Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.33, NO.3, Jun. 2023 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.3.401

ARMv8 환경에서 NIST LWC SPARKLE 효율적 구현*

신 한 범,[1†] 김 규 상,[1] 이 명 훈,[1] 김 인 성,[1] 김 선 엽,[1]
권 동 근,[1] 김 성 겸,[2] 서 석 충,[3‡] 홍 석 희[4]
[1,4]고려대학교 (대학원생, 교수), [2]삼성전자 (연구원), [3]국민대학교 (부교수)

Efficient Implementation of NIST LWC SPARKLE on 64-Bit ARMv8*

Hanbeom Shin,[1†] Gyusang Kim,[1] Myeonghoon Lee,[1] Insung Kim,[1] Sunyeop Kim,[1]
Donggeun Kwon,[1] Seonggyeom Kim,[2] Seogchung Seo,[3‡] Seokhie Hong[4]
[1,4]Korea University (Graduate student, Professor),
[2]Samsung Electronics (Researcher), [3]Kookmin University (Associate Professor)

요 약

본 논문에서는 NIST LWC 최종후보 중 하나인 SPARKLE을 64-비트 ARMv8 프로세서 상에서 최적화하는 방안에 대해 제안한다. 제안 방법은 두 가지로서 ARM A64 명령어를 이용한 구현과 NEON ASIMD 명령어를 이용한 구현이다. A64 기반 제안구현은 ARMv8 상에서 가용한 레지스터를 효율적으로 사용할 수 있도록 레지스터 스케줄링을 수행하여 최적화한다. 최적화된 A64 기반 제안구현을 활용할 경우 Raspberry Pi 4B에서 C언어 참조 구현보다 1.69~1.81배 빠른 속도를 얻을 수 있다. 두 번째로, ASIMD 기반 제안구현은 하나의 벡터 명령어를 통해 3개 이상의 ARX-box를 병렬적으로 수행하도록 데이터를 병렬적으로 구성하여 최적화한다. 최적화된 ASIMD 기반 제안구현은 A64 기반 제안구현보다 일반적인 속도는 떨어지지만, SPARKLE256에서 SPARKLE512로 블록 크기가 증가할 때 A64 기반 제안구현에서는 속도가 2.1배 느려지는 것에 비해 ASIMD 기반 제안구현에서는 1.2배밖에 느려지지 않다는 장점이 있다. 따라서 기존 SPARKLE보다 더 큰 블록 크기를 갖는 SPARKLE 변형 블록 암호 또는 순열 설계 시 ASIMD 기반 제안구현이 더 효율적이므로 유용한 자료로써 활용 가능하다.

ABSTRACT

In this paper, we propose optimization methods for implementing SPARKLE, one of the NIST LWC finalists, on a 64-bit ARMv8 processor. The proposed methods consist of two approaches: an implementation using ARM A64 instructions and another using NEON ASIMD instructions. The A64-based implementation is optimized by performing register scheduling to efficiently utilize the available registers on the ARMv8 architecture. By utilizing the optimized A64-based implementation, we can achieve speeds that are 1.69 to 1.81 times faster than the C reference implementation on a Raspberry Pi 4B. The ASIMD-based implementation, on the other hand, optimizes data by parallelizing the ARX-boxes to perform more than three of them concurrently through a single vector instruction. While the general speed of the optimized ASIMD-based implementation is lower than that of the A64-based implementation, it only slows down by 1.2 times compared to the 2.1 times slowdown observed in the A64-based implementation as the block size increases from SPARKLE256 to SPARKLE512. This is an advantage of the ASIMD-based implementation. Therefore, the ASIMD-based implementation is more efficient for SPARKLE variant block cipher or permutation designs with larger block sizes than the original SPARKLE, making it a useful resource.

Keywords: SPARKLE, Implementation, NIST LWC, ARMv8

Received(04. 19. 2023), Modified(05. 25. 2023), Accepted(05. 25. 2023)
* 이 성과는 2023년도 정보통신기획평가원의 지원(No. 2021-0-00796, 상시적 보안품질 보장을 위한 6G 자율보안 내재화 기반기술 연구, 100%)을 받아 수행된 연구임
† 주저자, newonetiger@korea.ac.kr
‡ 교신저자, scseo@kookmin.ac.kr(Corresponding author)

I. 서 론

IoT 환경에서의 효율적인 데이터 보호를 위해 경량암호 알고리즘의 필요성이 대두되었으며 효율적이고 안전한 경량암호 표준화를 위해 NIST (National Institute of Standards and Technology, 미국 국립표준기술연구소)는 2016년도부터 LWC(LightWeight Cryptography) 공모전을 수행하였다[8]. NIST LWC 최종후보들은 연산의 효율성과 높은 보안성으로 각광받고 있는 AEAD(Authenticated Encryption with Associated Data) 알고리즘이고 ASCON[9], Elephant[10], GIFT-COFB[11], Grain-128AEAD[12], SPARKLE(SCHWAEMM과 ESCH)[1] 등 10종으로 구성되어 있다. 이러한 NIST LWC 최종후보들에 대해, 경량성을 위해 연산 성능을 개선하는 연구가 최근 많이 진행되고 있다.

SPARKLE 순열(permutation)[1]은 SPARX 블록 암호[6]에서 더 큰 블록 크기와 고정된 키를 사용하는 구조이다. SPARKLE 순열을 기반으로 한 인증 암호화 알고리즘 SCHWAEMM과 해시함수 ESCH[1]는 NIST LWC 최종후보 중 하나이다. SPARKLE 순열은 덧셈, 회전 그리고 XOR 연산을 구성요소로 갖는 ARX(Addition, Rotation, eXclusive-or) 구조를 따른다. 이러한 ARX 구조는 연산에서 산술 및 논리 연산에서 추가적인 클럭 사이클(clock cycle) 없이 한쪽 입력 데이터에 대해 시프트(shift) 연산을 적용할 수 있는 배럴 시프터(barrel shifter)를 이용하면 효율적인 구현이 가능하다.

[1]에서는 C언어와 ARMv7 환경에서의 어셈블리 참조구현을 제시하고 있고, ARMv7 환경에서의 어셈블리 참조구현은 ARM A32 명령어를 이용하고 있다. ARMv7 환경에서의 어셈블리 참조구현은 ARM A32에서 지원하는 배럴 시프터와 ROR(ROtate-Right) 연산을 이용하여 SPARKLE 순열을 최적화한다. 뿐만 아니라 SPARKLE 순열에 대해 GPU[13], FPGA[14], 양자 회로[15] 등의 다양한 환경에서의 최적화 구현 연구가 진행되고 있다.

Table 1. SPARKLE Implementations

Implementation	Reference
ARMv7 A32	[1]
GPU	[13]
FPGA	[14]
Quantum Circuit	[15]
ARMv8 ARM64	Section 3.1
ARMv8 NEON ASIMD	Section 3.2

본 논문에서는 64-비트 ARMv8 환경에서 SPARKLE 순열 최적화 구현 방법을 제안한다. 제안 방법은 두 가지로서 64-비트 ARM A64 명령어를 이용한 구현과 NEON ASIMD(Advanced Single Instruction Multiple Data) 명령어를 이용한 구현이다.

ARM A64에서는 A32와 달리 ADD 연산에 대해 ROR 연산을 배럴 시프터로 이용하는 것이 불가능하다는 단점이 존재한다. 하지만 A32에 비해 더 많은 범용 레지스터를 갖고 있는 점과 64-비트 레지스터에서 32-비트 레지스터만 사용 가능하다는 점을 활용한다. A64 기반 제안 구현은 ARMv8 상에서 가용한 레지스터를 효율적으로 사용할 수 있도록 레지스터 스케줄링을 수행하여 최적화한다.

NEON ASIMD에서는 배럴 시프터가 지원되지 않고 ROR 연산 또한 지원하지 않는다는 단점이 존재한다. 하지만 NEON 레지스터를 이용하여 하나의 벡터 명령어를 통해 3개 이상의 ARX-box를 병렬적으로 수행하도록 데이터를 병렬적으로 구성하여 최적화 구현한다.

Raspberry Pi 4B에서 A64 기반 제안구현은 C 언어 참조구현[1]과 비교하여 1.69~1.81배 향상된 속도를 갖는다. ASIMD 기반 제안구현은 A64 기반 제안구현보다 일반적인 속도는 떨어진다. 하지만 SPARKLE256에서 SPARKLE512로 블록 크기가 증가할 때, A64 기반 제안구현에서는 속도가 2.1배 느려지는 것에 비해 ASIMD 기반 제안구현에서는 1.2배밖에 느려지지 않다는 장점이 있다. 따라서 기존 SPARKLE보다 더 큰 블록 크기를 갖는 SPARKLE 변형 블록 암호 또는 순열에서는 ASIMD 기반 구현이 더 효율적일 수 있다. 이로 인해 SPARKLE 변형 블록 암호 또는 순열 설계 시 유용한 자료로써 활용 가능하다. 제안구현은 https://github.com/shb115/SPARKLE_ARMv8에서 이용 가능하다.

본 논문의 구조는 다음과 같다. Ⅱ장에서는 SPARKLE과 ARMv8 프로세서를 간단하게 소개한다. 제안하는 구현 방법은 Ⅲ장에서 서술하며, Ⅳ

장에서는 구현 결과에 대해 분석한다. V장에서 본 논문의 결론을 맺는다.

II. 배경지식

2.1 SPARKLE

SPARKLE[1]은 256-, 384-, 512-비트의 입출력으로 가지며, 각 블록 크기에 대해 SPARKLE256, SPARKLE384, SPARKLE512로 부른다. 각 블록 크기에 대해 슬림 버전(slim version)과 큰 버전(big version)을 각각 지원하며 SPARKLE256에 대해 7-, 10-스텝(step), SPARKLE384에 대해 7-, 11-스텝, SPARKLE512에 대해 8-, 12-스텝을 지원한다.

Fig. 1.은 SPARKLE의 한 스텝을 그림으로 나타낸 것이다. n_b는 ARX-box 브랜치(branch) 수이고 SPARKLE의 각 블록 크기에 대해 $n_b=4,6,8$이며 $h_b=n_b/2$이다. 각 $z_i(0 \le i < n_b)$는 (x_i, y_i)로 이루어진 64-비트 입력이다. SPARKLE의 한 스텝은 상수 $c_i(0 \le i \le 7)$와 스텝 s를 각각 y_0, y_1에 XOR 연산하는 과정, n_b개의 64-비트 ARX-box A_{c_i}로 이루어진 비선형 계층과 그 출력값들을 결합하고 섞어주는 선형 계층 L_{n_b}로 이루어져 있다. 선형 계층 L_{n_b}은 선형 페이스텔(Feistel) 함수 M_{h_b}과 브랜치 순열로 이루어져 있다.

Algorithm 1.은 SPARKLE256$_{n_s}$를 알고리즘으로 나타낸 것이다. 이 때 n_s는 스텝의 수이다.

Fig. 3.은 ARX-box Alzette A_c를 그림으로 나타낸 것이고 x, y, c, u, v는 32-비트이다. A_c는 4-라운드로 구성되어 있고 64-비트 입출력을 가진다.

Algorithm 2.는 SPARKLE256에서 사용되는 선형 계층 L_4을 알고리즘으로 나타낸 것이다.

Algorithm 1. SPARKLE256$_{n_s}$
Input/Output: $((x_0,y_0),...,(x_3,y_3)) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^4$
1. $(c_0,c_1) \leftarrow$ (0xB7E15162, 0xBF715880)
2. $(c_2,c_3) \leftarrow$ (0x38B4DA56, 0x324E7738)
3. $(c_4,c_5) \leftarrow$ (0xBB1185EB, 0x4F7C7B57)
4. $(c_6,c_7) \leftarrow$ (0xCFBFA1C8, 0xC2B3293D)
5. for all $s \in [0, n_s-1]$ do
6. $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
7. $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
8. for all $i \in [0,3]$ do
9. $(x_i,y_i) \leftarrow A_{c_i}(x_i,y_i)$
10. end for
11. $((x_0,y_0),...,(x_3,y_3)) \leftarrow L_4((x_0,y_0),...,(x_3,y_3))$
12. end for
13. return $((x_0,y_0),...,(x_3,y_3))$

Fig. 2. SPARKLE256$_{n_s}$ [1]

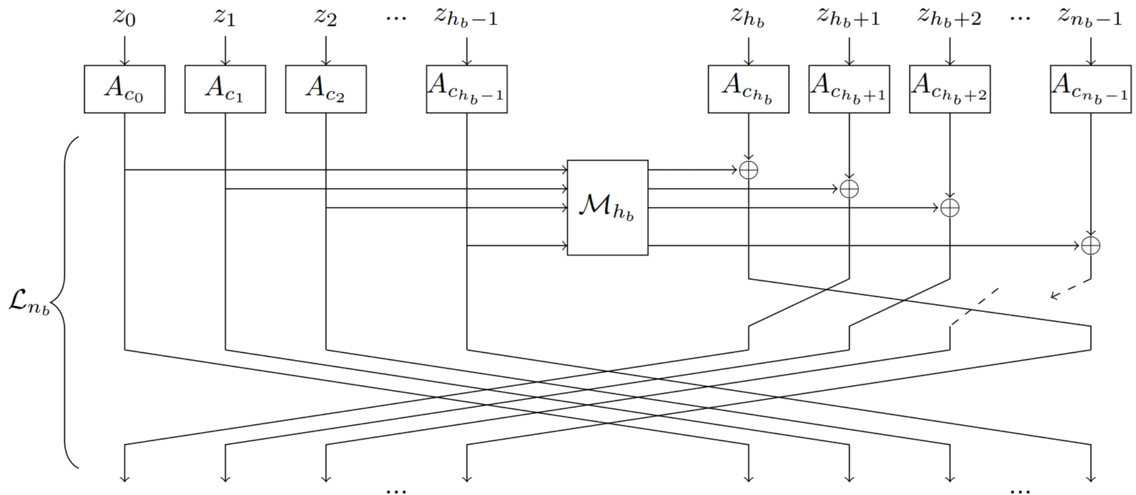

Fig. 1. The overall structure of a step of SPARKLE [1]

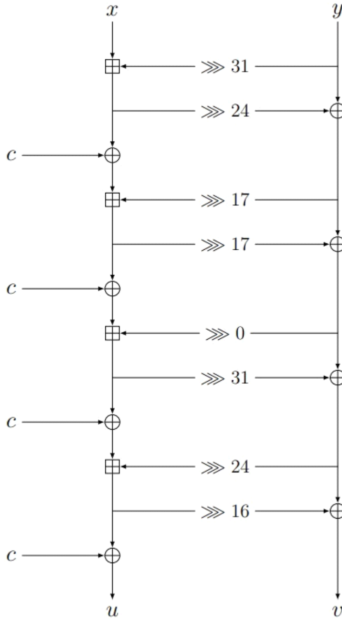

Fig. 3. Structure of ARX-box Alzette A_c [7]

Algorithm 2.의 1~5번째 줄이 선형 페이스텔 함수 M_{h_b}이고 6~7번째 줄이 브랜치 순열이다.

SPARKLE의 구성요소들은 배럴 시프터와 ROR 연산을 지원하는 프로세서에서 효과적이다.

2.2 64-비트 ARMv8 프로세서

ARM 프로세서는 저전력과 고성능 때문에 임베디드 산업에 널리 활용된다. ARMv8은 ARM 프로세서와 NEON 엔진을 제공한다[2]. ARM 프로세서는 병렬적인 프로세싱을 지원하지는 않지만, 산술 및 논리 연산에서 추가적인 클럭 사이클 없이 한쪽 입력 데이터에 대해 시프트 연산을 적용할 수 있는 배럴시프터를 제공한다. ARMv8 프로세서는 64-비트 범용 레지스터 x0-x30와 A64 명령어 집합을 제공한다. 64-비트 레지스터들은 32-비트 w0-w30으로 자유롭게 이용 가능하다.

NEON 엔진은 128-비트 벡터 레지스터 v0-v31을 제공하고 병렬 프로세싱이 가능한 ASIMD 명령어 집합을 제공한다. 이 병렬 프로세싱은 128-비트 벡터 레지스터 내에서 64-, 32-, 16-, 8-비트 단위로 수행될 수 있다.

Table 2.는 SPARKLE 순열 최적화 구현에 활용되는 A64와 ASIMD 명령어들의 요약이다.

Algorithm 2. Linear Layer L_4

Input/Output: $((x_0,y_0),\ldots,(x_3,y_3)) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^4$

1. $(t_x,t_y) \leftarrow (x_0 \oplus x_1, y_0 \oplus y_1)$
2. $t_x \leftarrow (t_x \oplus (t_x \ll 16)) \lll 16$
3. $t_y \leftarrow (t_y \oplus (t_y \ll 16)) \lll 16$
4. $(y_2,y_3) \leftarrow (y_2 \oplus y_0 \oplus t_x, y_3 \oplus y_1 \oplus t_x)$
5. $(x_2,x_3) \leftarrow (x_2 \oplus x_0 \oplus t_y, x_3 \oplus x_1 \oplus t_y)$ // M_2
6. $(x_0,x_1,x_2,x_3) \leftarrow (x_3,x_2,x_0,x_1)$
7. $(y_0,y_1,y_2,y_3) \leftarrow (y_3,y_2,y_0,y_1)$ // branch perm
8. **return** $((x_0,y_0),\ldots,(x_3,y_3))$

Fig. 4. Linear Layer L_4 [1]

Table 2. ARMv8 Instructions set

ARM A64 Instructions		
Instruction	Operand	Description
EOR	X_n, X_m, X_d	Exclusive-or operation, $X_n = X_m \oplus X_d$
ADD	X_n, X_m, X_d	Addition operation, $X_n = X_m + X_d$
ROR	$X_n, X_m, \#d$	Rotate right, $X_n = X_m \ggg d$
NEON ASIMD Instructions		
Instruction	Operand	Description
DUP	V_n, X_n	Duplicate element to vector, $X_n \rightarrow V_n$
UMOV	X_n, V_n	Unsigned move vector element to general-purpose register, $V_n \rightarrow X_n$
SHL	$V_n, V_m, \#d$	Shift left, $(V_m, d) \rightarrow V_n$
SRI	$V_n, V_m, \#d$	Shift right and insert, $(V_m, d) \rightarrow V_n$
REV32	V_n, V_m	Reverse elements in 32-bit words, $(V_m, 32) \rightarrow V_n$
EXT	$V_n, V_m, V_d, \#k$	Extract vector form pair of vectors, $(V_m, V_d, k) \rightarrow V_n$
EOR, ADD	V_n, V_m, V_d	Exclusive-or and Addition operation, $V_n = V_m \oplus V_d$, $(V_m, V_d) \rightarrow V_n$

III. Sparkle 구현 최적화

3.1 ARM A64를 이용한 구현 최적화

〔1〕에서는 ARMv7 환경에서의 어셈블리 참조구현을 제시한다. ARMv7 프로세서는 32-비트 범용 레지스터 r0-r15와 A32 명령어 집합을 제공한다. SPARKLE의 블록 크기는 256-, 384-, 512-비트이고 범용 레지스터 중 최소 1개는 임시 레지스터로 이용해야 하기 때문에 상수 $c_0,\ldots,c_7$을 범용 레지스터에 모두 저장할 수 없다. 특히 SPARKLE512에서는 상태 값 (x_i,y_i)조차 범용 레지스터에 모두 저장할 수 없다. 따라서 〔1〕의 ARMv7 참조구현은 범용 레지스터에 저장할 수 없는 상수들에 대해 Fig. 5.의 MOV32를 이용해서 상수를 매번 임시 레지스터에 저장하여 연산을 수행한다. Fig. 5.에서 MOVW와 MOVT는 A32 명령어로써 16-비트 단위로 값을 레지스터에 저장하는 연산이다. SPARKLE512에서는 범용 레지스터에 저장할 수 없는 상태 값들을 스택(stack)에 저장해두고 필요할 때마다 임시 레지스터에 저장하여 연산한다.

하지만 64-비트 ARMv8 프로세서는 총 31개의 범용 레지스터를 지원한다. 따라서 상태 값들뿐만 아니라 상수 $c_0,\ldots,c_7$를 모두 범용 레지스터에 저장하여 사용할 수 있다.

A32에서는 ADD 연산에 대해 배럴 시프터를 이용하여 ROR 연산을 추가 클럭 사이클 없이 이용할 수 있다. 하지만 A64에서는 ADD 연산에 대해 배럴 시프터를 이용하여 ROR 연산을 추가 클럭 사이클 없이 이용할 수 없다는 단점이 존재한다〔2〕. 따라서 A64에서는 ADD 연산과 ROR 연산을 각각 실행하는 것이 최소한의 연산만으로 A32에서와 동일한 연산을 수행할 수 있다.

64-비트 레지스터에 32-비트 상태 값 2개를 저장하여 연산을 진행하는 방식은 ADD 연산으로 인한 캐리(carry)에 의해 비효율적인 것을 확인하였다. 따라서 ARMv8에서 64-비트 레지스터들을 32-비트 레지스터로 자유롭게 이용 가능하다는 점을 이용하여, 범용 레지스터에 32-비트 상태 값을 저장하여 연산을 진행하는 것이 가장 효율적이다. 따라서 본 절의 제안구현에서 각 블록 크기에 대한 레지스터 스케줄링은 Table 3.과 같다.

```
.macro MOV32 ri:req, ci:req
    movw    \ri, #:lower16:\ci // lower 16-bit
    movt    \ri, #:upper16:\ci // lower 16-bit
.endm
```

Fig. 5. MOV32

Table 3. Register scheduling in 3.1

	Register Scheduling	
SPARKLE256	w3-w10	x_i,y_i
	w19-w26	c_i
	w1-w2	temp
SPARKLE384	w3-w14	x_i,y_i
	w19-w26	c_i
	w1-w2	temp
SPARKLE512	w3-w18	x_i,y_i
	w19-w26	c_i
	w1-w2	temp

3.2 NEON ASIMD를 이용한 구현 최적화

〔1〕의 5장 “구현적 측면”에서는 배럴 시프터를 이용한 구현과 더불어 NEON의 벡터 레지스터에 대해 언급하고 있다. ARMv7 환경에서도 NEON ASIMD 명령어를 지원하지만, ARMv7 참조구현〔1〕에는 NEON ASIMD 명령어를 이용한 구현이 존재하지 않는다. 따라서 본 절에서는 NEON ASIMD 명령어를 이용한 최적화 구현을 제안한다.

NEON 벡터 레지스터를 이용하면 ARX-box 또는 선형 계층을 병렬적으로 연산할 수 있다. 하지만 두 연산을 모두 병렬적으로 연산하는 것은 데이터 구조상 불가능하다. ARX-box의 연산이 선형 계층의 연산보다 더 많은 비용을 필요하므로 ARX-box를 병렬적으로 연산하는 것이 효율적이라는 것을 확인하였다.

따라서 ARX-box를 병렬적으로 연산하기 위해 SPARKLE256에서는 v0에 $x_0,\ldots,x_3$을 저장하고 v1에 $y_0,\ldots,y_3$을 저장하여 4개의 ARX-box를 1개의 벡터 ARX-box만으로 연산할 수 있다. SPARKLE384에서는 v0에 $x_0,\ldots,x_2$, v1에 $y_0,\ldots,y_2$, v2에 $x_3,\ldots,x_5$, v3에 $y_3,\ldots,y_5$를 저장하고 SPARKLE512에서는 v0에 $x_0,\ldots,x_3$, v1에 $y_0,\ldots,y_3$, v2에 $x_4,\ldots,x_7$, v3에 $y_4,\ldots,y_7$를 저장하여 각각 6, 8개의 ARX-box를 2, 2개의 벡터

ARX-box만으로 수행할 수 있다. ARX-box에는 상태 값 이외에도 상수가 필요하고 상태 값과 상수의 인덱스(index)가 일치해야 한다. SPARKLE256에서는 v2에 $c_0, \ldots, c_3$를 저장하고 SPARKLE384에서는 v4에 $c_0, \ldots, c_2$, v5에 $c_3, \ldots, c_5$, SPARKLE512에서는 v4에 $c_0, \ldots, c_3$, v5에 $c_4, \ldots, c_7$를 저장하면 상태 값과 상수의 인덱스를 일치시키며 벡터 ARX-box를 연산할 수 있다. Table 4.는 본 절의 제안구현에서 각 블록 크기에 대한 벡터 레지스터 스케줄링이고 Fig. 6.은 SPARKLE256에서의 벡터 레지스터 구조를 그림으로 나타낸 것이다.

NEON ASIMD에서 배럴 시프터를 지원하지 않는 점에 대해 3.1절과 동일하게 ADD 연산과 ROR 연산 역할을 수행하는 연산들로 분리하여 수행한다. ROR 연산을 지원하지 않는 점에 대해서는 SHL과 SRI를 이용하여 2개의 명령어만으로 ROR 연산의 역할을 수행하도록 하거나, ROR 16에 대해서는 REV32를 이용하면 1개의 명령어만으로 ROR 연산의 역할을 수행할 수 있다. Fig. 7.은 본 절의 SPARKLE256 제안구현에서 ARX-box 4-라운드 중 1-라운드를 연산하는 과정이다.

Table 4. Register scheduling in 3.2

	Register Scheduling	
SPARKLE256	v0-v1	x_i, y_i
	v2-v3	c_i
	v13	temp
SPARKLE384	v0-v3	x_i, y_i
	v4-v5	c_i
	v13	temp
SPARKLE512	v0-v3	x_i, y_i
	v4-v5	c_i
	v13	temp

v0	x_0	x_1	x_2	x_3
v1	y_0	y_1	y_2	y_3
v2	c_0	c_1	c_2	c_3
v3	c_4	c_5	c_6	c_7

Fig. 6. Structure of vector register in SPARKLE256

```
shl   v4.4s, v1.4s, #1
sri   v4.4s, v1.4s, #31 // rotate right v1 31
add   v0.4s, v0.4s, v4.4s // v0 = v0 + v4
shl   v4.4s, v0.4s, #8
sri   v4.4s, v0.4s, #24 // rotate right v0 24
eor   v1.16b, v1.16b, v4.16b // v1 = v1 XOR v4
eor   v0.16b, v0.16b, v2.16b // v1 = v1 XOR c
```

Fig. 7. SPARKLE256 ARX-box 1-round

3.1절의 제안구현에서는 선형 계층의 $x_i \leftarrow x_i \oplus t_y$ 연산에 대해 각 블록 크기에 대해 8, 12, 16번의 EOR 연산을 사용하였다. 하지만 벡터 레지스터를 이용하면 각 블록 크기에 대해 2, 4, 4번의 ASIMD EOR 연산만으로 구현할 수 있다.

하지만 t_x, t_y를 계산하기 위해서는 벡터 레지스터의 원소들을 함께 연산하는 과정에서 3.1절의 제안구현보다 많은 연산이 필요하다. A64 기반 제안구현에서는 각 블록 크기에 대해 t_x, t_y를 계산하기 위해 2, 3, 4번의 EOR 연산이 필요하였지만, 2, 3, 4번의 UMOV 연산과 1, 1, 1번의 ROR, 1, 1, 1번의 DUP 연산을 추가적으로 사용해야 했다. Fig. 8.은 본 절의 SPAKRLE384 제안구현에서 t_x, t_y를 계산하는 과정이다.

3.1절의 제안구현에서는 각 레지스터 별로 연산하기 때문에 브랜치 순열 연산이 필요하지 않다. 하지만 벡터 레지스터를 이용하는 경우에는 상태 값과 상수의 인덱스를 일치시켜야 하므로 브랜치 순열 연산이 필요하다. Fig. 9.은 브랜치 순열의 결과를 그림으로 나타낸 것이다.

SPARKLE256에 대해 2번의 EXT, 2번의 REV64 연산을 이용, SPAKRLE384, SPAKRLE512에 대해 각각 4, 2번의 EXT를 이용하면 쉽게 브랜치 순열을 구현할 수 있다. 이는 MOV, UMOV를 이용하여 스왑(swap)하는 것보

```
umov  w12, v0.s[0] // w12 : temp register
umov  w13, v0.s[1] // w13 : temp register
umov  w14, v0.s[2] // w14 : temp register
eor   w12, w12, w13
eor    w12, w12, w14 // w12 = w12 XOR w13 XOR
w14
eor   w12, w12, w12, lsl #16
ror   w12, w12, #16 // w12 = (w12 << 16) <<< 16
dup   v6.4s, w12 // w12 = t_x or t_y
```

Fig. 8. Compute t_x, t_y in SPARKLE384

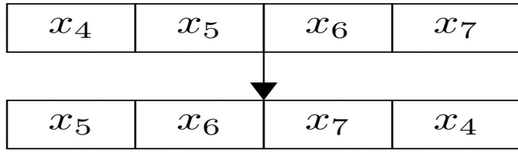

Fig. 9. Conclusion of branch permutation

```
ext    v2.16b, v2.16b, v2.16b, #4
ext    v3.16b, v3.16b, v3.16b, #4
```

Fig. 10. Branch permutation in SPARKLE512

다 더 적은 비용을 사용하여 브랜치 순열을 구현할 수 있다. Fig. 10.은 SPARKLE512에서 브랜치 순열을 구현한 것이다.

SPARKLE의 선형 계층은 벡터 레지스터에 적합하지 않아 많은 오버헤드를 발생시킨다는 것을 알 수 있다.

IV. 구현 결과 비교

본 논문의 제안구현들은 Cortex-A72 CPU를 가진 Linux raspberrypi 5.15.32-v8+ 장치상에서 평가되었다. 운영체제는 Ubuntu 22.04 LTS이고 컴파일러는 GCC version 11.2.0를 사용하였다. 모든 구현들은 -O3 옵션을 사용하였다.

Table 5.에는 SPARKLE 순열 최적화 구현의 성능 결과들이 나타나 있다. Table 5.의 구현에서 C는 [1]의 참조구현, ARM A64는 3.1절의 A64 기반 제안구현 그리고 NEON ASIMD는 3.2절의 ASIMD 기반 제안구현이다. SPAKRLE의 각 블록 크기에 대해 큰 버전의 결과를 얻었으며, 시간은 해당 장치상에서 얻어진 실행시간을 기반으로 계산하였다. 시간의 단위는 ns이며 값은 천만 개 데이터의 평균이다.

A64 기반 제안구현은 C 참조구현에 비해 SPARKLE256에서는 약 1.72배, SPARKLE384에서는 약 1.81배, SPARKLE512에서는 약 1.69배 효율적이란 것을 알 수 있다.

ASIMD 기반 구현물은 ARX-box 연산을 병렬적으로 수행할 수 있지만, 선형 계층의 연산이 벡터 레지스터에 비효율적이기 때문에 A64 기반 제안구현보다 비효율적이라는 것을 알 수 있다. 따라서 ASIMD 기반 구현을 고려한다면 다른 방식으로 선형 계층을 설계하는 것이 더 효과적이라는 것을 알

Table 5. Conclusion of implementation

	Implementation	Time(ns)
SPARKLE256	C [1]	399.692
	ARM A64 [Section 3.1]	232.688
	NEON ASIMD [Section 3.2]	608.751
SPARKLE384	C [1]	609.941
	ARM A64 [Section 3.1]	336.197
	NEON ASIMD [Section 3.2]	662.404
SPARKLE512	C [1]	849.483
	ARM A64 [Section 3.1]	503.476
	NEON ASIMD [Section 3.2]	730.499

수 있다.

Fig. 11.은 x축을 SPARKLE의 블록 크기, y축을 실행시간으로 하는 그래프이다. ASIMD 기반 제안구현은 A64 기반 제안구현에 비해 느리지만, SPARKLE256에서 SPARKLE512로 블록 크기가 증가할 때 A64 기반 제안구현에서는 속도가 2.1배 느려지는 것에 비해 ASIMD 기반 제안구현에서는 1.2배밖에 느려지지 않다는 장점이 있다. 따라서 기존 SPARKLE보다 더 큰 블록 크기를 갖는 SPARKLE 변형 블록 암호 또는 순열을 설계 및 구현한다면 NEON ASIMD를 이용한 구현이 더 효율적이라는 것을 알 수 있다.

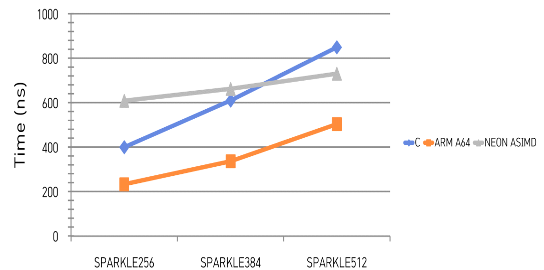

Fig. 11. Graph of implementation conclusion

V. 결 론

본 논문에서는 SPARKLE 순열을 ARMv8 상에서 최적화 구현하고 그 성능을 연산속도로 확인하였

다. ARM A64, NEON ASIMD 명령어들을 이용한 최적화 구현을 각각 제안하였고 모든 블록 크기에 대해 A64 기반 제안구현이 가장 효율적임을 알 수 있었다. 하지만 블록 크기의 증가에 따른 속도는 ASIMD 기반 제안구현이 가장 뛰어났다. 더 큰 안전성을 요구하는 현재 추세에 맞추어 기존 SPARKLE보다 더 큰 블록 크기를 사용하는 SPARKLE 변형 블록 암호 또는 순열을 설계 및 구현하였을 때, ASIMD 기반 제안구현이 가장 효율적임을 알 수 있다.

References

〔1〕 C. Beierle, A. Biryukov, and L.C. dos Santos, "Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family," NIST round 2 , 2019.

〔2〕 ARM Developer, "ARM A64 Instruction Set Architecture," https://developer.arm.com/documentation/ddi0596/latest/, 2023.04.01.

〔3〕 J. Song, Y. Kim, and S,C, Seo, "High-Speed Fault Attack Resistant Implementation of PIPO Block Cipher on ARM Cortex-A," IEEE Access, vol. 9, pp. 162893-162908, Dec. 2021.

〔4〕 J. Song, and S,C, Seo, "Secure and fast implementation of ARX-Based block ciphers using ASIMD instructions in ARMv8 platforms," IEEE Access, vol. 8, pp. 193138-193153, Oct. 2020.

〔5〕 H. Seo, "High speed implementation of LEA on ARMv8," Journal of the Korea Institute of Information and Communication Engineering, 21(10), pp. 1929-1934, Oct. 2017.

〔6〕 D. Dinu, L. Perrin, and A. Udovenko, "Design strategies for ARX with provable bounds: Sparx and LAX," Advances in Cryptology–ASIACRYPT 2016, vol. 10031, pp. 484-513, Dec. 2016.

〔7〕 C. Beierle, A. Biryukov, and L.C. dos Santos, "Alzette: A 64-Bit ARX-box: (Feat. CRAX and TRAX)," Advances in Cryptology–CRYPTO 2020, vol. 12172, pp. 419-448, Aug, 2020.

〔8〕 K. McKay, L. Bassham, and M. Sonmez Turan, "Report on lightweight cryptography," No. NIST Internal or Interagency Report (NISTIR) 8114 (Draft). National Institute of Standards and Technology, 2016.

〔9〕 C. Dobraunig, M. Eichlseder, and F. Mendel, "Ascon v1.2: Lightweight authenticated encryption and hashing," Journal of Cryptology, 34(33), pp. 1-42, Jun. 2021.

〔10〕 B. Menink, "Elephant v2," 2021.

〔11〕 S, Banik, A. Chakraborti, and A.Inoue, "Gift-cofb," Cryptology ePrint Archive, 2020.

〔12〕 M. Hell, T. Johansson, and A. Maximov, "Grain-128AEADv2-A lightweight AEAD stream cipher," NIST Lightweight Cryptography, Finalists, 2021.

〔13〕 W.K. Lee, K. Jang, and G. Song, "Efficient Implementation of Lightweight Hash Functions on GPU and Quantum Computers for IoT Applications," IEEE Access, vol. 10, pp. 59661-59674, Jun. 2022.

〔14〕 S, Khan, W.K. Lee, and A. Karmakar, "Area-time Efficient Implementation of NIST Lightweight Hash Functions Targeting IoT Applications," IEEE Internet of Things Journal, vol. 10(9), pp. 8083-8095, Dec. 2022.

〔15〕 Y. Yang, K. Jang, and H. Kim, "Grover on SPARKLE," Information Security Applications: WISA 2022, vol. 13720, pp. 44-59, Feb. 2023.

〈저자소개〉

신 한 범 (Hanbeom Shin) 학생회원
2022년 2월: 서울시립대 수학과 졸업
2022년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 암호 알고리즘 설계 및 분석, 대칭키 암호

김 규 상 (Gyusang Kim) 학생회원
2020년 8월: 연세대학교 수학과 졸업
2020년 9월~현재: 고려대학교 정보보호대학원 석박사 통합과정
〈관심분야〉 부채널 분석

이 명 훈 (Myeonghoon Lee) 학생회원
2020년 2월: 고려대학교 수학과 졸업
2020년 3월~현재: 고려대학교 정보보호대학원 석박사 통합과정
〈관심분야〉 후양자암호, 대칭키 암호

김 인 성 (Insung Kim) 학생회원
2022년 2월: 서울시립대 수학과 졸업
2022년 3월~현재: 고려대학교 정보보호대학원 석박사 통합과정
〈관심분야〉 암호 알고리즘 설계 및 분석, 대칭키 암호

김 선 엽 (Sunyeop Kim) 학생회원
2019년 8월: 고려대학교 수학과 졸업
2019년 9월~현재: 고려대학교 정보보호대학원 석박사 통합과정
〈관심분야〉 암호 알고리즘 설계 및 분석, 대칭키 암호

권 동 근 (Donggeun Kwon) 학생회원
2018년 2월: 고려대학교 수학과 졸업
2018년 3월~2020년 2월: 고려대학교 정보보호학과 석사
2020년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 부채널 공격, 딥러닝, 머신러닝 기반 암호분석, 암호 알고리즘 설계 및 분석, 대칭키 암호

김 성 겸 (Seonggyeom Kim) 정회원
2016년 8월: 한양대학교 수학과 졸업
2016년 9월~2018년 8월: 고려대학교 정보보호학과 석사
2019년 3월~2023년 2월: 고려대학교 정보보호학과 박사
2023년 3월~현재: 삼성전자
〈관심분야〉 암호 알고리즘 설계 및 분석, 대칭키 암호

서 석 충 (Seogchung Seo) 정회원
2011년 8월: 고려대학교 정보보호대학원 박사
2013년 11월: 삼성전자 종합기술원 전문연구원
2014년 4월: 삼성전자 DMC 연구소 책임연구원
2019년 2월: 국가보안기술연구소 선임연구원
2019년 3월~현재: 국민대학교 금융정보보안학과 부교수
〈관심분야〉 암호최적화, 공개키 암호, 암호모듈검증, 네트워크보안

홍 석 희 (Seokhie Hong) 종신회원
2001년: 고려대학교 수학과 박사
1999년 8월~2004년 2월: ㈜시큐리티 테크놀리지 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