JKIICE Journal of the Korea Institute of Information and Communication Engineering

한국정보통신학회논문지(J. Korea Inst. Inf. Commun. Eng.) Vol. 21, No. 10 : 1929~1934 Oct. 2017

ARMv8 상에서 LEA 암호화 고속 구현

서화정*

High Speed Implementation of LEA on ARMv8

Hwa-jeong Seo*

Department of IT Engineering, Hansung University, Seoul 02876, Korea

요 약

경량 블록암호화 (Lightweight Encryption Algorithm, LEA)는 연산의 효율성과 높은 보안성으로 인해 가장 각광받고 있는 블록암호화 알고리듬이다. 해당 블록암호화는 실제 응용프로그램에서도 많이 사용되고 있으며 서비스 가용성을 높이기 위해 연산 성능을 개선하는 연구가 많이 진행되고 있다. 본 논문에서는 최신 ARMv8 프로세서 상에서 LEA 연산을 최적화하는 방안에 대해 제안한다. 구현은 새로운 SIMD 명령어 셋인 NEON을 통해 최적화되었으며 병렬화된 연산을 통해 동시에 24 번의 암호화 연산을 수행하도록 한다. 메모리 접근 횟수를 줄이기 위해 활용가능한 모든 NEON 레지스터에 중간 계산값을 할당하여 활용하였다. 해당 구현 결과는 속도 관점에서 평가되었으며 ARMv8 상에서 LEA 암호 구현은 Apple A7 그리고 Apple A9 프로세서 상에서 각각 2.4 cycles/byte 그리고 2.2 cycles/byte 안에 수행 가능함을 확인할 수 있었다.

ABSTRACT

Lightweight block cipher (Lightweight Encryption Algorithm, LEA) , is the most promising block cipher algorithm due to its efficient implementation feature and high security level. The LEA block cipher is widely used in real-field applications and there are many efforts to enhance the performance of LEA in terms of execution timing to achieve the high availability under any circumstances. In this paper, we enhance the performance of LEA block cipher, particularly on ARMv8 processors. The LEA implementation is optimized by using new SIMD instructions namely NEON engine and 24 LEA encryption operations are simultaneously performed in parallel way. In order to reduce the number of memory access, we utilized the all NEON registers to retain the intermediate results. Finally, we evaluated the performance of the LEA implementation, and the proposed implementations on Apple A7 and Apple A9 achieved the 2.4 cycles/byte and 2.2 cycles/byte, respectively.

키워드 : ARMv8, LEA, NEON, 병렬 구현

Key word : ARMv8, LEA, NEON, Parallel Implementation

Received 11 June 2017, **Revised** 15 June 2017, **Accepted** 02 July 2017

* **Corresponding Author** Hwa-jeong Seo(E-mail:hwajeong@hansung.ac.kr, Tel:+82-2-760-8033)
Department of IT Engineering, Hansung University, Seoul 02876, Korea

Open Access https://doi.org/10.6109/jkiice.2017.21.10.1929 print ISSN: 2234-4772 online ISSN: 2288-4165

Ⅰ. 서 론

경량 암호화 알고리듬 (Lightweight Encryption Algorithm, LEA)은 국가보안기술연구소에서 개발되어 WISA'13에서 발표되었다[1]. LEA 알고지즘은 덧셈, 회전 그리고 XOR 연산을 기본으로 하는 Addition, Rotation, eXclusive-or (ARX) 구조를 따른다. 해당 구조는 소프트웨어와 하드웨어 구현 시 효율적이며 시간차 공격에도 안전한 특성을 가지고 있다. ICISC'13에서는 ARMv7 프로세서 상에서 LEA 암호화를 구현한 결과가 소개되었다[2]. 해당 연구에서는 ARM 프로세서의 Single Instruction Multiple Data (SIMD) 연산자인 NEON을 이용하여 LEA 연산을 병렬화할 수 있는 방안이 제안되었다.

2014년도에는 웹 프로그래밍 언어인 자바스크립트 상에서 LEA를 구현한 결과가 소개되었다 [3]. 해당 논문에서는 다양한 웹브라우저 상에서 동작 시 LEA를 통해 암호화된 패킷이 가장 효율적으로 암호화 됨을 확인할 수 있었다.

WISA'15에서는 8-비트 AVR 프로세서, 16-비트 MSP 프로세서 등 저전력으로 동작하는 경량 사물인터넷 디바이스 상에서의 최적화 구현결과가 소개되었으며 LEA 상에서 효율적인 구현이 가능함이 증명되었다 [4].

WISA'16에서는 ARMv7 프로세서 상에서 ARM과 NEON 명령어 셋을 혼용하여 연산하는 방안이 제시되었다[5]. 이를 기반으로 하여 동시에 다수의 암호화 연산 수행이 가능하도록 하였다. 하지만 기존의 연구는 32 비트 ARMv7 상에서의 최적화 구현에 대해서만 이루어졌고 최근에 많이 사용되는 64 비트 ARMv8 상에서의 연구는 진행되지 않았다. 64 비트 ARMv8은 애플의 경우 iPhone 5S 그리고 삼성의 경우 Galaxy S6부터 사용하고 있는 프로세서 구조로써 대부분의 모바일 프로세서는 64 비트라고 할 수 있다. 본 논문에서는 64 비트 ARMv8 상에서 LEA를 최적화 구현해봄으로써 많이 사용되는 모바일 환경 상에서의 암호화 부하를 최적화하도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 경량 암호화 알고리듬 LEA와 타겟 프로세서인 ARMv8에 대해 알아본다. 3장에서는 최적화 기법에 대해 확인해 본다. 4장에서는 본 논문에서 제안한 기법을 통해 얻어낸 성능 향상에 대해 확인해 본다. 5장에서는 본 논문을 마무리한다.

Ⅱ. 관련 연구

2.1. LEA 대칭키 암호화 알고리듬

LEA 대칭키 암호화 알고리듬은 국가보안기술 연구소에 의해 2013년에 국제학회 WISA에서 발표되었다[1]. 기존의 AES 대칭키 암호화 알고리듬에서 Substitution Permutation Network (SPN) 구조를 활용하여 설계되었다. 반면에 LEA 대칭키 암호화는 Addition Rotation eXclusive-or (ARX) 구조를 사용하여 복잡한 S-box 연산 없이 암호화를 수행한다. ARX 구조 상에서는 암호화 연산이 고정된 시간 안에 수행이 가능하여 이전 SPN 구조 상에서 취약했던 Timing 공격에도 강인한 장점을 가진다. 또한 소프트웨어와 하드웨어 상에서 높은 성능을 보이는 장점을 가진다[6].

2.2. ARMv7 프로세서와 ARMv8 프로세서 비교

Advanced RISC Machine (ARM)은 Instruction Set Architecture (ISA) 디자인을 가진 고성능의 임베디드 프로세서이다. 특히 스마트폰의 초창기 모델에서는 32-비트 ARMv7 프로세서가 사용되었으며 최신 모델에서는 64-비트 ARMv8 프로세서가 사용되고 있다. 해당 모델들에서는 데이터 병렬화가 가능한 SIMD 명령어가 NEON 이라는 이름으로 지원된다. ARMv7의 경우에는 총 16개의 128-비트 NEON 레지스터를 지원한다. 반면에 ARMv8의 경우에는 총 31개의 128-비트 NEON 레지스터를 지원하며 목적지 주소와 시작지 주소를 교차해서 사용하는 것이 대부분의 명령어 셋에서 가능하게 되었다.

표 1에는 ARMv8에서 정의된 명령어 셋 중 암호화에 사용되는 명령어만을 정의하여 나타내고 있다. TRN 명령어는 레지스터의 상위단 혹은 하위단의 레지스터만을 묶어서 출력해 주는 역할을 한다. DUP은 ARM 레지스터의 값을 NEON 레지스터에 중복하여 복사해주게 된다. SHL은 레지스터를 왼쪽으로 이동시키는 역할을 한다. SRI는 레지스터에 값을 삽입하면서 오른쪽으로 이동시키는 역할을 한다. EOR은 exclusive-or 연산을 수행한다. 마지막으로 ADD 명령어는 덧셈 연산을

수행한다.

Table. 1 NEON instruction set for ARMv8

Instruction	Description
TRN	Transpose
DUP	Duplication
SHL	Logical shift to left direction
SRI	Logical shift to right direction with insertion
EOR	Exclusive-or
ADD	Addition

Ⅲ. 제안하는 LEA 구현 기법

64-비트 ARMv8 프로세서는 총 31개의 128-비트 NEON 레지스터를 지원한다. 레지스터의 경우 Arithmetic Logical Unit (ALU)에 대한 접근 속도가 메모리와 비교해 빠르기 때문에 연산에 소모되는 시간이 짧다는 장점을 가진다. 따라서 가능한 많은 평문을 레지스터에 동시에 불러온 이후에 동시에 많은 암호화 연산을 병렬 수행하게 될 경우 성능을 효율적으로 높일 수 있는 특징을 가지고 있다. 타겟 프로세서 상에서의 연산 수행은 그림 1과 같다. 먼저 메모리에 저장된 24개의 128-비트 평문을 레지스터로 불러온다. 해당 연산을 통해 총 24개의 레지스터가 사용되게 되며 나머지 7개의 레지스터는 중간 결과값을 저장하는 임시 레지스터로 활용하였다.

그 다음에는 병렬로 연산을 수행하기 위하여 평문을 벡터화된 형식으로 정렬하는 과정을 거친다. 해당 연산을 통해 동일한 연산이 수행되는 4개의 32-비트 부분 평문들이 하나의 128-비트 NEON 레지스터에 할당되도록 하였다. 그 다음에는 총 24개의 평문이 동시에 라운드 함수를 수행하도록 하였다. LEA-128을 기준으로 라운드 함수는 총 24회 반복되며 24회를 반복한 이후에는 암호문이 출력되게 된다. 하지만 해당 암호문의 경우에는 벡터화된 형식으로 저장되어 있기 때문에 다시 초기 정렬로 재정렬하는 과정을 거치게 된다. 마지막으로 24개 평문에 대한 암호문이 생성되면 결과값을 저장한다.

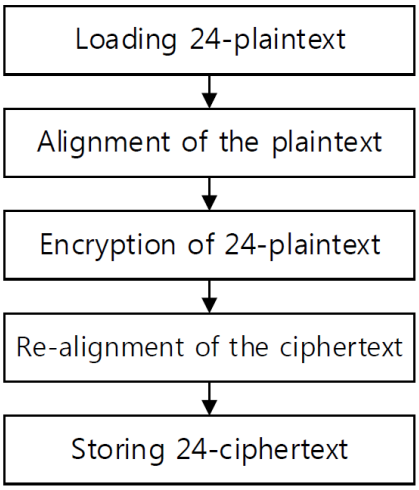

Fig. 1 LEA encryption on ARMv8

표 2에서는 24개의 평문을 메모리 주소 (x1)으로부터 4개씩 총 6번 NEON 레지스터 (v0-v23)으로 불러오는 연산을 하고 있다. 효율적인 메모리 접근을 위해 post-incremental 기법이 적용된 메모리 접근 명령어 셋을 사용하였다. 해당 메모리 접근 연산자는 메모리로부터 값을 불러온 이후에 메모리 주소값이 불러온 값의 크기만큼 자동으로 증가하기 때문에 주소를 계산하기 위한 시간 소모를 줄일 수 있다.

Table. 2 Loading 24-plaintext in NEON instruction

```
ld1.16b {v0, v1, v2, v3}, [x1], #64
ld1.16b {v4, v5, v6, v7}, [x1], #64
ld1.16b {v8 , v9 , v10, v11}, [x1], #64
ld1.16b {v12, v13, v14, v15}, [x1], #64
ld1.16b {v16, v17, v18, v19}, [x1], #64
ld1.16b {v20, v21, v22, v23}, [x1], #64
```

표 3에서는 메모리로부터 불러온 평문을 벡터화된 형식으로 재정렬하는 코드가 설명되어 있다. 만약 데이터가 (1, 2, 3, 4)라는 방식으로 레지스터에 들어오게 된다면 각각의 데이터에서 첫 번째 인자는 첫 번째 인자들끼리 그리고 두 번째 인자는 두 번째 인자들끼리 묶어서 한번 연산 시 모든 데이터에 동일한 연산이 수행되도록 구현하였다. 해당 벡터화 정렬을 위해서 ARMv7에서는 벡터화 명령어 셋에서 인자를 두 개만

받아서 상호간에 Transpose 연산이 되었지만 ARMv8의 경우에는 총 3개의 인자를 받아서 첫 번째 인자에 두 번째 인자와 세 번째 인자의 Transpose 값이 출력되도록 하였다. 또한 ARMv8에서는 trn1의 경우 두 레지스터의 하위값만 모아주고 trn2의 경우 상위값만을 모아주게 된다. 따라서 ARMv8에서는 두 인자에 대한 Transpose 값을 모두 도출하기 위해서 두 번의 Transpose 연산이 각각 수행되도록 하였다. 표 3에서는 총 4개의 평문에 대한 Transpose 연산이 수행되며 24개의 평문을 정렬하기 위해 총 6번 반복되어 연산이 수행되었다. 첫 번째 줄의 연산은 v0 레지스터와 v1 레지스터의 하위 64-비트 값을 합쳐서 v28 레지스터에 저장해 주게 되고 두 번째 줄의 연산은 v0 레지스터와 v1 레지스터의 상위 64-비트 값을 합쳐서 v29 레지스터에 저장해 주게 된다. 3번째 그리고 4번째 줄의 경우 v2 레지스터와 v3 레지스터의 값을 Transpose 연산 수행 후 v30 레지스터와 v31 레지스터에 저장해 주게 된다. 이렇게 Transpose 가 수행되게 된 이후에 다시 v28 레지스터와 v30 레지스터 그리고 v29 레지스터와 v31 레지스터를 Transpose 연산을 통해 동일한 오프셋에 위치하는 값들을 모두 모아 주게 된다.

Table. 3 Alignment of the plaintext in NEON instruction

```
trn1 v28.4s, v0.4s, v1.4s
trn2 v29.4s, v0.4s, v1.4s

trn1 v30.4s, v2.4s, v3.4s
trn2 v31.4s, v2.4s, v3.4s

trn1 v0.2d, v28.2d, v30.2d
trn2 v2.2d, v28.2d, v30.2d

trn1 v1.2d, v29.2d, v31.2d
trn2 v3.2d, v29.2d, v31.2d
```

표 4에서는 LEA 암호화에 사용되는 라운드 함수를 NEON 명령어 셋으로 구현한 결과를 나타내고 있다. 총 6 번의 XOR, 3번의 덧셈 그리고 3번의 회전 연산을 수행하게 된다. 해당 연산 결과는 총 4개의 평문에 대한 라운드 함수 결과값을 도출하게 되며 24개의 평문을 암호화하기 위해 6번 반복하여 수행되게 된다. 특히 shl과 sri의 조합은 한번의 회전 연산을 구현하기 위해 NEON 상에서 자주 사용되는 기법이다.

표 5에는 표 3에서 벡터화로 정렬했던 암호문을 다시 일반적인 정렬의 형태로 재정렬 해주는 코드를 나타내고 있다. 해당 재정렬과정은 이전 표 3에서 수행된 연산을 정반대로 수행하게 된다. 표 5에 나타난 연산은 4개의 암호문에 대한 재정렬 연산을 나타내며 24개의 암호문 정렬을 위해서는 해당 연산은 총 6번 반복하여 수행해야 한다.

Table. 4 Encryption of the plaintext in NEON instruction

```
eor.16b v3, v3, v29
eor.16b v27, v2, v31
add.4s v27, v27, v3

shl.4s v3, v27, #29
sri.4s v3, v27, #3

eor.16b v2, v2, v29
eor.16b v27, v1, v30
add.4s v27, v27, v2

shl.4s v2, v27, #27
sri.4s v2, v27, #5

eor.16b v1, v1, v29
eor.16b v27, v0, v28
add.4s v27, v27, v1

shl.4s v1, v27, #9
sri.4s v1, v27, #23
```

Table. 5 Re-alignment of the plaintext in NEON instruction

```
trn1 v28.2d, v0.2d, v2.2d
trn2 v30.2d, v0.2d, v2.2d

trn1 v29.2d, v1.2d, v3.2d
trn2 v31.2d, v1.2d, v3.2d

trn1 v0.4s, v28.4s, v29.4s
trn2 v1.4s, v28.4s, v29.4s

trn1 v2.4s, v30.4s, v31.4s
trn2 v3.4s, v30.4s, v31.4s
```

표 6에서는 암호문을 메모리에 저장하는 연산을 수행하게 된다. 메모리의 주소 (x0)에 결과값 (v0-v23)은 post incremental 방식으로 512-비트씩 저장되게 되며 총 6번의 연산을 수행하여 24개의 암호문을 저장하게

된다.

Table. 6 Storing 24-plaintext in NEON instruction

```
st1.16b {v0 ,v1 ,v2 ,v3 }, [x0], #64
st1.16b {v4 ,v5 ,v6 ,v7 }, [x0], #64
st1.16b {v8 ,v9 ,v10,v11}, [x0], #64
st1.16b {v12,v13,v14,v15}, [x0], #64
st1.16b {v16,v17,v18,v19}, [x0], #64
st1.16b {v20,v21,v22,v23}, [x0], #64
```

Ⅳ. 성능 평가

본 장에서는 ARMv8 상에서의 LEA 대칭키 암호화를 구현한 결과에 대해 성능을 분석하고 평가한다. 프로그램은 Xcode를 사용하여 작성되었으며 연산속도 측정은 Apple iPad mini 2와 Apple iPad 9.7 상에서 수행되었다. 두 플랫폼은 각각 1.3GHz Apple A7 그리고 1.84GHz Apple A9 프로세서를 가지고 있으며 64-비트 ARMv8 구조를 따른다. 프로그램은 C 언어와 어셈블리를 통해 구현되었으며 컴파일은 -Ofast 옵션을 사용하였다. iOS에서는 objective C를 제공하기 때문에 어셈블리 코딩이 바로 가능하다. 실제 동작 시간은 해당 디바이스 상에서 얻어진 실행시간을 기반으로 계산하였다.

Table. 7 Comparison of LEA implementations on ARMv7 and ARMv8 architectures (c/b: cycle/byte)

Method	Speed (c/b)	Architecture	Processor
Hong et al. [1]	20.1	ARM9	ARM926EJ-S
Seo et al. [2]	10.1	ARMv7	Cortex-A9
Seo et al. [5]	8	ARMv7	Cortex-A9
This work	2.4	ARMv8	Apple A7
	2.2	ARMv8	Apple A9

표 7에는 LEA 암호화 알고리듬을 ARM 프로세서 상에서 구현한 결과들이 나타나 있다. Hong et al.이 ARM9 구조 상에서 구현한 결과에서는 20.1 cycles/byte 안에 연산 수행이 가능함을 확인할 수 있었다. 그 다음에 ARMv7 상에서 NEON을 사용해 구현한 결과물에서는 최고 8 cycles/byte까지 성능이 향상됨을 확인할 수 있었다. 지금까지의 LEA 암호화 알고리듬 구현은 32-비트 ARMv7에서만 수행되었고 최신 프로세서인 64-비트 ARMv8에서는 구현되지 않았다. 최신 프로세서의 경우 이전 프로세서에 비해 레지스터와 명령어 셋이 개선되었으므로 이를 활용할 경우 보다 좋은 결과를 얻을 수 있다. 따라서 본 논문에서는 ARMv8에 최적화된 구현 프로그램을 ARMv8 코어를 차용한 두 개의 프로세서 Apple A7과 Apple A9 상에서 성능을 확인해 봄으로써 객관화된 지표를 얻을 수 있었다. 표 7에서와 같이 본 논문의 결과물을 ARMv8 상에서 동작시킨 경우 Apple A7의 경우 2.4 cycles/byte의 성능을 얻을 수 있었다. 동일한 코드를 Apple A9에서 수행할 경우에는 2.2 cycles/byte의 성능을 얻을 수 있음을 확인할 수 있었다. 해당 코드는 ARMv8의 명령어 셋과 레지스터를 활용하였기 때문에 ARMv7을 차용한 다른 기법들과 동일한 플랫폼에서의 비교는 할 수 없었다.

Ⅴ. 결 론

본 논문에서는 경량 블록 암호화 알고리듬 LEA를 최신 ARM 프로세서인 ARMv8 구조 상에서 최적화 구현하고 그 성능을 연산속도로 확인해 보았다. ARMv8 상에서는 이전 ARMv7와 다른 명령어와 레지스터 구조를 가지기 때문에 이를 효과적으로 활용하기 위한 방안을 제시되어야 높은 성능 향상이 가능하다. 최적화 구현 결과 ARMv8 상에서 LEA 암호 구현은 Apple A7 그리고 Apple A9 프로세서 상에서 각각 2.4 cycles/byte 그리고 2.2 cycles/byte 안에 수행 가능함을 확인할 수 있었다. 본 논문의 구현 결과는 다른 ARX 구조를 가진 블록 암호화 알고리듬인 SPECK 그리고 SIMON에 적용 시에도 동일한 성능향상이 가능하다. 따라서 [7]에서 제안된 SPECK 및 SIMON에 대한 최적화 구현 연구를 차후 연구로 진행해 나갈 예정이다.

ACKNOWLEDGMENTS

This research was financially supported by Hansung University.

REFERENCES

[1] D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, "LEA: A 128-bit block cipher for fast encryption on common processors," *In Information Security Applications, WISA 2013*, pp. 3-27, 2013.

[2] H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi, and H. Kim, "Parallel implementations of LEA," *In Information Security and Cryptology, ICISC 2013*, pp. 256-274, 2013.

[3] H. Seo and H. Kim, "Low-power encryption algorithm block cipher in JavaScript," *Journal of Information and Communication Convergence Engineering*, vol. 12, no. 4, pp. 252-256, Dec. 2014.

[4] H. Seo, Z. Liu, J. Choi, T. Park, and H. Kim, "Compact implementations of LEA block cipher for low-end microprocessors," *In Information Security Applications WISA 2015*, pp. 28-40, 2015.

[5] H. Seo, T. Park, S. Heo, G. Seo, B. Bae, Z. Hu, L. Zhou, Y. Nogami, Y. Zhu, H. Kim, "Parallel Implementations of LEA, Revisted," *In Information Security Applications, WISA 2016*, pp. 318-330, 2016.

[6] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright, "Fast software AES encryption," *In Fast Software Encryption FSE 2010*, pp. 75-93, 2010.

[7] T. Park, H. Seo, H, Kim, "Parallel Implementations of SIMON and SPECK," *IEEE International Conference on Platform Technology and Service*, pp. 1-6, 2016.

서화정(Hwa-jeong Seo)

2010년 2월: 부산대학교 컴퓨터공학과 학사 졸업
2012년 2월: 부산대학교 컴퓨터공학과 석사 졸업
2012년 3월 ~ 2016년 1월: 부산대학교 컴퓨터공학과 박사 졸업
2016년 1월~2017년 3월: 싱가포르 과학기술청
2017년 4월~현재: 한성대학교 IT 융합공학부 조교수
※관심분야 : 정보보호, 암호화 구현, IoT