Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.1, Feb. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.1.41

다양한 환경에서의 형태보존 암호 FEA에 대한 최적 구현*

박 철 희,† 정 수 용, 홍 도 원,‡ 서 창 호
공주대학교

Optimal Implementation of Format Preserving Encryption Algorithm FEA in Various Environments*

Cheolhee Park,† Sooyong Jeong, Dowon Hong,‡ Changho Seo
Kongju National University

요 약

형태보존 암호는 평문의 형태를 그대로 보존하여 암호화를 수행한다. 따라서 암호화 전·후에 대한 데이터베이스의 구조 변경을 최소화 시킬 수 있다. 예를 들어 신용카드, 주민등록번호와 같은 데이터에 대한 암호화를 수행할 경우 형태보존 암호는 평문과 동일한 형태의 암호문을 출력하기 때문에 형태가 갖춰진 기존의 데이터베이스 구조를 그대로 유지할 수 있다. 현재 미국표준기술연구소(NIST)는 형태보존 암호의 표준으로써 FF1과 FF3를 권장하고 있다. 최근 국내의 경우 매우 효율적인 형태보존 암호 알고리즘인 FEA를 형태보존 암호의 표준으로 채택하였다. 본 논문에서는 대한민국 형태보존 암호의 표준 알고리즘인 FEA를 분석하고 다양한 환경에서 최적 구현하여 FEA의 성능을 측정한다.

ABSTRACT

Format preserving encryption(FPE) performs encryption with preserving the size and format of plain-text. Therefore, it is possible to minimize the structural change of the database before and after the encryption. For example, when encrypting data such as credit card number or social security number, it is possible to maintain the existing database structure because FPE outputs the same form of cipher-text as plain-text. Currently, the National Institute of Standards and Technology (NIST) recommends FF1 and FF3 as standards for FPE. Recently, in Korea, FEA, which is a very efficient FPE algorithm, has been adopted as the standard of FPE. In this paper, we analyze FEA and measure the performance of FEA by optimizing it in various environments.

Keywords: Format preserving encryption, FEA, SIMD, NEON

I. 서 론

암호화 알고리즘은 개인정보와 데이터의 보안을 위해 반드시 필요한 암호 프리미티브이며 암호화 방식, 입력 및 출력 등에 따라 다양한 알고리즘이 존재한다. 그중 형태보존 암호는 평문의 형태를 그대로 보존하여 암호화를 수행하는 암호화 방식을 말한다. 형태보존 암호화 방식은 입력 메시지의 형태가 그대로 보존된 암호문을 출력하기 때문에 암호화 전·후에 대한 데이터베이스의 구조 변경을 최소화시킬 수 있다. 예를 들어 신용카드 또는 주민등록번호와 같이 주어진 형태가 존재하는 데이터의 암호화를 수행할

Received(09. 29. 2017), Modified(11. 13. 2017), Accepted(11. 13. 2017)
* 본 논문은 2016년도 정부(과학기술정보통신부/교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(2016R1A4A1011761, 2016R1D1A1B03931071).
† 주저자, newpch89@kongju.ac.kr
‡ 교신저자, dwhong@kongju.ac.kr(Corresponding author)

경우, 형태보존 암호는 입력 메시지의 형태 및 크기가 동일한 암호문을 출력하기 때문에 형태가 갖춰진 기존의 데이터베이스 구조를 그대로 유지할 수 있다(출력 크기가 고정된 AES와 같은 블록암호를 통해 암호화를 수행할 경우 입력과 출력의 형태 및 크기가 보존되지 않는다). 이러한 형태보존 암호의 필요성이 대두되고 있으며 미국표준기술연구소(NIST)는 형태보존 암호의 표준 알고리즘으로써 FF1과 FF3를 권장하고 있다[1-3]. 최근 국내의 경우 매우 효율적인 형태보존 암호 알고리즘인 FEA를 제안하여 이를 대한민국 TTA 표준으로 채택하였다[4,5]. 본 논문에서는 대한민국 형태보존 암호의 표준 알고리즘인 FEA를 분석하고 다양한 환경에서 최적 구현하여 FEA의 성능을 측정한다.

II. 형태보존 암호 알고리즘 FEA[4][5]

2015년 대한민국 한국정보통신기술협회(TTA) 표준으로 재정된 형태보존 암호 알고리즘 FEA는 평문의 형태를 그대로 보존하도록 암호화를 수행하며 암호화 방식에 따라 FEA-1과 FEA-2로 나뉜다. Feistel 구조를 갖는 FEA는 8bit ~ 128bit ($[2^8, 2^{128}]$) 크기의 평문과 길이 128, 192 또는 256bit의 비밀키를 입력으로 하여 평문과 형태 및 크기가 동일한 암호문을 출력한다. 또한 FEA는 타입에 따라 FEA-1과 FEA-2로 나뉘며 트윅의 크기 및 라운드 수에 차이를 갖는다. FEA-1의 경우 평문의 크기가 n일 때 트윅의 크기 $|T|=128-n$이며, 키 길이 128, 192 및 256bit에 대하여 각각 12, 14 및 16 라운드를 반복한다. FEA-2의 경우 트윅의 크기 $|T|=128$이며, 키 길이 128, 192 및 256bit에 대하여 각각 18, 21 및 24 라운드를 반복한다.

2.1 FEA의 암·복호화 함수

FEA는 키 스케줄함수, 트윅 스케줄 함수 및 암호화 함수로 이루어지며, 라운드 키를 역순으로 입력하여 암호화 함수로 복호화를 수행한다. FEA의 암호화 과정은 Fig. 1과 같이 Feistel 라운드 구조를 반복 수행한다. 이때 X_a와 X_b는 각각 평문의 왼쪽 및 오른쪽을 나타내며 평문의 크기를 n이라고 할 때 $|X_a|=\lceil n/2 \rceil$ 이며 $|X_b|=\lfloor n/2 \rfloor$ 이다. 또한 T_a와 T_b는 라운드 트윅을 나타내고 RK_a와 RK_b는 라운드 키를 나타내며 이에 대한 설명은 트윅 스케줄 과정 및 키 스케줄 과정에서 자세히 설명한다.

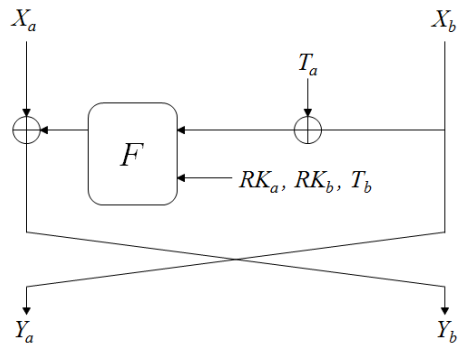

Fig. 1. Round structure of FEA

입력 값의 치환과 확산을 수행하는 라운드 함수 F는 Fig. 2와 같다. 라운드 함수의 치환 계층을 수행하는 S-box는 각각 8bit의 값을 입력받아 8bit의 값을 출력한다[4,5]. 즉, 입력 값 상위 4bit와 하위 4bit를 각각 행과 열의 색인으로 하여 주어진 S-box 테이블에 대응되는 값을 출력으로 한다.

라운드 함수의 확산 계층을 수행하는 M은 이진 유한체 상의 8차 기약 다항식 $p(t)=t^8+t^6+t^5+t^4+1$을 이용하여 정의된 유한체 $GF(2^8)$ 상의 선형 함수이며 Table 1과 같이 정의된 8×8 행렬 m과의 곱으로 표현할 수 있다. m의 i행 j열 원소를 $m(i,j)$라고 하고 $(0\le i,\ j\le 7)$, $X_0, \cdots, X_7$이 8 비트 값들이라면 $DL(X_0 \| \cdots \| X_7)$의 상위 i번째 바이트는 다음과 같다.

$$\oplus^7{}_{j=0} m(i,j) \cdot X_j \ .$$

이때 바이트 곱셈 $\cdot$은 유한체 $GF(2^8)$ 상의 곱셈이다. 즉, C_p를 $p(t)$에 대응하는 8 비트 값(16진수)인 0x71이라고 하면, 두 개의 8 비트 값 $a=a_7\cdots a_0$와 b에 대한 곱셈 연산의 결과는 다음과 같다.

$$a \cdot b = a_0 b \oplus a_1 mul2(b) \oplus a_2 mul2^2(b) \oplus \cdots \oplus a_7 mul2^7(b) \ .$$

여기서 $mul2$는 8 비트 입출력을 가지는 함수로, $x=x_7\cdots x_0$일 때 다음과 같이 정의한다.

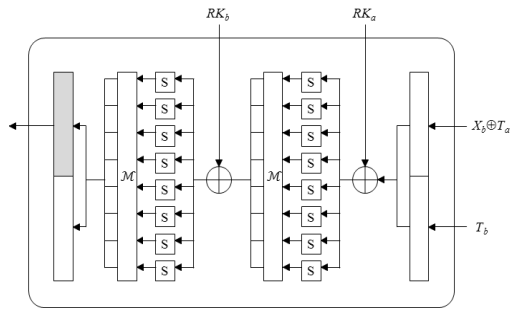

Fig. 2. Round function F

Table 1. Matrix m

$$m = \begin{bmatrix} 28 & 1a & 7b & 78 & c3 & d0 & 42 & 40 \\ 1a & 7b & 78 & c3 & d0 & 42 & 40 & 28 \\ 7b & 78 & c3 & d0 & 42 & 40 & 28 & 1a \\ 78 & c3 & d0 & 42 & 40 & 28 & 1a & 7b \\ c3 & d0 & 42 & 40 & 28 & 1a & 7b & 78 \\ d0 & 42 & 40 & 28 & 1a & 7b & 78 & c3 \\ 42 & 40 & 28 & 1a & 7b & 78 & c3 & d0 \\ 40 & 28 & 1a & 7b & 78 & c3 & d0 & 42 \end{bmatrix}$$

$$mul2(x) = (x \ll 1) \oplus x_7 C_p \ .$$

2.2 FEA의 키 스케줄

FEA의 키 스케줄 과정은 Fig. 3과 같이 k-bit 비밀키 K와 평문 비트 길이 n을 입력으로 하며 라운드 수 만큼의 128bit 라운드 키들을 출력한다(키 스케줄의 치환과 확산 과정은 암호화 함수에서의 치환, 확산 과정과 동일함). 이때 4개의 64 비트 값 K_a, K_b, K_c, K_d는 128, 192, 또는 256 비트 비밀키가 사용되었을 때의 키 스케줄을 일괄적으로 서술하는데 사용된다. K_a, K_b, K_c, K_d의 초기 값은 비밀 키 길이에 따라서 다음과 같이 설정된다.

- K가 128 비트일 때 $K_a \| K_b = K$이고, $K_c = K_d = 0^{64}$
- K가 192 비트일 때 $K_a \| K_b \| K_c = K$이고, $K_d = 0^{64}$
- K가 256 비트일 때 $K_a \| K_b \| K_c \| K_d = K$

키 스케줄은 각 라운드에서 두 개의 라운드 키를 출력한다. 따라서 라운드 수를 r이라고 할 때 Fig. 3에 나타난 과정을 $\lceil r/2 \rceil$ 번 반복한다. 각 반복 과정에서 사용되는 라운드 상수 RC는 타입, 키 길이 및 수행 라운드에 따라 주어진 값을 사용한다[4,5].

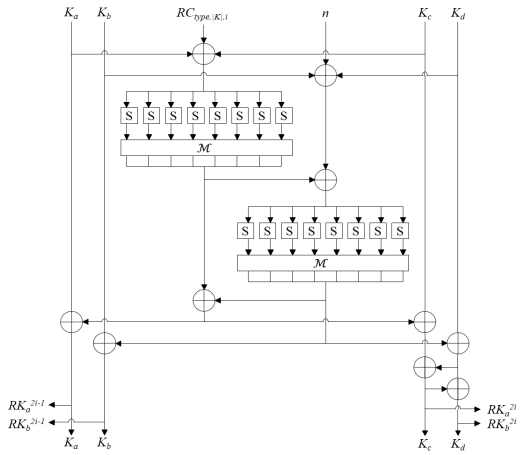

Fig. 3. key schedule round

2.3 FEA의 트윅 스케줄

FEA의 트윅 스케줄은 타입에 따라 제 1형과 제 2형으로 구분되며 서로 다른 과정을 거친다. 트윅 스케줄을 통해 출력된 라운드 트윅은 암호화 과정과 복호화 과정에서 순서만 변경하여 사용한다.

2.3.1 제1형 트윅 스케줄

FEA-1은 평문의 크기를 n이라고 할 때 $(128-n)$-bit 크기의 트윅 T를 사용하며 암호화 과정에서 각 라운드 마다 트윅 스케줄의 출력인 T_a, T_b를 사용한다. 이때 T_a는 0이며 T_b는 홀수 라운드의 경우 T의 상위 $(64-|X_a|)$-bit를 사용하며 나머지 bit들을 짝수 라운드의 라운드 트윅으로 사용한다.

2.3.2 제2형 트윅 스케줄

FEA-2은 128bit 크기의 트윅 T를 사용하며 암호화 과정에서 각 라운드 마다 트윅 스케줄의 출력인 T_a, T_b를 사용한다. 라운드 수를 r이라고 할 때 $r \equiv 1 \bmod 3$ 번째 라운드는 T_a, T_b 모두 0이며 $r \equiv 2 \bmod 3$ 번째 라운드에서는 T의 왼쪽 절반을 $T_a \| T_b$로 설정한다. 또한 $r \equiv 0 \bmod 3$ 번째 라운드에서는 T의 오른쪽 절반을 $T_a \| T_b$로 설정한다.

2.4 FEA의 직렬 구현

FEA 알고리즘의 직렬 구현은 크게 초기화 과정, 키 스케줄 과정 및 암호화 과정으로 이루어지며 키 스케줄 및 암호화 함수는 S-box look-up 과정 및 확산 계층에서의 Galois field 곱셈 연산을 포함한다.

초기화 과정은 우선 입력된 메시지 배열에 대하여 메시지 크기를 계산하고 메시지의 오른쪽 및 왼쪽 부분을 분할한 뒤 서로 다른 두 개의 64bit 변수에 각각 대입한다. 또한 초기화 과정에서는 입력된 트윅 배열에 대하여 트윅 스케줄을 수행한다. 초기화 과정은 매우 빠른 속도로 수행되며 타 함수의 수행시간에 비해 무시할 만큼 작다.

키 스케줄 및 암호화 과정에서는 Xor연산, S-box look-up 과정 그리고 Galois field 곱셈연산을 수행한다. 이때 64bit 변수에 대한 한 번의 확산 함수 호출은 64번의 Galois field 곱셈 연산을 수행하기 때문에 FEA 알고리즘 내에서 가장 많은 시간을 소비한다. FEA의 확산 계층은 8bit 크기의 모든 입력 값 0x00 ~ 0xFF에 대하여 0x28, 0x1A, 0x7B, 0x78, 0xC3, 0xD0, 0x42 그리고 0x40과 Galois field 곱셈 연산을 수행한다. 따라서 256개의 원소로 구성된 8개의 배열을 사전에 계산하여 확산 계층에 대한 look-up table로써 활용할 수 있다. 이 경우 치환 계층을 확산 계층에 포함시킬 수 있으며 S-box look-up과정을 생략할 수 있다. 본 논문에서는 치환 및 확산 계층에 대한 통합적인 look-up table을 구성하여 직렬 코드와의 수행 속도를 함께 비교하였다. 이때 해당 look-up table은 총 256×8 byte의 메모리를 필요로 하며, 메모리공간과 속도의 trade-off로써 직렬 코드와 look-up table 적용 코드를 고려할 수 있다.

```
1:    uint8_t GF_mul(u8 a, u8 b)
2:    {
3:        uint8_t result = 0, t;
4:        while (a != 0)
6:        {
7:            if ((a & 1) != 0)
8:                result ^= b;
9:            t = (b & 0x80);
10:           b <<= 1;
11:           if (t != 0)
12:               b ^= 0x71;
13:           a >>= 1;
14:       }
15:       return result;
16:   }
```

Fig. 4. Serial C language source code for Galois field multiplication

III. SIMD를 지원하는 고사양 디바이스 상에서의 FEA 병렬 구현

이번 장에서는 SIMD 구현이 가능한 고사양 디바이스 상에서 FEA의 병렬 구현을 설명한다. SIMD는 Single Instruction Multiple Data의 약자로 한 번의 명령으로 여러 개의 데이터를 동시에 처리하는 방식을 말한다. FEA는 전체적으로 64bit 단위로 연산을 수행하기 때문에 FEA의 알고리즘 내부 과정을 병렬로 처리할 경우 최대 4개의 독립적인 변수를 동시에 처리할 수 있다(256bit 레지스터를 활용한 경우). 그러나 FEA 내부 과정에서 64bit 단위의 연산이 독립적으로 수행되는 부분은 드물다. 또한 해당 부분을 병렬로 처리할지라도 메모리로부터 레지스터로, 레지스터로부터 메모리로 할당되는 데이터의 이동이 오버헤드를 발생시키기 때문에 64bit 연산에 대한 병렬적용의 성능향상효과는 매우 낮다.

8bit 단위로 연산이 수행되는 확산 계층은 서로 독립적인 연산을 수행하며 한 번의 확산 함수 호출은 64번의 Galois field 곱셈 연산을 수행한다. 해당 과정은 FEA의 전체 과정에서 가장 많은 연산을 수행하며 FEA 암호화 과정의 전체 소비시간 중 대부분을 차지한다. 따라서 이번 장에서는 FEA의 Galois field 곱셈 연산에 대한 병렬을 고려하여 FEA를 최적 구현한다(치환 계층의 경우 단순히 look-up table을 참조하기 때문에 병렬을 고려하지 않는다).

3.1 Intel(R) Core(TM) i7-4790 상에서 AVX2를 활용한 FEA의 병렬 구현

FEA의 Galois field 곱셈 연산을 나타내는 Fig. 4는 우선 첫 번째 변수인 a의 최하위 bit를 판단하여 1인 경우 두 번째 변수 b와 0으로 초기화 된 결과 값에 대하여 Xor연산을 수행하고 0인 경우 해당 과정을 통과한다. 그 후 두 번째 변수 b의 최상

Table 2. Used AVX2 commands

AVX2 command	Explanation
vmovdqu	Load 256bit data (memory ⇆ register)
vpbroadcastb	Broadcast 8bit data to all pack in a register
vpxor	Compute the bitwise XOR of 256bit registers
vpand	Compute the bitwise AND of 256bit registers
vpcmpgtb	Compare packed 8bit integers for greater-than, and store 0xFF
vpsrlw (vpsllw)	Shift packed 16bit integers in a right(left)

위 비트를 판단하고(이때 변수 t를 사용하여 입력 변수 b의 최상위 비트를 저장) b를 왼쪽으로 1만큼 이동시킨다. 이때 b의 최상위 비트가 1인 경우 왼쪽으로 1만큼 이동된 b와 $p(t)=t^8+t^6+t^5+t^4+1$의 16진수 값인 0x71에 대하여 Xor연산을 수행한다(b mod $p(t)$). 마지막으로 첫 번째 변수 a를 오른쪽으로 1만큼 이동시키면서 위의 과정을 a가 0이 될 때까지 반복한다(최대 8번 반복).

8bit 크기의 단일 변수에 대한 연산의 흐름인 Fig. 4를 병렬 코드로 구현하기 위해 다음과 같은 사항을 고려한다.

1) 32개의 모든 변수들에 대한 최하위 비트의 판단
2) 각각 32개의 변수를 포함하는 2개의 레지스터에 대한 Xor 연산
3) 32개의 모든 변수들에 대한 최상위 비트의 판단
4) 32개의 모든 변수들에 대한 크기 1 만큼의 bit 이동(왼쪽 및 오른쪽)

[1]위의 4가지 사항을 고려하여 FEA의 Galois field 곱셈 연산에 대한 병렬을 구현하였으며 Table 2는 사용된 AVX2 어셈블리 명령어를 나타낸다. 최상위 및 최하위 비트를 판단하여 Xor을 수행하기 위해 AVX2 명령어 vpbroadcastb, vpand 그리고 vpcmpgtb를 활용하였다. 즉, 32개 8bit 변수들의 최상위(또는 최하위) 비트를 판단하기 위해 우선 레

1) 해당 구현은 AVX2 어셈블리로 구현되었음

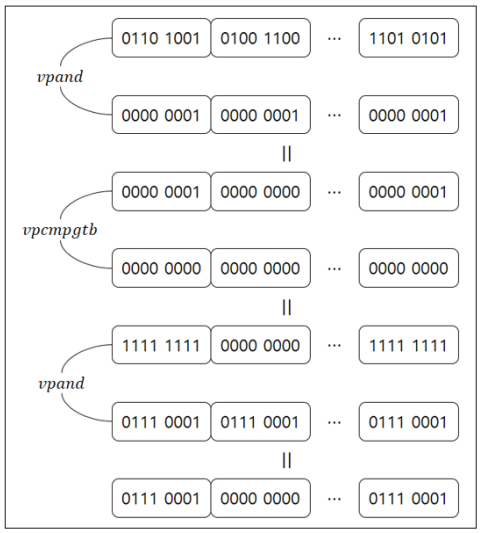

Fig. 5. Example of AVX2 parallel flow for conditional statements in Galois field multiplication

```
1:  _asm {
2:      vmovdqu ymm0, X1;
3:      vpbroadcastq ymm1, X2;
4:      vpbroadcastb ymm2, a;
5:      vpand ymm3, ymm2, ymm0;
6:      vpcmpgtb ymm3, ymm3, ymm5;
7:      vpand ymm3, ymm3, ymm1;
8:  }
```

Fig. 6. Assembly code for Fig.5

지스터에 0x80(또는 0x01)을 브로드캐스트[2](해당 과정은 Fig. 6의 줄 4를 나타내며 변수 a는 0x80 또는 0x01이다. 추가적으로 Fig. 6의 줄 2 ~ 3에서 X1은과 X2는 Fig. 4의 변수 a와 b에 대한 병렬 레지스터를 구성하는 과정을 나타낸다.) 한 후 0x80(또는 0x01)로 채워진 레지스터와 변수들로 채워진 레지스터에 대하여 and(vpand) 연산을 수행한다(Fig. 6의 줄 5). 따라서 모든 변수들은 최상위(또는 최하위) 비트만을 남기게 된다. 그 후, Fig. 4의 줄 14 ~ 15 및 줄 10 ~ 11 에 해당하는 Xor 연산을 병렬로 수행하기 위해 최상위(또는 최하위) 비트들만이 남겨진 레지스터와 제로벡터를 비교하여(Fig. 6의 줄 6) 최상위(또는 최하위) 비트가 1인

2) 크기 32의 8bit 배열을 사전에 정의하여 대입연산을 통해 수행될 수 있지만 해당 과정은 32개의 메모리 주소를 참조하기 때문에 1개의 메모리 주소를 참조하는 브로드캐스트 함수보다 비효율적이다.

Table 3. Performance comparison of FPE algorithm in Intel(R) Core(TM) i7-4790

Method / Algorithm	1) Serial code				2) SIMD (AVX2)	
	Naive code		look-up table applied			
	key schedule	encryption	key schedule	encryption	key schedule	encryption
FEA-1-128	0.007 ms	0.014 ms	0.0014 ms	0.0028 ms	0.0007 ms	0.0014 ms
FEA-1-192	0.009 ms	0.016 ms	0.0016 ms	0.0032 ms	0.0009 ms	0.0016 ms
FEA-1-256	0.010 ms	0.019 ms	0.0019 ms	0.0037 ms	0.0010 ms	0.0018 ms
FEA-2-128	0.011 ms	0.019 ms	0.0021 ms	0.0043 ms	0.0011 ms	0.0021 ms
FEA-2-192	0.013 ms	0.023 ms	0.0028 ms	0.0052 ms	0.0014 ms	0.0025 ms
FEA-2-256	0.013 ms	0.026 ms	0.0028 ms	0.0057 ms	0.0015 ms	0.0028 ms

pack 들을 0xFF로 변경하고, $p(t)$의 16진수인 0x71(또는 두 번째 변수들)로 구성된 레지스터와 and연산을 수행한다(Fig. 6의 줄 7). 이 과정은 최상위(또는 최하위) 비트가 1인 pack들에 대해서만 0x71(또는 두 번째 변수들)을 대입하기 위한 과정이다(Fig. 5). 이 과정에 대해 AVX2 곱셈연산을 고려 할 수 있지만 AVX2의 경우 곱셈연산은 최소 16bit부터 지원하기 때문에 8bit 크기의 데이터를 다루는 본 과정에서는 곱셈연산을 사용하지 않는다. 이때 두 번째 변수인 var2는 주어진 8×8 행렬 m의 원소들이며 두 번째 변수들을 레지스터에 대입(vmovdqu)하는 과정은 사전에 수행된다(0x71의 경우 브로드캐스트). Fig. 4에 나타난 비트 단위 이동연산을 병렬로 수행하기 위해 AVX2 명령어 vpsrlw(vpsllw)를 활용하였다. AVX2의 비트 이동연산은 최소 16bit 단위로 수행되기 때문에 8bit 크기의 변수를 다루는 FEA의 과정에서는 추가적인 연산을 필요로 한다. 해당 과정은 Fig. 7와 같이

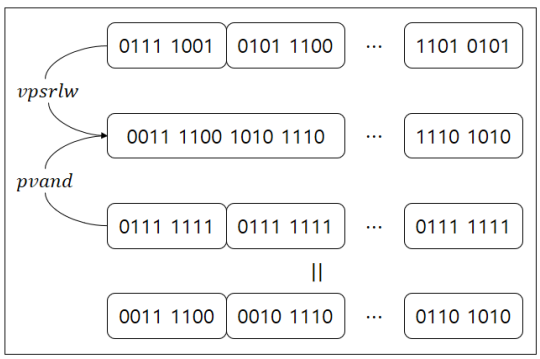

Fig. 7. Example of 8bit unit 1-right shift in AVX2

vpsrlw(vpsllw)를 활용하여 16bit 단위로 이동연산을 수행한 후(Fig. 8의 줄 2) 왼쪽 이동연산의 경우 0xFE(오른쪽의 경우 0x7F)로 구성된 레지스터와 and 연산을 수행한다(Fig. 8의 줄 3 ~ 4는 해당 과정을 나타내며 변수 a는 0x7F를 나타낸다.). 즉, 16bit 크기로 비트이동 연산을 수행한 후 8bit 단위의 최하위(또는 최상위) 비트를 0으로 치환함으로써 8bit 크기의 비트이동을 수행한다. FEA는 한 번의 확산 계층에서 직렬 구현의 경우 Galois field 곱셈 연산을 총 64번 수행하지만 병렬 구현의 경우 2번의 Galois field 곱셈 연산만으로 확산 계층을 완료시킬 수 있다.

Table 3은 PC환경인 Intel(R) Core(TM) i7-4790 상에서 FEA의 직렬구현 코드와 AVX2 병렬구현 코드의 성능을 비교한 비교표이며 각각의 과정을 100,000번 반복 수행한 후 측정된 시간을 100,000으로 나누어 나타내었다. 이때 직렬 코드는 Ubuntu Desktop 운영체제상에서 GCC 컴파일러를 통해 성능을 측정하였으며 AVX2를 활용한 병렬 코드의 경우 Windows 10 64bit 운영체제상에서 Visual studio 2015 컴파일러를 통해 성능을 측정하였다. 또한 메시지의 크기는 시간 측정 결과에 영향을 미치지 않기 때문에(실질적으로 무시할 만큼 작은 영향을 미침) 63bit 크기에 대해서만 측정을 수

```
1:  _asm {
2:      vpsrlw ymm0, ymm0, 1;
3:      vpbroadcastb ymm1, a;
4:      vpand ymm0, ymm0, ymm1;
5:  }
```

Fig. 8. Assembly code for Fig. 7

Table 5. Performance comparison of FPE algorithm in Intel(R) Core(TM) i5-4590

Method / Algorithm	1) Serial code				2) SIMD (SSE2)	
	Naive code		look-up table applied			
	key schedule	encryption	key schedule	encryption	key schedule	encryption
FEA-1-128	0.008 ms	0.016 ms	0.0016 ms	0.0032 ms	0.0011 ms	0.0021 ms
FEA-1-192	0.009 ms	0.017 ms	0.0017 ms	0.0035 ms	0.0012 ms	0.0024 ms
FEA-1-256	0.010 ms	0.020 ms	0.0020 ms	0.0041 ms	0.0014 ms	0.0028 ms
FEA-2-128	0.011 ms	0.021 ms	0.0024 ms	0.0046 ms	0.0016 ms	0.0032 ms
FEA-2-192	0.013 ms	0.025 ms	0.0030 ms	0.0057 ms	0.0020 ms	0.0037 ms
FEA-2-256	0.015 ms	0.028 ms	0.0031 ms	0.0062 ms	0.0022 ms	0.0042 ms

행하였다.

직렬 코드의 경우 동일한 키 길이에서 look-up table이 적용된 구현은 순수 직렬코드에 비해 약 5배 정도 효율적인 성능을 보였으며 병렬 구현의 경우 직렬 구현 보다 약 10배, look-up table이 적용된 구현 보다 약 2배 정도 효율적인 성능을 보였다. 추가적으로, 병렬 구현된 FEA 알고리즘에 대하여 10,000,000 번의 암호화를 수행한 경우 약 3초 정도 소요되었다.[3)]

3.2 Intel(R) Core(TM) i5-4590 상에서 SSE2를 활용한 FEA의 병렬 구현

SSE2는 128bit 크기의 레지스터를 활용하기 때문에 최대 16개의 데이터를 동시에 처리할 수 있다. 이번 절에서는 AVX2를 활용한 병렬 구현과 마찬가지로 SSE2 어셈블리를 활용하여 Galois field 곱셈 연산을 병렬 처리하였으며 Table 4는 사용된 SSE2 명령어를 나타낸다. 사용된 명령어의 역할은 앞선 AVX2의 구현에서와 동일하지만 128bit 레지스터에 대한 연산을 수행한다. 또한 SSE2의 경우 브로드캐스트 함수를 지원하지 않기 때문에 128bit 크기의 배열을 사전에 정의하여 대입연산을 수행한다.

Table 4. Used SSE commands

SSE command	Explanation
movdqu	Load 128bit data (memory ⇆ register)
pxor	Compute the bitwise XOR of 128bit registers
pand	Compute the bitwise AND of 128bit registers
pcmpgtb	Compare packed 8bit integers for greater-than, and store 0xFF
psrlw (psllw)	Shift packed 16bit integers in a right(left)

Table 5는 PC환경인 Intel(R) Core(TM) i5-4590 상에서 FEA의 직렬구현 코드와 SSE 병렬구현 코드의 성능을 비교한 비교표이며 3.1절과 동일한 방식으로 성능을 측정하였다. 이때 직렬 코드는 Ubuntu Desktop 운영체제상에서 GCC 컴파일러를 통해 성능을 측정하였으며 SSE를 활용한 병렬 코드의 경우 Windows 10 64bit 운영체제상에서 Visual studio 2015 컴파일러를 통해 성능을 측정하였다.

직렬 코드의 경우 동일한 키 길이에서 look-up table이 적용된 구현은 순수 직렬코드에 비해 약 5배 정도 효율적인 성능을 보였으며 병렬 구현의 경우 직렬 구현 보다 약 7.5배, look-up table이 적용된 구현 보다 약 1.45배 정도 효율적인 성능을 보였다.

3.3 ARM Cortex-A53 상에서 ARM-NEON을 활용한 FEA의 병렬 구현

이번 절에서는 ARM Cortex-A53 상에서 FEA의 성능을 측정한다. ARM Cortex-A53는 NEON SIMD를 지원하며 128bit 크기의 레지

3) 해당 과정은 AVX2 구현 및 thread 병렬을 통해 수행되었음

Table 7. Performance comparison of FPE algorithm in ARM Cortex-A53

Method / Algorithm	1) Serial code				2) SIMD (NEON)	
	Naive code		look-up table applied			
	key schedule	encryption	key schedule	encryption	key schedule	encryption
FEA-1-128	0.056 ms	0.127 ms	0.010 ms	0.024 ms	0.0038 ms	0.0074 ms
FEA-1-192	0.065 ms	0.145 ms	0.011 ms	0.029 ms	0.0044 ms	0.0086 ms
FEA-1-256	0.076 ms	0.154 ms	0.015 ms	0.031 ms	0.0051 ms	0.0099 ms
FEA-2-128	0.085 ms	0.165 ms	0.016 ms	0.037 ms	0.0057 ms	0.0110 ms
FEA-2-192	0.102 ms	0.191 ms	0.022 ms	0.043 ms	0.0070 ms	0.0128 ms
FEA-2-256	0.111 ms	0.220 ms	0.024 ms	0.050 ms	0.0077 ms	0.0147 ms

스터를 활용한다.

[4]AVX2 및 SSE2의 구현과 마찬가지로 NEON SIMD를 활용하여 Galois field 곱셈 연산을 병렬 처리하였으며 Table 6은 사용된 NEON intrinsic 명령어를 나타낸다.

ARM-NEON은 SSE 및 AVX와 달리 연산을 수행할 때 사전에 두 개의 레지스터를 동일한 단위의 구성으로 변환해야 한다. 예를 들어 Xor 연산의 경우 첫 번째 레지스터를 64bit 단위의 변수들로 구성하고 두 번째 레지스터를 16bit 단위의 변수들로 구성하였을 때, SSE 나 AVX는 단순히 두 개의 레지스터에 대한 Xor(pxor 또는 vpxor) 연산을 수행할 수 있지만 NEON의 경우 서로 다른 크기의 변수로 이루어진 두 레지스터를 동일한 크기의 변수 구성으로 변환해야만 한다. 함수 vreinterpretq_u8_u64는 해당 과정을 수행하는 함수이며 64bit 변수 구성으로 이루어진 레지스터를 8bit 구성으로 재해석한다. 또한 Galois field 곱셈 연산과정 중 최상위(또는 최하위) 비트가 1인 경우 해당 위치(pack)에 0x71(또는 두 번째 변수 var2)을 대입하는 과정에서 곱셈연산을 고려할 수 있다. SSE나 AVX의 경우 곱셈연산은 최소 16bit 단위부터 지원하기 때문에 곱셈연산을 고려하지 않았지만 NEON의 경우 8bit 단위의 병렬 곱셈 연산이 가능하기 때문에 Fig. 9과 같이 위의 과정을 구현하였다(Fig. 9은 Fig. 5에 해당하는 과정의 NEON 연산 과정임).

Table 6. Used NEON commands

SSE command	Explanation
vld1q_u8	Assign an array of 8bit data to a 128bit register
vdupq_n_u64	Broadcast a 64bit variable to a 128bit register
vreinterpretq_u8_u64	Converts a 128bit register consisting of two 64-bit variables into 16-8bit variables
vandq_u8	Compute the bitwise AND in 8bit unit
veorq_u8	Compute the bitwise XOR in 8bit unit
vmulq_u8	Multiplication operation in 8bit unit
vshrq_u8 (vshlq_u8)	Shift packed 8bit integers in a right(left)

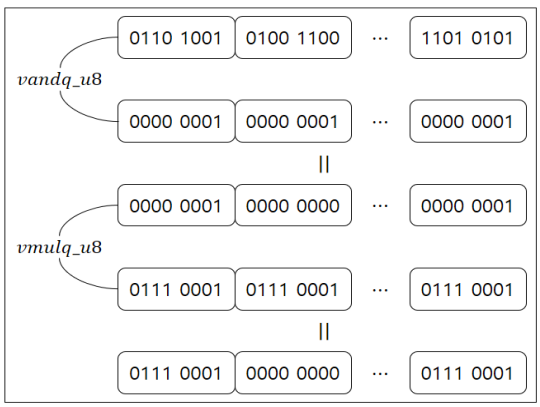

Fig. 9. Example of NEON parallel flow for conditional statements in Galois field multiplication

4) 해당 구현은 NEON intrinsic으로 구현되었음

Table 8. Performance comparison of FPE algorithm in 32-bit ARM core and 8-bit AVR

Environment / Algorithm	1) AT91SAM3X8E (32-bit ARM core)				2) ATmega328/P (8-bit AVR)	
	Naive code		look-up table applied			
	key schedule	encryption	key schedule	encryption	key schedule	encryption
FEA-1-128	1.5 ms	3.1 ms	0.4 ms	0.8 ms	6.0 ms	13.0 ms
FEA-1-192	1.8 ms	3.6 ms	0.4 ms	1.0 ms	7.0 ms	15.0 ms
FEA-1-256	2.0 ms	4.1 ms	0.5 ms	1.2 ms	8.0 ms	17.0 ms
FEA-2-128	2.3 ms	4.6 ms	0.6 ms	1.3 ms	9.0 ms	19.0 ms
FEA-2-192	2.8 ms	5.4 ms	0.7 ms	1.5 ms	12.0 ms	22.0 ms
FEA-2-256	3.0 ms	6.1 ms	0.8 ms	1.7 ms	13.0 ms	25.0 ms

또한 NEON의 경우 8bit 단위의 비트이동 연산이 병렬로 가능하기 때문에 Fig. 7와 같은 추가적인 과정을 생략할 수 있다.

Table 7은 ARM Cortex-A53 상에서 FEA의 직렬구현 코드와 NEON 병렬구현 코드의 성능을 비교한 비교표이며 3.1절과 동일한 방식으로 성능을 측정하였다. 이때 모든 측정은 Ubuntu MATE 운영체제상에서 GCC 컴파일러를 통해 성능을 측정하였다.

직렬 코드의 경우 동일한 키 길이에서 look-up table이 적용된 구현은 순수 직렬코드에 비해 약 5.2배 정도 효율적인 성능을 보였다. 병렬 구현의 경우 직렬 구현 보다 약 15배, look-up table이 적용된 구현 보다 약 3배 정도 효율적인 성능을 보였으며 ARM Cortex-A53 환경에서의 NEON 병렬 효율이 PC 환경에서의 SSE 및 AVX2의 병렬보다 효율적으로 측정되었다.

IV. IoT 및 embedded 환경을 고려한 32-bit ARM core Processor와 8-bit AVR Processor에서의 FEA 구현

이번 장에서는 IoT 및 Embedded 환경을 고려하여 저 사양 환경인 AT91SAM3X8E(32-bit ARM core Processor)와 ATmega328/P(8-bit AVR Processor)상에서 FEA를 구현한다.

IoT 와 Embedded 환경을 가정한 AT91SAM3X8E 및 ATmega328/P는 32bit의 작업레지스터 크기를 갖는다. 즉, 위의 환경에서 최대로 선언할 수 있는 변수의 크기는 32bit이다. 그러나 FEA의 경우 전반적으로 64bit 크기의 변수에 대한 연산을 수행하기 때문에 2, 3 및 4장에서 구현된 직렬코드를 위의 환경에 적용하는 것은 불가능하다. 따라서 이번 장에서는 32bit의 작업레지스터 크기를 갖는 저 사양 디바이스를 타겟으로 FEA를 구현한다.

FEA의 내부 과정은 xor과 and 연산으로 수행될 수 있으며 변수 크기에 영향을 미치는 비트단위 이동과 같은 연산은 8bit 크기로 수행되기 때문에 연산에 대한 분할은 고려사항에서 제외된다. 그러나 모든 64bit 변수들을 32bit 변수 두 개로 분할해야만 한다.

Table 8은 AT91SAM3X8E(32-bit ARM core Processor)와 ATmega328/P(8-bit AVR Processor)상에서 위의 고려사항을 적용하여 구현된 FEA에 대한 성능을 나타낸다. 이때 모든 측정은 Arduino IDE 컴파일러를 통해 측정하였다.

AT91SAM3X8E(32-bit ARM core Processor)상에서는 순수 직렬코드와 look-up table이 적용된 코드를 측정하였다. look-up table이 적용된 구현의 경우 순수 직렬코드에 비해 약 2 ~ 3배 정도 효율적인 성능을 보였다. ATmega328/P (8-bit AVR Processor)의 경우 256×8 byte 만큼의 메모리를 할당할 수 없기 때문에 순수 직렬코드에 대한 성능만을 측정하였다.

V. 결론 및 향후 연구

본 논문에서는 대한민국 형태보존 암호의 표준 알고리즘인 FEA를 다양한 환경에서 구현하고 그 성능을 측정하였다. PC 및 모바일 스마트폰을 가정한 환경에서는 FEA의 순수 직렬코드, 치환 및 확산

계층에 대한 통합 look-up table이 적용된 코드 및 확산 계층에 대한 병렬처리 코드를 구현하여 그 성능을 측정하였다. 이때 동일한 환경에서 look-up table이 적용된 코드는 순수 직렬코드보다 매우 효율적으로 측정되었으며 병렬처리코드가 가장 효율적으로 측정되었다.

IoT 와 Embedded 환경을 가정한 AT91SAM3X8E 및 ATmega328/P의 경우에도 FEA 알고리즘은 원활히 수행될 수 있음을 확인하였다. AT91SAM3X8E의 경우 look-up table 적용이 가능하였으며 이때 순수 직렬코드 대비 약 3.8배 정도 효율적임을 확인하였다. ATmega328/P에서는 2KB 크기의 look-up table을 적용하는 것이 불가능하였으며 이를 가능하게 하기 위한 연구를 진행하고 있다.

References

[1] M. Dworkin. "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", NIST Special Publication 800-38G, 2016.

[2] M. Bellare, P. Rogaway, T. Spies, "The FFX Mode of Operation for Format-Preserving Encryption," Draft 1.1. Submission to NIST, Feb. 2010.

[3] E. Brier, T. Peyrin, and J. Stern, "BPS: a formatpreserving encryption proposal," Submission to NIST, 2010.

[4] J.-K. Lee, B. Koo, D. Roh, W.-H. Kim, and D. Kwon, "Format-preserving encryption algorithms using families of tweakable blockciphers," Information Security and Cryptology - ICISC 2014. Lecture Notes in Comput. Sci., vol. 8949, pp. 132-159, 2015

[5] TTAK.KO-12.0275 "Format-Preserving Encryption Algorithm FEA," 2015.

〈저자소개〉

박 철 희 (Cheolhee Park) 학생회원
2014년 2월: 공주대학교 응용수학과 학사
2017년 2월: 공주대학교 수학과 석사
2017년 3월~현재: 공주대학교 수학과 박사 재학
〈관심분야〉 암호모듈 구현, 데이터 보호기술, 프라이버시 보호기술

정 수 용 (Sooyong Jeong) 학생회원
2011년 3월~현재: 공주대학교 응용수학과 학사 재학
〈관심분야〉 암호모듈 구현, 데이터 보안

홍 도 원 (Dowon Hong) 종신회원
1994년 2월: 고려대학교 수학과 학사
2000년 2월: 고려대학교 수학과 박사
2000년 4월~2012년 2월: 한국전자통신연구원 팀장, 책임연구원
2012년 3월~현재: 공주대학교 응용수학과 교수
〈관심분야〉 암호기술, 프라이버시 보호기술

서 창 호 (Changho Seo) 종신회원
1990년: 고려대학교 수학과 학사
1992년: 고려대학교 수학과 이학석사
1996년: 고려대학교 수학과 이학박사
1996년~1996년: 국방과학연구소 선임연구원
1996년~2000년: 한국전자통신연구원 선임연구원, 팀장
2000년~현재: 공주대학교 응용수학과 교수
〈관심분야〉 암호알고리즘, PKI, 무선인터넷 보안 등