A Fault-Resistant AES Implementation Using Differential Characteristic of Input and Output

JeongSoo Park
Hoseo University
Asan, ChungNam, Korea
sizeplay@nate.com

KiSeok Bae
Kyungpook National University
Daegu, Korea
gith@ee.knu.ac.kr

YongJe Choi, DooHo Choi
ETRI
Daejeon, Korea
{choiyj, dhchoi}@etri.re.kr

JaeCheol Ha*
Hoseo University
Asan, ChungNam, Korea
jcha@hoseo.edu

Abstract

The goal of a fault injection attack is to extract a secret key which is embedded in a cryptographic device by injecting a fault during execution of the algorithm. In particular, an attacker can extract the master key of the advanced encryption standard (AES) using only a one-byte fault injection. We propose a new countermeasure method resistant to fault injection attacks by checking the differential byte of the input and output in the encryption process and key expansion process, respectively. Based on the result of computer simulations and practical experiments, we suggest that our proposed AES implementation against fault attack has a superior error detection ability and improved efficiency compared with previous existing methods.

Keywords: fault attack, countermeasure, AES, differential characteristic

1 Introduction

Many hardware implementations of cryptographic algorithm can be countered by various physical attacks, especially fault injection attacks. A fault injection attack on a block cipher algorithm is usually referred to as a differential fault analysis(DFA). These DFA attacks are actually a security threat for cryptographic devices in which the secret key for encryption is embedded. The focus of a DFA attack is to retrieve the secret key by analyzing the differences between the pair of correct and faulty outputs obtained from a malicious error injection during execution of the algorithm. The DFA was first proposed in 1997 by Biham and Shamir as an attack on DES [2]. Similar attacks have been proposed for AES [15, 7, 13, 16], Triple-DES [8], and CLEFIA [4]. Especially, the AES [10] has been considered the main target of DFA attacks because of its popularity and as a representative of a block cipher.

Most DFAs on the AES can be roughly classified into two types according to the injected location of the fault model. In the first type of DFA, the intermediate data during the encryption is the target to be corrupted by fault injection. Attackers use a fault propagation property in which a one-byte fault injected before the *MixColums* function is diffused in the next four bytes by the *MixColums* function. As an example of a well-known DFA result based on this fault model, the fault attack described by Piret and

*Corresponding author: Dept. of Information Security, Hoseo University, 165 Sechul-Ri, Baebang-Eup, Asan-Si, Chungnam, 336-795 Korea, Tel: +82-41-540-5991

Quasiquater allows the extraction of a 128-bit secret key from two pair of correct and faulty outputs with computation complexity of about 2^{40} [15].

The second type is considered a fault propagation of the key expansion process. A fault injected during the key expansion process passes through the three transformations of the key expansion and also affects the *AddRoundKey* of the encryption process. Then a modification of the fault injected into the key expansion process and a diffusion during the encryption process are analyzed. Giraud first proposed a fault attack in which a one-bit fault on the encryption process or a one-byte fault on the key expansion process can reveal the entire secret key [7]. In 2008, Kim and Quisquater introduced an enhanced method to find the secret key with 8 pairs of correct and faulty output by a fault injection during the key expansion process [13]. Recently, Tunstall et al. proposed DFA attacks in which a secret key can be extracted by only a one-byte fault on the encryption process [16].

Various countermeasures have been introduced to protect AES implementation against DFA attacks. At first, concurrent error detection (CED), which decrypts an ciphertext to compare with the original plaintext, was introduced in terms of a hardware countermeasure [12]. Thereafter, many studies proposed software-based countermeasures including the use of parity bit [11, 1, 17], CRC method [18], and error detection methods based on the relation of the input and output of the multiplicative inversion [14, 6]. However, previous existing methods have a large amount of overhead when performing error detection, and their error detection rates have been unsatisfactory.

In this paper, we present a new countermeasure method to prevent DFA attacks on AES efficiently. Our proposed method is an error detection method that inspects the difference characteristic between the input and output in the round level or algorithm level. By calculating the differential bytes against the input and output of functions in the encryption process and key expansion process, we can discriminate whether a fault is injected during the execution of the cryptographic algorithm. Finally, we verify our proposed method in terms of its efficiency and the error detection ability against a DFA attack using computer simulations as well as practical fault injection experiments.

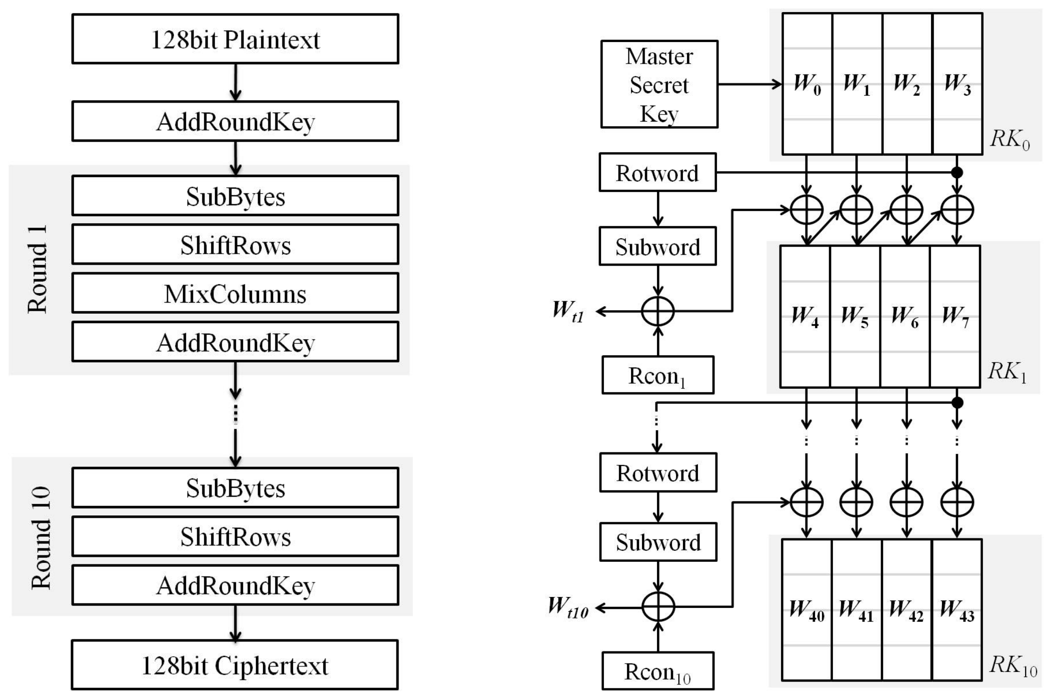
2 Preliminaries

2.1 Advanced Encryption Standard

The FIPS-approved cryptographic algorithm AES is defined for 128-bit blocks and key sizes of 128, 192 and 256 bits [10]. The 128-bit plaintext is viewed as a 4 × 4 byte matrix, called *State* byte corresponding in some way to the elements of F_{2^8}.

The AES operates on the *States* by iterating transformation rounds as shown in Figure 1(a). The initial round consists the *AddRoundKey* operation, the next rounds consist of applying successively the transformations *SubBytes, ShiftRows, MixColumns* and *AddRoundKey*, but the last round omits the *MixColumns* transformation. Defending on the key size, the number of rounds is altered as 10, 12 or 14. We dealed with the 128-bit AES due to its widespread usage.

The *SubBytes* is a non-linear byte substitution in an *SBox*. This *SBOX* is the composition of two transformations: an inversion in F_{2^8} and an affine transformation. Here, We denote this function as SB, and we denote *Inverse SubBytes* as SB^{-1}. *ShiftRows* is a cyclic shift operation on each of the four rows of the state. The first row is unchanged, the second is cyclically shifted by one byte to the left, the third by two bytes and the fourth by three bytes. We denote *ShiftRows* and its inverse as SR and SR^{-1}. *MixColumns* considers each column of the *State* matrix as coefficients of a degree three polynomial and multiplies them modulo z^4+1 with a fixed polynomial. We denote the *MixColumns* as MC and its inverse as MC^{-1}. *AddRoundKey* is a bit-wise XOR operation between the state and the round key. We denote the *AddRoundKey* as ARK.



(a) Encryption Process (b) Key Expansion Process

Figure 1: AES Structure

The 128-bit AES algorithm takes the master secret key, denoted as MK, and performs a key expansion routine to generate 11 round keys. The key schedule has a recursive structure that uses a linear array of 4-bytes words, denoted *W[i]*. There are three transformation functions in the key expansion process as follows. *RotWord* takes a word $[a_0, a_1, a_2, a_3]$ as an input, performs a cyclic permutations. *SubWord* is a non-linear function using a SB operation on a column. *Rcon$_r$* is a round constant with round number *r*.

Figure 1(b) shows the AES key expansion process. RK_0 is the initial round key identical to the master secret key and RK_r is the *r*-th round key generated by the key expansion process.

2.2 Previous DFA Results on AES

There are two categorized DFA attacks on the AES algorithm according to the location of the fault injection. The first type of DFA attack is when the *State* values during the encryption process are infected by a fault. An attacker uses a fault propagation property in which one erroneous byte affects four output bytes of the MC function. The other type of DFA attack injects a fault during the key expansion process. Since the concept of a DFA attack has been introduced, DFA on AES under real environments was proposed by Piret and Quisquater (P-Q) [15]. They described that an attacker can extract a 128-bit key from at least two pair of correct and faulty outputs. The principle of the P-Q method is to inject a random byte fault before the MC in the 8th round. The corresponding differential at the input of the last MC has four non-zero bytes, one per column of the *State* array, as shown in Figure 2.

Meanwhile, Giraud proposed a DFA attack that finds the secret key with the assumption of a fault model in which a one-bit error of the *State* occurs during the execution of encryption or a one-byte error occurs in the key expansion process [7]. Based on [7], an attacker can extract the secret key with 50 faulty outputs which is injected as a one-bit fault during the encryption process. However, this fault model have a limitation in feasibility since it is quite a hard and precise technique for inducing a bit-wise

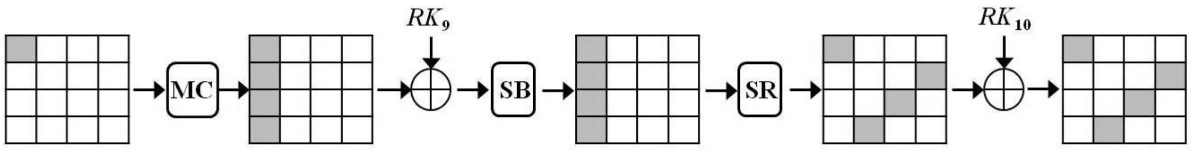


Figure 2: Propagation of a Differential Fault

fault. In the case of the byte fault model, an attacker can deduce the secret key using roughly 250 faulty outputs. Continuously, Kim and Quasiquater proposed a DFA attack using a byte fault injection on the key expansion process which needs 8 pairs of correct and faulty output to obtain the secret key. In 2009, Tunstall et al. proposed an enhanced DFA method using a well-placed fault of one-byte [16]. Similar to previous attack methods, an attacker induces a fault on a byte of the input in the 8th round to perform a fault analysis. With one pair of correct and fault ciphertext, the secret key is recovered with computation complexity of 2^8. For this DFA attack, it is now known that it needs the least number of faulty outputs in terms of the fault model, which injects a fault during the encryption process.

2.3 Previous Countermeasures against DFA

The main progress for DFA on AES have been analyzing a faulty output which is corrupted during the encryption process. Thus, countermeasures against DFA also focus on protecting the encryption process rather than the key expansion. The typical countermeasure for AES, CED [12], consists of comparing the original message with the decrypted output. Since this CED method dramatically increases the time overhead, the error detection process takes a long time.

In 2003, Karry et al. first introduced an error detection method based on a parity bit to protect against a fault injection attack on symmetric cryptographic algorithms with a substitution-permutation network(SPN) structure [11]. The parity bit is widely used for error-detection in a network. In the case of the one parity bit method, error detection for one-bit is effective, but the defection of numerous bits is too weak. Therefore, Bertoni et al. proposed a method using a multiple parity bit code instead of one parity bit [1]. Continuously, the simultaneous detection method in the round level, which is designed as one parity bit for each of the 16 SB functions, was also introduced by Wu et al. [17]. Since the detection rate is determined by the number of parity bits, the designer can use more parity bits to improve the rate. Since a present DFA on AES is able to recover the secret key from only a one-byte fault injection on the encryption process, the countermeasure method based on parity bits is not sufficient to protect against various types of DFA attacks.

Yen and Wu proposed an error detecting method based on an $(n+1,n)$ cyclic redundancy check(CRC) method [18]. The CRC method is a code used to detect accidental changes in transmitted data by a check value. The value n of CRC code is decided among 4, 8 and 16, and a lower value of n increases the number of parity bits and the detection rate. Since additional parity bits increase the overhead of the AES implementation, the designer should consider deciding against the value n. Additionally, an extra module to predict the CRC value adds to the overhead of the implementation.

In addition, there is another error detection method which compare the parity bits of input with the output's ones [14, 6]. A comparison process between a parity bit of the input of SB and a predicted parity bit according to the input and another comparison process between a parity bit of the output and a predicted parity bit of the output are performed simultaneously. However, these methods have a drawback in that the error detection rate is low.

3 A Novel Countermeasure against DFA on AES

In order to protect AES implementation against fault attacks, we propose a countermeasure based on detection that implicitly checks whether a fault is injected, by inspecting the differential bytes of every functions in the encryption process and key expansion process. Namely, in the case of the encryption process, a calculation of the differential bytes between the input and output of the round functions such as SB, SR, MC, and ARK is performed during the execution of the encryption, and is used to check whether a corruption is occurred during the round.qh In the case of the key expansion process, we use a fundamental principle of key generation to check for a fault injection on any round key by calculating a differential byte between the last column array of the $r-1$ round and the first column array of the r round. To enhance understanding of the proposed countermeasure, we show the location of the input and output of each functions in Figure 3. Here, a one-byte is represented by I_i or O_i.

I_0	I_4	I_8	I_{12}
I_1	I_5	I_9	I_{13}
I_2	I_6	I_{10}	I_{14}
I_3	I_7	I_{11}	I_{15}

O_0	O_4	O_8	O_{12}
O_1	O_5	O_9	O_{13}
O_2	O_6	O_{10}	O_{14}
O_3	O_7	O_{11}	O_{15}

Figure 3: Input and Output of AES Functions

For the efficiency of the check procedure, the proposed countermeasure do not require a space of 16 bytes for keeping the differences against the input and output during the encryption process. We use a XORed result of two one-byte values; the first one is a result of the XOR calculation among 16 input bytes, and the other value is the calculation result against 16 output bytes. The case of the key expansion process also use a result of a one-byte, which is a result of the XOR calculation from among the elements of the column arrays.

3.1 The Encryption Process

In this section, we explain the construction of a check element through the generation of the differential byte for each round function, respectively.

3.1.1 The Differential Characteristic of the Round Functions

At first, we investigate the differential characteristic of *SubBytes*. We generate two one-byte of XORed result of the 16 input bytes SB and the 16 outputs, relatively. Here, we denote the result of the input bytes or output bytes by "a differential byte of the input or output," and final check element, which is a result of the XOR calculation between the differential byte of the input and output, is denoted by "a differential byte of the SB function." The differential bytes of the input and output of the SB function is defined by Equation 1 and 2. The differential byte of SB function is given in Equation 3.

$$I_{SB} = I_0 \oplus I_1 \oplus \cdots I_{15} \tag{1}$$

$$O_{SB} = O_0 \oplus O_1 \oplus \cdots O_{15} \tag{2}$$

$$D_{SB} = I_{SB} \oplus O_{SB} \tag{3}$$

Input(I_i)	Output(O_i)	Differential byte(D_{SB_i})
00	63	63
01	7C	7D
:	:	:
FF	FF	E9

Table 1: Differential Table of SubBytes

According to Equations 1 and 2, D_{SB} is represented by Equation 4.

$$D_{SB} = (I_0 \oplus O_0) \oplus (I_1 \oplus O_1) \oplus \cdots (I_{15} \oplus O_{15}) = D_{SB_0} \oplus D_{SB_1} \oplus \cdots D_{SB_{15}} \tag{4}$$

It is necessary to store the pre-computed table of D_{SB} consisted of the differential bytes corresponding to the inputs, for checking whether the SB function operates without any corruption. A differential table can be computed in advance shown in Table 1, since the input of the SB has bounds from 0 to 255. The check procedure consists of a calculation using Equation 3 between the input and the output after the execution of the SB function, and then comparing it with a differential byte guessed by the differential table with according to the inputs. If a fault is injected at any state in the SB function, a XORed differential byte of the input and output and a XORed differential byte from differential table are different in opposition to the Equation4.

Secondly, we describe the differential byte of the SR and MC functions. Since the SR function is a simple cyclic shift operation of the *State*, the result of the XOR calculation against the input and the result against the output are equivalent. Thus, the differential byte of the SR function become zero.

$$D_{SR} = 0. \tag{5}$$

For calculating a differential byte of the MC function, to begin with, we consider a differential byte against four output bytes of an one column and its input bytes.

$$\begin{aligned} O_0 &= 2I_0 \oplus 3I_1 \oplus I_2 \oplus I_3, \\ O_1 &= I_0 \oplus 2I_1 \oplus 3I_2 \oplus I_3, \\ O_2 &= I_0 \oplus I_1 \oplus 2I_2 \oplus 3I_3, \\ O_3 &= 3I_0 \oplus I_1 \oplus I_2 \oplus 2I_3. \end{aligned} \tag{6}$$

Continuously, we calculate the differential bytes of the input and output, respectively, and compare the two results shown in Equation 7. The differential bytes of one column becomes zero, since the differential bytes of the input and output are equal.

$$O_0 \oplus O_1 \oplus O_2 \oplus O_3 = I_0 \oplus I_1 \oplus I_2 \oplus I_3 \tag{7}$$

Applied to another columns as above result, a differential byte of the entire MC function becomes zero.

$$D_{MC} = 0. \tag{8}$$

These results of a differential byte against SR and MC originate from the fact that the two functions are a kind of linear function.

Finally, we considered a differential byte of the ARK function. Since the function operates the XOR calculation between each *State* and a round key, a differential result between a differential byte of the input and output become the round key itself.

$$I_{ARK} = I_0 \oplus I_1 \oplus \cdots I_{15} \tag{9}$$

$$O_{ARK} = O_0 \oplus O_1 \oplus \cdots O_{15} = I_0 \oplus RK_0 \oplus I_1 \oplus RK_1 \oplus \cdots I_{15} \oplus RK_{15} \tag{10}$$

$$D_{ARK} = I_{ARK} \oplus O_{ARK} = RK_0 \oplus RK_1 \oplus \cdots RK_{15} \tag{11}$$

But be warned, if the input was already corrupted before the execution of a function, there is no way to detect whether the round function operated correctly or not. In order to rise the range of the detection, we consider extending our method to the round level.

3.1.2 The Differential Bytes in Round Level

Here, we extend the scope of the check element to a round level. As mentioned above, the differential bytes of the SR and MC function are zero and the differential byte of the SB function is the result of the XOR calculation of the pre-computed table value corresponding to the 16 inputs. In the case of the ARK function, the differential byte become the result of the XOR calculation of the 16 bytes of the round key. Thus, a differential byte of the round corresponding to the input and output containing four functions is given by Equation 12. Here, I_{Rnd} and O_{Rnd} are the results of the byte-wise XOR calculation against the input and output of the round.

$$D_{Rnd} = I_{Rnd} \oplus O_{Rnd} = D_{SB} \oplus D_{SR} \oplus D_{MC} \oplus D_{ARK} = (\bigoplus_{i=0}^{15} D_{SB_i}) \oplus (\bigoplus_{i=0}^{15} RK_i) \tag{12}$$

The detection procedure against the round level is as follow:

1. Calculate the differential byte for I_{Rnd}, which corresponds to the input before the execution of a target round.
2. Calculate both of the differential byte for the SB function ($\bigoplus_{i=0}^{15} D_{SB_i}$) using the Table 1 and the differential byte of the round key ($\bigoplus_{i=0}^{15} RK_i$)s.
3. After the execution of the round, calculate the output differential byte for O_{Rnd}.
4. Check whether a XOR calculation between the results of the 2nd step is equivalent to $I_{Rnd} \oplus O_{Rnd}$. If these results are same, there is no infection on the target round.

Despite this consideration, the detection mechanism has a flaw that the input is corrupted during transition to next round. For investigation among all the inputs, we need our method to extend entire algorithm level. Before demonstrating the algorithm level, we analyze the generation of round key in advance.

3.2 The Key Expansion Process

The proposed method for the encryption process can be extended to the key expansion process using a property of the round key generation. The last column of the previous round key is performed by three transformations such as *RotWord*, *SubWord*, and *Rcon* and then, operates the XOR calculation with the first column of the previous round key shown in Figure 1(b). The main property of the key expansion process is that the next round key is generated by the previous round key. Therefore, the input for calculating a difference value is the last column array of the previous round key and the output is the value after performing three transformations. Thus, the differential value of the key expansion becomes a word-wise value. In the case of first round key generation, the equations are given by as follows. Here, we denote the result after three transformations as W_t.

$$\begin{aligned} W_6 &= W_3 \oplus W_7, \\ W_5 &= W_2 \oplus W_6, \\ W_4 &= W_1 \oplus W_5, \\ W_t &= W_0 \oplus W_4. \end{aligned} \tag{13}$$

Then, W_t is represented as

$$W_t = W_0 \oplus W_1 \oplus W_2 \oplus W_3 \oplus W_7. \tag{14}$$

The differential value of the first round key is the difference between W_3 as the input and W_t as the output. Thus, it is given as

$$W_t \oplus W_3 = W_0 \oplus W_1 \oplus W_2 \oplus W_7. \tag{15}$$

Because of the property of the key expansion process, the differential value is in effect equivalent to the round level of the encryption process. Thus, we can generate 10 differential value corresponding to each round. Entire differential bytes of the round key are given by

$$\begin{aligned} D_1 &= W_{t1} \oplus W_3 = W_0 \oplus W_1 \oplus W_2 \oplus W_7, \\ D_2 &= W_{t2} \oplus W_7 = W_4 \oplus W_5 \oplus W_6 \oplus W_{11}, \\ &\vdots \\ D_{10} &= W_{t10} \oplus W_{39} = W_{36} \oplus W_{37} \oplus W_{38} \oplus W_{43}. \end{aligned} \tag{16}$$

The check procedure whether a fault is injected is the comparison of the XOR calculation for all the round keys with the XOR calculation for all the differential words corresponding to the round keys. The check element D_T and D_W are given as

$$\begin{aligned} D_t &= D_1 \oplus D_2 \oplus \cdots \oplus D_{10}, \\ D_w &= W_0 \oplus W_1 \oplus W_2 \oplus W_4 \oplus \cdots \oplus W_{39} \oplus W_{43}, \\ D_w &= D_t. \end{aligned} \tag{17}$$

D_t is the differential value between the input and output of *SubWord*. For simplifying the calculation for differential value, we perform the XOR calculation between 4 elements of the array to obtain one-byte differential value since the column array word consists of 4 bytes of the *State*. Thus, we can represent D_t as by $DB_{t,0}||DB_{t,1}||DB_{t,2}||DB_{t,3}$. Additionally, for lowering the computation overhead, we use the pre-computed difference table of SB instead of *SubWord*'s one.

The byte-wise differential value DB_T is given by

$$\begin{aligned} DB_T &= \bigoplus_{j=0}^{3} DB_{t,j}, \\ &= (\bigoplus_{r=1}^{10}\bigoplus_{j=0}^{3} D_{SB_{r,j}}) \oplus (\bigoplus_{r=1}^{10}\bigoplus_{j=0}^{3} D_{Rcon_{r,j}}). \end{aligned} \tag{18}$$

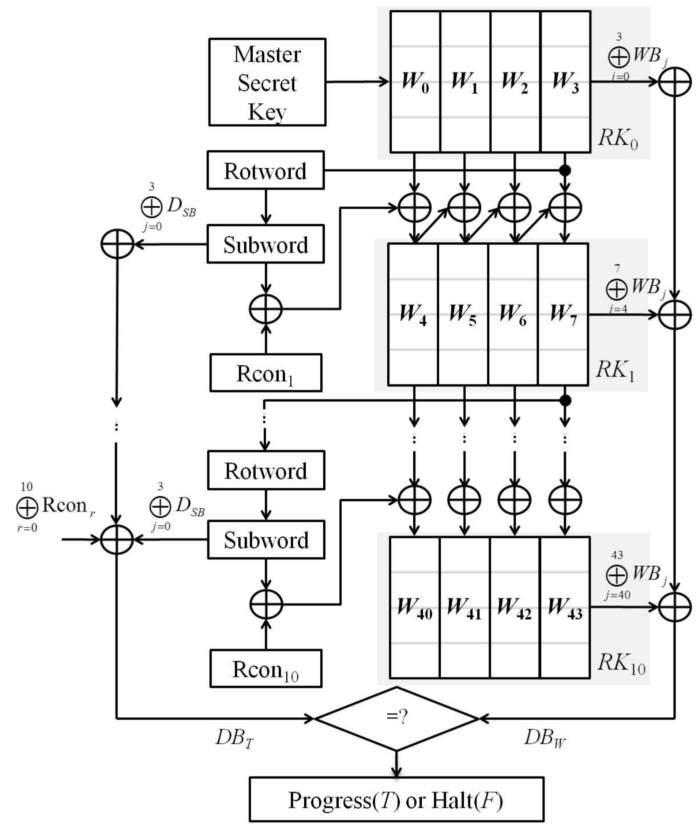

Master Secret Key
W_0 W_1 W_2 W_3
RK_0
$\bigoplus_{j=0}^{3} WB_j$
Rotword
Subword
$\bigoplus_{j=0}^{3} D_{SB}$
Rcon$_1$
W_4 W_5 W_6 W_7
RK_1
$\bigoplus_{j=4}^{7} WB_j$
$\bigoplus_{r=0}^{10}$ Rcon$_r$
Rcon$_{10}$
W_{40} W_{41} W_{42} W_{43}
RK_{10}
$\bigoplus_{j=40}^{43} WB_j$
=?
DB_T
DB_W
Progress(T) or Halt(F)

Figure 4: Countermeasure Architecture for the Key Expansion Process

We also represent D_w as by $DB_{w,0}||DB_{w,1}||DB_{w,2}||DB_{w,3}$ and $W[i]$ as by $WB_{i,0}||WB_{i,1}||WB_{i,2}||WB_{i,3}$. Then the XOR calculation result for all the round keys are given as Equation 19 in word-wise and byte-wise, respectively.

$$\begin{aligned} DB_W &= \bigoplus_{j=0}^{3} DB_{w,j}, \\ &= \bigoplus_{k=0}^{43}\left(\bigoplus_{j=0}^{3} WB_{k,j}\right), \text{where } k \neq 3, 40, 41, 42. \end{aligned} \tag{19}$$

In short, the check procedure for the key expansion process is the comparison between DB_T and DB_W. The differential byte DB_T is computed during the key expansion process. On the other hand, DB_W is computed after the entire round key generation. If two values are same, there is no corruption during the key expansion. Then the algorithm performs the encryption process in succession. However, if the two values are different, the AES implementation stops operating instantly and returns an error message.

The remaining round keys are checked by the following equations separately.

$$\begin{aligned} WB_3 &= WB_6 \oplus WB_7, \\ WB_{40} &= WB_{41} \oplus WB_{37}, \\ WB_{41} &= WB_{42} \oplus WB_{38}, \\ WB_{42} &= WB_{43} \oplus WB_{39}. \end{aligned} \tag{20}$$

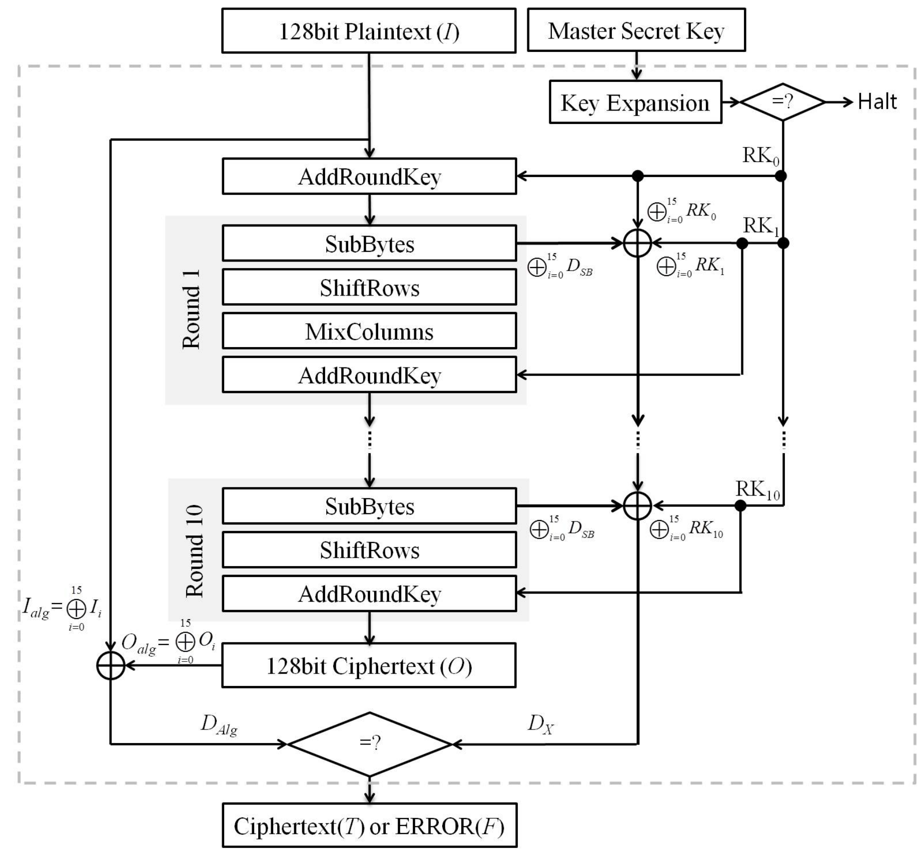


Figure 5: Secure AES Architecture with Fault Detection

During the calculation of the differential byte DB_T, the differential byte of *RotWord* is zero since this function is just a cyclic rotation. The case of *RCon* which does computations with the fixed round constants can be pre-computed. The entire procedure of the proposed secure AES implementation for the key expansion process is described in Figure 4.

3.3 Entire Algorithm Level

In order to protect AES algorithm regardless of a corruption of the input during passing to the following round, we apply our proposed method to an algorithm level, which inspects the entire inputs and output of AES algorithm. Since our method in the round level can detect any corruption during the execution of round process, it also is impossible to obtain the faulty output in terms of the algorithm level. If a certain input of any round was corrupted, a checking procedure using the input and output of the algorithm can detect in opposition to the round level. A proposed countermeasure against fault injection attack adapted to the algorithm level is described in Figure 5. Here, we assumed that a validation test of round key, which mentioned above, may perform during the key expansion process.

As shown in Figure 5, a differential byte between the differential bytes of a plaintext as the input and a ciphertext as the output is given by Equation 21.

$$D_{Alg} = I_{Alg} \oplus O_{Alg} = (\bigoplus_{i=0}^{15} I_i) \oplus (\bigoplus_{i=0}^{15} O_i) \tag{21}$$

According to Section 3.1, a validation test in the round level constructs the XOR calculation between

Algorithm 1 Secure AES Algorithm with Fault Detection

```
Input: Plaintext($I$), Master key($MK$)
Output: Ciphertext($O$) or Error message(ERROR!)
1: (precomputation) Calculate the differential bytes of the input and output of SubBytes and Store.
2: Perform the key expansion process with validation
3: If the key expansion process is valid, begin the encryption process. If not, halt the execution of AES algorithm.
4: Calculate the differential byte of the plaintext(input) ($I_{Alg} = \bigoplus_{i=0}^{15} I_i$).
5: Calculate the differential byte of the initial round key $RK_0$ ($D_{X_0} = \bigoplus_{i=0}^{15} RK_{0,i}$).
6: During the execution of rounds, calculate the differential bytes of the SubBytes and round keys, respectively ($D_{X^r_{SB}} = \bigoplus_{i=0}^{15} D_{SB_{r,i}}$ , $D_{X^r_{RK}} = \bigoplus_{i=0}^{15} RK_{r,i}$ ,where $1 \leq r \leq 10$).
7: Calculate the check element $D_X$ ($D_X = \bigoplus_{r=1}^{10} (D_{X^r_{SB}} \oplus D_{X^r_{RK}}) \oplus D_{X_0}$)
8: Calculate the differential byte of the ciphertext(output), and XOR computation with $I_{Alg}$ to obtain the differential byte of entire algorithm $D_{Alg}$ ($O_{Alg} = \bigoplus_{i=0}^{15} O_i$, $D_{Alg} = I_{Alg} \oplus O_{Alg}$).
9: Compare $D_X$ to $D_{Alg}$, and then return ciphertext $O$ if two values are equal. If not, return the error message.
```

the differential bytes of SB and RK of the target round. In order to check throughout entire rounds, thus we need to calculate XOR calculation between the differential bytes of SB and RK of each rounds cumulatively. A check element for confirming whether entire round process execute correctly is denoted by Equation 22. Here, $r(1 \leq r \leq 10)$ means the number of rounds, and $i(0 \leq i \leq 15)$ means the number of byte for the *State*.

$$D_X \quad = \quad \bigoplus_{r=1}^{10}\{(\bigoplus_{i=0}^{15} D_{SB_{r,i}}) \oplus (\bigoplus_{i=0}^{15} RK_{r,i})\} \oplus \bigoplus_{i=0}^{15} RK_{0,i} \tag{22}$$

If a fault is injected at any point in the algorithm, D_X and D_{Alg} are different, so that the algorithm indicates a fault and returns an error message. If not, the architecture returns a correct ciphertext. The entire procedure of the proposed secure AES implementation against fault injection attack is described as Algorithm 1.

However, there have been some fault attacks using multiple fault injections. These kind of fault models assumed that several faults are injected on various locations of the target algorithm during the execution. In our proposed method, if the XOR calculation of the entire fault, which is injected during the execution of the same function, is equal, then it is impossible to detect the fault injection. That means, the differential byte of the corrupted function is zero. Fortunately, this kind of fault model is hard to implement in real environments, since the cryptographic device cannot endure multiple fault injections and the measurement for timing and location of the fault injection is complicated. Thus such kind of fault techniques against AES implementation have not been practically presented yet. Thus in this paper, we considered the fault attack based on a single fault model only, and our proposed countermeasure can detect any fault corruption in terms of single fault model.

The number of error bit	The number of fault injection trials	Error detection rate (%)		
		Parity bit (the number of detection)	CRC method (n=4)	Proposed method
1	5120	100 (5120)	100	100
2	17920	5.6 (1008)	100	100
3	35840	100 (35840)	100	100
4	44800	11..2 (5040)	100	100
5	35840	100 (35840)	100	100
6	17920	16.9 (3024)	100	100
7	5120	100 (5120)	100	100
8	640	22.5 (114)	100	100

Table 2: Simulation Results - Error Detection Rate

4 Performance Evaluation and Practical Experimental Results

In this section, we present the simulation results of our proposed countermeasure and the aforementioned protected AES implementations such as the parity based method [1] and CRC method [18] to evaluate the performance of error detection. The first stage was performed by simulation to compare the effectiveness and efficiency of the proposed implementation with another methods. The second stage consisted of experiments using a laser on a decapsulated surface of a microprocessor to induce a fault.

4.1 Simulation Result and Evaluation

The simulation set up consisted of three different AES implementations: parity-based countermeasures, the CRC method, and our countermeasure for the algorithm level. Table 2 shows the error-detection rate for each of the countermeasures in terms of the error detection rate. We assumed that a fault could affect only one byte of the *State* during the execution of the AES algorithm because the most effective fault attack among previous existing attacks on AES is assumed as a byte-wise fault injected on the *State* during the encryption process. We simulated all of the possible cases such as various size of the fault within 8-bits and various locations of the fault was inducted for every round, functions, and *State*. For example, in the case of a 2-bits fault injection, there were 40 instances for the operation of functions and 16 bytes of the *State* during the execution of 10 rounds. Then we simulated the error-detection rate through 17920 $(= 40 * 16 * C_8^2)$ instances of fault injections with the simulation environments as described in [3].

When an odd bit error occurred, the countermeasure of Bertoni et al. [1] allowed for the detection of an error with 100 percents certainty, since this method is based on the even parity bit perceiving an error. When an even bit error occurred, the method using the even parity bit could not detect the error well, but the error propagation property of the MC operation helped the method to detect the error with probability. The CRC method [18] was able to detect all possible errors with a 100 percent success rate for the condition $n = 4$ shown in Table 2. However, the computation complexity was much greater than the parity based method since an extra module to predict the CRC values is more complicated.

In our proposed method, the XOR calculation of the differential bytes for each of the inputs of SB, D_{SB_i}, and the XOR calculation for each of the round keys were needed. And a temporal or permanent

Methods	**Parity bit**		**CRC method (n=4)**		**Proposed method**	
Memory for SB	256x9 bits		-		256x2 Bytes	
Computation complexity	SB	10x(9x9b)	SB	10x(32B+16B+4M)	In/out	2x16B
	SR	10x(16B)	SR	10x(16B)	Key	10x16B
	MC	9x(16B+16x4b)	MC	9x(16B)	SB	10x16B
	ARK	11x(16B+16b)	ARK	11x(16B+4B)	Comparison	2B
	Total :	29x16B+1562b	Total :	60x16B+44B+40M	Total :	22x16B+2B

(*B=Byte* XOR, *b=Bit* XOR, *M=Affine Multiplication*)

Table 3: The Costs of AES Implementation with Fault Detection

memory space was required to store predicted differential table of SB. As shown in Table 3, we tabulate the comparison test against algorithm level by referring results described in [18], which investigated the computational cost in the operation level. Despite of some additional memory, our method looks a reasonable countermeasure due to superiority on the computational speed between other methods. In a time overhead, averaged execution time of our proposed method is 37*ms* in comparison with naive AES implementation as 31*ms*. But it may be a tiny overhead in opposition to other methods such as parity based method is 71*ms*. As the results, our proposed method is quite efficient in terms of countermeasure against single fault model.

4.2 Practical Fault Experiment and Results

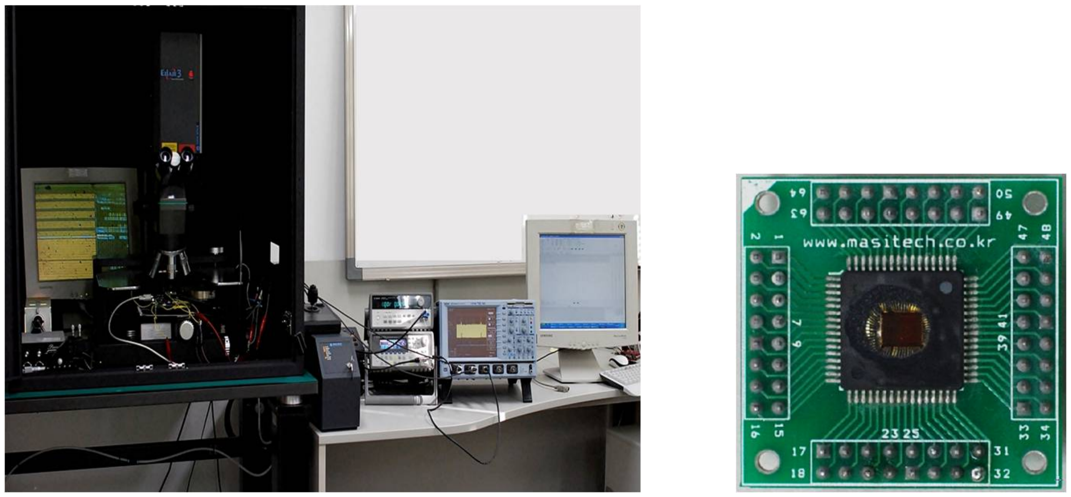
In real environments, we assumed an unknown random single-byte fault model to verify the security of our countermeasure method against DFA attack. Therefore, we applied to the P-Q attack to our protected AES implementation using an ATmega128 microprocessor with a practical fault injection.

To implement a physical fault injection, we implemented the protected AES method on the Atmega128 microprocessor [5]. Our implementation followed the structure shown in Figure 5. Then, we corrupted a *State* in the 8th round before execution of he MC function by the injection of a fault. We used the fault injection tool EZ Laze 3 [9], which can target a laser beam on the surface of a decapsulated chip. Figure 6 shows our experimental setup.

During the experiments, we monitored the power signal using a digital oscilloscope, and controlled several I/O signals to distinguish between the operations and the number of rounds. Figure 7 shows the power signals of a normal execution of the protected AES implementation and a faulty AES encryption process.

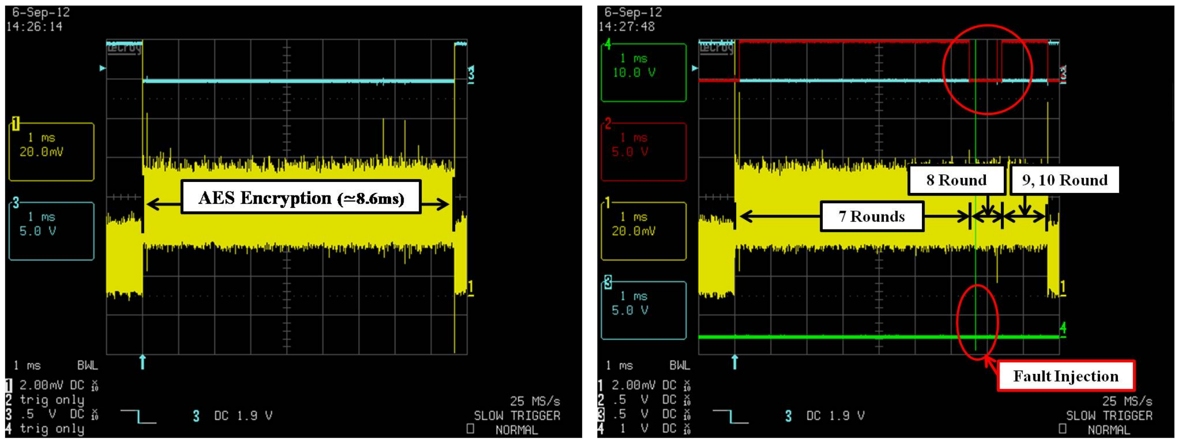
According to Figure 7(a), the entire AES encryption process took 8.6*ms* shown in the low state of the first I/O signal. As shown in Figure 7(b), the second I/O signal showed the period of the 8th round, which was followed after the 7th round, and the lower circled area indicates the fault injection. Despite several trials of fault injection, we obtained only correct ciphertexts or error messages shown in Figure 8(b) .

In Figure 8, "39 25 ··· 0B 32" is the correct ciphertext of the example input and key given in Appendix B of FIPS 197 [10]. In the case of the naive AES implementation, an attacker can obtain an intended faulty output "B6 6D ··· EC 5E" shown in Figure 8(a). However, in the case of our proposed implementation, an attacker indicates a successful fault injection by the error message but cannot obtain any intended faulty outputs. As shown in Figure 8(b), the other messages that consist of "FF" were regarded as an error message, which is implemented in order to represent the error. Thus, the proposed method can defeat the DFA attack in a practical random byte fault.



(a) Experimental Setup (b) Decapsulated Atmega128

Figure 6: Experiment Setup and Target Chip

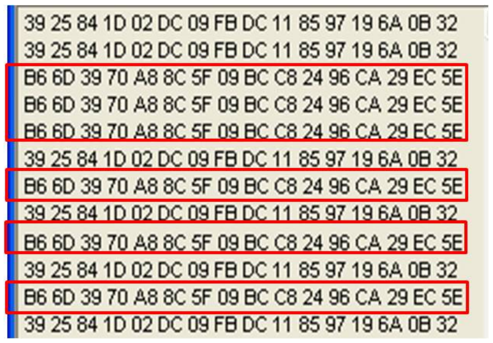


(a) Protected AES Encryption (b) Fault Attack on AES Encryption

Figure 7: Captured Power Signal of Proposed AES Implementation

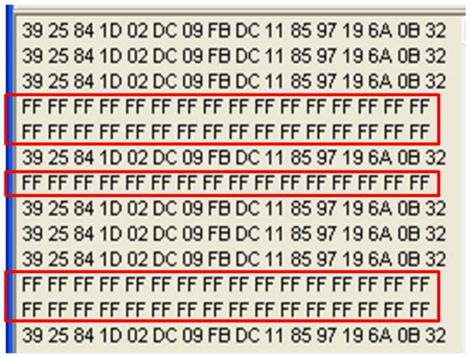
5 Conclusion

In this paper, we present a novel countermeasure that protects all the variables of the AES algorithm against fault injection attacks. Our countermeasure is based on an infective computation strategy that checks the correctness of the intermediate values at the algorithm level. An attacker cannot obtain any faulty outputs since all of values during the encryption process and key expansion process will typically be protected. The efficiency and security of the proposed method against DFA attack was verified by computer simulation as well as by practical fault injection experiments. As we demonstrated, our proposed method is adequate in protecting AES implementation against fault attacks with low additional overhead and superior error detection.



```
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
B6 6D 39 70 A8 8C 5F 09 BC C8 24 96 CA 29 EC 5E
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
```

(a) Naive AES Implementation



```
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
```

(b) Protected AES Implementation

Figure 8: Outputs from the Target Chip

Acknowledgments

This work was supported by SCARF project which is the R&D program of KCC/KCA [Development of the Technology of Side Channel Attack Countermeasure Primitives and Security Validation]

References

[1] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE Transactions on Computers*, 52(4):492–505, April 2003.

[2] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Proc. of 17th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'97), Santa Barbara, California, USA, LNCS*, volume 1294, pages 513–525. Springer-Verlag, August 1997.

[3] K. Bousselam, G. Di Natale, M. Flottes, and B. Rouzeyre. *Fault Detection in Crypto-Devices*. InTech, 2010.

[4] H. Chen, W. Wu, and D. Feng. Differential fault analysis on CLEFIA. In *Proc. of the 9th International Conference on Information and Communications Security (ICICS'07), Zhengzhou, China, LNCS*, volume 4861, pages 284–295. Springer-Verlag, December 2007.

[5] A. Corporation. Specification of Atmega 128L chip. `http://www.atmel.com/Images/doc2467.pdf`, 2011.

[6] G. Di Natale, M. Flottes, and B. Rouzeyre. An on-line fault detection scheme for sboxes in secure circuits. In *Proc. of the 13th IEEE International On-Line Testing Symposium (IOLTS'07), Montpellier, France*, pages 57–62. IEEE, July 2007.

[7] C. Giraud. DFA on AES. In *Proc. of the 4th International Conference on Advanced Encryption Standard (AES'04), Bonn, Germany, LNCS*, volume 3373, pages 27–41. Springer-Verlag, May 2003.

[8] L. Hemme. A differential fault attack against early rounds of (triple-)DES. In *Proc. of the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'04), Massachusetts, USA, LNCS*, volume 3156, pages 170–217. Springer-Verlag, August 2004.

[9] E. S. Industries. SEzLaze Laser Cutting System. http://www.new-wave.com/, 2012.

[10] N. I. o. S. Information Technology Laboratory and Technology. Announcing the advanced encryption standardsAES. FIPS 197, November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[11] R. Karri, G. Kuznetsov, and M. Goessel. Parity-based concurrent error detection of substitution-permutation network block ciphers. In *Proc. of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03), Cologne, Germany, LNCS*, volume 2779, pages 113–124. Springer-Verlag, September 2003.

[12] R. Karri, K. Wu, P. Mishra, and Y. Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1509–1517, December 2002.

[13] C. Kim and J.-J. Quisquater. New differential fault analysis on AES key schedule: Two faults are enough. In *Proc. of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS'08), London, UK, LNCS*, volume 5189, pages 48–60. Springer-Verlag, September 2008.

[14] M. Mozaffari-Kermani and A. Reyhani-Masoleh. A structure-independent approach for fault detection hardware implementations of the advanced encryption standard. In *Proc. of the 4th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'07), London, UK*, pages 47–53. IEEE, September 2007.

[15] G. Piret and J.-J. Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In *Proc. of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03), Cologne, Germany, LNCS*, volume 2779, pages 77–88. Springer-Verlag, September 2003.

[16] M. Tunstall, D. Mukhopadhyay, and S. Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *Proc. of the 5th IFIP WG 11.2 International Workshop on Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication (WISTP'11), Crete, Greece, LNCS*, volume 6633, pages 224–233. Springer-Verlag, June 2011.

[17] K. Wu, R. Karri, G. Kuznetsov, and M. Goessel. Low cost concurrent error detection for the advanced encryption standard. In *Proc. of the 2004 International Test Conference (ITC'04), Charlotte, North Carolina, USA*, pages 1242–1248. IEEE, October 2004.

[18] C.-H. Yen and B.-F. Wu. Simple error detection methods for hardware implementation of advanced encryption standard. *IEEE Transactions on Computers*, 55(6):720–731, June 2006.

Jeong-Soo Park is currently a master course student working in information security at Hoseo University, Asan, Rep. of Korea. His main area of interest are Smartphone security, side channel analysis, and wireless network security. He received his bachelor in computer engineering from Hoseo University in 2011.

KiSeok Bae received the BE and ME in electrical engineering and computer science from Kyungpook National University, Rep. of Korea, in 2006 and 2008, respectively. Currently, he is a PhD student at Communication & Information Security lab in Kyungpook National University. His research interests include side-channel analysis and information security.

Yong-Je Choi received his BSEE and MS from Chonnam National University, Kwangju, Rep. of Korea, in 1996 and 1999, respectively. He is currently a senior member of technical staff at ETRI, Daejeon, Rep. of Korea. His research interests include VLSI design, crypto processor design, side channel analysis, and information security.

DooHo Choi received the BS in mathematics from Sungkyunkwan University, Seoul, Korea, in 1994, and the MS and PhD in mathematics from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1996, 2002, respectively. He has been a senior researcher at ETRI, Daejeon, Korea, since January 2002. His current research interests are side-channel analysis and its resistant crypto design, security technologies of RFID and wireless sensor network, lightweight cryptographic protocol/module design, and cryptography based on non-commutativity. He was an editor of the ITU-T Rec. X.1171.

JaeCheol Ha received the BE, ME, and PhD in electronics engineering from Kyungpook National University, Rep. of Korea, in 1989, 1993, and 1998, respectively. He is currently a professor of the Department of Information and Security at Hoseo University, Asan, Korea. During 1998 to 2006, he also worked as a professor in the Department of Information and Communication at Korea Nazarene University, Cheonan, Korea. In 2006, he was a visiting researcher at the Information Security Institute of Queensland University of Technology, Australia. His research interests include network security, smart card security, and side-channel attacks.