Trends in Mathematics
Information Center for Mathematical Sciences
Volume 8, Number 1, June, 2005, Pages 1–15

DIFFERENTIAL POWER ATTACK AND MASKING METHOD

YOO-JIN BAEK, MI-JUNG NOH

Abstract. The differential power attack (DPA) tries to find out secret information from the power consumption signals derived during some cryptographic operations. Various countermeasures have been proposed against it and the masking method is known to be one of the most powerful algorithmic countermeasures against the first-order DPA. In this article, we give a brief survey about DPA and the masking method. Especially, we introduce some masking methods applicable to various arithmetic operations and Boolean operations, including modular additions and finite field multiplications. We then apply them to implement AES, SEED and SHA-1 DPA-securely and present the detailed implementation results.

1. Introduction

Side-channel attacks view cryptographic algorithms as concretely implemented programs running on some physical devices and deal with side-channel information obtained during cryptographic operations to get secret information in the devices. Especially, the differential power attack (DPA) [5] uses the power consumption signal as its side-channel information. Various countermeasures have been proposed against DPA and the masking method [6] is known to be one of the most powerful algorithmic countermeasures against the first-order DPA.

The first target of DPA was tamper-resistant devices, such as smart cards, but it can be applied to other cryptographic devices as well. The reason why DPA works is that the power consumption signal from a device is strongly related to the internal state of the device, and therefore it may leak critical information about the keying-parameters involved. In particular, we adopt the usual power leakage model which says that the power consumption information leaks the Hamming weight of the data being processed and under this model the first-order DPA investigates the statistical properties of the power consumption information at each sample time [7].

The idea of the masking method is that before a certain cryptographic operation involving a secret key is performed, the input data is scrambled using a randomly chosen value so that the Hamming weight of the data looks random to the outside world. This clearly prevents an adversary from performing the first-order DPA.

Key words and phrases. Differential Power Attack (DPA), Masking Method, Finite Field Multiplier, Adder, AES, SEED, SHA-1.

©2005 Information Center for Mathematical Sciences

In this article, we present the basic idea of DPA and introduce some masking methods applicable to various arithmetic operations and Boolean operations, including modular additions and finite field multiplications. The presented masking methods have various applications. For example, they can be applied to DPA-securely implement cryptographic algorithms which use a finite field multiplication as an internal operation. The typical example is AES (Advanced Encryption Standard) [1]. Also, they can be applied to algorithms which use both Boolean operations and arithmetic operations. SEED [4] and various hash functions including SHA-1 [10] belong to this case.

This paper is organized as follows. In Section 2, we introduce DPA, the masking method and the general masking problem. Also, we give a brief introduction to basic logic gates, a full adder, a ripple carry adder and finite field arithmetics. Section 3 describes how to apply the masking method to basic logic gates and adders. In Section 4, we give another masking method applicable to finite field inversions. Finally, we give a brief overview of AES, SEED and SHA-1 and present their DPA-secure implementation results in Section 5.

2. Preliminaries

2.1. **Differential Power Attack and Masking Method.** The differential power attack (DPA) which was first introduced by Kocher et al. in [5], tries to recover some secret information from power consumption signals derived during cryptographic computations. The reasons why DPA works are that 1) physical devices consume a different amount of power when operating on logical 1's compared to operating on logical 0's and 2) in the case of card applications, power supply from the outside world into a device makes the power consumption easy to be measured. In particular, this paper adopts the power leakage model that the power consumption information exposes the Hamming weight of the data being processed and considers the first-order DPA which investigates the statistical properties of power signals at each sample time. In the sequel, DPA stands for the first-order DPA.

We can perform DPA as follows: First, we gather power consumption signals for randomly chosen plaintexts, and then partition the set of manipulated plaintexts into several subsets according to some (guessed) key-dependent bits. Finally, we analyze the average of power signals in each partition. The idea of DPA is that for the rightly guessed key bits, power consumption is statistically related to the data computed with those bits. On the other hand, for the wrongly guessed key-bits, the relation between those data are likely to be random. In this way, one hopes to obtain useful information about the secret key.

The masking method [6] is known to be one of the most powerful software countermeasures against the first-order DPA. The main idea of the masking method is that before a certain cryptographic operation involving a secret key is performed, the input data should be masked so that the Hamming weight of the data being processed looks random to the outside world. More precisely, to apply the masking method, we first scramble (or mask) the input plaintext using a random value (the masking phase), then perform the corresponding cryptographic operation with

the scrambled data and finally descramble (or unmask) the resulting data to get a desired ciphertext (the unmasking phase).

In applying the masking method to cryptographic algorithms, we usually encounter the following masking techniques (used in the masking phase):

Definition 1. *For a k-bit binary string x, a Boolean mask of x is any tuple (x', r) such that $x = x' \oplus r$, where $\oplus$ denotes the eXclusive-OR operation. For a cryptographic use, r must be randomly chosen.*

Definition 2. *For a k-bit binary string x, an arithmetic mask of x is any tuple (x', r) such that $x = x' \pm r \pmod{2^k}$. For a cryptographic use, r must be randomly chosen.*

Now, to apply the masking method, we usually come up with a generic problem called a *masking problem*: for a given function $f : \{0,1\}^k \to \{0,1\}^n$, construct an efficiently computable (probabilistic) function $F : \{0,1\}^{2k} \to \{0,1\}^{2n}$ with the following properties:

1) Given any mask (x', r) of $x \in \{0,1\}^k$, F outputs a mask of $f(x)$, i.e. $F(x', r) = (f(x) \oplus s, s)$ or $F(x', r) = (f(x) \pm s \pmod{2^n}, s)$ for random $s \in \{0,1\}^n$.
2) The Hamming weight distributions of intermediate results in the computation must be independent of x.

Hence, for example, if f is a linear function (with respect to $\oplus$), we can set $F(x', r) = (f(x'), f(r))$, which solves the masking problem for f. Note that for a linear function f and $x = x' \oplus r$

$$f(x') \oplus f(r) = f(x \oplus r) \oplus f(r) = f(x).$$

For another example, if g is an affine function, i.e., there exists an $n \times k$ matrix A and a vector $b \in \{0,1\}^n$ such that $g(x) = A \cdot x \oplus b$, we can set $G(x', r) = (g(x'), A \cdot r)$, which solves the masking problem for g. But solving the masking problem for non-linear functions is not an easy task and the main focus of this paper is on applying the masking method to non-linear functions. In particular, many cryptographic primitives use the multiplicative inverse operation in a finite field or an arithmetic adder as non-linear functions. Thus, we mainly deal with the problems for these two non-linear functions. Hence, the related masking problems can be restated as follows:

Masking Problem for Inversion in a Finite Field For a given finite field $\mathrm{GF}(q)$ and $(x', r) \in \mathrm{GF}(q) \times \mathrm{GF}(q)$ with $x = x' + r$, compute $(x^{-1} + t, t)$ for some randomly chosen $t \in \mathrm{GF}(q)$, where the Hamming weight distributions of intermediate results must be independent of x during the computation.

Masking Problem for Modular Addition For a given integer k and $(x', r), (y', s) \in \{0,1\}^k \times \{0,1\}^k$ with $x = x' \oplus r$ and $y = y' \oplus s$, construct $((x + y \pmod{2^k}) \oplus t, t)$ for some randomly chosen $t \in \{0,1\}^k$, where the Hamming weight distributions of

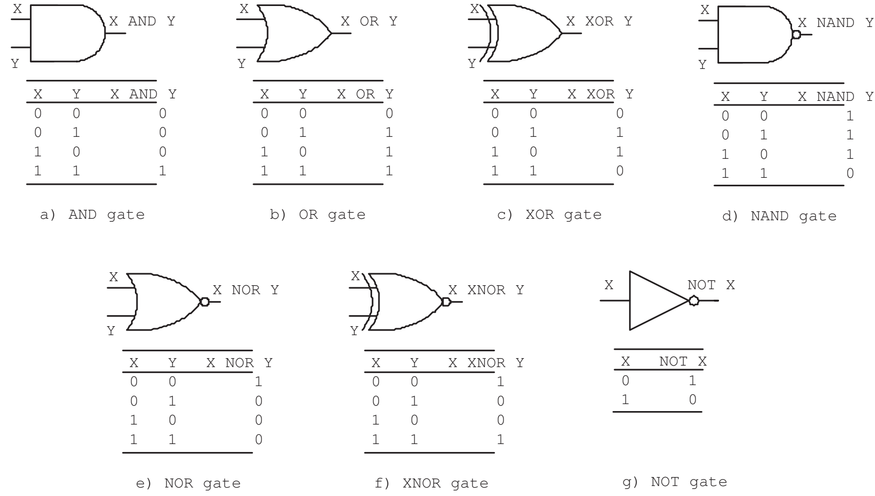

FIGURE 1. Basic Logic Gates

intermediate results must be independent of x and y during the computation.

In the following, we will give some algorithms to solve the above problems.

2.2. **Logic Gates and Adders.** Logic gates act as the basic building blocks for any combinational digital logic circuit. There are 7 basic logic gates, say, AND, OR, XOR, NOT, NAND, NOR, XNOR and Figure 1 shows the truth tables and symbols for these elements. In the sequel, we will denote AND, OR and XOR by $\wedge, \vee$ and $\oplus$, respectively.

To add operands with more than one bit, we must consider carries between bit positions. The basic building block for this operation is called a full adder. It takes X, Y and CIN (carry bit) as inputs and outputs two bits, say, S (sum) and $COUT$ (carry bit). The exact relations between these bits are given by the below equations and the circuit that performs the equations is shown in Figure 2 with its representing logic symbol:

$$\begin{aligned} S &= X \oplus Y \oplus CIN \\ COUT &= (X \wedge Y) \oplus (X \wedge CIN) \oplus (Y \wedge CIN). \end{aligned}$$

Two binary words, each with n bits, can be added using a ripple carry adder. Figure 3 shows the example circuit for a 4-bit ripple carry adder. In the figure, the first carry input c_0 is normally set to 0. Of course, there are different addition schemes which provide trade-offs between delay and area. The details can be found in [3].

In summary, an adder have a recursive structure and if we want to apply a masking method to it, it is very useful to use the structure.

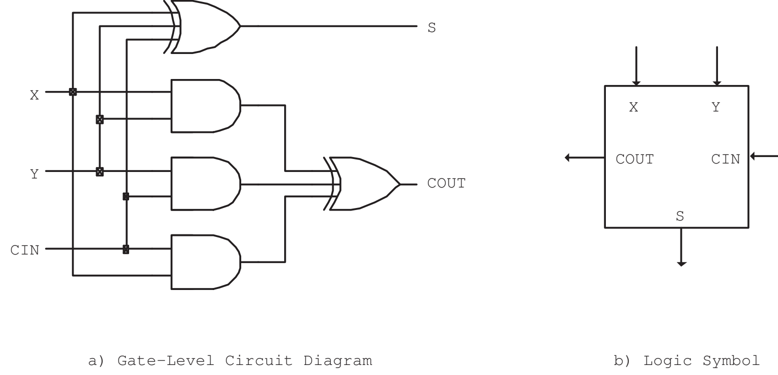

FIGURE 2. Full Adder

2.3. **Finite Field Arithmetic.** Many block ciphers including AES and SEED use the operations in a finite field $\mathrm{GF}(2^n)$ for some n. In this section, we give a brief explanation for the structure of $\mathrm{GF}(2^n)$.

Let $f(x)$ be an irreducible polynomial of degree n over GF(2). Then one can construct the finite field

$$\mathrm{GF}(2^n) \simeq \mathrm{GF}(2)[x]/(f(x)),$$

which consists of 2^n elements each of which can be uniquely written as a polynomial of degree less than n over GF(2). The addition in $\mathrm{GF}(2^n)$ is the usual polynomial addition over GF(2) and the multiplication is the usual polynomial multiplication over GF(2), followed by the modular reduction by $f(x)$. For any non-zero $g(x) \in GF(2^n)$, one can find $h(x), k(x) \in \mathrm{GF}(2)[x]$ such that

$$g(x)h(x) + f(x)k(x) = 1.$$

Then $h(x) \pmod{f(x)}$ is the multiplicative inverse of $g(x)$ in $\mathrm{GF}(2^n)$.

3. Adders with Masked Values

In [2], the author proposed a way how to apply the masking method to basic logic gates and we will review it in this section.

Note that XOR and NOT gates are affine functions (viewed as mathematical functions) and NAND, NOR, XNOR gates can be constructed as compositions of AND, OR and XOR gates with the NOT gate, respectively. Hence, to devise masking methods for all the basic logic gates, it is sufficient to do only for AND and OR gates, so we first restate the masking problems for these two gates.

Masking Problem for the AND Gate For given $(x', r), (y', s) \in \{0,1\} \times \{0,1\}$ with $x = x' \oplus r$ and $y = y' \oplus s$, compute $((x \wedge y) \oplus t, t)$ for a random bit t, where the Hamming weight distributions of intermediate results must be independent of x and y during the computation.

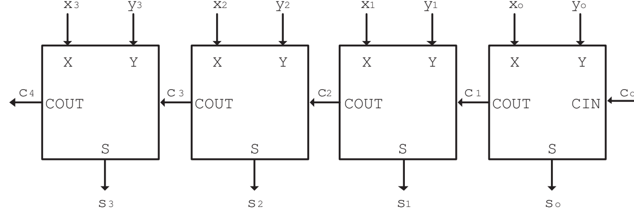

FIGURE 3. Ripple Carry Adder

Masking Problem for the OR Gate For given $(x', r), (y', s) \in \{0,1\} \times \{0,1\}$ with $x = x' \oplus r$ and $y = y' \oplus s$, compute $((x \vee y) \oplus t, t)$ for a random bit t, where the Hamming weight distributions of intermediate results must be independent of x and y during the computation.

Now, to solve the above problems, we use the following two simple equations

$$
\begin{aligned}
(x \oplus r) \wedge (y \oplus s) &= (x \wedge y) \oplus (x \wedge s) \oplus (r \wedge y) \oplus (r \wedge s), \\
x \vee y &= x \oplus y \oplus (x \wedge y).
\end{aligned}
$$

and accordingly get Algorithm 1 and Algorithm 2 below.

Algorithm 1 (Masking Method for the AND Gate)
Input: $x'(= x \oplus r), r, y'(= y \oplus s), s \in \{0,1\}$
Output: a random Boolean mask of $x \wedge y$
1. Calculate $a = y' \oplus (x' \wedge y')$.
2. Calculate $b = a \oplus (x' \wedge s)$.
3. Calculate $c = y' \oplus (r \wedge y')$.
4. Calculate $d = c \oplus (r \wedge s)$.
5. Return (b, d).

Algorithm 2 (Masking Method for the OR Gate)
Input: $x'(= x \oplus r), r, y'(= y \oplus s), s \in \{0,1\}$
Output: a random Boolean mask of $x \vee y$
1. Calculate $a = (x' \vee y') \oplus (x' \wedge s)$.
2. Calculate $b = (r \vee y') \oplus (r \wedge s)$.
3. Return (a, b).

To justify the algorithms, we need the following propositions.

Proposition 3. *The output of Algorithm 1 is a random Boolean mask of* $x \wedge y$.

Proof. Since

$$\begin{aligned} b \oplus d &= y' \oplus (x' \wedge y') \oplus (x' \wedge s) \oplus y' \oplus (r \wedge y') \oplus (r \wedge s) \\ &= (x' \wedge y') \oplus (x' \wedge s) \oplus (r \wedge y') \oplus (r \wedge s) \\ &= ((x \oplus r) \wedge (y \oplus s)) \oplus ((x \oplus r) \wedge s) \oplus (r \wedge (y \oplus s)) \oplus (r \wedge s) \\ &= x \wedge y \end{aligned}$$

and for any $\alpha, \beta \in \{0, 1\}$

$$\Pr(y' \oplus (r \wedge y') \oplus (r \wedge s) = 0 \mid x = \alpha, y = \beta) = \frac{1}{2},$$

(b, d) is a random Boolean mask of $x \wedge y$ as claimed. Note that the above probability is taken over the random choices of r and s. □

Proposition 4. *The probabilistic distributions of intermediate values of Algorithm 1 are independent of x and y, which implies that their Hamming weight distributions are independent of x and y.*

Proof. In the algorithm, there are 6 intermediate values to be considered, say, $x' \wedge y', x' \wedge s, r \wedge y', r \wedge s, y' \oplus (x' \wedge y')$ and $y' \oplus (r \wedge y')$. Now, for any α and β in $\{0, 1\}$, we have

$$\begin{aligned} & \Pr(x'y' = 0 \mid x = \alpha, y = \beta) \\ = & \Pr(x's = 0 \mid x = \alpha, y = \beta) \\ = & \Pr(ry' = 0 \mid x = \alpha, y = \beta) \\ = & \Pr(rs = 0 | x = \alpha, y = \beta) \\ = & \Pr(y' + x'y' = 0 \mid x = \alpha, y = \beta) \\ = & \Pr(y' + ry' = 0 \mid x = \alpha, y = \beta) \\ = & 3/4 \end{aligned}$$

so the assertions follow. □

Proposition 5. *The output of Algorithm 2 is a random Boolean mask of $x \vee y$.*

Proof. Since

$$\begin{aligned} a \oplus b &= (x' \vee y') \oplus (x' \wedge s) \oplus (r \vee y') \oplus (r \wedge s) \\ &= ((x \oplus r) \vee (y \oplus s)) \oplus ((x \oplus r) \wedge s) \oplus (r \vee (y \oplus s)) \oplus (r \wedge s) \\ &= x \vee y \end{aligned}$$

and for any $\alpha, \beta \in \{0, 1\}$

$$\Pr((r \vee y') \oplus (r \wedge s) = 0 \mid x = \alpha, y = \beta) = \frac{1}{2},$$

(a, b) is a random Boolean mask of $x \vee y$. As usual, the above probability is taken over the random choices of r and s. □

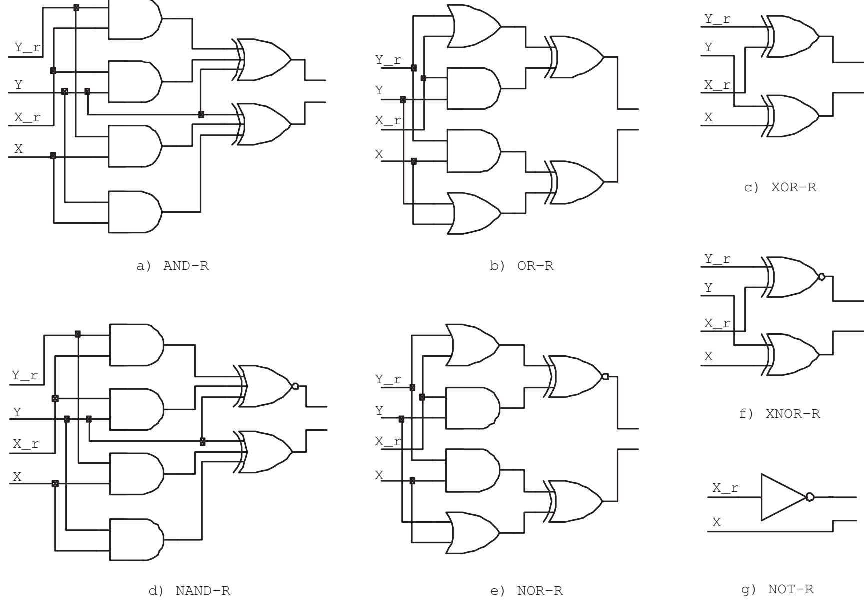

FIGURE 4. Basic Gates Manipulating Masked Values

Proposition 6. *The probabilistic distributions of intermediate values of Algorithm 2 are independent of x and y, which implies that their Hamming weight distributions are independent of x and y.*

Proof. In the algorithm, there are 4 intermediate values to be considered, say, $x' \vee y', x' \wedge s, r \vee y'$ and $r \wedge s$. Now, for any α and β in $\{0, 1\}$, we have

$$\begin{aligned} & \Pr(x' \wedge s = 0 \mid x = \alpha, y = \beta) \\ = \; & \Pr(r \wedge s = 0 \mid x = \alpha, y = \beta) \\ = \; & 3/4, \end{aligned}$$

$$\begin{aligned} & \Pr(x' \vee y' = 0 \mid x = \alpha, y = \beta) \\ = \; & \Pr(r \vee y' = 0 \mid x = \alpha, y = \beta) \\ = \; & 1/4, \end{aligned}$$

so the assertions follow. □

From the above propositions, we can conclude that Algorithm 1 and Algorithm 2 give complete solutions to the masking problems of AND and OR gates, respectively. Now, using AND and OR gates as building blocks, we can get masking algorithms for other logic gates, too. Figure 4 shows the resulted circuits.

After devising logical gates manipulating masked values, we apply those gates to a full adder. For example, the full adder in Figure 2 uses 3 AND gates and 2 3-input XOR gates and so if we substitute those gates with AND-R gates and XOR-R gates

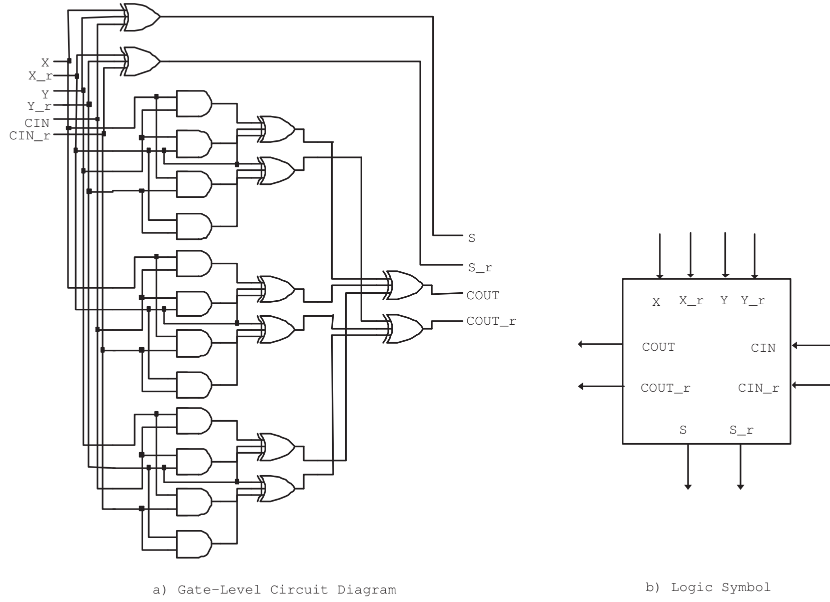

Figure 5. Full Adder Manipulating Masked Values

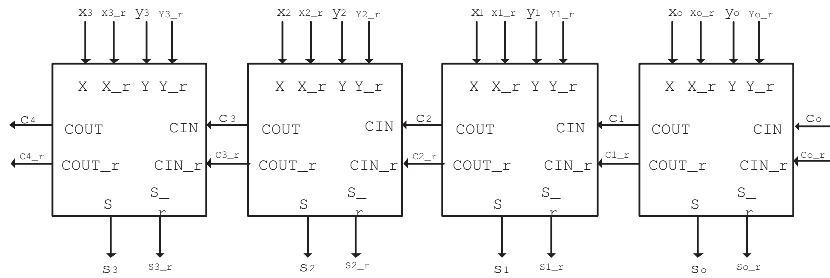

Figure 6. 4-Bit Ripple Adder Manipulating Masked Values

in Figure 4 (with minor modifications), we can get the masking algorithm for the adder. The resulting circuit can be found in Figure 5 with its logic symbol.

Finally, this new full adder can be used to generate a ripple carry adder which can manipulate masked values. Figure 6 shows the circuit of this ripple carry adder. We emphasize that the resulting ripple carry adder is secure against DPA and so can be used for implementing cryptographic algorithms securely against DPA.

4. Inversion in $GF(2^8)$ with Masked Values

In this section, we show how to implement the (multiplicative) inversion over $GF(2^8)$ using the arithmetics in $GF(((2^2)^2)^2)$ [9, 11] and then propose a size-efficient masking method for the operation. The details can be found in [8]. In the sequel,

we will use the following field structures:

$$\begin{aligned}
\mathrm{GF}(2^2) &\simeq \mathrm{GF}(2)[z]/(z^2+z+1),\\
\mathrm{GF}((2^2)^2) &\simeq \mathrm{GF}(2^2)[y]/(y^2+y+\phi), \phi=(10)_2\in\mathrm{GF}(2^2),\\
\mathrm{GF}(((2^2)^2)^2) &\simeq \mathrm{GF}((2^2)^2)[x]/(x^2+x+\lambda), \lambda=(1100)_2\in\mathrm{GF}((2^2)^2),\\
\mathrm{GF}(2^8) &\simeq \mathrm{GF}(2)[x]/(x^8+x^4+x^3+x+1).
\end{aligned}$$

Note that $\mathrm{GF}(2^8)\simeq\mathrm{GF}(((2^2)^2)^2)$ and the isomorphism $\sigma:\mathrm{GF}(2^8)\to\mathrm{GF}(((2^2)^2)^2)$ can be explicitly given by the following matrix

$$\sigma=\begin{pmatrix}
1&1&0&0&0&0&1&0\\
0&1&0&0&1&0&1&0\\
0&1&1&1&1&0&0&1\\
0&1&1&0&0&0&1&1\\
0&1&1&1&0&1&0&1\\
0&0&1&1&0&1&0&1\\
0&1&1&1&1&0&1&1\\
0&0&0&0&0&1&0&1
\end{pmatrix}.$$

Now, viewing $\mathrm{GF}(2^8)$ as $\mathrm{GF}(((2^2)^2)^2)$ via the above isomorphism, $g\in\mathrm{GF}(((2^2)^2)^2)$ can be considered as $g=ax+b$ for some $a,b\in\mathrm{GF}((2^2)^2)$. Hence $g^{-1}=(ax+b)^{-1}\in\mathrm{GF}(((2^2)^2)^2)$ can be computed by

$$(ax+b)^{-1}=\frac{1}{a^2\lambda+b(a+b)}(ax+a+b), \tag{1}$$

where all the operations like $a^2, a^2\lambda, b(a+b)$ and $(a^2\lambda+b(a+b))^{-1}$ take place in $\mathrm{GF}((2^2)^2)$. So the inverse operation in $\mathrm{GF}(2^8)$ can be computed using only three multiplications, one squaring, one constant multiplication by λ, three eXclusive-OR operations, and one inversion over $\mathrm{GF}((2^2)^2)$ (and one group isomorphism computation). Note that squaring and constant multiplication by λ in $\mathrm{GF}((2^2)^2)$ are linear functions so we can easily apply the masking method to those functions.

Now, $a^{-1}=(a_1y+a_0)^{-1}\in\mathrm{GF}((2^2)^2)$ can be computed by

$$a^{-1}=\frac{1}{a_1^2\phi+a_0(a_1+a_0)}(a_1y+(a_1+a_0)), \tag{2}$$

where all the operations like $a_1^2, a_1^2\phi, a_0(a_1+a_0)$ and $(a_1^2\phi+a_0(a_1+a_0))^{-1}$ occur in $\mathrm{GF}(2^2)$. Also, the multiplication in $\mathrm{GF}((2^2)^2)$ can be realized by using the operations over $\mathrm{GF}(2^2)$ as follows: for $a=a_1y+a_0, b=b_1y+b_0\in\mathrm{GF}((2^2)^2)$ with $a_1,a_0,b_1,b_0\in\mathrm{GF}(2^2)$

$$ab=((a_1x+a_0)(b_1x+b_0)+a_0b_0)y+(a_1b_1\phi+a_0b_0). \tag{3}$$

Hence multiplication and inversion over $\mathrm{GF}((2^2)^2)$ can be implemented using multiplications, squaring, constant multiplications by ϕ, and inversion over $\mathrm{GF}(2^2)$. Note that inversion in $\mathrm{GF}(2^2)$ is a linear function because $a^{-1}=a^2$ for all $a\in\mathrm{GF}(2^2)$. Also, squaring and constant multiplication by ϕ in $\mathrm{GF}(2^2)$ are linear functions so we can easily apply the masking method to those functions. In summary, if we can apply the masking method to the multiplication in $\mathrm{GF}(2^2)$, we can apply

it to the the inversion in $\mathrm{GF}(2^8)$ and the following algorithm gives us the solution:

Algorith 3 (Masking Method for Finite Field Multiplier)
Input: $x'(=x+r), r, y'(=y+s), s \in \mathrm{GF}(q)$
Output: $(xy+t, t)$ for some random $t \in \mathrm{GF}(q)$
1. Calculate $a = y' + x'y'$.
2. Calculate $b = a + x's$.
3. Calculate $c = y' + ry'$.
4. Calculate $d = c + rs$.
5. Return (b, d).

Algorithm 3 can be justified as follows:

Proposition 7. *The output (b, d) of the above algorithm is a random mask of xy.*

Proof. Since $b+d = ((y'+x'y')+x's)+((y'+ry')+rs) = x'y'+x's+ry'+rs = xy$ and for any $\alpha, \beta, \gamma \in \mathrm{GF}(q)$

$$\Pr(y' + ry' + rs = \gamma \mid x = \alpha, y = \beta) = 1/q,$$

(b, d) is a random mask of xy. Note that the above probability is taken over the random choices of r and s. □

Lemma 8. *The probabilistic distributions of intermediate values of the above algorithm is independent of x and y.*

Proof. In the algorithm, there are 6 intermediate values to be considered, say, $x'y', x's, ry', rs, y' + x'y'$ and $y' + ry'$. Now, for any α and β in $\mathrm{GF}(q)$, we have:

$$\begin{aligned} & \Pr(x'y' = \gamma \mid x = \alpha, y = \beta) \\ = & \Pr(x's = \gamma \mid x = \alpha, y = \beta) \\ = & \Pr(ry' = \gamma \mid x = \alpha, y = \beta) \\ = & \Pr(rs = \gamma | x = \alpha, y = \beta) \\ = & \Pr(y' + x'y' = \gamma \mid x = \alpha, y = \beta) \\ = & \Pr(y' + ry' = \gamma \mid x = \alpha, y = \beta) \\ = & \begin{cases} \dfrac{2q-1}{q^2} & \text{if } \gamma = 0 \\ \\ \dfrac{q-1}{q^2}, & \text{if } \gamma \neq 0, \end{cases} \end{aligned}$$

where the probability is taken over the random choices of r and s. The assertion now follows.

□

Proposition 9. *The Hamming weight distributions of intermediate values of the above algorithm is independent of x and y.*

Proof. For a bit sting x, let $|x|$ denote the Hamming weight of x and put $f(x', r, y', s)$ to stand for one of the intermediate values of the above algorithm. Then, for any α and β from GF(q) and any positive integer n, we have

$$\begin{aligned} & \Pr(|f(x', r, y', s)| = n \mid x = \alpha, y = \beta) \\ = & \sum_{|\gamma| = n} \Pr(f(x', r, y', s) = \gamma \mid x = \alpha, y = \beta), \end{aligned}$$

where the summation is taken over all $\gamma \in \mathrm{GF}(q)$ with $|\gamma| = n$ and the probability is taken over the random choices of r and s. Now, the assertion follows from the above lemma. □

Remark Algorithm 1 is a special case of Algorithm 3 because the multiplication over GF(2) is a bitwise AND operation.

5. Applications

In this section, we present the implementation results for AES, SEED and SHA-1 with the presented masking methods [2, 8]. The hardware implementations were done using Verilog-HDL, which resulted in DPA-secure implementations of those algorithms. The simulation and the synthesis were conducted with Cadence NC-Verilog and Synopsys Design-Compiler using Samsung smart-card library (*smart130*), which is a 0.18 μm CMOS technology. The results are summarized in Table 1. For completeness, we include the specifications of the algorithms in the following subsections.

5.1. **AES.**

AES uses the following primitive functions [1]:

- ShiftRows - Each row in a 4 by 4 byte-array (called a state) of data is shifted 0,1,2 or 3 bytes to the left, producing a new state.
- SubBytes - Replaces each byte in a state with its substitute in an S-Box. The S-Box is an invertible function which is constructed by a composition of two transforms: first, each byte is replaced with its multiplicative inverse in GF(2^8) and then, an affine transformation is applied.
- AddRoundKey - Simply XOR-es an input with a sub-key.
- MixColumns - Each column in a state is considered as polynomial over GF(2^8) and is multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = \{0x03\}x^3 + \{0x01\}x^2 + \{0x01\}x + \{0x02\}$.

ShiftRows and MixColumns are byte-wise linear operations and AddRoundKey is an affine function. Thus, as noted before, we can apply easily the masking method to these operations. On the other hand, SubBytes is a parallel application of nonlinear functions called S-Boxes and each S-box is composed of the multiplicative inverse function in $\mathrm{GF}(2^8) \simeq \mathrm{GF}(2)[x]/(x^8 + x^4 + x^3 + x + 1)$, followed by an affine transformation. Hence, if we can efficiently apply the masking method to the multiplicative inverse function in $GF(2^8)$, then we can also do that to the whole AES algorithm, which is the point of this paper.

We implemented AES in Verilog-HDL using Algorithm 3. The implementation is optimized for power and area, rather than performance, and therefore needs 16 clock cycles for each round. The implementation result is summarized in Table 1.

5.2. **SEED.** SEED is a Feistel-type block cipher with 16 rounds and 128-bit block and key length. It is now a national industrial association standard in Korea [4] and is believed to be robust against known attacks including DC (Differential Cryptanalysis) and LC (Linear Cryptanalysis). The overall specification is as follows: an 128-bit input is divided into two 64-bit blocks. The resulting right 64-bit block and a 64-bit round key generated from the key scheduling algorithm are inputs to the round function F. The input block of F is divided into two 32-bit subblocks (C, D) and the output $(C', D') = F(C, D)$ is computed as follows:

$$\begin{aligned} C' &= G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})] + G\{(C \oplus K_{i,0}) \oplus \\ &\quad (D \oplus K_{i,1})\}] + G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})], \\ D' &= G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})] + G\{(C \oplus K_{i,0}) \oplus \\ &\quad (D \oplus K_{i,1})\}], \end{aligned}$$

where the round key is a concatenation of $K_{i,0}$ and $K_{i,1}$. The symbol $+$ stands for the addition modulo 2^{32}. The function G has two layers: a layer of two 8×8 S-boxes which are defined as a monomial (say, x^{247} and x^{251}) over $\mathrm{GF}(2^8) \simeq \mathrm{GF}(2)[x]/(x^8 + x^6 + x^5 + x + 1)$ followed by an affine function in $\mathrm{GF}(2^8)$ and a layer of a block permutation of sixteen 8-bit sub-blocks. The complete specification including the key scheduling algorithm can be found in [4].

The presented masking algorithm for adders is applied to $+$ operation in F function of SEED. The input data to SEED is initially masked with a Boolean mask and the new algorithm ensures that the masked data does not get corrupted during arithmetic operations. Hence the output data of SEED can be properly unmasked with the appropriate Boolean mask computed, while preventing any leakage of critical information related to operation taking place or the keying-parameters involved.

We implemented SEED in Verilog-HDL using our masking algorithm for addition modulo 2^{32} and using Algorithm 3 for G-function. Our implementation is optimized for power and area and so needs 7 clock cycles for each round. The implementation result is summarized in Table 1.

5.3. **SHA-1.** SHA-1 produces a 160-bit hash value for a message of length less than 2^{64} bits. For the purpose, it first applies a specific padding rule to a message, which results in a padded message of length $512n$ bits for some n. And then it sequentially processes blocks of 512 bits. A sequence of logical functions f, g, h are used in the SHA-1 computation, each of which operates on three 32-bit words u, v, w and produces a 32-bit word as output. f, g and h are defined as follows: for

	Gate Count	Critical Path	Maximum Throughput @ Maximum Clock Frequency
AES w/o masking	17.4K	49 ns	16 Mbps @ 20 MHz
AES w masking	25.9K	59 ns	12 Mbps @ 15 MHz
SEED w/o masking	9.5K	32 ns	33.96 Mbps @ 30 MHz
SEED w masking	16.2K	63 ns	11.32 Mbps @ 10 MHz
SHA-1 w/o masking	8.8K	12 ns	512 Mbps @ 80 MHz
SHA-1 w masking	24.8K	43 ns	128 Mbps @ 20 MHz

TABLE 1. Implementation Results

words u, v, w,

$$\begin{aligned} f(u,v,w) &= (u \wedge v) \vee (\overline{u} \wedge w), \\ g(u,v,w) &= (u \wedge v) \vee (u \wedge w) \vee (v \wedge w), \\ h(u,v,w) &= u \oplus v \oplus w, \end{aligned}$$

where $\overline{u}$ denotes the bit-wise negation of u. A sequence of constant words y_1, y_2, y_3, y_4 is also used in the SHA-1 computation, which is given by $y_1 = 0x5a827999, y_2 = 0x6ed9eba1, y_3 = 0x8f1bbcdc$ and $y_4 = 0xca62c1d6$. To generate the message digest, the padded message is represented as the n 16-word blocks $M_1, M_2, ..., M_n$. The processing of each M_i involves 80 steps. Before processing any blocks, the chaining variables $(H_1, H_2, H_3, H_4, H_5)$ are initialized as follows: $H_0 = 0x67452301, H_1 = 0xefcdab89, H_2 = 0x98badcfe, H_3 = 0x10325476, H_4 = 0xc3d2e1f0$. Then, to process M_i, we proceed as follows: (The below process is called the SHA-1 core function)

1. Divide M_i into 16 words $W[0], W[1], ..., W[15]$.
2. For $16 \leq t \leq 79$, let $W[t] = (W[t-3] \oplus W[t-8] \oplus W[t-14] \oplus W[t-16] \hookleftarrow 1)$.
3. $(A, B, C, D, E) = (H_1, H_2, H_3, H_4, H_5)$;.
4. For t = 0 to 79 do,
   $temp = (A \hookleftarrow 5) + f_t(B, C, D) + E + W[t] + y_t$;
   $(A, B, C, D, E) = (temp, A, B \hookleftarrow 30, C, D)$;
5. $(H_1, H_2, H_3, H_4, H_5) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$;.

Here, for a word a and a positive integer s, $a \hookleftarrow s$ stands for the left-rotation of a by s position and + represents the addition modulo 2^{32}, to which the result of this paper can be applied.

Although a hash function itself doesn't involve any secret information during its computation, it can be used as a primitive function for the HMAC function. HMAC is a mechanism that provide an integrity check based on a secret key and cryptographic hash functions. The definition of HMAC requires a secret key k and a cryptographic hash function H. We denote by B the byte-length of such blocks (B=64 for SHA-1), and by L the byte-length of hash outputs (L=20 for SHA-1).

The key k can be of any length up to B and the minimal recommended length for K is L bytes. During the HMAC computation, we need the following constants: $ipad =$ the byte $0x36$ repeated B times, $opad =$ the byte $0x5C$ repeated B times. Finally, we perform:

$$\mathrm{HMAC}(k, \mathrm{text}) = \mathrm{H}(k \oplus opad, \mathrm{H}(k \oplus ipad, \mathrm{text})).$$

We implemented the SHA-1 core function (which manipulates the 512-bit block) in Verilog-HDL, using our masking method for an arithmetic adder. Our implementation needs 80 clock cycles for the computation. The implementation result is summarized in Table 1.

6. Conclusion

In this paper, we give a brief survey about DPA and the masking method. Especially, we introduce some masking methods applicable to various arithmetic operations and Boolean operations including modular additions and finite field multiplications. Then, we apply them to implement AES, SEED and SHA-1 DPA-securely.

References

[1] National Institute of Standards and Technology, *Federal Information Processing Standards Publication 197, Anouncing the Advanced Encryption Standard(AES)*, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[2] Y.J. Baek, *DPA-Resistant Logic Gates Applicable to Cryptographic Algorithm Implementations with Masking Method*, preprint.

[3] M.D. Ercegovac and T. Land, *Digital Arithmetic*, Morgan Kaufmann Publishers, 2004.

[4] Korea Information Security Agency (KISA), *SEED Algorithm Specification*, available at http://www.kisa.or.kr.

[5] P. Kocher, J. Jaffe and B. Jun, *Differential power analysis*, Proceedings of Crypto '99, LNCS vol. 1666, Springer-Verlag, 1999, pp. 388–397.

[6] T. Messerges, *Securing the AES finalists against power analysis attacks*, Proceedings of Fast Software Encryption Workshop 2000, LNCS vol. 1978, Springer-Verlag, 2000, pp. 150–165.

[7] T. Messerges, *Using Second-Order Power Analysis to Attack DPA Resistant Software*, Proceedings of Cryptographic Hardware and Embedded Systems: CHES 2000, LNCS vol. 1965, Springer-Verlag, 2000, pp. 238–251.

[8] M.J. Noh, Y.J. Baek and K.M. Ahn, *A Low-Power and Secure AES Coprocessor Protecting Differential Power Attack with 0.18μm CMOS Smart Card Technology*, Conference Proceedings of The International Embedded Solutions Event GSPx, 2004.

[9] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao and P. Rohatgi, *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*, Proceedings of Cryptographic Hardware and Embedded Systems: CHES 2001, LNCS vol. 2162, Springer-Verlag, 2001, pp. 175–188.

[10] National Institute of Standards and Technology, *Federal Information Processing Standards Publication 180-2, Anouncing the Secure Hash Standard*, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, 2002.

[11] A. Satoh, S. Morioka, K. Takano and S.Munetoh, *A compact Rijndael Hardware Architecture with S-Box Optimization*, Proceedings of Asiacrypt 2001, LNCS Vol. 2248, 2001, pp. 239–254.

[12] J.F. Wakerly, *Digital Design Principles and Practices*, Prentice-Hall International Editions, 1990.

SoC R&D Center, Samsung Electronics, Korea
E-mail address: {yoojin.baek, mjnoh}@samsung.com