Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.26, NO.6, Dec. 2016 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2016.26.6.1443

부채널 분석에 안전한 밸런스 인코딩 기법에 관한 연구*

윤 진 영,† 김 한 빛, 김 희 석,‡ 홍 석 희
고려대학교

Study for Balanced Encoding Method against Side Channel Analysis*

JinYeong Yoon,† HanBit Kim, HeeSeok Kim,‡ SeokHie Hong
Korea University

요 약

하드웨어 기반의 Dual-rail Logic 스타일을 소프트웨어로 구현한 밸런스 인코딩 기법은 추가적인 저장 공간이 필요 없는 효과적인 부채널 분석 대응방법이다. 밸런스 인코딩 기법을 이용하여 암호 알고리즘을 구현하면 암호 알고리즘이 연산되는 동안 입력 값에 상관없이 비밀 정보를 포함하고 있는 중간 값은 항상 일정한 해밍 웨이트 및 해밍 디스턴스를 유지하게 되어 부채널 분석을 어렵게 만드는 효과가 있다. 그러나 기존 연구에서는 밸런스 인코딩 기법을 적용한 Constant XOR 연산만 제안되어 있어 PRINCE와 같이 XOR 연산만으로 구성이 가능한 암호 알고리즘에만 적용이 가능하다는 제한사항이 있다. 따라서 본 논문에서는 ARX 구조 기반의 다양한 대칭키 암호 알고리즘에도 적용이 가능하고, 효율적인 메모리 관리를 위해 Look-up table을 사용하지 않는 새로운 Constant AND, Constant Shift 연산 알고리즘을 최초로 제안하였으며, 상호 정보량 분석을 통해 안전성을 확인하였다.

ABSTRACT

Balanced encoding method that implement Dual-rail logic style based on hardware technique to software is efficient countermeasure against side-channel analysis without additional memory. Since balanced encoding keep Hamming weight and/or Hamming distance of intermediate values constantly, using this method can be effective as countermeasure against side channel analysis due to elimination of intermediate values having HW and/or HD relating to secret key. However, former studies were presented for Constant XOR operation, which can only be applied to crypto algorithm that can be constructed XOR operation, such as PRINCE. Therefore, our first proposal of new Constant ADD, Shift operations can be applied to various symmetric crypto algorithms based on ARX. Moreover, we did not used look-up table to obtain efficiency in memory usage. Also, we confirmed security of proposed Constant operations with Mutual Information Analysis.

Keywords: Balanced Encoding, Dual-rail with Precharge Logic, Side Channel Analysis, Constant operations

I. 서 론

초소형 무인항공기(드론) 개발 기술이 발달함에 따라 이를 군사적으로 이용하여 감시에 적발되지 않으면서 상대국의 중요 영상정보를 획득하기 위한 세계 각국의 노력이 계속되고 있다. 특히 2016년 미군이 언론에 공개한 정찰용 초소형 무인항공기 '블랙호넷(PD-100)'[1]의 경우, 크기는 가로 20.23cm・세로 8.89cm・높이 5.08cm이며, 무게는 18.25g에 불과할 정도로 매우 작아 공중에서 정찰 임무를 수행 시 육안으로 이를 식별하기란 쉽지 않다. 이러한 초소형 무인항공기를 운용하기 위해서는 내부에 다양한 정보통신기술들이 적용되는데, 그 중 본부와

Received(09. 30. 2016), Modified(10. 17. 2016), Accepted(10. 20. 2016)
* 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 대학 ICT연구센터육성 지원사업의 연구결과로 수행되었음. (IITP-2016-R0992-16-1017)
† 주저자, yjiny0406@gmail.com
‡ 교신저자, 80khs@korea.ac.kr(Corresponding author)

송·수신하는 무인항공기 운항 통제 신호와 영상 정보를 보호하기 위한 암호기술은 가장 중요한 핵심 요소라 할 수 있다.

위와 같이 중요 정보를 보호하기 위해 암호기술을 적용하는 방법은 암호 알고리즘을 소프트웨어로 구현하여 운영 프로그램에 적용하는 방법, 그리고 이를 하드웨어로 구현하여 디바이스 내부에 별도로 장착하는 방법이 있다. 하지만 이 같은 초소형 무인항공기의 경우, 협소한 내부 공간으로 인해 하드웨어 기반의 암호화 방법 적용이 제한될 가능성이 있다. 이러한 경우에는 소프트웨어 기반 암호화 방법을 적용하면 추가적인 디바이스 설치 없이 각종 중요 정보를 보호한다는 본래 목적을 달성할 수 있다. 그러나 암호학적으로 안전한 암호 알고리즘을 이용하여 중요 정보를 암호화 하더라도 실제 운용 과정에서 발생하는 부가적인 정보(전력, 전자기파 등)들로 인해 비밀정보가 노출될 수 있음이 이미 알려져 있다[2].

1996년 Paul Kocher가 [2]에서 이러한 부채널 분석에 대한 개념을 제시한 이후, 부채널 분석을 방어하기 위한 대응기법(countermeasure)에 대한 연구가 활발히 진행되고 있으며, 대표적인 부채널 분석 대응기법으로는 마스킹(Masking)[3]과 하이딩(Hiding) [4]이 있다. 마스킹은 암호 알고리즘이 연산되는 동안 비밀정보를 포함하고 있는 중간 값에 랜덤 값을 이용하여 공격자가 중간 값을 예측할 수 없도록 하는 방법이며, 하이딩은 디바이스 내에서 암호 알고리즘이 연산되면서 발생하는 leakage(전력, 전자기파 등)가 비밀정보를 포함하고 있는 데이터와 관련성이 없는 것처럼 보이도록 만드는 방법이다.

이와 같은 부채널 대응기법으로 사용되고 있는 예로 Dual-Rail with Precharge Logic(이하 DPL)이 있다[4]. DPL을 사용하여 암호 알고리즘을 구현하면 데이터가 연산되는 과정에서 데이터의 해밍 웨이트와 해밍 디스턴스가 항상 일정하여 부채널 분석 위협을 완화시킬 수 있다.

Hoogvorst 등은 2011년에 DPL의 하드웨어 기반 기술을 소프트웨어로 구현하는 방안을 제시하였는데[5], 저자들은 아이디어의 안전성을 뒷받침하는 실험 결과를 제시하지 않았으며, 오직 일정한 해밍 웨이트만 고려하였다. 이후, 2014년 Chen 등은 [6]에서 Dual-rail 기법을 모티브로 한 '밸런스 인코딩 기법'과 이를 이용한 Constant XOR 연산 알고리즘을 제안하였는데, 이를 암호 알고리즘에 적용하면 연산 간 모든 중간 값의 해밍 웨이트 뿐만 아니라 해밍 디스턴스까지 항상 일정한 값을 유지하도록 하여 부채널 분석 위협을 완화할 수 있다. 그리고 이를 64비트 경량 대칭키 암호 알고리즘인 PRINCE에 적용하여 상호정보량 분석을 통해 안전성도 확인하였다. 하지만 Chen 등은 오직 Constant XOR 연산 알고리즘만 제안하여 ARX 기반의 다른 대칭키 암호 알고리즘에 밸런스 인코딩 기법을 적용할 수 없다는 제한사항을 남겼다.

본 논문에서는 HIGHT[7], LEA[8], SIMON[9] 등과 같은 ARX 기반 경량 대칭키 암호 알고리즘에 밸런스 인코딩 기법을 적용하여 메모리 사용을 절약함과 동시에 부채널 분석에 대한 위협을 완화시킬 수 있는 Constant AND와 Constant Shift 연산을 최초로 제안한다. 그리고 이를 Chen 등이 제안한 Constant XOR 연산과 함께 Coron 등이 [10]에서 하드웨어 기반의 Kogge-Stone Carry Look -Ahead Adder를 AND, Shift, XOR 등 Boolean 연산만으로 구성할 수 있도록 제안한 알고리즘에 적용하여 Constant ADD 연산도 구현하였다. 본 논문에서 제안한 Constant 연산 알고리즘을 이용하여 암호 알고리즘을 구현하면 입력 값에 상관없이 비밀 정보를 포함하고 있는 중간 값은 항상 일정한 해밍 웨이트와 해밍 디스턴스를 유지하며, Look-up table을 사용하지 않기 때문에 메모리 또한 절약할 수 있다. 물론, 본 논문에서 제안하는 밸런스 인코딩 기법이 적용된 Constant 연산 알고리즘을 사용하여 암호 알고리즘을 구현하더라도 차분전력분석공격[3]이나 bit-wise CPA[11] 등을 비롯한 다양한 부채널 공격을 완벽하게 방어할 수는 없다. 하지만 공격자로 하여금 더 많은 부채널 정보 수집을 필요로 한다거나, 공격하는데 있어 더 많은 시간을 소모하게 하는 등 시간 복잡도를 상승시킴으로써 부채널 분석을 더 어렵게 만드는 효과가 있다.

본 논문의 구성은 다음과 같다. 2장에서 DPL과 밸런스 인코딩에 대해 설명한 후, 3장에서는 밸런스 인코딩 기법을 적용한 새로운 Constant 연산 알고리즘을 제안한다. 그리고 4장에서는 시뮬레이션을 통한 안전성 검증 결과를 제시하고, 5장에서 제안하는 Constant 연산 알고리즘의 성능을 비교 후 결론을 맺는다.

II. 관련 연구

2.1 Dual-Rail with Precharge Logic(DPL)

DPL은 하드웨어 기반의 부채널 대응기법으로 주로 사용되는데, 1비트 데이터를 처리하는데 2개의 wire를 사용하는 dual-rail logic과 circuit의 모든 신호를 0 또는 1로 만드는 precharge logic이 결합되어 있는 구조이다[4, 5, 6, 11]. DPL을 사용하는 목적은 게이트 레벨에서 항상 일정한 액티비티를 달성함으로써 연산되는 데이터 값에 상관없이 항상 일정한 leakage가 발생하도록 하는 것이다. dual-rail logic 형태의 회로에서는 logic bit A를 생성하는 모든 게이트는 logic bit A의 보수인 $\overline{A}$를 수반하는데, logic bit 쌍 $(A, \overline{A})$가 logic bit A를 나타내며, logic bit 쌍의 해밍 웨이트는 항상 1로 일정하다. 또한, precharge logic에서는 연속되는 데이터 간 항상 일정한 해밍 디스턴스를 유지하기 위해 logic bit 쌍을 evaluation 하기 전에 (0, 0)으로 미리 충전하기 때문에 이후 게이트 값이 (0, 0)에서 (1, 0) 또는 (0, 1)로 변하면서 발생하는 leakage는 일정한 해밍 디스턴스를 유지한다.

2.2 밸런스 인코딩(Balanced Encoding)

Chen 등이 2014년 CARDIS에서 발표한 '밸런스 인코딩' 기법이란 암호 알고리즘이 연산되는 동안 처리되는 데이터에 dual-rail logic style을 적용한 것을 말하는데, 하나의 비트 A를 보수 $\overline{A}$와 함께 $(A, \overline{A})$로 인코딩 하는 방식이다. [6]에서는 이를 경량 대칭키 암호 알고리즘 PRINCE에 적용하였으며, 타겟 플랫폼이 8비트 프로세서인 관계로 데이터를 nibble 단위로 인코딩 하였다. 특히 [6]에서는 암호 알고리즘이 연산되는 동안 중간 값이 항상 일정한 해밍 웨이트와 해밍 디스턴스를 유지하기 위해 연산별 상이한 인코딩 형태를 적용하였다. KeyAddition과 Mixcolumns 연산에서는 nibble을 encoding I ($\overline{b_3}b_3\overline{b_2}b_2\overline{b_1}b_1\overline{b_0}b_0$) 형태로 인코딩 하였으나, S-Box 연산에서는 데이터를 encoding I 형태로 인코딩 시 일정한 해밍 디스턴스 유지가 제한되어 encoding II($b_0\overline{b_2}b_1b_3\overline{b_1}b_2\overline{b_0}\overline{b_3}$) 형태로 인코딩하였다. 특히 Chen 등은 MixColumns 부분 연산 시 적용할 수 있는 새로운 Constant XOR 연산을 8-bit Atmel AVR Assembler로 작성하였으며, 구체적인 알고리즘은 [6]을 참고하기 바란다.

III. 제안하는 Constant 연산 알고리즘

본 장에서는 밸런스 인코딩 기법을 적용한 새로운 Constant AND, Shift 연산 알고리즘을 제안한다. 알고리즘은 8비트 레지스터를 사용하는 디바이스를 대상으로 Atmel AVR 8-bit instruction set을 참고하여 작성하였으며, 주요 명령어에 대한 기능은 [17]을 참고하면 된다. 각 알고리즘의 INPUT 값은 nibble에 $\overline{b_3}b_3\overline{b_2}b_2\overline{b_1}b_1\overline{b_0}b_0$ 형태의 밸런스 인코딩을 적용

Algorithm 1 ConstAND

STEP 1 ⟨Generate x_L, x_R, y_L, y_R⟩

Input: r4 = x ($x : \overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$)

Output: r5 = x_L ($x_L : \overline{x_3}x_3\overline{x_2}x_2\overline{x_3}x_3\overline{x_2}x_2$)

```
1  CLR   r5
2  CLR   r6
3  LDi   r5, b 11110000
4  LDi   r6, b 11110000
5  AND   r5, r4
6  AND   r6, r4
7  SWAP  r6
8  OR    r5, r6
```

STEP 2 ⟨Generate z_L, z_R⟩

Input: r4 = x_L ($\overline{x_3}x_3\overline{x_2}x_2\overline{x_3}x_3\overline{x_2}x_2$),
r5 = y_L ($\overline{y_3}y_3\overline{y_2}y_2\overline{y_3}y_3\overline{y_2}y_2$)

Output: r6 = z_L ($0000\overline{z_3}z_3\overline{z_2}z_2$)

```
1  CLR   r6
2  LDi   r6, b 11110000
3  EOR   r6, r4
4  CLR   r7
5  MOV   r7, r6
6  AND   r6, r5
7  EOR   r7, r5
8  ANDi  r7, b 10101010
9  EOR   r6, r7
10 ANDi  r6, b 00001111
```

STEP 3 ⟨Construct z⟩

Input: r4 = z_L ($0000\overline{z_3}z_3\overline{z_2}z_2$),
r5 = z_R ($0000\overline{z_1}z_1\overline{z_0}z_0$)

Output: r4 = z ($\overline{z_3}z_3\overline{z_2}z_2\overline{z_1}z_1\overline{z_0}z_0$)

```
1  SWAP  r4
2  OR    r4, r5
```

하였다. 각 알고리즘에서는 임의 값이 레지스터에 저장되는 것을 방지하기 위해 레지스터를 사용하기 전 'CLR' 명령어로 초기화 하였으며, 중간 값의 해밍 웨이트 및 해밍 디스턴스가 입력 값과 무관하게 일정하도록 설계하였다. 알고리즘 안전성에 대한 자세한 분석은 4장에서 설명한다.

3.1 Constant AND

Constant AND 연산 알고리즘에서는 입력 값을 $x : \overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$, $y : \overline{y_3}y_3\overline{y_2}y_2\overline{y_1}y_1\overline{y_0}y_0$ 라 하고, 이를 AND 연산한 결과 값을 $z : \overline{z_3}z_3\overline{z_2}z_2\overline{z_1}z_1\overline{z_0}z_0$로 정의하며, 다음과 같은 3단계로 구성되어 있다.

STEP 1, 3은 [6]에서 제안된 Constant XOR 연산 알고리즘의 아이디어를 모티브로 설계하였는데, STEP 1에서는 입력 값 x를 절반으로 나눈 후 'AND', 'SWAP' 명령어 등을 사용하여 $x_L : \overline{x_3}x_3\overline{x_2}x_2\overline{x_3}x_3\overline{x_2}x_2$를 생성하며, 유사한 방법을 통해 $x_R : \overline{x_1}x_1\overline{x_0}x_0\overline{x_1}x_1\overline{x_0}x_0$와 y_L, y_R도 생성한다. STEP 2는 이전 단계에서 생성한 x_L, y_L를 이용하여 z_L 생성하고, 동일 방법으로 x_R, y_R를 이용하여 z_R를 생성하는 단계로 본 논문에서 새롭게 제안하였다. 특히, 6~9번째 줄은 중간 값의 일정한 해밍 웨이트와 해밍 디스턴스를 위해 다소 복잡한 과정을 거치게 되는데 이에 대한 설명은 4장에서 실시한다. 그리고 마지막으로 STEP 3은 입력 값 z_L, z_R를 이용하여 $x \wedge y$ 연산의 결과 값인 z를 최종적으로 구성하는 단계로서, SWAP과 OR 명령어를 사용하면 된다.

3.2 Constant Shift

Constant Shift 연산은 Shift left 1~4까지 총 4가지 알고리즘을 제안하였다. Algorithm 2 (ConstShift 1)와 Algorithm 4(ConstShift 3)의 core idea는 'MUL' 명령어 사용이다. MUL 명령어는 두 개의 레지스터에 저장되어 있는 값을 곱한 후 그 결과 값을 r0, r1에 저장하는데, Algorithm 2와 Algorithm 4에서는 입력 값 r4 : $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$ 와 r5 : $\overline{x_7}x_7\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$가 어떤 값과 곱해지느냐에 따라 r0과 r1에 저장되는 값들이 얼마나 왼쪽으로 shift 되는지 결정된다. 이에, Algorithm 2에서는 r4와 r5의 값들을 왼쪽으로 두 칸 shift 하기 위해 4번째 줄에서 r6에 '00000100' 이라는 값을 저장하여 r4, r5에 곱해주었으며, Algorithm 4에서는 r6에 '01000000' 이라는 값을 저장하여 r4와 곱한 후 ANDi, SWAP, LSL, OR 명령어 등을 이용하여 r4와 r5의 값들을 왼쪽으로 여섯 칸 shift 하였다.

Algorithm 3(ConstShift 2)과 Algorithm 5 (ConstShift 4)는 AND와 SWAP, MOV 명령어 등을 이용하여 입력 값을 왼쪽으로 각각 네 칸, 여덟 칸

Algorithm 2 ConstShift 1

Input: r4 = $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$,
r5 = $\overline{x_7}x_7\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$

Output: r4 = $\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_010$,
r5 = $\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4\overline{x_3}x_3$

```
1  CLR   r0
2  CLR   r1
3  CLR   r6
4  LDi   r6, b 00000100
5  MUL   r4, r6
6  ORi   r0, b 00000010
7  CLR   r4
8  MOV   r4, r0
9  CLR   r7
10 MOV   r7, r1
11 CLR   r0
12 CLR   r1
13 MUL   r5, r6
14 OR    r0, r7
15 CLR   r5
16 MOV   r5, r0
```

Algorithm 3 ConstShift 2

Input: r4 = $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$,
r5 = $\overline{x_7}x_7\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$

Output: r4 = $\overline{x_1}x_1\overline{x_0}x_01010$,
r5 = $\overline{x_5}x_5\overline{x_4}x_4\overline{x_3}x_3\overline{x_2}x_2$

```
1  CLR   r6
2  LDi   r6, b 11110000
3  AND   r6, r4
4  SWAP  r6
5  ANDi  r4, b 00001111
6  SWAP  r4
7  ORi   r4, b 00001010
8  ANDi  r5, b 00001111
9  SWAP  r5
10 OR    r5, r6
```

shift 하였다. 그리고 왼쪽으로 shift 후 '0'으로 채워지는 오른쪽 부분도 '10'과 같은 밸런스 인코딩 형태를 유지하도록 하였는데, 이를 통해 오류 주입 공격(Fault Injection Attack)[12, 13]에 의해 '00', '11'과 같이 비트 플립이 발생하여 동일한 비트 값이 연속되는 경우도 식별할 수 있다.

3.3 Constant ADD

1973년 Kogge와 Stone에 의해 제안된 Kogge-Stone Carry Look-Ahead Adder(이하 Kogge-Stone Adder)는 입력 값 x, y에 대해 'carry(올림 수)가 발생할 가능성이 있는 경우'와 'carry가 반드시 발생하는 경우'에 대해 값을 미리 생성한 후, 그 값을 이용하여 4비트를 동시에 병렬(parallel) 처리함으로써 효율적인 연산이 가능하도록 고안된 하드웨어 기반 알고리즘이다[14]. Coron 등은 이러한 Kogge-Stone Adder를 [10]에서 AND, Shift, XOR와 같은 Boolean 연산만으로 구현할 수 있도록 Algorithm 6을 제안하였다. 여기에 [6]에서 제안된 Constant XOR 연산과 본 논문에서 새롭게 제안한 Constant AND, Shift 연산을 적용하면 Constant ADD 연산으로 구현이 가능하다.

Algorithm 4 ConstShift 3

Input: r4 = $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$,
r5 = $\overline{x_7}x_7\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$

Output: r4 = $\overline{x_0}x_0101010$,
r5 = $\overline{x_4}x_4\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1$

```
1  CLR   r0
2  CLR   r1
3  CLR   r6
4  LDi   r6, b 01000000
5  MUL   r4, r6
6  ANDi  r5, b 00000011
7  SWAP  r5
8  LSL   r5, 2
9  OR    r5, r1
10 CLR   r4
11 MOV   r4, r0
12 ORi   r4, b 00101010
```

Algorithm 5 ConstShift 4

Input: r4 = $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$,
r5 = $\overline{x_7}x_7\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$

Output: r4 = 10101010,
r5 = $\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$

```
1  CLR   r5
2  MOV   r5, r4
3  CLR   r4
4  ORi   r4, b 10101010
```

Algorithm 6 Kogge-Stone Adder

Input: $x, y \in 0,1^k$,
and $n = \max(\lceil \log_2(k-1) \rceil, 1)$

Output: $z = x + y \bmod 2^k$

```
1  P ← x ⊕ y
2  G ← x ∧ y
3  for i := 1 to n−1 do
4      G ← (P ∧ (G ≪ 2^(i−1))) ⊕ G
5      P ← P ∧ (P ≪ 2^(i−1))
6  end for
7  G ← (P ∧ (G ≪ 2^(n−1))) ⊕ G
8  return x ⊕ y ⊕ (2G)
```

IV. 안전성 검증

본 장에서는 3장에서 제안한 Constant AND, Shift 연산 알고리즘에 대해 중간 값의 해밍 웨이트와 해밍 디스턴스가 입력 값과 무관하게 일정함을 보이고, 상호 정보량 분석(Mutual Information Analysis)에 기반한 시뮬레이션을 통해 안전성을 검증한다.

4.1 Constant AND 중간 값 분석

Constant AND 알고리즘의 STEP 1은 3장에서 언급한 바와 같이 입력 값 x, y에 대해 x_L, x_R, y_L, y_R를 생성하는 단계이며, 각 중간 값에 대한 해밍 웨이트 및 해밍 디스턴스는 [Table 1]과 같이 입력 값에 상관없이 2 또는 4로 항상 일정함을 확인할 수 있다.

STEP 2는 이전 단계에서 생성한 (x_L, y_L)와 (x_R, y_R)를 이용하여 각각 z_L와 z_R를 연산하는 단계로서 각 레지스터의 중간 값과 해밍 웨이트 및 해밍

디스턴스는 〔Table 2〕와 같으며, 〔Table 2〕의 6~9번째 줄의 중간 값을 표현하는데 사용하는 기호들은 다음과 같이 정의한다.

- $\alpha_i = x_i \wedge \overline{y_i}$
- $\beta_i = \overline{x_i} \wedge y_i$
- $\gamma_i = \overline{x_i} \wedge \overline{y_i}$
- $z_i' = x_i \oplus y_i$
- $\square_3 = \alpha_3 \oplus \overline{z'_3}$
- $\square_2 = \alpha_2 \oplus \overline{z'_2}$

알고리즘의 중간 값들은 (x_i, y_i) 값에 의존하며, 〔Table 3〕의 진리표를 참고하여 〔Table 2〕의 중간 값들의 해밍 웨이트와 해밍 디스턴스를 확인해보면 2 또는 4로 항상 일정함을 알 수 있다. 9번째 줄 연산 후 r6에 저장되는 값인 '$\square_3\beta_3\square_2\beta_2\,\overline{z_3}z_3\overline{z_2}z_2$'의 해밍 웨이트 또한 ($x_2$, y_2)와 (x_3, y_3) 값에 의존하는데, 〔Table 4〕와 같이 (x_2, y_2)와 (x_3, y_3) 값에 따라 발생할 수 있는 16가지 모든 경우를 확인하더라도 해밍 웨이트가 4로 일정함을 알 수 있다.

Table 1. (Security of Algorithm 1) HW/HD of Intermediate Values in Step 1

	Reg.	Intermediate value	HW/HD
1	r5	$\cdot\,00000000$	0 / 0
2	r6	$\cdot\,00000000$	0 / 0
3	r5	$\cdot\,11110000$	4 / 4
4	r6	$\cdot\,11110000$	4 / 4
5	r5	$\cdot\,\overline{x_3}x_3\overline{x_2}x_20000$	2 / 2
6	r6	$\cdot\,\overline{x_3}x_3\overline{x_2}x_20000$	2 / 2
7	r6	$\cdot\,0000\overline{x_3}x_3\overline{x_2}x_2$	2 / 4
8	r5	$\cdot\,\overline{x_3}x_3\overline{x_2}x_2\overline{x_3}x_3\overline{x_2}x_2$	4 / 2

Table 2. (Security of Algorithm 1) HW/HD of Intermediate Values in Step 2

	Reg.	Intermediate value	HW/HD
1	r6	$\cdot\,00000000$	0 / 0
2	r6	$\cdot\,11110000$	4 / 4
3	r6	$\cdot\,x_3\overline{x_3}x_2\overline{x_2}\overline{x_3}x_3\overline{x_2}x_2$	4 / 4
4	r7	$\cdot\,00000000$	0 / 0
5	r7	$\cdot\,x_3\overline{x_3}x_2\overline{x_2}\overline{x_3}x_3\overline{x_2}x_2$	4 / 4
6	r6	$\cdot\,\alpha_3\beta_3\alpha_2\beta_2\gamma_3z_3\gamma_2z_2$	2 / 2
7	r7	$\cdot\,\overline{z_3'}\overline{z_3'}\overline{z_2'}\overline{z_2'}z_3'z_3'z_2'z_2'$	4 / 4
8	r7	$\cdot\,\overline{z_3'}0\overline{z_2'}0z_3'0z_2'0$	2 / 4
9	r6	$\cdot\,\square_3\beta_3\square_2\beta_2\,\overline{z_3}z_3\overline{z_2}z_2$	4 / 2
10	r6	$\cdot\,0000\,\overline{z_3}z_3\overline{z_2}z_2$	2 / 2

Table 3. Truth-table

(x, y)	α ($x\wedge\overline{y}$)	β ($\overline{x}\wedge y$)	γ ($\overline{x}\wedge\overline{y}$)	z' ($x\oplus y$)	z ($x\wedge y$)
(1, 1)	0	0	0	0	1
(1, 0)	1	0	0	1	0
(0, 1)	0	1	0	1	0
(0, 0)	0	0	1	0	0

Table 4. HW of Intermediate Values in 9th Line of Table 2

(x_2, y_2, x_3, y_3)	$\alpha_3\oplus\overline{z'_3}$ / $\overline{z_3}$	β_3 / z_3	$\alpha_2\oplus\overline{z'_2}$ / $\overline{z_2}$	β_2 / z_2	HW
(1, 1, 1, 1)	1	0	1	0	4
	0	1	0	1	
(1, 1, 1, 0)	1	0	1	0	4
	1	0	0	1	
(1, 1, 0, 1)	0	1	1	0	4
	1	0	0	1	
(1, 1, 0, 0)	1	0	1	0	4
	1	0	0	1	
(1, 0, 1, 1)	1	0	1	0	4
	0	1	1	0	
(1, 0, 1, 0)	1	0	1	0	4
	1	0	1	0	
(1, 0, 0, 1)	0	1	1	0	4
	1	0	1	0	
(1, 0, 0, 0)	1	0	1	0	4
	1	0	1	0	
(0, 1, 1, 1)	1	0	0	1	4
	0	1	1	0	
(0, 1, 1, 0)	1	0	0	1	4
	1	0	1	0	
(0, 1, 0, 1)	0	1	0	1	4
	1	0	1	0	
(0, 1, 0, 0)	1	0	0	1	4
	1	0	1	0	
(0, 0, 1, 1)	1	0	1	0	4
	0	1	1	0	
(0, 0, 1, 0)	1	0	1	0	4
	1	0	1	0	
(0, 0, 0, 1)	0	1	1	0	4
	1	0	1	0	
(0, 0, 0, 0)	1	0	1	0	4
	1	0	1	0	

4.2 Constant Shift 중간 값 분석

3장에서 설명한 바와 같이 Algorithm 2 (ConstShift 1)와 Algorithm 4(ConstShift 3)는 공통적으로 'MUL' 명령어를 사용하여 입력 값이 왼쪽으로 shift 하도록 설계하였는데, Algorithm 4의 경우에는 8번째 줄에서 'LSL' 명령어를 추가로 사용하였다. 일반적으로 입력 값을 1 bit 단위로 왼쪽으로 shift 하는 LSL 명령어를 사용하면 shift 연산을 쉽게 구현할 수 있다. 그러나 입력 값이 왼쪽으로 shift 함으로써 사라지는 값에 따라 해밍 웨이트나 해밍 디스턴스가 변하는 경우가 발생할 수 있다. 이에 Algorithm 4에서는 LSL 명령어로 인해 레지스터에 저장된 값의 해밍 웨이트나 해밍 디스턴스가 변하지 않는 범위에서 LSL 명령어를 제한적으로 사용하였다.

〔Table 5~8〕은 각 Constant Shift 연산 알고리즘의 중간 값을 보여주고 있으며, 해밍 웨이트 및 해밍 디스턴스가 일정하다는 것을 확인할 수 있다.

4.3 전력 모델 가정

본 절에서 사용하는 기호의 표기법은 다음과 같이 정의한다.

- $L(\ )$: leakage function
- $D(\ )$: deterministic part
- X : sensitive variable
- N : noise
- $N(\ ,\ )$: gaussian distribution
- α_i : 비트에 부여되는 가중치
- σ : variance of noise
- σ_α : bit leakage weights variance
- σ_β : 장비별 bit weights variance
- x_i : X의 i번째 비트
- μ : 비트의 평균

비트가 저장되는 레지스터들은 최초 공정에서 각각 고유의 전기적 물성으로 인해 비트 0과 1에 대한 평균 전력소모량이 0과 1에 근접하지만 정확히 0과 1이 아닐 수 있다는 '공정변이' 개념을 적용하여 조금 더 실질적인(practical) 환경을 가정하였다.

전력 모델에서 사용하는 leakage function ($L(X)$)은 deterministic part($D(X)$)와 noise part(N)로 구성되어 있으며, 식은 (1)과 같다.

$$L(X) = D(X) + N = \sum_{i=1}^{n} \alpha_i x_i + N(0, \sigma) \quad (1)$$

(가정 1) deterministic part는 각 비트마다 부여되는 가중치와 독립적으로 발생하는 noise들의 합으로 나타낼 수 있으며, 일반적으로 전력모델의 noise는 가우스 분포를 따른다.

$$\begin{aligned} x_i &= 0,\ \alpha_i \sim N(\mu_i^0, \sigma_\alpha) \\ x_i &= 1,\ \alpha_i \sim N(\mu_i^1, \sigma_\alpha) \end{aligned} \quad (2)$$

(가정 2) 레지스터 고유의 전기적 물성으로 인해 0과 1의 평균 전력소모 크기가 다르므로 각 비트마다 부여되는 가중치는 실험 모델에 독립적이며, 다음과 같은 가우스 분포를 따른다. 그리고 본 시뮬레이션에서는 $a=0$, $b=1$이라고 가정한다.

$$\begin{aligned} \mu_i^0 &\sim N(a, \sigma_\beta) \\ \mu_i^1 &\sim N(b, \sigma_\beta) \end{aligned} \quad (3)$$

Table 5. (Security of Algorithm 2) HW/HD of Intermediate values

	Reg.	Intermediate value	HW/HD
1	r0	· 00000000	0 / 0
2	r1	· 00000000	0 / 0
3	r6	· 00000000	0 / 0
4	r6	· 00000100	1 / 1
5	r1, r0	· r1: 000000 $\overline{x_3}x_3$ · r0: $\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$00	1 / 2 3 / 3
6	r0	· $\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$10	4 / 1
7	r4	· 00000000	0 / 0
8	r4	· $\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$10	4 / 4
9	r7	· 00000000	0 / 0
10	r7	· 000000 $\overline{x_3}x_3$	1 / 1
11	r0	· 00000000	0 / 4
12	r1	· 00000000	0 / 1
13	r0, r1	· r1: 000000 $\overline{x_7}x_7$ · r0: $\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4$00	1 / 1 3 / 3
14	r0	· r0: $\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4\overline{x_3}x_3$	4 / 1
15	r5	· 00000000	0 / 4
16	r5	· r5: $\overline{x_6}x_6\overline{x_5}x_5\overline{x_4}x_4\overline{x_3}x_3$	4 / 4

Table 6. (Security of Algorithm 3) HW/HD of Intermediate values

	Reg.	Intermediate value	HW/HD
1	r6	$\cdot 00000000$	0 / 0
2	r6	$\cdot 11110000$	4 / 4
3	r6	$\cdot \overline{x_3}x_3\overline{x_2}x_20000$	2 / 2
4	r6	$\cdot 0000\overline{x_3}x_3\overline{x_2}x_2$	2 / 4
5	r4	$\cdot 0000\overline{x_1}x_1\overline{x_0}x_0$	2 / 2
6	r4	$\cdot \overline{x_1}x_1\overline{x_0}x_00000$	2 / 4
7	r4	$\cdot \overline{x_1}x_1\overline{x_0}x_01010$	4 / 2
8	r5	$\cdot 0000\overline{x_5}x_5\overline{x_4}x_4$	2 / 2
9	r5	$\cdot \overline{x_5}x_5\overline{x_4}x_40000$	2 / 4
10	r5	$\cdot \overline{x_5}x_5\overline{x_4}x_4\overline{x_3}x_3\overline{x_2}x_2$	4 / 2

Table 7. (Security of Algorithm 4) HW/HD of Intermediate values

	Reg.	Intermediate value	HW/HD
1	r0	$\cdot 00000000$	0 / 0
2	r1	$\cdot 00000000$	0 / 0
3	r6	$\cdot 00000000$	0 / 0
4	r6	$\cdot 01000000$	1 / 1
5	r1, r0	$\cdot$ r1: $00\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1$ $\cdot$ r0: $\overline{x_0}x_0000000$	3 / 3 1 / 1
6	r5	$\cdot 000000\overline{x_4}x_4$	1 / 3
7	r5	$\cdot 00\overline{x_4}x_40000$	1 / 2
8	r5	$\cdot \overline{x_4}x_4000000$	1 / 2
9	r5	$\cdot \overline{x_4}x_4\overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1$	4 / 3
10	r4	$\cdot 00000000$	0 / 4
11	r4	$\cdot \overline{x_0}x_0000000$	1 / 1
12	r4	$\cdot \overline{x_0}x_0101010$	4 / 3

Table 8. (Security of Algorithm 5) HW/HD of Intermediate values

	Reg.	Intermediate value	HW/HD
1	r5	$\cdot 00000000$	0 / 4
2	r5	$\cdot \overline{x_3}x_3\overline{x_2}x_2\overline{x_1}x_1\overline{x_0}x_0$	4 / 4
3	r4	$\cdot 00000000$	0 / 4
4	r4	$\cdot 10101010$	4 / 4

4.4 시뮬레이션 설계

4.1절과 4.2절에서는 본 논문에서 제안하는 Constant 연산 알고리즘에 대한 중간 값과 그에 대한 해밍 웨이트 및 해밍 디스턴스를 분석함으로써 중간 값이 입력 값과 무관하게 일정한 해밍 웨이트와 해밍 디스턴스를 유지한다는 것을 보였다. 4.4절부터는 밸런스 인코딩을 적용했을 때 발생하는 leakage의 정보량이 그렇지 않은 경우에 비해 낮다는 것을 상호 정보량 분석을 통해 확인한다.

일반적으로 하나의 연산을 수행하기 위해서는 1 clock이 소모되는데, 밸런스 인코딩을 적용하면 하나의 연산을 수행하는데 여러 clock이 소모된다. 이는 unprotected case에서 leakage는 한 포인트에서 발생하지만, 밸런스 인코딩을 적용하면 동일한 leakage가 여러 포인트에서 발생하기 때문에 공격자에게 오히려 많은 정보를 제공할 우려가 있다는 것을 의미한다. 따라서 본 절에서는 밸런스 인코딩을 적용하지 않은 중간 값(unprotected)을 기준으로 하고, 밸런스 인코딩이 적용된 Constant AND와 Constant Shift 1~4에서 해밍 웨이트가 4인 중간 값(protected)을 대조군으로 설정하여 시뮬레이션을 수행하며, 공격자는 PoI(Point of Interest) 검출을 통해 동일 leakage가 발생하는 여러 포인트를 찾아서 공격에 사용할 수 있다고 가정하였다.

시뮬레이션은 아래와 같은 4가지 변수(X축)의 값들을 변화시켜가며 실시하였고, 각 실험을 100번 반복 후 상호 정보량(Y축)의 평균을 계산하였다.

- 시뮬레이션 1: 파형의 개수(n)
- 시뮬레이션 2: variance of noise(σ)
- 시뮬레이션 3: bit leakage weights variance(σ_α)
- 시뮬레이션 4: 장비별 bit weights variance(σ_β)

4.5 상호 정보량 분석(MIA)

시뮬레이션 결과에 대한 안전성 검증은 MATLAB R2015a를 이용하여 상호 정보량 분석[16]을 통해 실시하였다. 상호 정보량 분석이란 두 개의 서로 다른 random variable X, Y에 대해 X가 Y의 정보를 어느 정도 가지고 있는지 판단하는 척도로서 (4)와 같이 계산한다. 본 논문에서 X는 데이터(unprotected / protected)를, Y는 부채널 leakage를 의미하며, 밸런스 인코딩을 적용하지 않은 경우 발생하는 부채널 leakage에 포함된 중간 값의 정보량과 밸런스 인코딩을

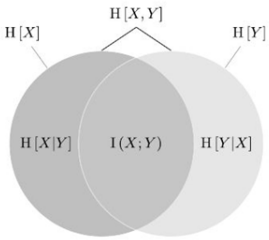

Fig. 1. Mutual Information Analysis(MIA)

적용하였을 때 발생하는 부채널 leakage에 포함된 중간 값의 정보량을 비교하였다. 이를 통해 밸런스 인코딩을 적용하면 동일 leakage가 여러 포인트에서 발생하더라도 밸런스 인코딩을 적용하지 않았을 때보다 leakage를 통해 누출되는 정보량이 적다는 것을 확인한다.

$$I(X;Y) = H(X) - H(X|Y) \quad (4)$$

4.6 시뮬레이션 결과

시뮬레이션을 통해 상호 정보량을 측정한 결과 모든 경우에서 밸런스 인코딩을 적용하였을 때 상호 정보량이 더 낮게 측정되었다. 이는 데이터에 밸런스 인코딩을 적용하면 leakage가 비밀정보를 포함하는 중간 값과 관련이 없는 것처럼 보이도록 하는 대응기법으로서의 목적을 달성했다고 할 수 있다.

시뮬레이션 1(Fig. 2)의 경우, 파형의 개수가 많아질수록 상호 정보량은 특정한 값으로 수렴하지만, 밸런스 인코딩을 적용한 경우 상호 정보량이 그렇지 않은 경우에 비해 현저히 낮다. 시뮬레이션 2(Fig. 3)와 시뮬레이션 3(Fig. 4)의 경우도 noise가 커질수록 정확한 분석은 제한되지만, noise가 적은 상황에서는 밸런스 인코딩을 적용했을 때 상호 정보량 값이 확연히 낮음을 확인할 수 있다. 레지스터의 공정변이 개념을 적용한 시뮬레이션 4(Fig. 5)는 장비별 비트 평균의 분산이 커질수록 밸런스 인코딩을 적용했을 때 상호 정보량값이 급격하게 증가하였는데, 이

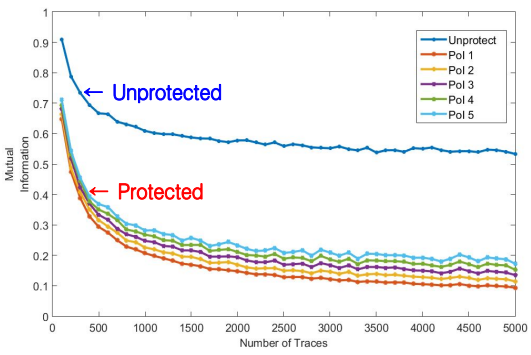

Fig. 2. Simulation 1 (σ=1, σ_α=0.1, σ_β=0.1, n=100~5,000)

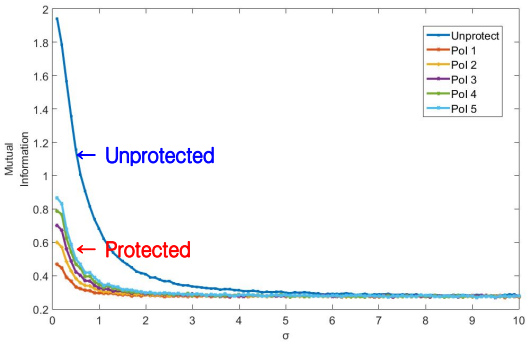

Fig. 3. Simulation 2 (σ=0.1~10, σ_α=0.1, σ_β=0.1, n=500)

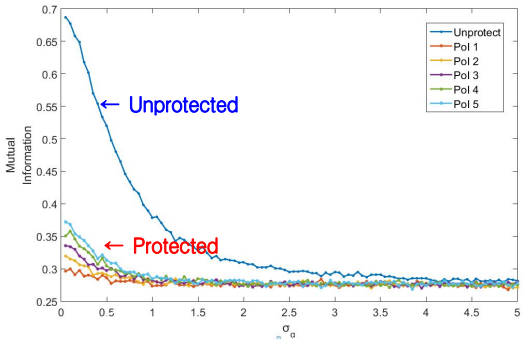

Fig. 4. Simulation 3 (σ=1, σ_α=0.05~5, σ_β=0.1, n=500)

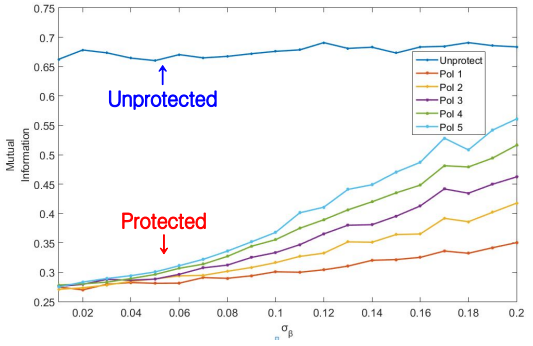

Fig. 5. Simulation 4 (σ=1, σ_α=0.1, σ_β=0.01~0.2, n=500)

는 반대로 비트마다특성이 확연해 질수록, 즉 비트 값에 따른 평균 전력소모량이 0과 1에 가까울수록 부채널 분석이 용이하다는 것을 보여준다.

Table 9. Performance of Proposals

Algorithm	Clock cycle
Constant AND	54 (STEP 1: 8×4, STEP 2: 10×2, STEP 3: 2)
Constant Shift 1	18
Constant Shift 2	10
Constant Shift 3	14
Constant Shift 4	4
Constant XOR [6]	47 (STEP 1: 8×4, STEP 2: 4×2, STEP 3: 7)
Constant ADD [10] (Kogge-Stone Adder)	637 (ConstAND: 6, ConstShift 1: 3, ConstShift 2: 2, ConstShift 4: 1, ConstXOR: 5)

V. 성능비교 및 결론

본 논문에서는 밸런스 인코딩 기법을 적용한 Constant AND, Shift 연산을 새롭게 제안하였으며, 시뮬레이션을 통해 제안하는 Constant 연산 알고리즘에 대한 안전성을 평가하였다. 본 논문에서 제안한 Constant 연산 알고리즘의 연산량은 [Table 9]와 같으며, 알고리즘에서 MUL 연산(2 clock)을 제외한 모든 연산은 1 clock으로 계산하였다. Constant ADD의 경우, [10]에서 제안된 Kogge-Stone Adder에 Constant AND·Shift·XOR를 적용하여 구현한 관계로 상대적으로 overhead가 많이 발생하였다. 밸런스 인코딩 기법을 적용하면 연산하는 데이터의 길이 증가 및 추가적인 연산 등으로 인해 효율성은 다소 저하될 수 있다. 하지만 초소형 무인 항공기나 초소형 IoT 디바이스처럼 제한된 내부 저장 공간으로 인해 하드웨어 기반 암호화 방법을 적용하기 어려운 환경에서는 소프트웨어 암호화 방법이 대안이 될 수 있으며, 이러한 경우에 밸런스 인코딩 기법을 적용한다면 부채널 분석 위협도 완화시킬 수 있다.

본 논문에서 제안하는 Constant 연산 기법은 [7], [8], [9]와 같은 ARX 기반 대칭키 암호 알고리즘에 적용이 가능하다. 향후에는 제안한 알고리즘의 효율성을 향상시키기 위한 최적화와 함께 본 논문에서 적용한 밸런스 인코딩 형태($\overline{b}_3 b_3 \overline{b}_2 b_2 \overline{b}_1 b_1 \overline{b}_0 b_0$) 외 다른 인코딩 형태에 따른 안전성 차이 여부 확인에 대한 추가적인 연구가 필요할 것으로 보인다.

References

[1] Segyeilbo, "http://www.segye.com/content/html/2016/08/10/20160810003908.html", Aug. 2016.

[2] Kocher, P.C., Jaffe, J., Jun, B., "Differential power analysis", CRYPTO 1999. LNCS, vol. 1666, pp. 388-397. Springer, Heidelberg, 1999.

[3] Goubin, L., Patarin, J., "DES and Differential Power Analysis. The 'Duplication' Method", CHES 1999, pp. 158-172, 1999.

[4] Tiri, K., Verbauwhede, I., "A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation", DATE 2004, pp. 246-251, 2004.

[5] Hoogvorst, P., Duc, G., Danger, J.-L., "Software implementation of dual-rail representation", COSADE 2011, pp. 73-81, 2011.

[6] Chen, C., Eisenbarth, T., Shahverdi, A., Ye, X., "Balanced encoding to mitigate power analysis: a case study", CARDIS 2014. LNCS, vol. 8968, pp. 49-63. Springer, Heidelberg, 2015.

[7] Hong, D.J., Sung, J.C., Hong, S.H., et al., "HIGHT: A New Block Cipher Suitable for Low-Resource Device", CHES 2006. LNCS, vol. 4249, pp. 46-59, Springer, Heidelberg, 2006.

[8] Park. J., et al., "128-Bit Block Cipher LEA", TTAK.KO-12.0223, Dec. 2013.

[9] Beaulieu. R., Shors. D., et al, "SIMON and SPECK: Block Ciphers for the Internet of Things", NIST Lightweight

Cryptography Workshop 2015.
〔10〕 Coron, J.S., Großschädl, J., Tibouchi, M., Vadnala, P.K., "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity", FSE 2015, Springer, Heidelberg, Aug, 2015.
〔11〕 Won, Y.S., Hodgers, P., O'Neill, M., Han, D.G., "On the Security of Balanced Encoding Countermeasures", CARDIS 2015, LNCS 9514, pp. 242-256, Springer, Heidelberg, 2016.
〔12〕 Biham, E., Shamir, A., "Differential Fault Analysis of Secret Key Cryptosystems", CRYPTO '97, LNCS 1294, pp. 513-525, Aug. 1997.
〔13〕 Boneh, D., DeMillo R. A., Lipton, R. J., "On the Importance of Checking Cryptographic Protocols for Faults", EOROCRYPTO '97, LNCS 1233, pp. 37-51, May, 1997.
〔14〕 Peter M Kogge, Harold S Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", Computers, IEEE Transactions on, 100(8), pp. 786-793, 1973.
〔15〕 Maghrebi, H., Servant, V., Bringer, J., "There is Wisdom in Harnessing the Strengths of your Enemy : Customized Encoding to Thwart Side -Channel Attacks(Extended Version)", FSE 2016, pp. 223-243, Springer, Heidelberg, July, 2016.
〔16〕 B. Gierlichs, L.Batina, P. Tuyls, B.Preneel, "Mutual information analysis - generic side-channel distinguisher", CHES 2008, pp. 426-442, Springer, Berlin, Aug, 2008.
〔17〕 Atmel, "http://www.atmel.com/Images/Atmel-0856-AVR-Instruction-Set-Manual.pdf"

〈저자소개〉

윤 진 영 (JinYeong Yoon) 정회원
2005년 2월: 육군사관학교 문학사
2015년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 부채널 공격 및 대응기법, 정보보호 정책

김 한 빛 (HanBit Kim) 학생회원
2014년 2월: 고려대학교 신소재공학 학사
2016년 2월: 고려대학교 정보보호대학원 석사
2016년 2월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 부채널 공격 및 대응기법, 암호시스템 안전성 분석 및 고속구현

김 희 석 (HeeSeok Kim) 정회원
2006년: 연세대학교 수학과 학사
2008년: 고려대학교 정보보호대학원 석사
2011년: 고려대학교 정보보호대학원 박사
2011년 9월~2012년 12월: Bristol University 박사후 연구원
2013년~2016년 8월: 한국과학기술정보연구원(KISTI) 선임연구원
2015년~2016년 8월: 과학기술연합대학원대학교(UST) 조교수
2016년 9월~현재: 고려대학교 과학기술대학 수학과 조교수
〈관심분야〉 부채널 공격, 암호시스템 안전성 분석 및 고속구현, 암호칩 설계 기술, 보안관제, 네트워크 보안

홍 석 희 (SeokHie Hong) 종신회원
1995년: 고려대학교 수학과 학사
1997년: 고려대학교 수학과 석사
2001년: 고려대학교 수학과 박사
1999년 8월~2004년 2월: ㈜시큐리티 테크놀로지 선임연구원
2003년 3월~2004년 2월: 고려대학교 정보보호기술연구센터 선임연구원
2004년 4월~2005년 2월: K.U. Leuven ESAT/SCD-COSIC 박사후 연구원
2005년 3월~2013년 8월: 고려대학교 정보보호대학원 부교수
2013년 9월~현재: 고려대학교 정보보호대학원 정교수
〈관심분야〉 대칭키 및 공개키 암호 알고리즘, 부채널 공격 및 대응기법, 디지털 포렌식