PAGE: Practical AES-GCM Encryption for Low-End Microcontrollers

https://youtu.be/aQCqSKftExM

김경호

한성대학교 HANSUNG UNIVERSITY

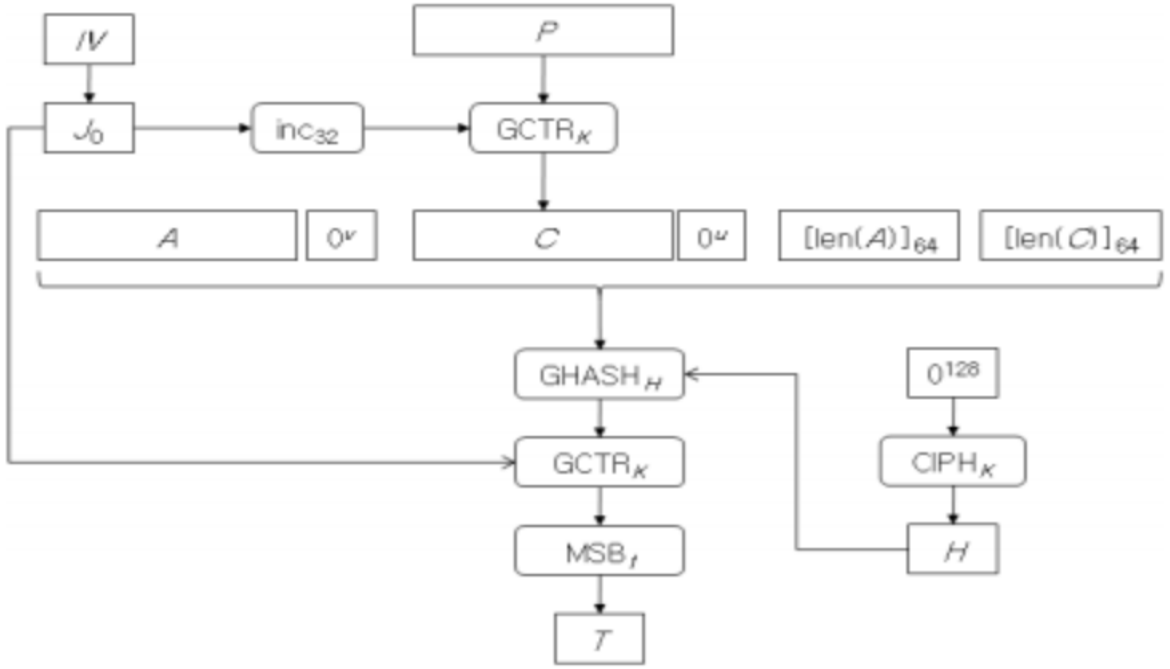
CryptoCraft LAB
https://crypto.modoo.at

Contents

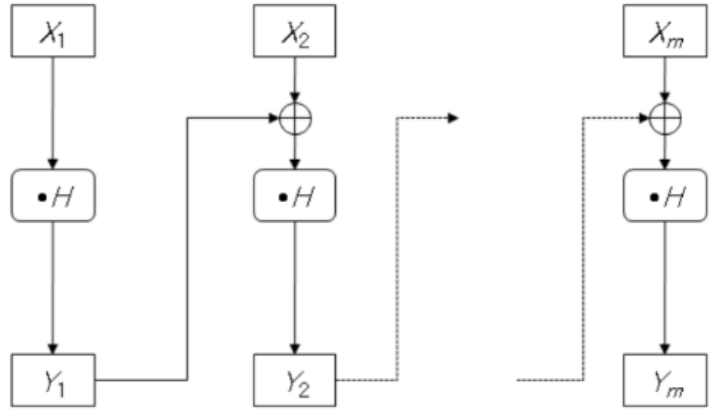
CryptoCraft LAB

1. Related work | AES-GCM

- 현재 가장 많이 사용하는 암호화 모드
- Message 인증을 포함하는 암호화 모드(GMAC을 이용한 무결성 검증)
- AES CTR 암호화 + Galois Message Authentication Code(GMAC)
- GF(2^{128})



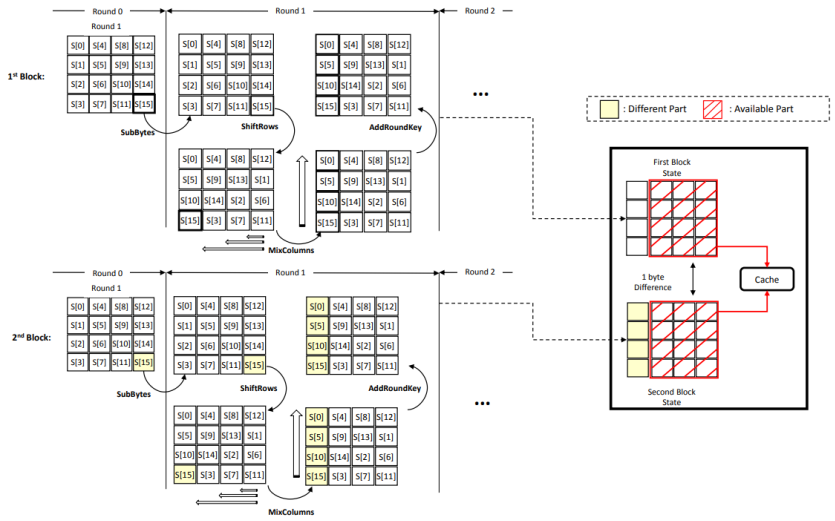
GCM Overview

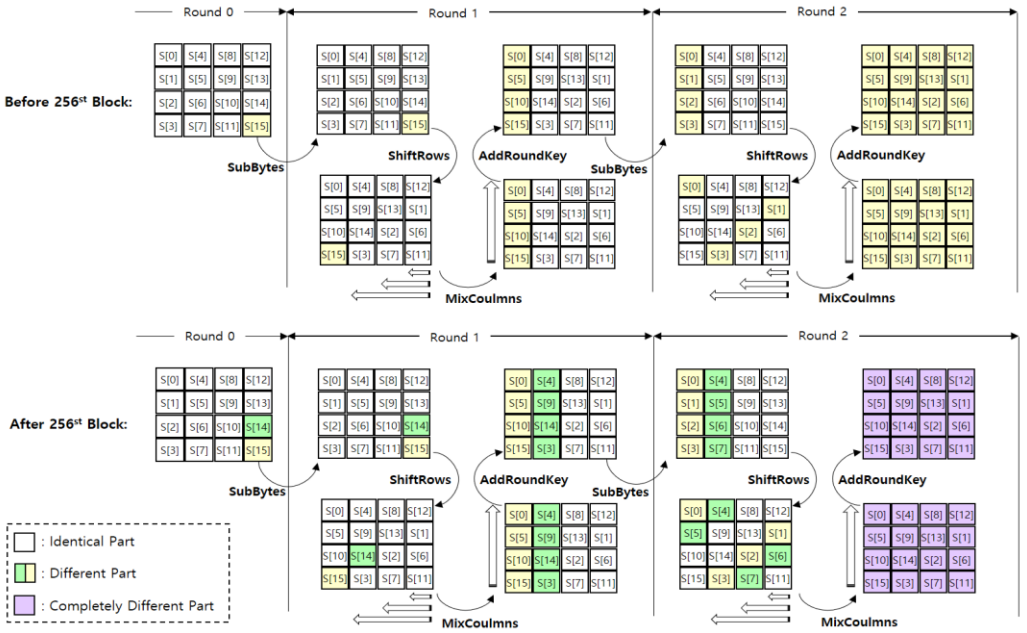


GHASH Overview

1. Related work | FACE

- 2018 CHES에서 발표된 AES 최적화 기법
- Look Up Table(LUT)을 이용한 연산 시간 최적화
- AES-CTR의 Counter 값의 특징을 이용한 LUT 저장
- Counter 값 변화에 따라 일정 주기마다 **LUT 업데이트 필요**
- LUT의 크기는 5KB ↑





1. Related work | 128bit Binary Field Multiplication

- GCM에서는 128bit Binary Field(BF) Multiplication 이용
- Multiplication은 연산 시간이 크기 때문에 최적화 필요

- LUT Method
  - LUT를 이용한 연산 시간 단축
  - Timing Attack, SPA 방어
  - 반복적인 LUT 접근에 따른 CPA 공격 취약성 밝혀짐

- Block Comb Method
  - Multiplicand(피승수)의 bit 값에 따라 연산
  - 1 -> 연산, 0 -> Pass
  - If, else로 인한 Timing Attack 취약

Algorithm 1 Constant exclusive-or operation with NOP [25]
Require: Operand A
Ensure: Result R
1: **if** exclusive operation is required **then**
2: $R \leftarrow R \oplus A$ {normal exclusive-or operation}
3: **else**
4: NOP {no operation}
5: **end if**
6: **return** R

Algorithm 2 Constant exclusive-or operation with zero variable [25]
Require: Operand A
Ensure: Result R
1: **if** exclusive operation is required **then**
2: $R \leftarrow R \oplus A$ {normal exclusive-or operation}
3: **else**
4: $R \leftarrow R \oplus 0$ {exclusive-or operation with zero value}
5: **end if**
6: **return** R

1. Related work | Research on Block Comb Method

- Secure GCM Implementation on AVR – Ziu et al
  - LUT Method에 대한 CPA 공격
  - Masked Block Comb(MBC) Method 제안
  - Karatsuba Algorithm을 통한 곱셈기 최적화 구현

Algorithm 5 Masked Block Comb on 32-bit

Require: 32-bit wise operands A and B
Ensure: Result $C \leftarrow A \cdot B$

```
for i from 7 by 1 to 0 do
    for j from 3 by 1 to 0 do
        BIT ← A[j]&(1 ≪ i)
        {MASK, T0} ← 0 − BIT
        for k from 3 by 1 to 0 do
            C[k + j] ← C[k + j] ⊕ (B[k]&MASK)
        end for
    end for
    C ← C ≪ 1
end for
return C
```

1. BIT 변수에 A[j]의 I 번째 bit set
2. BIT이 1이면 MASK = 0xff, 0이면 MASK = 0
3. MASK = 0xff 이면 동일한 결과 출력
4. MASK = 0 이면 XOR 0 -> 결과에 지장 x
5. 결과 1비트 왼쪽 shift 연산 후 8번 반복

1. Related work | Research on Block Comb Method

- SCA-Resistant GCM Implementation on AVR – Seo and Kim
  - MBC(Masked Block Comb) Method에 CPA 공격
  - Dummy XOR을 통한 SPA 방지
  - GHASH의 CPA를 막기 위한 Masking 제안
  - 8개의 Garbage 레지스터 사용

1: $R_{25} \leftarrow 0x07$ // Set displacement value for dummy ADD instruction for ILA
2: **for** $l = 0$ to 15 **do**
3: $R_l \leftarrow 0$
4: **end for**
5: **for** $l = 0$ to 3 **do**
6: $R_{16+l} \leftarrow A[l]$
7: $R_{21+l} \leftarrow B[l]$
8: **end for**
9: $R_{20} \leftarrow 0$
10: // Processing from 0-th bit to 6-th bit
11: **for** $l = 0$ to 6 **do**
12: **for** $m = 0$ to 3 **do**
13: **if** the l-th bit of R_{21+m}==1 **then**
14: $R_0 \leftarrow R_0 + R_{25}$ // Dummy ADD instruction for ILA
15: **for** $k = 0$ to 4 **do**
16: $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
17: **end for**
18: **else**
19: // Dummy XOR with the garbage registers
20: **for** $k = 0$ to 4 **do**
21: $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
22: **end for**
23: **end if**
24: **end for**
25: $(R_{20}, \ldots, R_{16}) \leftarrow (R_{20}, \ldots, R_{16}) \ll 1$
26: **end for**
27: // Processing the final 7-th bit
28: **for** $m = 0$ to 3 **do**
29: **if** the 7-th bit of R_{21+m}==1 **then**
30: $R_0 \leftarrow R_0 + R_{25}$ // Dummy ADD instruction for ILA
31: **for** $k = 0$ to 4 **do**
32: $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
33: **end for**
34: **else**
35: // Dummy XOR with the garbage registers
36: **for** $k = 0$ to 4 **do**
37: $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
38: **end for**
39: **end if**
40: **end for**

```
l_1_m_0:
    SBRS    R21, 1
    RJMP    l_1_m0_Dummy

    ADD     R0, RT
    EOR     R8, R16
    EOR     R9, R17
    EOR     R10, R18
    EOR     R11, R19
    EOR     R12, R20
    RJMP    l_1_m_1

l_1_m0_Dummy:
    EOR     R0, R16
    EOR     R1, R17
    EOR     R2, R18
    EOR     R3, R19
    EOR     R4, R20
    RJMP    l_1_m_1
```

1. Related work | Karatsuba

- 효율적인 연산을 통해 곱셈 연산을 최적화하는 알고리즘

- 기본적인 곱셈 알고리즘

$x = x_1B^m + x_0$
$y = y_1B^m + y_0$

$xy = (x_1B^m + x_0)(y_1B^m + y_0)$
$xy = x_1y_1B^{2m} + (x_1y_0 + x_0y_1)B^m + x_0y_0$

Ex) 1234 * 5678

12 34 = 12 × 10^2 + 34
56 78 = 56 × 10^2 + 78

z_2 = 12 × 56 = 672
z_0 = 34 × 78 = 2652
$z_1 = (12 + 34)(56 + 78) - z_2 - z_0 = 46 \times 134 - 672 - 2652 = 2840$
결과 = $z_2 \times 10^{2\times2} + z_1 \times 10^2 + z_0$ = 672 × 10000 + 2840 × 100 + 2652 = **7006652**

- Karatsuba Algorithm

$z_2 = x_1y_1$
$z_0 = x_0y_0$
$z_1 = (x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0) - x_1y_1 - x_0y_0$
$= x_1y_0 + x_0y_1$

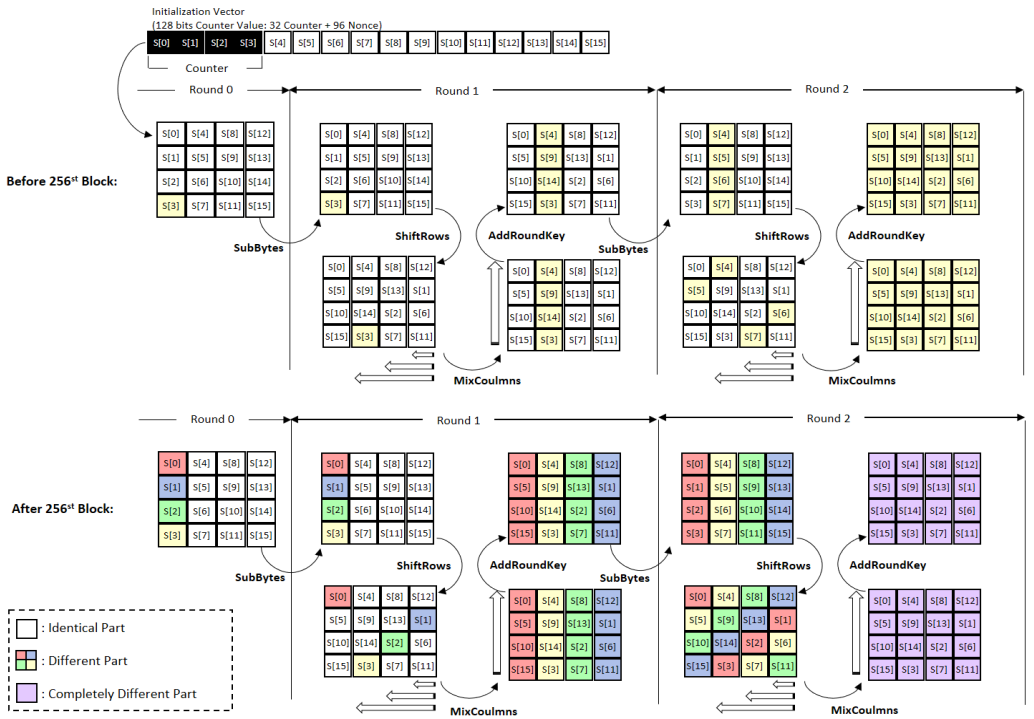
$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$

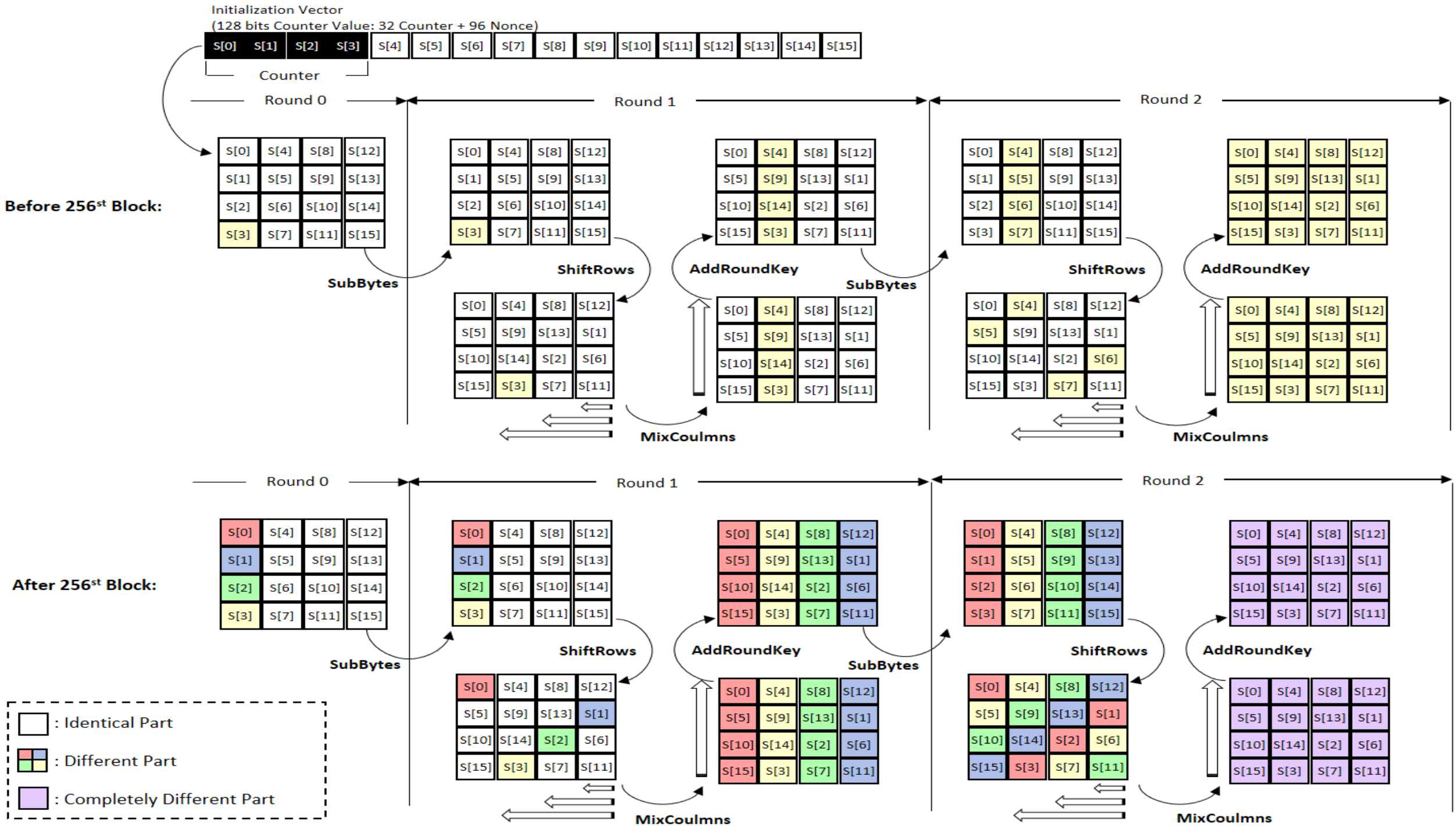
2. FACE-LIGHT

- FACE에 기반한 최적화 구현 기법
  - 저전력 프로세스에 최적화 된 구현 기법

- Counter 값 변화에 따라 **LUT 업데이트 필요 없음**

- 기존 FACE와 결합하여 성능 향상




Initialization Vector
(128 bits Counter Value: 32 Counter + 96 Nonce)
S[0] S[1] S[2] S[3] S[4] S[5] S[6] S[7] S[8] S[9] S[10] S[11] S[12] S[13] S[14] S[15]
Counter
Round 0
Round 1
Round 2
Before 256st Block:
SubBytes
ShiftRows
AddRoundKey
MixCoulmns
After 256st Block:
: Identical Part
: Different Part
: Completely Different Part

2. FACE-LIGHT

Initialization Vector
(128 bits Counter Value: 32 Counter + 96 Nonce)

00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

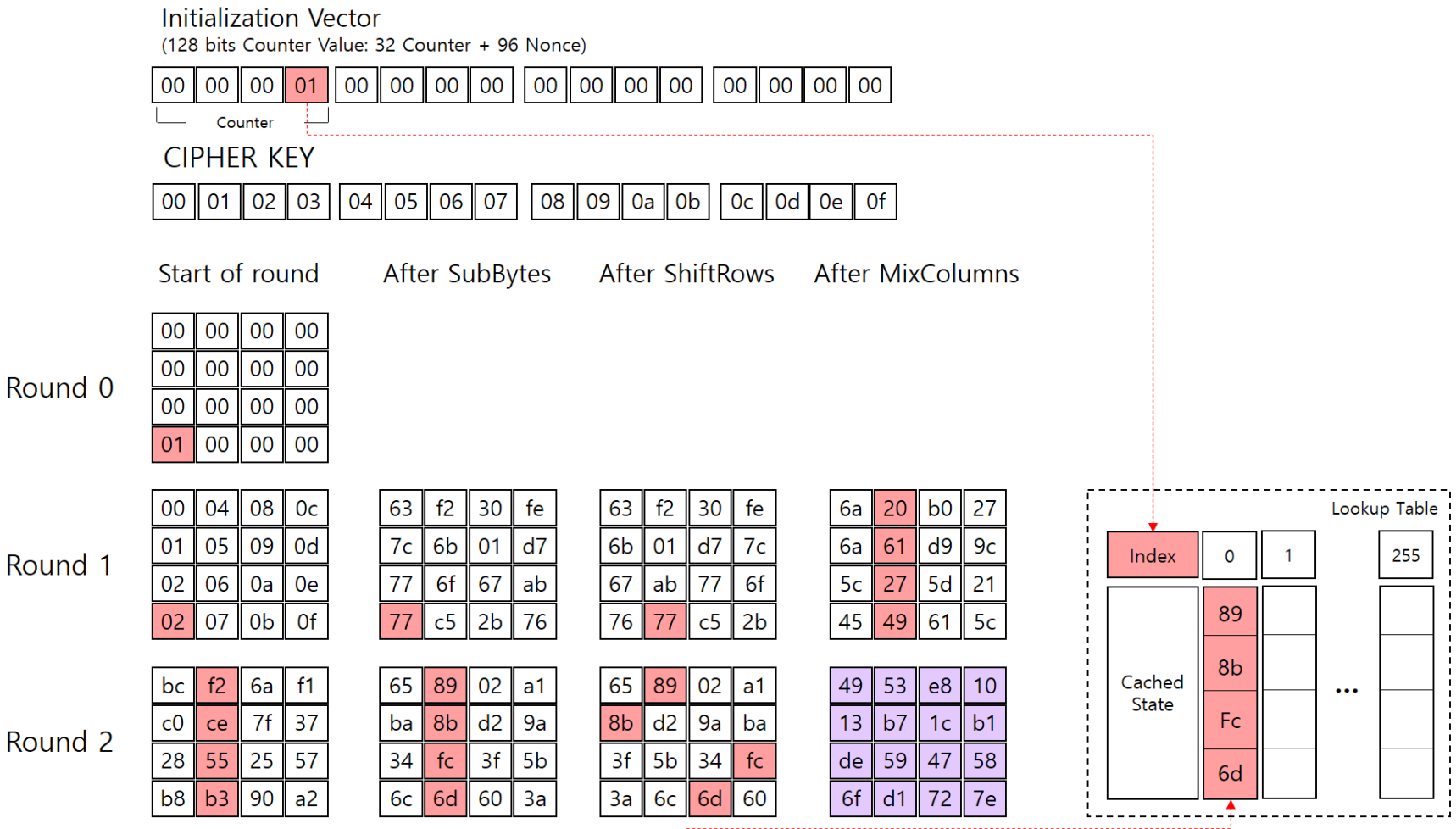
Counter

CIPHER KEY

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f

	Start of round	After SubBytes	After ShiftRows	After MixColumns
Round 0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00			
Round 1	00 04 08 0c 01 05 09 0d 02 06 0a 0e 03 07 0b 0f	63 f2 30 fe 7c 6b 01 d7 77 6f 67 ab 7b c5 2b 76	63 f2 30 fe 6b 01 d7 7c 67 ab 77 6f 76 7b c5 2b	6a 2c b0 27 6a 6d d9 9c 5c 33 5d 21 45 51 61 5c
Round 2	bc fe 6a f1 c0 c2 7f 37 28 41 25 57 b8 ab 90 a2	65 bb 02 a1 ba 25 d2 9a 34 83 3f 5b 6c 62 60 3a	65 bb 02 a1 25 d2 9a ba 3f 5b 34 83 3a 6c 62 60	a0 37 e7 6f 54 85 13 30 70 6b 56 a6 c1 87 6c 01

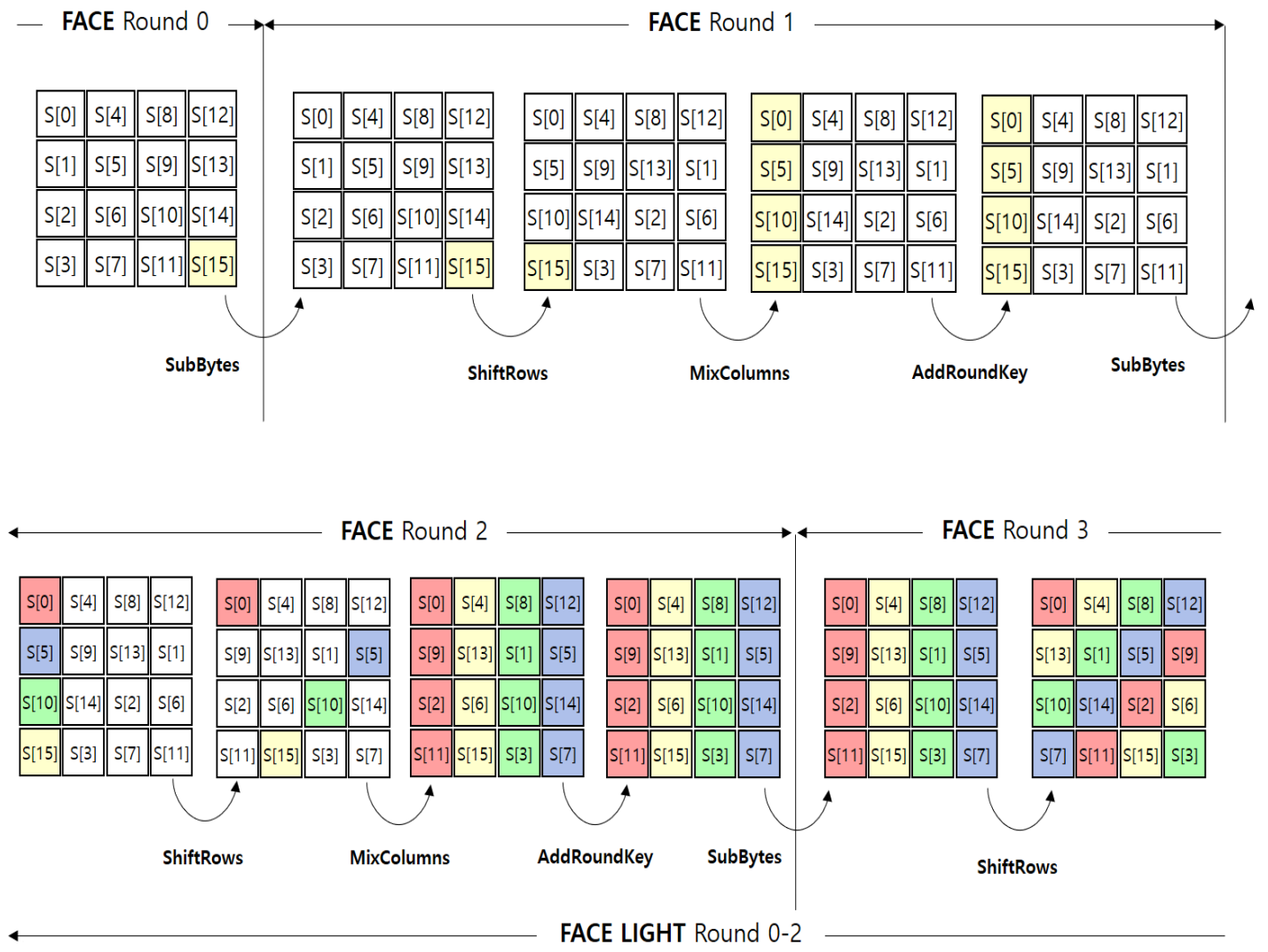
2. FACE-LIGHT


Initialization Vector
(128 bits Counter Value: 32 Counter + 96 Nonce)
00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
Counter
CIPHER KEY
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Start of round
After SubBytes
After ShiftRows
After MixColumns
Round 0
00 00 00 00
00 00 00 00
00 00 00 00
01 00 00 00
Round 1
00 04 08 0c
01 05 09 0d
02 06 0a 0e
02 07 0b 0f
63 f2 30 fe
7c 6b 01 d7
77 6f 67 ab
77 c5 2b 76
63 f2 30 fe
6b 01 d7 7c
67 ab 77 6f
76 77 c5 2b
6a 20 b0 27
6a 61 d9 9c
5c 27 5d 21
45 49 61 5c
Round 2
bc f2 6a f1
c0 ce 7f 37
28 55 25 57
b8 b3 90 a2
65 89 02 a1
ba 8b d2 9a
34 fc 3f 5b
6c 6d 60 3a
65 89 02 a1
8b d2 9a ba
3f 5b 34 fc
3a 6c 6d 60
49 53 e8 10
13 b7 1c b1
de 59 47 58
6f d1 72 7e
Lookup Table
Index
0
1
255
Cached State
89
8b
Fc
6d
...

2. FACE-LIGHT | Extended FACE

- Extended FACE
  - Original FACE
  - FACE-Light

- 연산 절감 효과
  - Subbytes
  - AddRoundKey

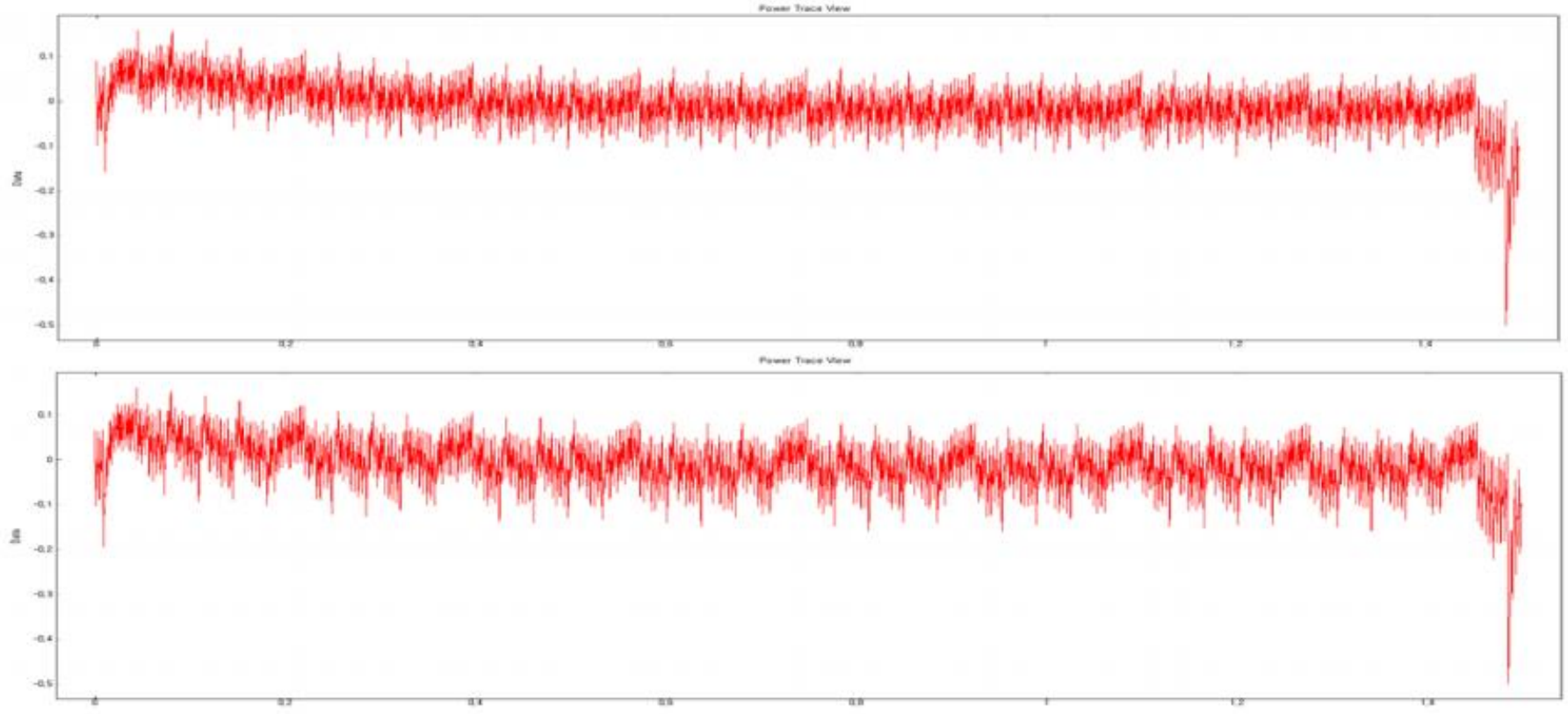


3. 128bit Binary field multiplication

- 기존 연구(SCA-Resistant GCM Implementation on AVR) 에서 성능 향상
  - 8개의 Garbage Register를 1개로 감소 및 곱셈 방식 교체로 Register 1개 확보
  - 확보한 8개의 레지스터를 이용한 Karatsuba Algorithm 최적화
  - Reduction 최적화 구현

3. 128bit Binary field multiplication

- 8개의 Garbage Register를 1개로 감소(동일 곱셈 연산)
  - 파형에 큰 변화가 없으므로 공격자가 판단 불가능



3. 128bit Binary field multiplication

Input: : 32-bit multiplicand A and 32-bit multiplier B ($R_{19}, \ldots, R_{16}$ A value, $R_{23}, \ldots, R_{20}$ B value)
Output: : Result C(64-bit) = $A \times B$ (Garbage result goes into (R_{24}) and real result goes into ($R_{15}, \ldots, R_8$))

...

1: $R_{25} \leftarrow 0x06$
2: **for** $l = 7$ to 0 **do**
3: **for** $m = 3$ to 0 **do**
4: **if** the l-th bit of R_{16+m}==1 **then**
5: $R_0 \leftarrow R_0 + R_{25}$
6: **for** $k = 0$ to 4 **do**
7: $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{20+k}$
8: **end for**
9: **else**
10: **for** $k = 0$ to 4 **do**
11: $R_{24} \leftarrow R_{24} \oplus R_{20+k}$
12: **end for**
13: **end if**
14: **end for**
15: $(R_{15}, \ldots, R_8) \leftarrow (R_{15}, \ldots, R_8) \ll 1$
16: **end for**

Our work

Algorithm 5 Masked Block Comb on 32-bit
Require: 32-bit wise operands A and B
Ensure: Result $C \leftarrow A \cdot B$
1: **for** i from 7 by 1 to 0 **do**
2: **for** j from 3 by 1 to 0 **do**
3: $BIT \leftarrow A[j] \& (1 \ll i)$
4: $\{MASK, T0\} \leftarrow 0 - BIT$
5: **for** k from 3 by 1 to 0 **do**
6: $C[k+j] \leftarrow C[k+j] \oplus (B[k] \& MASK)$
7: **end for**
8: **end for**
9: $C \leftarrow C \ll 1$
10: **end for**
11: **return** C

Ziu et al

1: $R_{25} \leftarrow 0x07$ // Set displacement value for dummy ADD instruction for ILA
2: **for** $l = 0$ to 15 **do**
3: $R_l \leftarrow 0$
4: **end for**
5: **for** $l = 0$ to 3 **do**
6: $R_{16+l} \leftarrow A[l]$
7: $R_{21+l} \leftarrow B[l]$
8: **end for**
9: $R_{20} \leftarrow 0$
10: // Processing from 0-th bit to 6-th bit
11: **for** $l = 0$ to 6 **do**
12: **for** $m = 0$ to 3 **do**
13: **if** the l-th bit of R_{21+m}==1 **then**
14: $R_0 \leftarrow R_0 + R_{25}$ // Dummy ADD instruction for ILA
15: **for** $k = 0$ to 4 **do**
16: $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
17: **end for**
18: **else**
19: // Dummy XOR with the garbage registers
20: **for** $k = 0$ to 4 **do**
21: $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
22: **end for**
23: **end if**
24: **end for**
25: $(R_{20}, \ldots, R_{16}) \leftarrow (R_{20}, \ldots, R_{16}) \ll 1$
26: **end for**
27: // Processing the final 7-th bit
28: **for** $m = 0$ to 3 **do**
29: **if** the 7-th bit of R_{21+m}==1 **then**
30: $R_0 \leftarrow R_0 + R_{25}$ // Dummy ADD instruction for ILA
31: **for** $k = 0$ to 4 **do**
32: $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
33: **end for**
34: **else**
35: // Dummy XOR with the garbage registers
36: **for** $k = 0$ to 4 **do**
37: $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
38: **end for**
39: **end if**
40: **end for**

Seo and Kim

- MBC에서 사용한 곱셈 방식 사용
- 40bit의 Multiplicand를 사용하는 기존 연구에 비해 1개의 레지스터 확보

3. 128bit Binary field multiplication

- 확보한 8개의 레지스터를 이용한 Karatsuba Algorithm 최적화

Algorithm 5 Karatsuba algorithm for 64-bit multiplication

Input: : 64-bit wise operands A and B
Output: : Result $C \leftarrow A \times B$

1: $L \leftarrow A[3 \sim 0] \times B[3 \sim 0]$
2: $H \leftarrow A[7 \sim 4] \times B[7 \sim 4]$
3: $M \leftarrow (A[7 \sim 4] \oplus A[3 \sim 0]) \times (B[7 \sim 4] \oplus B[3 \sim 0])$
4: $M \leftarrow M \oplus L \oplus H$
5: $C \leftarrow (H \ll 64) \oplus (M \ll 32) \oplus L$
6: **return** C

Algorithm 6 Proposed Level 1 Karatsuba Block-Comb in source code level

```
1: ROUND32

2: STD Z+0, C0
3: STD Z+1, C1
4: STD Z+2, C2
5: STD Z+3, C3

6: EOR C0, C4
7: EOR C1, C5
8: EOR C2, C6
9: EOR C3, C7

10: MOVW K0, C0
11: MOVW K2, C2
12: MOVW K4, C4

13: MOVW K6, C6

14: ROUND32

15: STD Z+12, C4
16: STD Z+13, C5
17: STD Z+14, C6
18: STD Z+15, C7

19: EOR K0, C0
20: EOR K1, C1
21: EOR K2, C2
22: EOR K3, C3

23: EOR C0, C4

24: EOR C1, C5
25: EOR C2, C6
26: EOR C3, C7

27: EOR K4, C0
28: EOR K5, C1
29: EOR K6, C2
30: EOR K7, C3

31: ROUND32

32: EOR C0, K0
33: EOR C1, K1
34: EOR C2, K2

35: EOR C3, K3

36: EOR C4, K4
37: EOR C5, K5
38: EOR C6, K6
39: EOR C7, K7

40: STD Z+4, C0
41: STD Z+5, C1
42: STD Z+6, C2
43: STD Z+7, C3
44: STD Z+8, C4
45: STD Z+9, C5
46: STD Z+10, C6
47: STD Z+11, C7
```

K0 ~ K7을 이용해 Karatsuba 과정에서 C[4] ~ C[11] 연산에 필요한 값을 스텍 메모리 사용 없이 처리

3. 128bit Binary field multiplication

- Reduction 최적화 구현
  - 128bit 곱셈의 결과를 Reduction하는 과정에서 소프트웨어 최적화
  - 8bit 환경에 알맞게 최적화 구현
  - Assembly 구현으로 320cc로 최적화

Algorithm 2 (Fast reduction modulo g)

Input: 256-bit string $[X_3 : X_2 : X_1 : X_0]$ where X_3, X_2, X_1, X_0 are 64-bit long each.

Step 1: Shift X_3 by 63-, 62- and 57-bit positions to the right, to compute:

$$A = X_3 \gg 63$$
$$B = X_3 \gg 62$$
$$C = X_3 \gg 57 \quad (23)$$

Step 2: XOR A, B, and C with X_2. Compute D as follows:

$$D = X_2 \oplus A \oplus B \oplus C \quad (24)$$

Step 3: Shift $[X_3 : D]$ by 1-, 2- and 7-bit positions to the left. Compute numbers:

$$[E_1 : E_0] = [X_3 : D] \ll 1$$
$$[F_1 : F_0] = [X_3 : D] \ll 2$$
$$[G_1 : G_0] = [X_3 : D] \ll 7 \quad (25)$$

Step 4: XOR $[E_1 : E_0]$, $[F_1 : F_0]$, and $[G_1 : G_0]$ with each other and $[X_3 : D]$. Compute a number $[H_1 : H_0]$ as follows:

$$[H_1 : H_0] = [X_3 \oplus E_1 \oplus F_1 \oplus G_1 : D \oplus E_0 \oplus F_0 \oplus G_0] \quad (26)$$

Output: $[X_1 \oplus H_1 : X_0 \oplus H_0]$ (the reduction result)

4. Evaluation | FACE-LIGHT

- 표준 AES보다 약 **22% 성능 향상**을 보임
- 추가적인 LUT 업데이트 시간 소요가 없음

Security level	Dinu et al. [17]	Otte et al. [18]	FACE-LIGHT (This work)	Extended FACE (This work)
AES-128	2,835	2,507	2,218	1,967
AES-192	N/A	2,991	2,702	2,449
AES-256	N/A	3,473	3,184	2,931

4. Evaluation | FACE-LIGHT

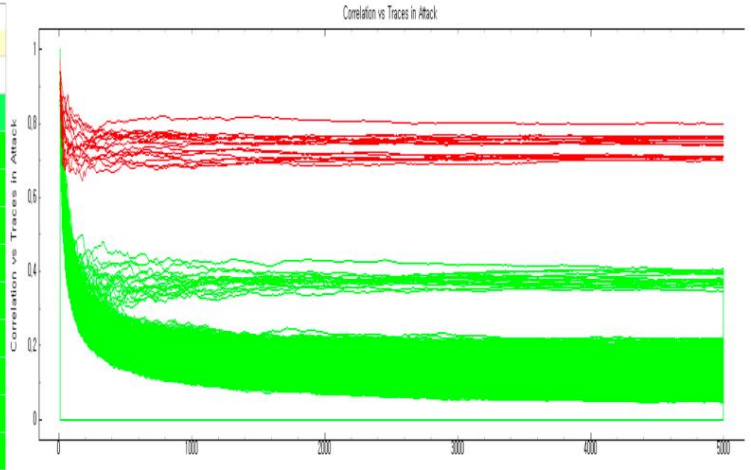
- FACE-LIGHT **8bits Microcontroller**에 최적화 됨
- LUT 업데이트가 없기 때문에 Constant Timing 유지
- 8bits 저전력 프로세서부터 제약없이 사용 가능

	FACE [9]	FACE–LIGHT (This work)
Table update	√	–
Constant timing	–	√
Target processor	32-bit or above	8-bit or above
Expandable Round	Round 2	Round 3

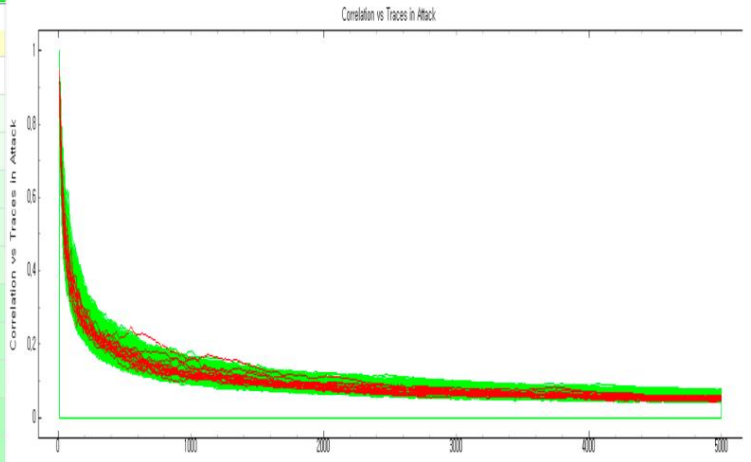
4. Evaluation | FACE-LIGHT

- **파형 분석 공격**(CPA, DPA)에 대한 **저항성**을 지님

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	2B 0.7013	7E 0.7601	15 0.7126	16 0.7651	28 0.7064	AE 0.7453	D2 0.7093	A6 0.7518	AB 0.7095	F7 0.7581	15 0.7409	88 0.7640	09 0.7067	CF 0.7530	4F 0.7115	3C 0.7998
1	2A 0.3791	7F 0.3795	14 0.3778	17 0.3984	29 0.3711	AF 0.3961	D3 0.3647	A7 0.3987	AA 0.3669	F6 0.4060	14 0.3464	89 0.3737	08 0.3563	CE 0.3780	4E 0.3653	3D 0.3930
2	24 0.1953	9A 0.2181	B3 0.2100	E3 0.2041	DC 0.1975	4A 0.2078	A7 0.2149	99 0.1848	5E 0.1874	13 0.2106	BF 0.1899	8D 0.1909	B1 0.1851	B4 0.1993	F5 0.2012	E1 0.1954
3	FB 0.1948	48 0.2059	1E 0.2012	38 0.1640	82 0.1939	C3 0.1845	60 0.2144	53 0.1823	3F 0.1819	8C 0.1918	7E 0.1888	FD 0.1892	91 0.1844	3A 0.1939	B1 0.2003	21 0.1892
4	20 0.1947	0B 0.2035	E5 0.1889	29 0.1821	47 0.1927	D5 0.1730	F9 0.2119	75 0.1794	97 0.1780	02 0.1888	B1 0.1811	33 0.1793	9D 0.1778	2B 0.1919	F7 0.1971	81 0.1878
5	DB 0.1881	88 0.1896	A1 0.1879	E5 0.1780	9C 0.1890	98 0.1719	FD 0.2074	8C 0.1770	EE 0.1767	C2 0.1817	8C 0.1783	9E 0.1782	D1 0.1724	F0 0.1833	0C 0.1935	C9 0.1790
6	06 0.1844	68 0.1885	E0 0.1864	5C 0.1769	D8 0.1830	58 0.1696	D9 0.2046	DA 0.1668	01 0.1747	E4 0.1671	7A 0.1740	7D 0.1775	99 0.1696	05 0.1765	BD 0.1857	96 0.1765
7	9F 0.1825	41 0.1845	5A 0.1864	36 0.1760	98 0.1783	E0 0.1674	64 0.1995	6C 0.1657	32 0.1719	C8 0.1634	56 0.1722	B7 0.1725	A7 0.1688	12 0.1630	B3 0.1816	02 0.1746
8	46 0.1791	75 0.1766	7E 0.1858	C6 0.1738	45 0.1778	5F 0.1671	20 0.1971	93 0.1644	28 0.1699	9A 0.1616	5A 0.1711	AE 0.1668	D4 0.1677	A2 0.1604	B9 0.1813	09 0.1713
9	F6 0.1756	6D 0.1747	5E 0.1858	18 0.1722	63 0.1766	BD 0.1663	42 0.1966	3A 0.1643	33 0.1692	CA 0.1608	C8 0.1704	58 0.1608	B9 0.1672	DC 0.1599	F9 0.1811	40 0.1712
10	64 0.1754	C6 0.1725	A3 0.1783	A5 0.1697	03 0.1756	90 0.1661	87 0.1952	51 0.1642	1B 0.1667	3D 0.1591	3D 0.1704	83 0.1607	35 0.1662	0C 0.1590	40 0.1808	49 0.1706



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE	201	87	49	195	145	140	231	80	238	146	220	216	29	249	86	151
0	42 0.0727	B4 0.0755	B8 0.0690	EF 0.0692	51 0.0683	69 0.0685	55 0.0728	7F 0.0686	7B 0.0773	CB 0.0727	78 0.0672	1D 0.0729	D8 0.0696	F6 0.0689	E2 0.0691	73 0.0720
1	20 0.0704	60 0.0679	48 0.0682	CF 0.0687	B3 0.0680	C1 0.0682	0F 0.0697	CB 0.0673	DF 0.0705	B5 0.0699	1E 0.0643	B9 0.0723	C1 0.0676	A1 0.0673	9C 0.0682	43 0.0704
2	F0 0.0695	1E 0.0649	CE 0.0680	A0 0.0676	B7 0.0679	7E 0.0647	06 0.0689	EB 0.0665	96 0.0652	22 0.0676	44 0.0642	0E 0.0686	0C 0.0664	AA 0.0666	01 0.0676	B6 0.0701
3	19 0.0682	44 0.0648	E5 0.0679	25 0.0676	A1 0.0672	99 0.0644	4A 0.0669	34 0.0658	06 0.0651	17 0.0643	C5 0.0638	85 0.0671	E6 0.0663	89 0.0647	39 0.0672	24 0.0686
4	2F 0.0680	84 0.0646	7A 0.0663	92 0.0672	4F 0.0671	D9 0.0634	60 0.0669	FA 0.0645	EE 0.0645	7D 0.0641	1F 0.0636	21 0.0667	78 0.0645	65 0.0639	F9 0.0666	D3 0.0686
5	44 0.0675	D4 0.0640	FF 0.0660	D4 0.0665	82 0.0670	D2 0.0633	C2 0.0655	A7 0.0640	8E 0.0639	EE 0.0638	DA 0.0635	3D 0.0667	BE 0.0644	5D 0.0636	86 0.0655	B7 0.0681
6	57 0.0674	2B 0.0636	7E 0.0654	84 0.0657	E4 0.0648	49 0.0623	13 0.0640	05 0.0638	70 0.0630	2A 0.0629	2B 0.0627	73 0.0657	41 0.0638	74 0.0634	92 0.0651	EE 0.0662
7	17 0.0665	7B 0.0635	57 0.0634	B0 0.0654	BD 0.0645	45 0.0621	93 0.0638	1E 0.0627	1B 0.0627	37 0.0628	46 0.0627	3E 0.0650	CC 0.0635	51 0.0630	D3 0.0645	C3 0.0662
8	C3 0.0661	63 0.0629	FC 0.0627	71 0.0638	75 0.0644	6F 0.0618	63 0.0635	AF 0.0625	94 0.0624	00 0.0623	E5 0.0626	EF 0.0649	A1 0.0632	B9 0.0629	F6 0.0642	D6 0.0661
9	79 0.0659	96 0.0626	9A 0.0620	C5 0.0632	21 0.0643	6C 0.0618	BB 0.0631	AB 0.0624	17 0.0624	C2 0.0615	FA 0.0620	47 0.0640	9D 0.0629	CD 0.0629	FE 0.0635	BF 0.0647
10	0D 0.0627	5F 0.0623	D3 0.0619	F4 0.0631	48 0.0640	C9 0.0618	9E 0.0628	DA 0.0619	AD 0.0616	AA 0.0615	D0 0.0620	B1 0.0638	34 0.0627	01 0.0628	E3 0.0630	A7 0.0646



4. Evaluation | FACE-LIGHT

- ARX 연산을 사용하는 LEA에 비해 **매우 빠른 마스킹 속도**
- 기존의 Masked AES 연구 결과에 비해 성능 향상
  - **FACE-LIGHT, 소프트웨어 최적화**

LEA-128 [19]	Masked LEA-128 [20]	**Masked AES-128 (This work)**
2,688	36,589	**6,219**

4. Evaluation | 128bit Binary Multiplication

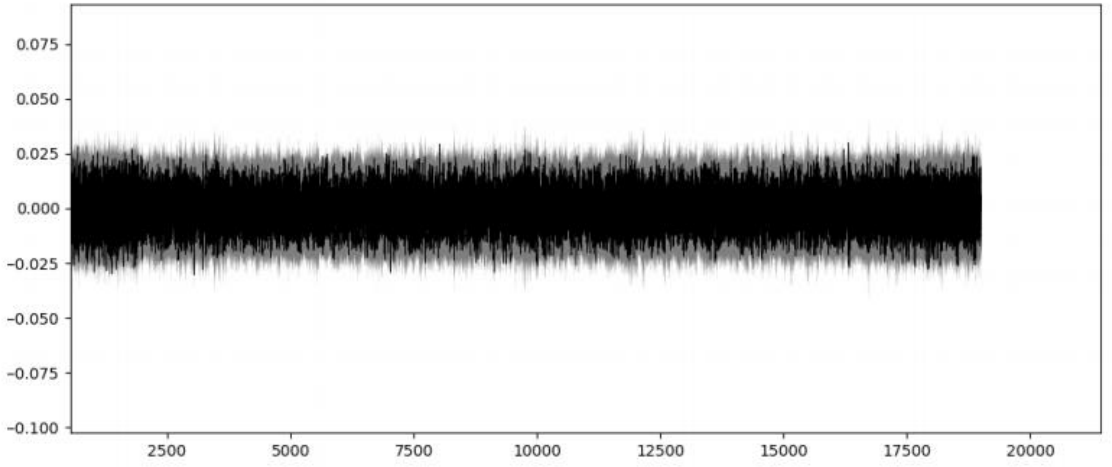
- 기존 연구에 비해 31% 성능 향상
- 기존 연구들과 비교 했을 때 가장 빠름

Bit	Method	Seo and Kim [10]	**This work**
32-bit	Proposed Block-Comb	490	**478**
64-bit	Proposed Karatsuba Block-Comb (Level 1)	1,330	**1,188**
128-bit	Proposed Karatsuba Block-Comb (Level 2)	5,675	**3,896**

Contributor	Method	TA/SPA Security	Timing (cc)
Liu et al. [15]	Karatsuba + Masked Block Comb	TA only	**14,878**
Seo and Kim [10]	Karatsuba + Block-Comb with Dummy XOR and ILA	TA/SPA both	**7,162**
This Work	Karatsuba + Block Comb with Dummy XOR and ILA	TA/SPA both	**4,230**

4. Evaluation | PAGE

- CPA에 취약한 기존 연구에 비해 Masking을 통한 CPA 방어
- 기존 AES-GCM 구현 (Ziu et al)에 비해 평균 44% 성능 향상



Algorithm	16-byte	64-byte	1024-byte
AES-GCM-128	807	510	415
AES-GCM-192	868	548	466
AES-GCM-256	928	586	477

Algorithm	16 bytes	64 bytes	1024 bytes
A: Proposed masked GCM	1,378	850	686
B: Ziu et al	2,047	1,242	990
overhead ration: (A-B)/B	0.48	0.46	0.44

5. Conclusion

- AES-GCM 최적화
- 암호화 부분인 AES-CTR을 FACE-LIGHT 기법을 통해 최적화
- 메시지 인증 부분의 128bit Multiplication을 최적화

Q & A