Function Masking: A New Countermeasure Against Side Channel Attack

Taesung Kim[1,3], Sungjun Ahn[2], Seungkwang Lee[1], and Dooho Choi[1,2(✉)]

[1] Electronics and Telecommunications Research Institute, Daejeon, South Korea
{taesung,skwang,dhchoi}@etri.re.kr
[2] Korea University of Science and Technology, Daejeon, South Korea
asj503@ust.ac.kr
[3] Korea Advanced Institute of Science and Technology, Daejeon, South Korea
ruthere@kaist.ac.kr

Abstract. Masking schemes have been developed to implement secure cryptographic algorithms against Side Channel Analysis(SCA) attacks. Technically, the first-order masking method is vulnerable to the second order Differential Power Analysis(2ODPA) attacks, but the current solutions against 2ODPA are expensive to implement. Moreover, worse performance will be shown if the cryptographic algorithms include boolean and arithmetic operations. In this paper, we propose a new masking scheme to resist SCA attacks, which is called the Function Masking. Function Masking method conceals functions instead of data in the algorithms and makes it resistant to attacks as much as 2ODPA. We apply our masking scheme to the HIGHT algorithm. The encryption of function masked HIGHT takes only 1.79 times more than one of the original algorithm, even though it needs 25 kbytes to store lookup tables in memory.

Keywords: Side channel attack · Countermeasure · Second-order masking

1 Introduction

A lot of researches have been published about various methods to secure implementations of different kinds of cryptographic algorithms, after Kocher *et al.* [11] introduced Simple Power Analysis(SPA) and Differential Power Analysis(DPA), types of power analysis. SCA attacks are physical attacks to find out secure data by using Side Channel Information such as power consumption, electromagnetic wave, timing, and so on during the execution. The attacks are based on the statistical dependency between the intermediate values and leaked information. It means that it is possible for adversaries to determine the entire secret key related to the intermediate values.

It is very common to randomize the sensitive variables by masking techniques when a countermeasure is used to protect implementations of block ciphers against SCA. One or several random values are added to the secret data during the execution of cryptographic algorithms, which means that every intermediate

K.-H. Rhee and J.H. Yi (Eds.): WISA 2014, LNCS 8909, pp. 331–342, 2015.
DOI: 10.1007/978-3-319-15087-1_26

value is independent of any secret variable. But, the first-order masking method is vulnerable to a second-order SCA. The second-order masking schemes should be considered to resist the attacks, but decrease performance.

Some cryptographic algorithms use the boolean and arithmetic operations to make the security. To counteract the SCA attacks, it is necessary to convert back and forth between the boolean masking and the arithmetic masking. Thus, Goubin *et al.* [3] has suggested a secure method to convert between masks, which is only applicable to the first-order masking. However, it is shown that it is impossible to apply the Goubin's method to the second-order masking schemes [4]. Vadnala *et al.* [4] has proposed masking conversion for the second-order masking, but it requires 1027 times more operations to convert 8-bit size of masks. It is not possible to use it on embedded devices.

In this paper, we suggest a new countermeasure method which randomizes all the intermediate values of cryptographic algorithms. We call it a function masking. Our scheme makes lookup tables which randomly convert all the functions and operations in the algorithms with encoding and decoding(we call it linear and non-linear function masks). The algorithms are reconstructed by using these lookup tables. Actually, this method is similar to white-box cryptography because of encoded lookup tables. However, our method dynamically inputs the round keys while white-box cryptography includes the round keys in lookup tables. Thus, it is possible to change the round keys depending on the environments and the attackers cannot predict all the intermediate values during the execution of cryptographic algorithms. We show the security of the function masking, apply it to HIGHT algorithm, and compare with second order masking.

The remainder of this paper is organized as follows. Section 2 describes the existing countermeasures of SCA and HIGHT cryptography algorithm which we applied the function masking. In Sect. 3, we introduce the concept of function masking and explain the implementation method of HIGHT algorithm to apply function masking. We show the security and performance analysis in Sect. 4. Finally, in Sect. 5, we offer the conclusion.

2 Related Work

2.1 Countermeasures Against Side Channel Attacks

To the best of our knowledge, the most widely used technique protecting against DPA is to mask key-dependent intermediate data by random values. This is called masking. For a key-dependent intermediate byte x and a random mask m, masking requires a function $f(x, m) = x \cdot m$, where $\cdot$ is defined as bitwise XOR(boolean masking), modulo addition (additive masking) or multiplication (multiplicative masking).

$$y \oplus m' = \mathbf{MaskedSbox}(x \oplus m \oplus k)$$
$$m, m' \ : \ \text{random values(mask)}, y = \mathbf{Sbox}(x \oplus k)$$

However, using only one mask which is called a first-order masking is vulnerable to a second-order DPA.

$$\begin{aligned} y_1 \oplus m' &= \mathbf{MaskedSbox}(x_1 \oplus m \oplus k_1), y_1 = \mathbf{Sbox}(x_1 \oplus k_1) \\ y_2 \oplus m' &= \mathbf{MaskedSbox}(x_2 \oplus m \oplus k_2), y_2 = \mathbf{Sbox}(x_2 \oplus k_2) \end{aligned} \tag{1}$$

$$y_1 \oplus m' \oplus y_2 \oplus m' = y_1 \oplus y_2 = \mathbf{Sbox}(x_1 \oplus k_1) \oplus \mathbf{Sbox}(x_2 \oplus k_2) \tag{2}$$

To be specific, Eqs. (1) and (2) show that an attacker can obtain a non-masked result value of XORing two S-box outputs by XORing two masked S-box outputs. This is due to the fact that m' is canceled out by the XOR operation. A second-order DPA is therefore started by making two target points of a power trace as one point using subtractions or multiplications. The next step is to mount DPA based on a hypothetical value computed by XORing two S-box outputs [2].

Protection of second-order DPA requires more than two masks, and all intermediate values have to be masked through out the execution of the algorithm. Especially, each of input and output bytes of S-box must use different masks. For this reason, a masked AES implementation requires 16 masked S-boxes. As a result, a high-order masking of AES gives rise to an efficient implementation of S-boxes. Unfortunately, Table 1 shows that implementing a high-order masking scheme affects the performance of AES. To be more precise, the countermeasures are 150–300 times slower than a straightforward implementation. This might be an intolerable performance for a practical solution. HIGHT algorithms, which we will apply function masking, includes both boolean and arithmetic operations. To properly apply data masking, it is required to use a secure boolean-from/to-arithmetic mask conversion without exposing non-masked intermediate values against a second-order DPA. Goubin proposed secure mask conversion which can hide sensitive intermediate in convert process [3]. However, this conversion can only resist for first-order DPA. Vadnala *et al.* [4] proposed new conversion method which can work for second-order DPA. This method as shown in Algorithm 1 requires $4 \times 2^k + 3$ operations for conversion of a k-bit mask. An arithmetic-to-boolean conversion also requires the similar number of operations. It must be a critical overhead when it is applied to all mask conversions.

Table 1. Performance of the high-order masking scheme in AES

Method	Cycles	RAM(bytes)	ROM(bytes)
Unprotected implementation			
No masking [7]	2×10^3	32	1150
Provably Secure second-order SCA resistant implementation			
[5]	675.4×10^3	0	768
[6]	265.5×10^3	0	816

Algorithm 1. Boolean to arithmetic conversion of 2nd order

Input: Boolean share: $x_1 = x \oplus x_2 \oplus x_3, x_2, x_3$
Output: Arithmetic share: $A_1 = (x - A_2) - A_3, A_2, A_3$
1: Randomly generate n-bit numbers r, A_2, A_3
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: **for** $a = 0$ to $2^n - 1$ **do**
4: $\quad a' \leftarrow a \oplus r'$
5: $\quad T[a'] \leftarrow ((x_1 \oplus a) - A_2) - A_3$
6: **end for**
7: $A_1 = T[r]$ **return** A_1, A_2, A_3

2.2 HIGHT Algorithm

The HIGHT(HIGh security and light weigHT) [8] is a symmetric cipher which encrypts and decrypts data with a 64-bit block cipher using a key of size 128 bits. It provides light-weight and low-powered hardware implementation for ubiquitous computing devices. We will briefly introduce the algorithm of HIGHT. The 64-bit plaintext and ciphertext are denoted by concatenations of 8 bytes such as $P = P_7 \| P_6 \| P_5 \| P_4 \| P_3 \| P_2 \| P_1 \| P_0$ and $C = C_7 \| C_6 \| C_5 \| C_4 \| C_3 \| C_2 \| C_1 \| C_0$. Round functions are consisted of several mathematical operations: $\boxplus$ addition mod 2^8, $\boxminus$ subtraction mod 2^8, $\oplus$ XOR, and $\lll r$ r-bit left rotation. The encryption of HIGHT algorithm is totally made up of initial transformation, round function, final transformation, and key schedule. It is described in detail below.

Algorithm 2. HIGHT encryption

Input: $P = P_7 \| P_6 \| P_5 \| P_4 \| P_3 \| P_2 \| P_1 \| P_0$
Output: $C = C_7 \| C_6 \| C_5 \| C_4 \| C_3 \| C_2 \| C_1 \| C_0$

$X_{0,i} = P_i \ for\ i = 1, 3, 5, 7$
$X_{0,0} = P_0 \boxplus WK_0$
$X_{0,2} = P_2 \oplus WK_1$
$X_{0,4} = P_4 \boxplus WK_2$
$X_{0,6} = P_6 \oplus WK_3$
for $i = 0$ *to* 31 **do**
$\quad X_{i+1,1} = X_{i,0};\ X_{i+1,3} = X_{i,2};\ X_{i+1,5} = X_{i,4};\ X_{i+1,7} = X_{i,6}$
$\quad\quad X_{i+1,0} = X_{i,7} \oplus (F_0(X_{i,6}) \boxplus SK_{4i+3})$
$\quad\quad X_{i+1,2} = X_{i,1} \boxplus (F_1(X_{i,0}) \oplus SK_{4i+2})$
$\quad\quad X_{i+1,4} = X_{i,3} \oplus (F_0(X_{i,2}) \boxplus SK_{4i+1})$
$\quad\quad X_{i+1,6} = X_{i,5} \boxplus (F_1(X_{i,4}) \oplus SK_{4i})$
end for
$C_0 = X_{32,1} \boxplus WK_4;\ C_1 = X_{32,2}$
$C_2 = X_{32,3} \oplus WK_5;\ C_3 = X_{32,4}$
$C_4 = X_{32,5} \oplus WK_6;\ C_5 = X_{32,6}$
$C_6 = X_{32,7} \boxplus WK_7;\ C_7 = X_{32,0}$

$WK_{0\leq i\leq 7}$ means whitening key and $SK_{0\leq i\leq 127}$ is subkey. Round function uses functions F_0 and F_1:

$$F_0 = (x \lll 1) \oplus (x \lll 2) \oplus (x \lll 7)$$
$$F_1 = (x \lll 3) \oplus (x \lll 4) \oplus (x \lll 6)$$

The decryption process is similar to the encryption of HIGHT.

3 Function Masking for Symmetric Cryptography Algorithm

3.1 Function Masking

Our function masking is inspired by a white-box implementation [9] of block ciphers. Protection of a key-customized encryption function E_k in a white-box implementation is replaced by $E'_k = G \cdot E_k \cdot F^{-1}$, where F and G are input and output encoding, respectively. Being chosen randomly without reference to k, the use of G and F unlikely weakens the ordinary black-box security of E_k. However, one of the serious problems of this solution is the large size of the lookup tables. Our motivation in this matter is that an attacker in a gray-box model is not fully privileged to access the lookup table. For this reason, we try to generate a dynamic-key lookup table which takes both a key and an operand as an input. To be specific, it can be represented by

$$E(k, x) = G(E(k, F^{-1}(x)))$$

where x is an operand to be involved with k.

By generating a lookup table for $E(k, x)$, we can significantly reduce the total size of the lookup table than a white-box implementation because the table can be shared throughout all rounds. Also, this yields an additional advantage over a white-box lookup table: it can support dynamic key applications. In other words, this method can be also used when a secret key is updated from time to time like in the case of a session key. A potential problem is how to design the lookup table within practical size because a key is added to an input to the table. In the following, we explain how to apply function masking to HIGHT in such an efficient way.

3.2 Applying Function Masking to HIGHT Algorithm

Function Masking Method

Encoding & Decoding. It is required to conceal all of the intermediate values. The function masking method uses non-linear, linear encoding and random masking. Chow *et al.* [9] has suggested input and output encodings to protect a table. An encoding is a bijection. Encodings are networked with input and output of tables. If a table T is prevented with chosen bijections G,H

$$T' = H \circ T \circ G^{-1}$$

G is the input encoding and H is the output encoding. In case of two tables for lookup operations, it is expressed in a networked fashion. For example, tables T_1 and T_2 are protected with encodings as follows.

$$T_2' \circ T_1' = (H \circ T_2 \circ G^{-1}) \circ (G \circ T_1 \circ H^{-1}) = H \circ T_2 \circ T_1 \circ H^{-1}$$

Encodings make all lookup tables to obfuscate in Function Masking method. Furthermore, linear functions L and M are used to achieve diffusion for security, defined by Shannon [12]. There is also random mask to conceal 2×4-bit output values. Random mask is used to encode the modular addition in round.

4-bit non-linear function mask G, H, G^{-1}, H^{-1}: $\{0,1\}^4 \rightarrow \{0,1\}^4$
8-bit linear function mask L, M, L^{-1}, M^{-1}: $\{0,1\}^8 \rightarrow \{0,1\}^8$
8-bit random mask $C_1, C_2 (0 \leq C_1 \leq 255,\ 0 \leq C_2 \leq 255)$

Several types of lookup tables (See Figs. 1, 2 and 3) could be generated with above masks.

Reduction of Lookup Table Size. The modular addition and XOR operation result in a value with two operands. If two input values are 8-bit, it can be shown that all of the $2^{16} (= 65536)$ possible output values produce distinct lookup tables. However, it is too much big to store in memory sometimes. To overcome this problem, it could be transformed into $2 \times 2^{12} (= 8192)$ lookup tables. Then, it could significantly reduce the size of lookup tables. At first, an 8-bit operand and a high 4-bit of another operand will become input values of the first lookup table. An 8-bit output of the first table and a low 4-bit of another operand produce an 8-bit result value of the modular addition or XOR operation by using the second table. For example, the XOR operation of two 8-bit operands can be computed with type III-1 and III-2 tables. There are also two tables of type IV-1 and IV-2 for the modular addition.

Applying Function Masking to HIGHT Algorithm. The Function Masking method is applied to the HIGHT algorithm. It is required to make 12 lookup tables of 5 types for HIGHT algorithm.

Initial Transformation downsizing the size of lookup tables is applied to the modular addition and XOR with $2 \times$8-bit input. P_0 and P_1 are encoded by a type I-2 table. In the case of P_0, the encoded value is added with Whitening Key by using two tables of type IV-1 and IV-2 tables. Then $X_{0,0}$ is obtained after changing the mask from type I-4 table. The intermediate value $X_{0,1}$ is the encoded value of P_1. Moreover, P_2 and P_3 are encoded by type I-1 table. The encoded value of P_2 is XORed with Whitening Key by using two tables of type III-1 and III-2 tables. Thus, a table lookup of type I-3 yields the intermediate value $X_{0,2}$. The intermediate value $X_{0,3}$ is the encoded value of P_3. P_4, P_5 and P_6, P_7 are the same process as above lookup operations P_0, P_1 and P_2, P_3 respectively.

Round Transformation. Let's take a close look at the first two 8-bit values of the round inputs shown in Fig. 4. Subkey is protected by encoding through type

I-1 table. Type II-1 and III-2 tables operate functions F and XOR. A high 4-bit of the encoded subkey and the 8-bit value $X_{i-1,0}$ are the input value of the type II-1 table. Thus, The output and a low 4-bit of the encoded subkey go into the type III-2 table. And the 8-bit value of $X_{i-1,1}$ and a high 4-bit of the XORed value make the 8-bit output by using a type IV-1 table. The intermediate value $X_{i,2}$ is obtained by a type IV-2 table with the central output and a low 4-bit of the XORed value. $X_{i-1,0}$ becomes $X_{i,1}$ just as it is.

The next process is similar to the previous process but the modular addition and XOR operation are out of order. A type I-2 table encodes a subkey to conceal. A high 4-bit of the encoded subkey and the 8-bit value $X_{i-1,2}$ are the input of a type II-2 table. The result and a low 4-bit of the encoded subkey calculate the modular addition by the type IV-2 table. After computing the modular addition by lookup operations, the output is divided into 2×4-bit values. Thus, the high 4-bit output and the value $X_{i-1,3}$ are the input of a type III-1 table. The 8-bit outcome value of the type III-1 table and the low 4-bit output make the intermediate value $X_{i,4}$. $X_{i,3}$ is gained by the $X_{i-1,2}$. The rest of process is the same as before. $X_{i,5}, X_{i,6}$ could be output of $X_{i-1,4}, X_{i-1,5}$ by the same process of the first one. The later process makes $X_{i,7}, X_{i,0}$ with input of $X_{i-1,6}$, $X_{i-1,7}$. Lastly all of the output values are rearranged by a left cyclic shift.

Final Transformation. It is easy to look into the final transformation since it is similar to the initial transformation. The value $X_{32,0}$ is added with a Whitening Key by using lookup tables, type IV-1 and IV-2 tables. A first byte C_0 of ciphertext is obtained after decoding table of a type V-1. C_1 is the output of the type V-1 from the intermediate value $X_{32,1}$. The value $X_{32,2}$ XOR with a Whitening key by using lookup tables of type III-1 and III-2. C_3 is obtained by a type V-2 table with an input value $X_{32,3}$. C_4, C_5 and C_6, C_7 are derived from $X_{32,4}, X_{32,5}$ and $X_{32,6}, X_{32,7}$ by the same process of $X_{32,0}, X_{32,1}$ and $X_{32,2}, X_{32,3}$ respectively.

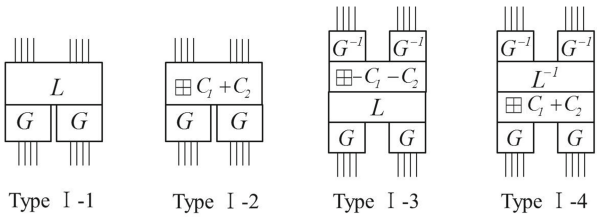

Fig. 1. Tables of Type I

4 Security and Performance Analysis

4.1 Security Analysis

To demonstrate the security of the proposed method against side channel attack, we mainly show that a masked intermediate value is independent from a

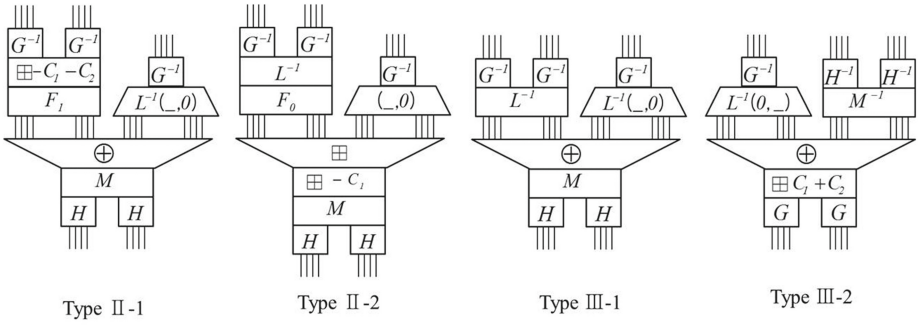

Fig. 2. Tables of Type II and Type III

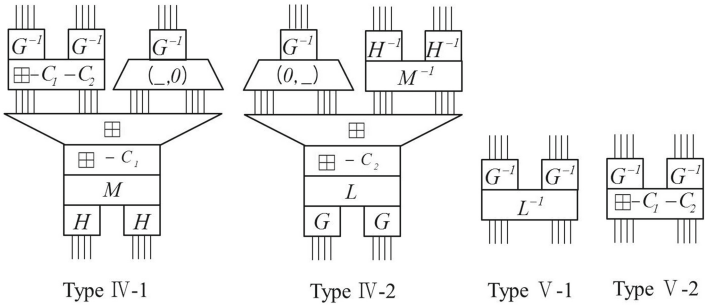

Fig. 3. Tables of Type IV and Type V

non-masked value. To do so, we first compare each bit of a masked and a non-masked intermediate values using the proposed and the original HIGHT implementations, respectively. The target intermediate value to be compared is X_2(third byte, see Algorithm 2) in the first round output because it is affected by the first byte of the first round key. The main step of single-bit DPA is to compute a differential trace after dividing power traces into two sets according the value of a target bit. The protection of DPA can be then justified if two bits of the non-masked and the masked X_2 at each bit position are different with probability 1/2. For the verification, we have performed encryption for 10,000,000 different plaintexts using the two HIGHT implementations, and also compared each bit of the masked and the non-masked values of X_2. As a result, Table 2 shows that they are different with a nearly 1/2 probability for every bit position. This property prevents a DPA attacker from constructing the correct sets of power traces and thus DPA is unlikely to work when using function masking.

In the case of CPA(correlation power analysis), an attacker computes a correlation value between the Hamming weights of a hypothetical value and the power consumption [13]. This is due to the fact that the power consumption of a micro-controller at a given point is known to be proportional or inversely proportional to the Hamming weight of a processed data. To demonstrate the

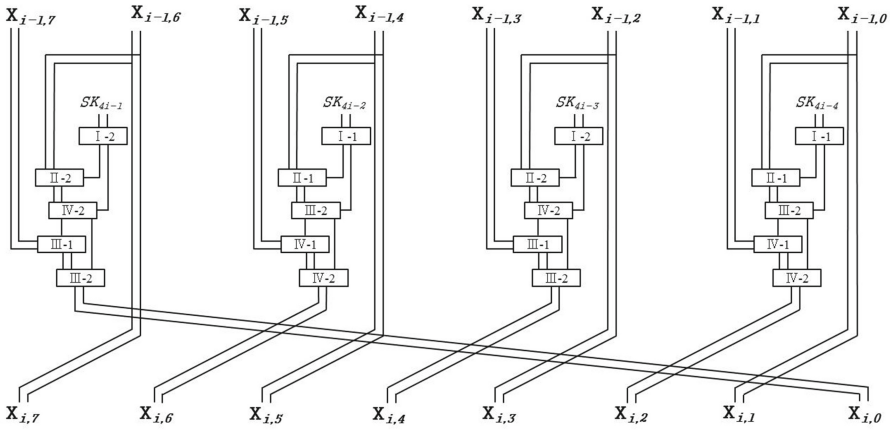

Fig. 4. Round transformation

Table 2. Probability of different bit between function masking and no masking intermediate

Bit position	1	2	3	4	5	6	7	8
Probability	49.99 %	50.00 %	50.01 %	50.00 %	50.01 %	49.97 %	50.00 %	49.99 %

protection of CPA, we show that the Hamming weights of a masked and a non-masked values of X_2 are independent from each other. Let HW_α denote the set of plaintexts that lead to the Hamming weight α of the non-masked value of X_2. Then, we have $\alpha \in [0, 8]$ because there are nine possible Hamming weights for an 8-bit value. We have performed encryption for 10,000,000 random plaintexts using the original HIGHT implementation and divided the plaintexts into HW_α, where $\alpha \in [0, 8]$. The next step is to show that the plaintexts in HW_α lead to well-distributed Hamming weights of X_2 in our implementation. For this purpose, we have repeated encryption on our proposed implementation for each set of plaintexts in HW_α, where $\alpha \in [0, 8]$. If the Hamming weights of the masked values of X_2 are uniformly distributed, they will show the probabilities for the Hamming weights of an 8-bit value shown in Table 3. For $\alpha \in$ [0,8], our experimental result shown in Table 4 gives us that the plaintexts in HW_α cause the Hamming weights of X_2 to be almost uniformly distributed in our implementation. This means that a masked and a non-masked values are not correlated to each other with overwhelming probability. We can therefore conclude that our function masking can also protect against CPA.

4.2 Performance Analysis

In this section, we compared the performance of the data masking and function masking. There is no secure implementation of HIGHT with second-order masking so far. Thus, it was tried to estimate the approximate overhead by calculating

Table 3. Probability distribution for the Hamming weight of a uniformly distributed 8-bit value [10]

HW	0	1	2	3	4	5	6	7	8
Prob	0.004	0.031	0.109	0.219	0.273	0.219	0.109	0.031	0.004

Table 4. Probability distribution for the Hamming weight of a function masked value

HW_α \ Masked HW	0	1	2	3	4	5	6	7	8
HW_0	0.0038	0.0307	0.1062	0.2201	0.2773	0.2196	0.1073	0.0313	0.0038
HW_1	0.0038	0.0311	0.1100	0.2189	0.2730	0.2197	0.1093	0.0303	0.0039
HW_2	0.0040	0.0312	0.1092	0.2185	0.2738	0.2190	0.1092	0.0312	0.0039
HW_3	0.0039	0.0312	0.1097	0.2189	0.2730	0.2186	0.1095	0.0313	0.0038
HW_4	0.0040	0.0312	0.1094	0.2186	0.2736	0.2189	0.1092	0.0312	0.0039
HW_5	0.0039	0.0312	0.1091	0.2188	0.2736	0.2189	0.1094	0.0312	0.0039
HW_6	0.0039	0.0312	0.1090	0.2191	0.2735	0.2189	0.1093	0.0312	0.0039
HW_7	0.0038	0.0311	0.1089	0.2183	0.2746	0.2187	0.1088	0.0317	0.0039
HW_8	0.0040	0.0314	0.1097	0.2150	0.2744	0.2189	0.1121	0.0306	0.0039

the number of operations required additional. Conversion of boolean and arithmetic mask is needed 10 times for one round function, when an implementation is used converting algorithm of [4]. Initial and final transformation are required two times mask conversion.

If data masking is applied at the beginning and end of 4 round, namely 8 rounds, initial and final transformation, the required operations of mask conversion are 86,268$(((8 \times 10) + (2 \times 2)) \times 1027)$ because one mask conversion needs 1,027 additional operations. In the case of no masking HIGHT, 392 operations are required because initial and final transformation need 4 operations in each and one round needs 12 operations where the HIGHT is composed of 32 rounds. Thus, it can be estimated that data masking version is over 200 times slower than the straightforward version. Even this is optimistic estimate excluding random number creation for mask conversion.

HIGHT applied function masking requires 16 times table lookup for initial transformation, 20 times for final transformation and 20 times for each round. For 8 rounds masking, table lookup will be 196 times. Since rest of unmasked 24 rounds require 288 operations, total operations for function masking are 484 times. Although this means function masking is 1.2 times slower than original HIGHT, actual runtime should be slower than the expectation because memory operation takes longer than ALU operation in CPU.

We implemented the function masked HIGHT in C language using a Intel core i7. Table 5 shows that lookup tables are around 25 Kbytes and it takes 1.79 times longer than original HIGHT.

Table 5. Lookup table size and time complexity of function masked HIGHT

Size of lookup tables			Time complexity	
Type I	4 tables	4×256	HIGHT (no masking)	754 cycles
Type II	2 tables	2×4096		
Type III	2 tables	2×4096	HIGHT (function masking)	1351 cycles
Type IV	2 tables	2×4096		
Type V	2 tables	2×256		
Total	26,112 bytes (25.5 kbytes)		Ratio	1.79 times

5 Conclusion

Prior works have documented the masking methods against the standard DPA attack. However, The masking method is vulnerable to the high-order DPA attacks since the attacks use correlation coefficient between two points or more. To resist the high-order attack, the high-order masking schemes have been proposed but it is not easy to implement in reality because of bad performance. In this study, it is possible to implement our function masking scheme which needs only a little overhead in reality. Thus, our scheme takes only 1.79 times more than the original HIGHT algorithm, but spends almost 200 times less than the second-order masking method. It means that it is possible to implement the masked HIGHT algorithm on the microprocessor against SCA by using 25 KB memory.

In the future, we should consider about the reduction of table size. The efficiency and security of the masked HIGHT should be verified by applying the function masking to the standard cryptographic algorithms, AES or ARIA. We expect that it is possible to compare with the high-order masked AES since many researches of high-order masking AES have been published. And it will be confirmed on the small processor devices as well as PC with different environments.

Acknowledgment. This work was supported by the K-SCARF project, the ICT R&D program of ETRI(Research on Key Leakage Analysis and Response Technologies).

References

1. Oswald, E., Mangard, S., Herbst, C., Tillich, S.: Practical second-order DPA attacks for masked smart card implementations of block ciphers. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 192–207. Springer, Heidelberg (2006)
2. Schramm, K., Paar, C.: Higher order masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006)
3. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, p. 3. Springer, Heidelberg (2001)

4. Vadnala, P.K., Großschädl, J.: Algorithms for switching between boolean and arithmetic masking of second order. In: Gierlichs, B., Guilley, S., Mukhopadhyay, D. (eds.) SPACE 2013. LNCS, vol. 8204, pp. 95–110. Springer, Heidelberg (2013)
5. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
6. Kim, H., Hong, S., Lim, J.: A fast and provably secure higher-order masking of AES S-box. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 95–107. Springer, Heidelberg (2011)
7. Fumaroli, G., Martinelli, A., Prouff, E., Rivain, M.: Affine masking against higher-order side channel analysis. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 262–280. Springer, Heidelberg (2011)
8. Hong, D., Sung, J., Hong, S.H., Lim, J.-I., Lee, S.-J., Koo, B.-S., Lee, C.-H., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J.-S., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)
9. Chow, S., Eisen, P., Johnson, H., Van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Nyberg, K., Heys, H. (eds.) Selected Areas in Cryptography. LNCS, vol. 2595, pp. 250–270. Springer, Heidelberg (2003)
10. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: revealing the secrets of smart cards, vol. 31. Springer, Heidelberg (2008)
11. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO' 99. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
12. Shannon, C.E.: Communication theory of secrecy systems. Bell Syst. Tech. J **28**(4), 656–715 (1949)
13. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)