SEED의 차분전력공격에 대한 강화 방안

장화선*, 박상배*, 김광조*

*한국정보통신대학원대학교, 국제정보보호기술연구소

Enhancing the Security of SEED against Differential Power Analysis

Hwasun Chang*, Sangbae Park*, and Kwangjo Kim*

*International Research center for Information Security

Information and Communications Univ.(ICU)

요 약

최근 부채널 공격으로 스마트카드 같은 장치의 비밀 키를 알아낼 수 있음이 알려지면서 많은 알고리즘에 대한 부채널 공격과 대응 방안이 연구되고 있다. DPA는 부채널 공격의 일종으로 암호화 연산 중 발생하는 전력 소모 곡선을 통계적으로 분석하여 키를 알아내는 공격이다. 한편 SEED가 국내 표준 블록 암호 알고리즘으로 널리 사용되고 있으나 SEED에 대한 DPA 연구 결과는 발표된 바가 없는 것 같다. 본 논문에서는 SEED에 DPA를 적용할 수 있음을 보이고 대응방안 예를 제시한다.

I. 서론

Kocher에 의해 Differential Power Analysis (DPA)가 소개된 이후 DES, AES, RSA, ECC 등 많은 알고리즘에 대해 공격방안과 대응 기술이 연구되었다[2, 3, 4, 5, 6, 7, 8]. 이는 최근 스마트카드가 tamper-resistant device로서 사용이 증가하고 있지만 DPA를 사용하면 스마트카드 내부의 비밀 키를 비교적 쉽게 알아낼 수 있다는 사실이 확인된 때문인 것 같다. 이런 이유로 Visa에서는 스마트카드 인증 시 DPA에 대한 대응방안이 구현되어 있는지 확인한다.

한편 SEED는 국내 표준 블록 암호 알고리즘으로 많은 곳에 이용되고 있다[1]. 그러나 SEED에 대한 DPA 결과는 발표된 사례가 없는 것 같다. 본 논문에서는 먼저 SEED에 대해 DPA를 적용할 수 있는지 살펴본다. 이 때 Akkar *et al.*이 제안한 전력 소모 모델과 공격법을 활용한다[14].

다음으로는 대응방안을 살펴본다. DPA에 대한 대응방안은 하드웨어에 의한 방법과 소프트웨어에 의한 방법이 있다. 최근에 생산되는 스마트카드는 칩 자체에 DPA에 대응할 수 있는 방안이 하드웨어로 구현되어 있는 경우가 많다. 하지만 Clavier는 하드웨어만 사용한 대응방안이 완전하지 않음을 보였고 실제로도 소프트웨어에 의한 대응방안을 같이 사용하는 경우가 많다 [4].

소프트웨어에 의한 대응방안으로 AES의 경우에는 state의 각 바이트마다 임의로 생성한 마스크를 xor하여 DPA에 사용될 중간 값을 없애는 방법이 제안되었다[3]. 하지만 이 방법을 적용하려면 매 실행 시 임의로 생성된 마스크마다 S-box를 새로 계산하여야 하고 이를 위해 많은 램과 계산이 요구된다. 이를 해결하기 위해 S-box의 역원을 구하는 연산 전에 덧셈 마스크를 곱셈 마스크로 변환하는 방법이 제안되었다[5]. 그러나 후에 이 방법은 DPA로 공격할 수 있음이 밝혀졌다[9]. 한편 롬에 미리 생성된 마스크와 재 계산된 S-box를 저장한 후 알고리즘 시작 시 임의로 한 세트를 선택하여 사용하는 방법이 제안되었다[10]. 이 방법은 스마트카드에서 비교적 풍부한 자원인 롬을 사용하여 저가 스마트카드에서도 구현이 가능하다는 장점이 있다. 이러한 마스크 방법을 SEED를 위한 대응방안으로 사용할 수 있을 것이

다.

본 논문의 구성은 다음과 같다. 2장에서는 DPA에 관련된 내용을 중심으로 SEED 알고리즘을 살펴본다. 3장에서는 SEED에 DPA를 적용하는 방법에 대해 논의한다. 4장에서는 SEED에 DPA 대응방안을 구현하는 방법에 대해 살펴본다.

II. SEED

SEED는 데이터와 키 길이가 모두 128비트인 Feistel 구조이고 16라운드로 구성된다. 차분 공격과 선형 공격에 강하도록 설계되었고 CRYPTREC에서 안전성 평가를 받았다. 여기서는 DPA 공격에 관련된 부분을 중심으로 SEED를 살펴본다.

1. 알고리즘 전체 구조

그림 1은 알고리즘의 전체 구조이다. 128비트 입력 평문을 2개의 64비트 블록 $(L_0(64), R_0(64))$으로 나눈 후 16개의 64비트 라운드 키를 이용하여 16라운드를 수행한 후 암호문으로 128비트 블록 $(L_{16}(64), R_{16}(64))$을 출력한다.

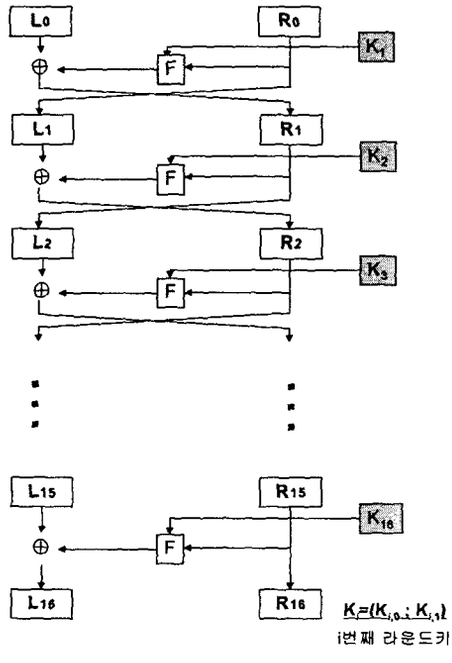

그림 1: SEED 전체 구조

2. F 함수

F 함수는 64비트 블록(C, D)와 64비트 라운드 키 $K_i = (K_{i,0}, K_{i,1})$를 입력으로 받아 64비트 블록 (C', D')을 출력한다.

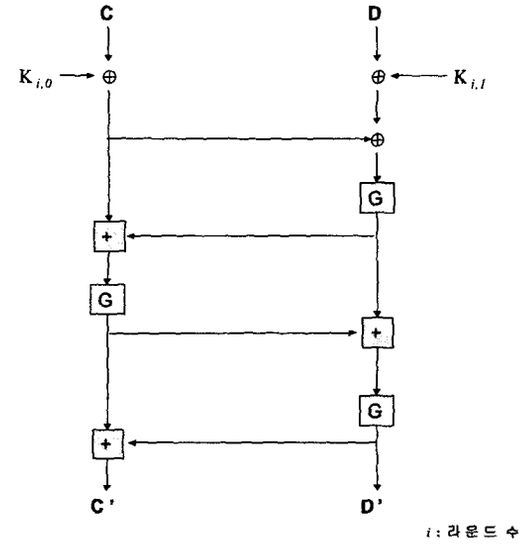

그림 2: F 함수 구조

3. G 함수

G 함수는 다음과 같이 기술된다.

$$Y_3 = S_2(X_3),\ Y_2 = S_1(X_2),\ Y_1 = S_2(X_1),\ Y_0 = S_1(X_0),$$

$$\begin{aligned} Z_3 &= (Y_0 \& m_3) \oplus (Y_1 \& m_0) \oplus (Y_2 \& m_1) \oplus (Y_3 \& m_2) \\ Z_2 &= (Y_0 \& m_2) \oplus (Y_1 \& m_3) \oplus (Y_2 \& m_0) \oplus (Y_3 \& m_1) \\ Z_1 &= (Y_0 \& m_1) \oplus (Y_1 \& m_2) \oplus (Y_2 \& m_3) \oplus (Y_3 \& m_0) \\ Z_0 &= (Y_0 \& m_0) \oplus (Y_1 \& m_1) \oplus (Y_2 \& m_2) \oplus (Y_3 \& m_3) \end{aligned}$$

$$(m_0 = 0\text{xfc},\ m_1 = 0\text{xf3},\ m_2 = 0\text{xcf},\ m_3 = 0\text{x3f})$$

4. 라운드 키 생성

라운드 i 에 사용되는 라운드 키 $K_i = (K_{i,0}, K_{i,1})$는 다음과 같은 방식으로 생성한다.

```
for(i = 1; i <= 16; i++){
    Ki,0 ← G(A+C-KCi);
    Ki,1 ← G(B-D+KCi);
    if(i % 2 == 1) A||B ← (A||B) >> 8;
    else C|| D ← (C||D) << 8; }
```

Ⅲ. DPA

여기서는 SEED에 DPA를 적용하는 방법에 대해 살펴본다. 우선 전력 소모 모델을 소개하고 DPA를 통해 각 라운드 키를 알아내는 방법을 살펴본다. 다음으로 찾아낸 라운드 키로부터 비밀키를 계산하는 방법에 대해 기술한다.

1. 전력 소모 모델

Akkar *et al.*은 고가 및 저가 두 종류의 스마트카드에 대해 실험으로 전력 소모 모델을 구하였다[14]. 실험에 따르면 고가 스마트카드는 선형 모델로 나타낼 수 있었다. 즉 입력 x에 대한 소모 전력 $P(x)$는 다음 식으로 나타낼 수 있다.

$$P(x) = \sum_{i=0}^{n-1} x_i P_i \qquad (1)$$

여기서 n은 x의 비트 수이고 x_i는 x의 i 번째 비트이며 P_i는 실험에 의해 구한 x_i에 대한 상수이다. P_i는 상수도 아니었고 항상 양의 값이거나 음의 값도 아니어서 Hamming weight에 의한 기존 전력 소모 모델이 불완전함을 보였다.

고가 및 저가 스마트카드 모두에 대해 x_i가 0인 경우와 1인 경우의 평균 소모 전력의 차이에 대한 실험도 이루어졌다. 실험 결과에 따르면 x_i가 0인 경우와 1인 경우사이에 평균 전력 소모에 차이가 있었으며 기존 DPA가 유효함이 확인되었다.

2. 라운드 키 복구

라운드 키 복구에는 x_i가 0인 경우와 1인 경우의 평균 전력 소모가 다르다는 사실을 이용할 수 있다. Akkar *et al.*의 실험에 의하면 i 값에 따라 x_i가 0인 경우의 평균 전력 소모가 가 클 수도 있고 1인 경우가 클 수도 있다. 여러 x에 대해 x가 이동되는 때의 전력 소모를 측정한 후 전체의 평균을 취하고 특정 비트가 0인 값들의 평균 A_0와 1인 값들의 평균 A_1을 구한다. 전체 평균은 A_0와 A_1 사이에 위치하므로 전체 평균에 대해 A_0와 A_1은 다른 쪽에 위치하게 된다. 이는 Akkar *et al.*이 DESX와 같은 알고리즘의 "whitening" 부분을 공격하여 키를 알아내기 위해 사용한 Binary Power Analysis(BPA)의 원리이다. BPA는 Akkar *et al.*이 DPA의 일종으로 제안한 방법이며 whitening은 암호화 전과 후에 데이터에 키의 xor를 취하여 알고리즘의 변경이나 반복 수행 없이 키 길이를 늘리는 기술이다. Akkar *et al.*은 BPA를 저가 스마트카드에 실제로 적용한 결과를 제시하여 실효성을 입증하였다. 본 논문에서는 BPA를 SEED의 라운드 키를 알아내기 위해 사용한다.

먼저 첫 번째 라운드 키 K_1을 복구하는 방법에 대해 살펴본다. K_1은 $K_{1,0}$와 $K_{1,1}$으로 이루어져 있으며 $K_{1,0}$는 F 함수에서 평문의 64번째부터 95번째 비트와 xor 된다. xor를 위한 구체적인 순서는 다음과 같다.

a. 평문의 해당 바이트를 가져온다.

b. 라운드 키의 해당 바이트와 평문의 저장된 바이트가 xor 된다.

c. xor 결과가 저장된다.

라운드 키의 한 비트를 주목할 때 그 값이 0이면 평문의 해당 비트는 그대로 결과가 된다. 만약 라운드 키의 한 비트가 1이면 평문의 해당 비트는 반전되어 결과로 저장된다. 이 사실을 이용하여 라운드 키 $K_{1,0}$의 j (=0...31)번째 비트 $K_{1,0}^j$를 구하는 절차는 다음과 같다.

a. 여러 개의 평문을 임의로 생성하고 이를 암호화하면서 전력 소모 곡선을 저장한다. 이로부터 평문과 전력 소모 곡선의 쌍 (Pt_k, Pc_k)를 얻을 수 있다.

b. $K_{1,0}$의 j 번째 비트 $K_{1,0}^j$는 평문의 $(64 + j)$번째 비트와 xor되므로 (Pt_k, Pc_k)를 평문 Pt_k의 $(64 + j)$번째 비트가 0인 것과 1인 것으로 분류한다. 0인 집합을 S_0, 1인 집합을 S_1이라고 한다.

c. S_0과 S_1 각각에 대해 평균 전력 소모 곡선 Pc_{avg0}와 Pc_{avg1}을 구한다. 구 곡선의 차이를 구하고 이를 Dc_{avg}라 한다. Dc_{avg}는 시간에 따른 평균 전력 소모의 차이를 나타내며 어느 한 순간은 평문 중 $(64 + j)$번째 비트를 포함하는 바이트가 가져와 지는 때 T_l이고 얼마 후에 xor된 결과가 저장되는 순간 T_s가 있을 것이다. S_0과 S_1는 평문 중 $(64 + j)$번째 비트 값이 다르므로 T_l에서 평균 전력 소모의 차이가 발생할 것이고 Dc_{avg}에서 카드 특성에 따라 위쪽 혹은 아래쪽으로 펄스를 관찰할 수 있을 것이다. 라운드 키의 해당 비트가 0이었다면 연산 결과 평문의 해당 비트가 변하지 않으므로 T_s에서 동일한 방향으로 펄스를 관찰할 수 있을 것이며 라운드 키의 해당 비트가 1이었다면 평문의 해당 비트가 반전되므로 T_s에서 T_l에서와는 다른 방향으로의 펄스가 관찰될 것이다.

위의 과정을 j가 0인 값부터 31인 값까지 반복하면 $K_{1,0}$를 모두 구할 수 있다. 같은 방법으로 $K_{1,1}$도 구할 수 있다. 첫 번째 라운드 키 K_1을 구했으므로 평문과 K_1으로부터 2번째 라운드로의

입력 중 오른쪽 부분인 R_1을 구할 수 있다. K_1에 사용한 것과 같은 방법을 사용하면 K_2도 구할 수 있다. 이와 같은 과정을 반복하면 모든 라운드 키를 찾아낼 수 있다.

한편으로 이 방법은 에러 검지 기능도 가지고 있다. 한 라운드 키를 잘못 계산하면 다음 라운드 키 계산을 위해 (Pt_k, Pc_k)를 분류하는 과정에서 제대로 분류할 수 없게 되고 Dc_{avg}를 구해도 펄스를 관찰할 수 없게 된다. 이러한 경우에는 바로 이전 라운드 키를 다시 확인해야 한다.

3. 비밀 키 복구

여기서는 BPA로부터 구한 라운드 키들을 이용하여 비밀 키를 복구하는 방법을 기술한다. 앞 장 4절에서 본 것처럼 SEED에서는 키 스케쥴링 시 비밀 키를 32비트 단위로 나누어 A, B, C, D라 하고 라운드를 거치면서 이 값들을 쉬프트하고 라운드 상수 KC_i와 모듈러 덧셈 및 뺄셈을 한 후 G 함수를 적용하여 라운드 키를 생성한다. i 라운드에서의 A, B, C, D를 각각 A_i, B_i, C_i, D_i라 하면 다음 식이 성립한다.

$$K_{i,0} = G(A_i + C_i - KC_i)$$
$$K_{i,1} = G(B_i - D_i + KC_i) \quad (2)$$

라운드 키 $K_{i,0}$와 $K_{i,1}$들로부터 비밀 키를 구하기 위해서는 G 함수의 역함수를 구할 수 있어야 하고 $A_i + C_i$와 $B_i - D_i$로부터 비밀 키 A, B, C, D를 구할 수 있어야 한다.

G 함수의 역함수는 G 함수를 전단사함수로 볼 수 있어서 함수 값으로부터 함수 입력을 유일하게 결정할 수 있다. 전수검사를 펜티엄3 PC에서 C 프로그램으로 확인한 결과, 1분정도면 G 함수의 출력 값으로부터 입력 값을 찾을 수 있었다.

다음으로 $A_i + C_i - KC_i$ $(=G^{-1}(K_{i,0}))$와 $B_i - D_i + KC_i$ $(=G^{-1}(K_{i,1}))$로부터 암호화 키 A, B, C, D를 구하는 방법에 대해 논의한다. 각 라운드 키에 G^{-1}를 적용하고 라운드 상수를 더하거나 빼면 $A_i + C_i = T_{i,0}$와 $B_i - D_i = T_{i,1}$의 값을 알 수 있다. SEED의 키 스케쥴에 따르면 다음이 성립한다.

$$A_1 + C_1 = T_{1,0} = A + C$$
$$B_1 - D_1 = T_{1,1} = B - D$$
$$A_9 + C_9 = T_{9,0} = B + D \quad (3)$$
$$B_9 - D_9 = T_{9,1} = A - C$$

이 식으로부터 다음과 같이 A'을 계산할 수 있다.

$$2A = T_{1,0} + T_{9,1}$$
$$A0 = 2A \gg 1 \quad (4)$$

A'은 최상위 비트를 뺀 나머지 비트가 A와 같아진다. A'의 최상의 비트는 무조건 0이 되고 A의 최상위 비트는 아직 정해지지 않았다. D'은 다음 식으로부터 구할 수 있다.

$$2D = T_{9,0} - T_{1,1}$$
$$D0 = 2D \gg 1 \quad (5)$$

D'은 최상위 비트를 뺀 나머지 비트가 D와 같아진다. 식 (3)과 A' 및 D'으로부터 B'과 C'을 다음과 같이 구할 수 있다.

$$B0 = T_{1,1} + D0$$
$$C0 = T_{1,0} - A0 \quad (6)$$

D'이 D에서 최상위 비트만 정해지지 않은 값이므로 B'은 B와 최상위 비트만 다를 수 있다. 마찬가지로 C'은 C와 최상위 비트만 다를 수 있다. A'과 D'의 최상위 비트가 맞는지는 $T_{2,0}$와 $T_{2,1}$를 이용하여 알 수 있다. $T_{2,0} = A_2 + C_2$ 이고 $T_{2,1} = B_2 - D_2$이며 $A_2 || B_2 \leftarrow (A||B) >> 8$ 이다. A를 비트 단위로 나누어 $A^{31}A^{30}...A^0$로 나타내고 B도 같은 방법으로 $B^{31}B^{30}...B^0$로 나타내면 $A_2 = B^7...B^0A^{31}...A^8$이고 $B_2 = A^7...A^0B^{31}...B^8$이다. C_2와 D_2는 C와 D 그대로이다. $T_{2,0} = A_2 + C_2$ 를 이와 같은 표기를 이용하여 나타내면 다음과 같다.

$$B^7...B^0A^{31}...A^8 + C^{31}.....C^0 \quad (7)$$
$$T_{2,0} \quad !$$

식 (7)에서 아직 정해지지 않은 값은 A^{31}과 C^{31}이며 C^{31}은 A^{31}에서 얻은 값이다. A^{31}이 맞다면 C^{31}도 맞게 되고 위 등식은 성립한다. 한편 A^{31}이 틀리면 C^{31}도 틀리고 위 식은 성립하지 않게 된다. 위 식이 성립하면 A^{31}와 C^{31}을 그대로 두고 성립하지 않으면 A^{31}와 C^{31}을 반전하여 최종 A와 C의 값을 구할 수 있다. B와 D의 최종 값도 같은 방법으로 $T_{2,1}$을 이용하여 구할 수 있다.

위의 방법을 컴퓨터 프로그램으로 시뮬레이션한 결과 펜티엄3 컴퓨터에서 4~6분의 시간으로 라운드 키들에서 비밀 키를 정확히 찾아낼 수 있었다.

제안된 방법에서는 첫 번째 라운드 키와 아홉 번째 라운드 키를 알아야 한다. 두 번째 라운드 키는 몰라도 비밀 키를 구할 수 있다. A, B, C, D의 최상위 비트를 찾기 위해 A와 C의 최상위 비

트가 바뀌지 않은 상태와 바뀐 상태 그리고 *B*와 *D*의 최상위 비트가 바뀌지 않은 상태와 바뀐 상태 총 네 가지 경우에 대해 평문을 암호화하고 원래의 암호문이 나오는지 확인하면 맞는 조합을 찾을 수 있다. 실제로 필요한 암호화 계산은 세 번이다. 세 가지 조합이 맞지 않는 경우 네 번째 조합이 맞는 것으로 볼 수 있을 것이다. 두번째 라운드 키도 아는 경우에는 앞서 설명한 방법을 이용하여 암호화를 하지 않고도 비밀 키를 유일하게 결정할 수 있다.

아홉 번째 라운드 키를 찾기 위해서는 제안된 BPA를 이용하여 세 번째부터 여덟 번째 라운드 키도 모두 찾아야 한다. 하지만 광학적 공격에 의해 오류 공격에서는 앞의 라운드 키를 몰라도 해당 라운드 키를 바로 찾을 수 있다[15]. 라운드 키의 한 비트를 0으로 만들고 암호화를 한 후 생성된 암호문을 정상 암호문과 비교하여 같으면 해당 라운드 키 비트가 0이고 다르면 1로 결정할 수 있을 것이다. 이러한 상황에서는 제안된 비밀 키 복구 방법이 효율적으로 사용될 수 있을 것이다.

IV. 대응 방안

대응 방안으로 더미 연산을 중간에 임의로 수행하는 방법, 임의로 변하는 내부 클럭을 사용하는 방법, 연산 순서를 임의로 하는 방법, 전원에 노이즈를 싣는 방법 등을 생각할 수 있다. 하지만 이러한 방법은 공격자가 더 많은 시간을 소비하도록 할 수는 있지만 패턴 인식, 동기화 등의 처리로 무력화될 수 있다고 한다[4, 14].

여기서는 AES에 사용된 마스크 방법을 사용해 본다. 마스크 방법은 처리되는 데이터나 키에 임의의 값을 논리적으로 xor하거나 산술적으로 더하여 공격자가 중간 계산 결과를 알 수 없도록 하는 방법이다. AES와 달리 SEED에서는 모듈러 덧셈과 뺄셈이 사용되므로 논리적 마스크와 산술 마스크 간에 변환을 해 주어야 한다. 두 마스크 간 변환은 Messerges에 의해 처음 제안되었다[3]. 이후에 Coron *et al.*은 Messerges 방법의 문제점을 지적하였다[11]. Goubin *et al.*에 의해 보완된 방법이 제안되었고 Coron *et al.*은 더 효율적인 방법을 제안하였다[12, 13].

한 라운드의 입력 마스크와 출력 마스크가 다르면, 암호화가 진행되면서 라운드마다 마스크가 변하게 된다. 마스크가 변하면 변화된 마스크마다 재 계산된 S-box를 계산해서 혹은 저장했다가 사용해야 한다. S-box 재 계산은 입력 마스크가 M_{in}이고 출력 마스크가 M_{out}이며 x에 대한 S-box 출력을 S(x)라고 할 때 출력이 $S(x \oplus M_{in}) \oplus M_{out}$로 되도록 하면 된다. 이렇게 재 계산된 S-box는 ROM이나 EEPROM에 저장해야 하며 스마트카드 메모리의 한계로 많은 수를 저장할 수는 없다. 입출력 마스크가 다른 경우에는 16라운드에 대한 마스크와 해당 재 계산된 S-box가 저장되어야 하므로 메모리 사용량이 많아진다. 그리고 이러한 제약으로 각 암호화에 사용할 수 있는 각 라운드별 마스크 조합이 많을 수 없다. 그러나 입력 마스크와 출력 마스크가 같으면 라운드 단위로 저장된 마스크 중 하나를 선택하여 사용할 수 있게 되어 만들 수 있는 마스크의 조합이 많아진다. 마스크의 조합이 많아지면 higher-order DPA에 강해진다. Higher-order DPA는 알고리즘 내부 구현을 알아야 적용이 가능한 것으로 알려져 있으므로 대부분의 공격자는 적용이 어렵다. 이런 이유로 저가 스마트카드에서는 higher-order DPA의 위협을 감수할 수 있을 것이며 이러한 경우에 입출력 마스크가 같은 방식을 사용하면 한 라운드에서 사용한 마스크를 모든 라운드에서 반복 사용할 수 있고 마스크를 위한 메모리 사용량이 많이 줄어든다.

각 라운드의 입력 마스크와 출력 마스크가 같도록 하기 위해서는 평문 중 F 함수로 입력되는 64비트에만 마스크를 적용하고 이 마스크가 F 함수 출력에서 그대로 나타나도록 한다. 이 때, C와 D에 대한 마스크가 같은 경우에는 두 값이 xor되면서 마스크가 없어지는 시점이 생기므로 두 부분에 다른 마스크 M_1, M_2를 각각 적용한다. G 함수 내부에서는 바이트 단위로 처리되므로 M_1, M_2도 바이트 단위로 같은 값을 갖도록 한다 ($M_1=m_1m_1m_1m_1$, $M_2=m_2m_2m_2m_2$). G 함수 내부의 모든 S-box의 출력 마스크가 같은 경우에는 G 함수 내부의 xor 연산 후 마스크가 없는 시점이 생기므로 S_1에 해당하는 재 계산된 S-box를 S_1'과 S_1"와 같이 두 가지 출력 마스크를 갖도록 준비 한다. 두 출력 마스크의 xor는 M_2의 한 바이트 값 m_2와 같도록 한다. S_2의 출력 마스크는 또 다른 값 m_3로 한다. 이와 같이 하면 G 함수의 출력 마스크는 M_2가 된다. 이러한 점을 고려한 F 함수에서의 마스크 적용은 그림 3과 같다.

G함수를 처리하기 전과 F 함수의 출력 전에 마스크는 논리적 마스크로 변경되어야 하고 산술적 연산 전에는 마스크가 산술 마스크로 변경되어야 한다. G함수에 들어가는 데이터는 논리적 마스크 M_1이 적용되고 나오는 데이터는 논리적 마스크 M_2가 적용되어 있다. *C*와 *D*에는 시작 시 각각 마스크 M_1과 M_2가 적용되고 F 함수 출력 *C*'과 *D*'에는 각각 같은 마스크 M_1, M_2가 적용되게 된다. 마

지막 라운드에서는 출력 마스크를 다시 xor하여 마스크가 없는 데이터가 출력되도록 한다. 라운드 사이에서 마스크를 변경하고자 할 때는 변경 전 마스크와 변경 후 마스크를 xor 한 후 *C*와 *D*에 적용한다.

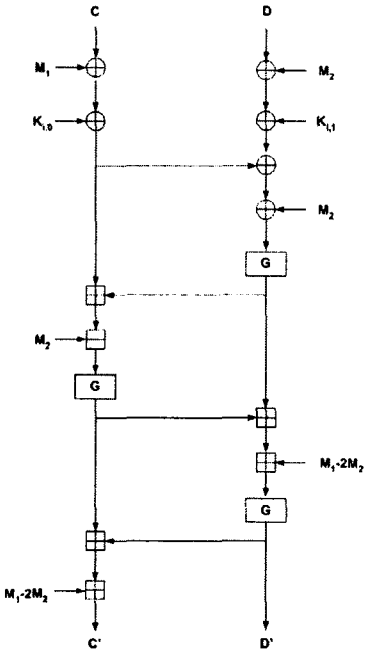

그림 3: 마스크가 적용된 F 함수

V. 결 론

본 논문에서는 국내 블록 암호 표준인 SEED에도 DPA가 적용될 수 있음을 확인하였다. Akkar *et al.*이 실험으로 유도한 전력 소모 모델과 공격법을 사용하였으므로 제안한 공격은 실제로 가능한 것으로 볼 수 있을 것이다. 다음으로는 이러한 공격을 막기 위한 방법으로 모듈러 연산을 고려한 고정 마스크 방법을 제안하였다. 마스크에 의해 모든 중간 값이 바뀌므로 DPA는 적용할 수 없게 될 것이다. 또 EEPROM이나 ROM에 미리 계산된 마스크를 저장한 후 임의로 선택하여 사용하면 S-box 재 계산을 위한 RAM 사용과 계산 시간이 필요 없을 것이다. 제안한 방법에서는 F 함수의 입력 마스크가 출력에 유지되도록 하여 저가 스마트카드에서 마스크를 위한 메모리 사용량이 적도록 했으며 higher-order DPA를 고려하는 경우에도 정해진 메모리로 많은 마스크 조합을 사용할 수 있도록 하였다.

향후 작업으로는 실제 스마트카드의 SEED에 제안한 방법을 적용해 보고, 제안한 대응방안을 적용한 후 그 효과를 확인해 보고자 한다. 또 대응 방안을 최적화하고 안전성을 정밀하게 분석하려고 한다.

참고문헌

[1] 한국정보통신기술협회 (TTA), "128비트 블록 암호알고리즘 표준", 정보통신단체표준 TTA.KO-12.0004, 1999년 9월.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis", Advances in Cryptology - Crypto 1999, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.

[3] T. Messerges, "Securing the AES Finalists Against Power Analysis Attacks", FSE 2000, LNCS 1978, pp. 150-164, Springer-Verlag, 2001.

[4] C. Clavier, J. Coron, and N. Dabbous, "Differential Power Analysis in the Presence of Hardware Countermeasures", CHES 2000, LNCS 1965, pp. 252-263, Springer-Verlag, 2000.

[5] M. Akkar and C. Giraud, "An Implementation of DES and AES, Secure against Some Attacks", CHES 2001, LNCS 2162, pp. 309-318, Springer-Verlag, 2001.

[6] T. Messerges, "Using Second-Order Power Analysis to Attack DPA Resistant Software", CHES 2000, LNCS 1965, pp. 238-251, Springer-Verlag, 2000.

[7] S. Yen, "Amplified Differential Power Cryptanalysis on Rijdael Implementations with Exponentially Fewer Power Traces", ACISP 2003, LNCS 2727, pp. 106-117, Springer-Verlag, 2003.

[8] R. Bevan and E. Knudsen, "Ways to Enhance Differential Power Analysis", ICISC 2002, LNCS 2578, pp. 327-342, Springer-Verlag, 2003.

[9] J. Golic and C. Tymen, "Multiplicative Masking and Power Analysis of AES", CHES 2002, LNCS 2523, pp. 198-212, Springer-Verlag, 2003.

[10] K. Itoh, M. Takenaka, and N. Torii, "DPA Countermeasure Based on the Masking Method", ICICS 2001, LNCS 2288, pp. 440-456, Springer-Verlag, 2001.

[11] J. Coron and L. Goubin, "On Boolean and

Arithmetic Masking against Differential Power Analysis", CHES 2000, LNCS 1965, pp. 231-237, Springer-Verlag, 2000.

[12] L. Goubin, "A Sound Method for Switching between Boolean and Arithmetic Masking", CHES 2001, LNCS 2162, pp. 3-15, Springer-Verlag, 2001.

[13] J. Coron and A. Tchulkine, "A New Algorithm for Switching from Arithmetic to Boolean Masking", CHES 2003, LNCS 2779, pp. 89-97, Springer-Verlag, 2003.

[14] M. Akkar, R. Bevan, P. Dischamp, and D. Moyart, "Power Analysis, What Is Now Possible...", ASIACRYPT 2000, LNCS 1976, pp. 489-502, Springer-Verlag, 2000.

[15] S. Skorobogatov and R. Anderson, "Optical Fault Induction Attacks", CHES 2002, LNCS 2523, pp. 2-12, Springer-Verlag, 2003.