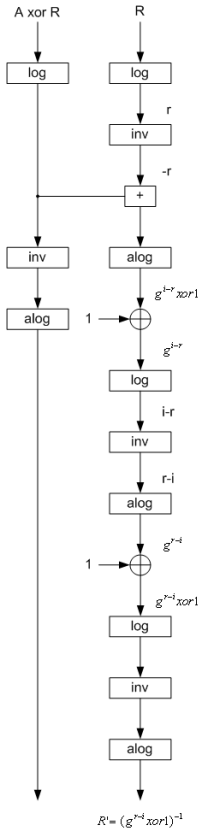

Fig. 2. Computing inversion on masked data with log/alog tables

VI. Attacks on smart cards implementation

A. Zero Attack

Zero attack is based on the fact that a multiplicative mask only non-zero values. In other words,if the actual data byte A is zero, then for any mask X,$(A \oplus X) = 0$. Moreover, multiplicative and simplified masking techniques had as subtle flaw, namely, they were vulnerable to a zero attack.

First of all, notice that all manipulations on discrete logarithms are fully protected as long as random masks R change from one run to another. On the other hand, detecting that an intermediate value γ^{i-r} is zero provides some dangerous information. Indeed, γ^{i-r} is in fact equivalent to $A \oplus R^{-1}$, where $A = (data \oplus key)$, R^{-1} is a mask. $\gamma^{i-r} = 0$ implies that either $A = 0$ or $R^{-1} = 0$, Hence an attacker may systematically try all 256 possible values for *data*in order to find the oe which turns A into zero. In order to protect the inversion on masked data from this situation, we will have to implement log/alog table lookups in a secure way.

B. Differential Power Attack

Most smart-cards use CMOS technology which consumes power only when some change occurs in the logic state of the chip. Also, smart card chips are clock driven. However, a few processes, such as on chip noise generators, operate independently of the clock and consume a small, possibly random amount of power continuously. Each clock edge triggers a sequence of power consuming events within the chip which brings it to the next state.

The instantaneous power consumption of the chip, shortly after a clock edge, is a combination of the instantaneous power consumption components from each of the events that have occurred thus far. We assume that the power consumption function of the chip is the *sum* of the power consumption functions of all the events that have taken place. We will use the term *relevant state* to subsystems that processor is accessing in that cycle. At the start of the cycle, the smart card can be in one of several states depending on the input and processing done in earlier cycles. Let S denote that set of possible relevant states when control reaches this cycle. Let ε be the space of all possible events that can occur in that cycle. For each $s \in S$, and each $e \in \varepsilon$, let *occurs(e,s)* be the binary function which is 1 if e occurs when the relevant state is s and 0 otherwise. Let *delay(e,s)* be the time delay of event e in the state s from the clock edge and let *f(e,t)* denote the power consumption impulse function of event e with respect to time t the power consumption function of the chip in that cycle with state s and time t after the clock edge can be written as

$$P(s,t) = \sum f(e, t - delay(e,s)) * occur(e,s) \ .$$

In reality, due to random asynchronous power consuming components in the chip and noise introduced within itself, the actual power is better modelled by adding noise components to it

$$\begin{aligned} P(s,t) &= N_c(t) + \sum f(e, t - delay(e,s) + \\ &\quad N_d(e,s) + N(e,t)) * occur(e,s) \ . \end{aligned}$$

Where $N_e(e,t)$ is a (small) Gaussian noise component associated with the power consumption function of event e, $N_d(e,s)$ is a small Gaussian noise component affecting the delay function and N_c is the small Gaussian external noise component. From power equation show that there is a strong dependence between the power consumption function and the relevant state s at that cycle. This is because different events occur in different states and even if the same set of events were to occur in two different states, their timing could be different. This dependence is at the core of all differential power analysis attacks with seek to exploit asymmetries in the power consumption function with respect to state. These asymmetries are particularly pronounced in low end smart cards where the relevant state space is quite small.

Consider two different probability distributions *D1* and *D2* on the relevant state s before the clock edge of a certain cycle. From power equation, it is very likely that the distribution of the instantaneous power when the state is drawn from *D2* and these cases can be distinguished by statistical tests on power samples. This difference and distinguishability between the two distributions is the basic for differential power attack. In most well knows attacks, the distribution *D1* and *D2* are very simple, e.g.,*D1* is the uniform distribution on the set of all states which have a particular state bit 1 and *D2* is the uniform distribution on the set of all states which have that bit 0. The difference in the power distribution for these two cases represents the effect of that particular state bit on the net power consumption.

C. Timing Attack

Implementations of cryptographic algorithms often perform computations in non-constant time, due to performance optimizations. If such operations involve secret parameters, these timing variations can leak some information and, provided enough knowledge of the implementation is at hand, a careful statistical analysis could even lead to the total recovery of these secret parameters.

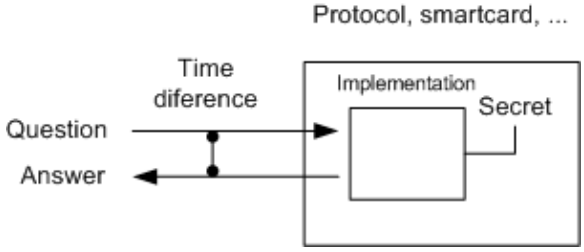

Fig. 3. The timing attack principle

For the initialization phase, we build a table for describing every possible first key byte and N different possible values of the first plain text byte, if the multiplication ε will require an additional XOR or not.

This table T with thus have $256 \times N$ entries as follows

$$\begin{aligned} \forall\ 0 \le i \le 255,\ 0 \le j \le N,\ T[i][j] &= 1 \\ \text{if the first bit of } ByteSub(iXorj) &= 1, \\ \text{otherwise} &= 0. \end{aligned}$$

Each line i corresponds to a possible value for R_1 the first byte of the round key; each column j corresponds to a different value of the first plain text byte.

For the measure phase, similarly to the initialization phase, we build N sets of M messages, where the first byte of every message from set S_i is equal to i; the other bytes are random. Thus, for every message from subset S_i, the multiplication ε will be exactly the same.

For now, Let encrypting these messages and measuring the computation times. If M is large enough, we can expect the mean time for subset S_i to reflect the time taken by the multiplication ε, i.e. to be slightly bigger when that multiplication is long. We have thus built an oracle that, with some error probability, determines, for i in $[0, N-1]$, if the first bit of *ByteSub*(i XOR R_1), is set.

To determine R_1, it suffices to compare the oracle with the array T: the line that best reflects the oracle's predictions should correspond to the right value of R_1. Other bytes of the first round key can be recovered in the same way by turning our attention to the second, third, ..., etc. bytes of plain text.

Due to Rijndael's key schedule, once N_k (the key size, in 32-bit words) consecutive bytes of round key are known,

the complete round keys can be generated. We have thus broken the cipher in the case were the block size is greater or equal than the key size. If the block size is smaller, then a little more work is necessary, but the idea is just the same.

VII. Conclusion

Discrete logarithm is use to be basic to represent $GF(2^n)$ The field operations are effectively reduced to table look ups and simple operations like shift, *XOR* and integer arithmetic. We use log-alog table for the entire round, *SubByte* and *MixColumn* operations in encryption, and their counterparts in decryption process. Data masking techniques is used to against power analysis attack, e.g., Differential power attacks. Another attack is zero attack that is eliminated at the price of few additional *XOR* operations and on-the-fly table re-computations for every run of algorithm.

References

[1] C. S. Charles "An Overview of Smart Card Security," "`http://home.hkstar.com/alanchan/papers/smartCardSecurity`," 1997.

[2] M. Karpovsky, K. J. Kulikowski, and A. Taubin "Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard," *International Conference on Dependable Systems and Networks*, pp. 93-101, June. 2004.

[3] E. Trichina and L. Korkishko "Secure And Efficient AES Software Implementation For Smart Cards," "`http://eprint.iacr.org/2004/149`," 2004

[4] G. Hachez, F. Koeune, and J. J. Quisquater "Timing Attack: What Can be achieved by a powerful adversary? , *Proceedings of the 20th symposium on Information Theory in the Benelux*, pp. 63-70," 1999.

[5] L. Joachim and T. Markus "Efficient implementation of the AES-encryption algorithm for Smart-Cards," "`http://www.iaik.tugraz.ac.at/teaching/10seminare-projekte/01Telematik Bakkalaureat/EfficientAESImplemetation.pdf`," June 2004.

[6] F. Koeune and J. J. Quisquater "A Timing Attack against Rijndael. *Technical Report CG-1999/1*, June 1999.

[7] E. Thiagarajan and M. Gourishetty "Study of AES and its Efficient Software Implementation," "`http://islab.oregonstate.edu/koc/ece679/project/2003/thiagarajan-gourishetty.pdf`," 2003.