Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.3, Jun. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.3.309

RISC-V 프로세서상에서의 효율적인 ARIA 암호 확장 명령어*

이 진 재,[1†] 박 종 욱,[2] 김 민 재,[1] 김 호 원[3‡]
[1,3]부산대학교 (대학원생, 교수), [2]부산대학교 사물인터넷 연구센터 (연구원)

Efficient ARIA Cryptographic Extension to a RISC-V Processor*

Jin-jae Lee,[1†] Jong-uk Park,[2] Min-jae Kim,[1] Ho-won Kim[3‡]
[1,3]Pusan National University (Graduate student, Professor),
[2]Pusan National University Internet of Things Research Center (Researcher)

요 약

본 연구에서는 저성능 IoT 디바이스에서의 고속 암호화 연산을 지원하기 위해 블록암호 알고리즘 ARIA의 RISC-V 프로세서상에서의 고속 연산을 위한 확장 명령어 셋을 추가한다. 하드웨어상에서의 효율적인 구조로 ARIA 알고리즘을 구현하여 32bit 프로세서에서 동작하기 때문에 효과적인 확장 명령어 셋을 구현한다. 기존의 소프트웨어 암호화 연산과 비교하여 유의미한 성능 향상을 보인다.

ABSTRACT

In this study, an extension instruction set for high-speed operation of the ARIA block cipher algorithm on RISC-V processor is added to support high-speed cryptographic operation on low performance IoT devices. We propose the efficient ARIA cryptographic instruction set which runs on a conventional 32-bit processor. Compared to the existing software cryptographic operation, there is a significant performance improvement with proposed instruction set.

Keywords: RISC-V, ARIA, ISA

I. 서 론

IoT 디바이스의 확산으로 보안이 취약한 IoT 제품의 사용이 증가하고 있다. IoT 디바이스는 주로 저성능 프로세서를 사용하기 때문에 고성능 연산 기능을 요구하는 암호 알고리즘에 대응하지 못하고 있으며 보안을 지원하기 위해 프로세서 외부의 보안 모듈과의 연결이 요구되는 경우가 많다. 외부 모듈과의 연결은 잠재적인 보안 취약성을 증가시키고 프로세서의 성능감소의 원인이 된다.

최근 오픈소스 Instruction set architecture인 RISC-V ISA의 등장으로, IoT 디바이스와 같은 저성능 디바이스의 보안 문제 해결을 위해 명령어 셋 레벨에서 프로세서의 보안을 강화할 수 있는 방안이 연구되고 있다.

본 논문에서는 국산 블록암호 알고리즘 ARIA의 RISC-V 상에서의 custom 확장 명령어 셋을 추가하고 이를 이용한 검증 결과를 보인다.

Received(02. 23. 2021), Modified(05. 12. 2021), Accepted(05. 12. 2021)
* 본 논문은 2020년도 한국정보보호학회 동계학술대회에 발표한 우수논문을 개선 및 확장한 것임.
* 본 연구는 국토교통부의 스마트시티 혁신인재육성사업으로 지원되었습니다.
* 본 연구는 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (2019-0-01343,융합보안핵심인재양성사업)
† 주저자, jinjae@islab.re.kr
‡ 교신저자, howonkim@pusan.ac.kr(Corresponding author)

II. 관련 연구

2.1 암호 연산 가속화 기법

프로세서상에서의 암호 연산의 가속화 기법은 크게 소프트웨어상에서 암호 알고리즘의 연산구조 개선을 통한 방법과 하드웨어를 통한 가속화 기법으로 나눌 수 있다.

소프트웨어상에서의 암호 알고리즘 연산구조 개선은 미리 연산이 가능한 블록암호의 S-box 연산을 Look-up table의 구현을 통해 가속화 하는 방식과 [1][2][3]과 같이 암호 알고리즘의 수학적 특성을 이용한 가속화 방식이 주로 사용된다.

하드웨어를 통한 가속화 기법은 [4]와 같이 암호 연산을 수행하고자 하는 main-processor 이외에 하드웨어적으로 암호 알고리즘 연산을 수행하기 위한 별도의 co-processor를 구현하여 암호화 연산을 수행한 뒤에 연산 결과값을 main-processor에서 사용하는 방식이 있으며, 이러한 방식은 소프트웨어 가속화 기법에 비해 높은 성능을 보인다는 장점이 있지만, 하드웨어상의 면적이 증가하는 단점을 가진다.

이외에도 암호 알고리즘에 사용되는 연산에 대한 확장 명령어 셋을 구현하는 연구도 진행되고 있으며, 자주 사용되는 연산에 대한 명령어를 추가함으로써 면적의 증가는 최소화하고 연산 성능을 높일 수 있다는 장점을 가진다. 암호 확장 명령어에 대한 연구는 [5]와 같이 다양한 암호 알고리즘에서 공통적으로 사용되는 연산에 대한 명령어를 추가하거나 [6]처럼 특정 암호 알고리즘에 대한 전용 명령어를 추가하는 방식이 사용된다. 최근에는 [7][8]과 같이 RISC-V 프로세서상에서 새로운 확장 명령어를 추가하는 방식으로 암호 알고리즘의 가속화를 연구하는 사례가 증가하고 있다.

2.2 AES-NI

AES-NI 확장은 인텔의 CPU 하드웨어 내장 명령어 집합으로써 128bit 데이터의 암/복호화를 가속화 하기 위한 6가지 명령어를 제공하며 추가로 AES-NI 와 관련된 추가 명령어 PCLMULQDQ를 제공한다. ECB, CBC, CTR과 같은 암호화 모드를 지원하며, 대량 암/복호화를 사용하는 데이터 인증 및 CTR-DRBG 와 같은 알고리즘을 사용한 난수 생성의 용도에 적합하다.

기존 AES의 명령어와 다르게 AES-NI는 cpu내부 하드웨어에 종속적인 명령어로 구현되어 있다. AES 암호화/복호화 명령어에서 사용하는 모든 계산은 내부 하드웨어에서 연산을 수행하기 때문에 AES 연산 수행 중 내부 데이터와 물리적인 메모리상의 상관관계가 사라진다. 이런 특성 때문에 캐시 부채널 공격에 내성을 지니고 있으며, 기존 명령어 집합보다 높은 성능을 가지고 있다.

III. 배경지식

3.1 유한체 연산(Finite field)

유한체 상의 덧셈은 두 수의 덧셈을 수행 후 주어진 modulo 공간상의 원소로 표현하면 된다.

$GF(2^m)$의 덧셈은 modulo2의 공간이므로 0과 1로 나타낼 수 있다. 아래 식은 다항식 표현, 이진법 표현, 16진법 표현으로 다음의 연산은 모두 동치이다.

$$(x^6+x^2+x+1)+(x^7+x^6+x^2)=x^7+x^6+x^2$$
$$[010001111]_2 \oplus [10000011]_2 = [11000100]_2$$
$$[47]_{16} \oplus [83]_{16} = [c4]_{16}$$

유한체 상의 곱셈은 ARIA에서 Substitution과 Permutation 연산에 사용된다. $GF(2^m)$상의 곱 $(\bullet)$은 다항식의 곱을 irreducible polynomial (기약 다항식) $m(x)$로 나눈 나머지 값을 의미한다.

ARIA에 사용되는 irreducible polynomial은 $m(x) = x^8+x^4+x^3+x+1$이다. 예를 들어 $[57]_{16} \bullet [83]_{16} = [c1]_{16}$은 다음과 같이 나타낼 수 있다.

$$(x^6+x^4+x^2+x+1)(x^7+x+1)$$
$$=x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+x^1$$

위 식은 다항식의 곱셈을 수행한 것으로 위 식에서 irreducible polynomial로 나눈 나머지 값을 의미한다. 따라서 나타내면 다음과 같다.

$$x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+x^1$$
$$=(x^8+x^4+x^3+x+1)(x^5+x^3)+x^7+x^6+1$$

이므로 x^7+x^6+1이 유한체 상의 곱셈의 값으로 표현된다. 유한체 $GF(2^m)$는 Field이므로 교환법칙, 결합법칙, 항등원을 가지며 곱셈에 대한 역원 또한 존재한다.

3.2 ARIA

ARIA[9] 알고리즘의 기본 구조는 ISPN(Involution SPN)이다. 일반적인 SPN(Substitution-Permutation-Networks) 구조는 별도의 복호화기를 필요로 한다. 하지만 ISPN 구조는 Fig. 1.과 같이 별도의 복호화기를 필요로 하지 않는다. 키 크기는 128,192,256 bit가 사용되며 128 bit를 기준으로 입/출력을 생성한다. 또한, 각 라운드마다 128 bit 키를 생성하여 사용한다.

입/출력 블록 크기를 Nb, 암호화 키 블록 크기 Nk, 라운드 수를 Nr 라고 하면 Table 1.과 같다.

ARIA의 Round 함수는 크게 3부분으로 AddRoundKey, Substitution Layer, Diffusion Layer로 구성이 된다. AddRoundKey는 128bit 라운드 키를 입력과 Xor 연산을 한다. 치환 계층(Substitution Layer)은 2종류가 존재하고 각각은 8-bit 입/출력 S-box와 그들의 역변환으로 구성된다.

확산 계층(Diffusion Layer)은 16*16 involution 이진 행렬을 사용한 byte간의 확산 함수로 구성된다.

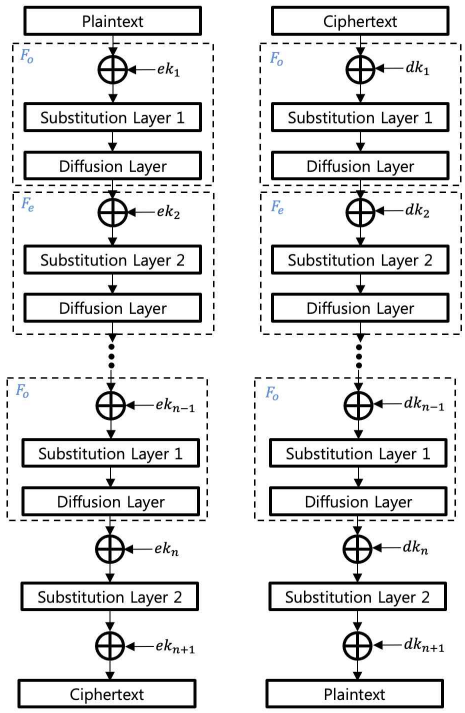

Fig. 1. Encryption/decryption of ARIA

Table 1. Parameter value by key length

	Nb	Nk	Nr
128	16	16	12
192	16	24	14
256	16	32	16

AddRoundKey 과정은 초기화 과정과 라운드 키 생성 과정으로 나뉜다. 키 초기화 과정은 초기 입력 키를 256bit로 확장하여 KL, KR 부분으로 나누고 상수값 C_1, C_2, C_3와 라운드 변환 함수(F)를 이용하여 4개의 128bit 값 W_0, W_1, W_2, W_3 을 생성한다. 그리고 라운드 키 생성 과정에는 4개의 128-bit W_0, W_1, W_2, W_3 을 조합하여 암호화 라운드 키(ek_n) 또는 복호화 라운드 키(dk_n)를 생성한다. 마지막 라운드에서 키 덧셈 계층이 2번 존재하므로 13, 15, 17개의 라운드 키를 생성해야 한다.

Substitution Layer는 8bit 입출력 S-box로 구성된다. ARIA에서 S-box는 x^{-1}와 x^{247}에 아핀 변환(affine transform)을 사용하여 S-box를 생성한다. Substitution Layer는 $S_1, S_2, S_1^{-1}, S_2^{-1}$로 구성이 된다. S-box의 성질은 다음을 만족한다.

$$S_1(x) = Bx^{-1} \oplus b, S_2(x) = Cx^{247} \oplus c$$

여기서 B, C는 8*8 정칙 행렬(non-singular m-atrix)와 b, c는 8*1 행렬의 조건을 만족한다. 또한, Substitution Layer는 2종류로 각각 S_1, S_2, S_1^{-1}, S_2^{-1}과 $S_1^{-1}, S_2^{-1}, S_1, S_2$로 구성된다. s

Diffusion Layer는 다른 블록 암호와 구별되는 주요 부분으로 16*16 involution 이진 행렬을 사용한다.

확산 함수는 $GF(2^8)^{16} \rightarrow GF(2^8)^{16}$와 같이 입력 128bit 값에 대하여 byte 단위의 행렬 곱을 수행한 결과 128bit를 출력해야 한다. ARIA의 확산 함수는 involution 구조($A^{-1} = A$)를 만족해야 하며 16*16 이진 행렬에서는 Symmetric 구조를 만족해야 한다는 뜻이다.

ARIA에서 라운드 변환 함수는 Fig. 2.와 같이 홀수 라운드 변환 함수(F_o)와 짝수 라운드 변환 함수(F_e)로 구분되며, 각 변환 함수는 Substitution Layer와 Diffusion Layer, AddRoundKey로 구성된다.

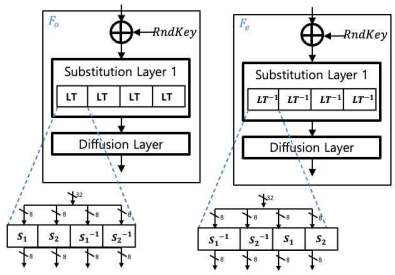

Fig. 2. Round function of ARIA

마지막 라운드에서는 Diffusion Layer가 사용되지 않고, 라운드 키 덧셈 연산으로 대체된다. 암호화 과정에서는 암호화 라운드 키(ek_n)가 사용되고 복호화 과정에서는 복호화 라운드 키(dk_n)가 사용된다.

3.3 RISC-V ISA

RISC-V ISA[10]는 오픈소스 instruction set architecture로 베이스가 되는 명령어 set을 바탕으로 개발자가 원하는 확장 명령어를 추가하여 개발할 수 있도록 지원된다. RISC-V ISA에서 기본적으로 지원하는 standard 확장 명령어 이외에도 개발자는 custom 확장 명령어를 직접 만들어 추가할 수가 있다.

RISC-V ISA에서 standard 확장 명령어 이외에 제공하는 custom 명령어를 위한 opcode는 custom-0부터 custom-3까지의 총 4개가 지원되고 각 opcode는 명령어 옵션을 통해 복수의 명령어로 구성될 수 있다.

Fig. 3.과 같이 (a)두 개의 레지스터 입력과 하나의 결과 레지스터를 사용하는 R 타입 명령어, (b) 하나의 레지스터 입력과 12bit immediate 입력, 하나의 결과 레지스터를 사용하는 I 타입 명령어, (c)두 개의 레지스터 입력과 12bit immediate 입력을 사용하는 S 타입 명령어, (d)20bit immediate 입력과 하나의 결과 레지스터를 사용하는 U 타입 명령어의 총 4가지로 구분할 수 있다.

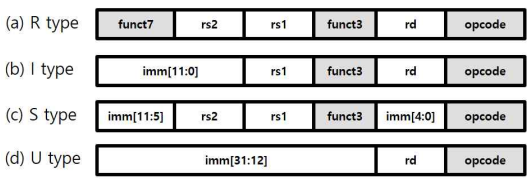

Fig. 3. instruction format of RISC-V

3.4 SPIKE

SPIKE[12]는 RISC-V ISA를 사용한 하드웨어 프로세서의 동작을 시뮬레이션하기 위해 제공되는 오픈소스 소프트웨어 시뮬레이터이다. Proxy kernel이라는 소형 kernel을 이용하여 single thread 및 multi thread 환경에서 RISC-V ISA용으로 컴파일 된 프로그램의 동작을 지원한다. 또한, SPIKE의 코드를 수정하여 개발자가 custom 명령어의 추가나 새로운 architecture의 적용 및 테스트가 가능하다.

IV. ARIA extension

본 연구에서는 RISC-V 프로세서상에서의 고속 ARIA 암/복호화를 지원하기 위해 총 10가지의 명령어를 사용한다. 명령어는 ARIA의 암호화 과정에서 키 확장과 암/복호화에 사용되는 치환 계층의 S-box 명령어 2종류와 확산 계층의 diffusion layer 명령어 8종류로 구성되며 각각의 명령어는 하나의 하드웨어 모듈에서 이루어지기 때문에, S-box 명령어의 경우 multiplexing이 다르게 이루어지고 diffusion layer에서의 동작은 실제로는 한 번의 연산으로 같은 라운드의 결과값을 구하게 된다.

4.1 ARIA Substitution Layer 명령어

하드웨어상에서 S-box를 LUT(Look Up Table) 방식으로 구현하는 것은 면적상에 비효율성을 가져오지만 한 라운드만에 연산이 가능하다는 장점이 있다. 라운드를 4라운드까지 늘리고 Galois Field operation을 사용하여 Table이 아닌 sbox를 구함으로써 클럭 주파수의 성능 향상과 면적의 장점을 얻는 architecture를 사용하였다.

이를 위해 Composite Fields를 사용한다. $GF(2^8)$는 GF(2)로부터 확장한 Field로 볼 수 있으며 $GF(((2^4)^2))$와 $GF(2^8)$ 는 isomorphic이라고 한다. 따라서 $GF(2^8)$의 원소 a(a $\in GF(2^8)$)는 $GF(((2^4)^2))$상의 원소 $a_h x + a_l (a_h, a_l \in 2^4)$

로 표현할 수 있다. 이러한 변형을 function map 이라고 정의하며 위와 같이 Field가 Composite 되어 있다고 하여 Composite Field라고 한다.

Composite Field의 구현은 $GF(((2^2)^2)^2)$, $GF((2^4)^2)$ 2가지 방법이 있다. $GF(((2^2)^2)^2)$[1] 방법은 $GF(2^2)$,$GF((2^2)^2)$의 곱셈 연산에서 각 입력을 공유하여 사용해 하드웨어 구현 시 면적을 최소화할 수 있다는 장점을 가지며, $GF((2^4)^2)$[2]은 $GF(((2^2)^2)^2)$ 보다 많은 게이트를 사용하지만 고속 연산을 할 수 있다는 장점을 가진다.

본 연구에서는 확장 명령어 추가를 통해 암호화 연산 속도를 빠르게하는 것을 목표로 하기 때문에 $GF((2^4)^2)$[2] 방법을 사용한다.

ARIA에서 필요한 S-box는 아래와 같은 조건을 만족해야 한다.

$$
\begin{aligned}
S_1(x) &= Ax^{-1} \oplus a,\\
S_1^{-1}(x) &= ((A^{-1}x) \oplus (A^{-1}a))^{-1}\\
S_2(x) &= Bx^{247} \oplus b = Bx^{-8} \oplus b\\
&= BCx^{-1} \oplus b = Dx^{-1} \oplus b\\
S_2^{-1} &= ((D^{-1}x) \oplus (D^{-1}b))^{-1}
\end{aligned}
$$

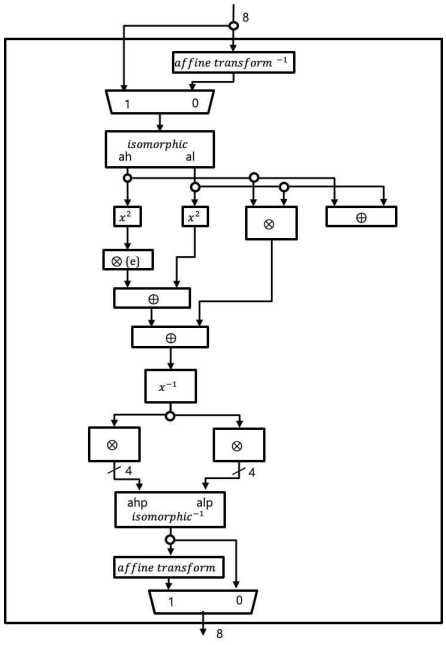

Fig. 4. S-box architecture

위의 식처럼 표현 가능하여 x^{-1}만 구한다면 $S_1, S_2, S_1^{-1}, S_2^{-1}$를 구할 수 있다.

Field의 차원이 이동되기 때문에 선형 변환 중 하나인 Affine Transform, Inverse를 구할 경우, Inverse Affine Transform을 이용해야 한다. Fig. 4.를 보면 암호화의 경우에는 마지막 부분에서 Affine Transform을 수행하고 Inverse는 처음에 Affine Transform을 수행해야 한다.[2][11] 아래는 Affine Transform을 위한 수식이다.

$$
\begin{aligned}
S_1[0] &= \overline{x_5} \oplus x_2 \oplus x_7\\
S_1[1] &= x_0 \oplus x_3 \oplus x_6\\
S_1[2] &= \overline{x_7} \oplus x_1 \oplus x_4\\
S_1[3] &= x_5 \oplus x_2 \oplus x_7\\
S_1[4] &= x_1 \oplus x_3 \oplus x_6\\
S_1[5] &= x_4 \oplus x_2 \oplus x_7\\
S_1[6] &= x_3 \oplus x_0 \oplus x_5\\
S_1[7] &= x_6 \oplus x_1 \oplus x_4
\end{aligned}
$$

$$
\begin{aligned}
S_1^{-1}[0] &= \overline{x_0} \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7\\
S_1^{-1}[1] &= \overline{x_5} \oplus x_0 \oplus x_1 \oplus x_6 \oplus x_7\\
S_1^{-1}[2] &= x_2 \oplus x_0 \oplus x_1 \oplus x_6 \oplus x_7\\
S_1^{-1}[3] &= x_7 \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3\\
S_1^{-1}[4] &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4\\
S_1^{-1}[5] &= \overline{x_1} \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5\\
S_1^{-1}[6] &= \overline{x_6} \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5\\
S_1^{-1}[7] &= x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7
\end{aligned}
$$

S_2-box를 위한 Affine Transform의 수식은 다음과 같이 나타낼 수 있다.

$$
\begin{aligned}
S_2[0] &= x_7 \oplus x_6 \oplus x_5 \oplus x_3 \oplus x_1\\
S_2[1] &= \overline{x}_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus x_3 \oplus x_2\\
S_2[2] &= x_7 \oplus x_5 \oplus x_4 \oplus x_2 \oplus x_1 \oplus x_0\\
S_2[3] &= x_7 \oplus x_6 \oplus x_1 \oplus x_0\\
S_2[4] &= x_7 \oplus x_6 \oplus x_1\\
S_2[5] &= \overline{x}_6 \oplus x_5 \oplus x_4 \oplus x_1 \oplus x_0\\
S_2[6] &= \overline{x}_7 \oplus x_6 \oplus x_2 \oplus x_1\\
S_2[7] &= \overline{x}_6 \oplus x_5 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0
\end{aligned}
$$

$$S_2^{-1}[0] = x_3 \oplus x_4$$
$$S_2^{-1}[1] = x_6 \oplus x_5 \oplus x_2$$
$$S_2^{-1}[2] = \overline{x}_6 \oplus x_4$$
$$S_2^{-1}[3] = \overline{x}_7 \oplus x_6 \oplus x_2 \oplus x_1 \oplus x_0$$
$$S_2^{-1}[4] = x_5 \oplus x_4 \oplus x_2 \oplus x_1 \oplus x_0$$
$$S_2^{-1}[5] = \overline{x}_7 \oplus x_6 \oplus x_4 \oplus x_2 \oplus x_1$$
$$S_2^{-1}[6] = x_7 \oplus x_5 \oplus x_4 \oplus x_3 \oplus x_2 \oplus x_0$$
$$S_2^{-1}[7] = x_7 \oplus x_6 \oplus x_3 \oplus x_0$$

GF(((2^4)^2))와 GF(2^8)의 isomorphic 특성을 이용하여 Composite Fields를 구성한다.

$a = a_h x + a_l ((a \in GF(2^8), a_h, a_l \in GF(2^4))$를 이용하여 Two-term Polynomial을 구성하여 inverse를 구하는 식은 아래와 같다.

$(a_h x + a_l) * (a'_h x + a'_l) = \{0\} x + \{1\}$ $(a_h, a_l, a'_h, a'_l \in GF(2^4))$와 irreducible polynomial인 $x^2 + \{1\}x + \{e\}$를 이용하면 $a_h x + a_l$의 inverse 값인 $a'_h x + a'_l$를 Extended Euclid algorithm으로 구할 수 있게 된다.

$$a'_h x + a'_l = (a_h * h)x + (a_h + a_l) * d,$$
$$d = ((a_h^2 * \{e\}) + (a_h * a_l + a_l^2)^{-1}$$

따라서 Inverse를 구하기 위해 architecture를 Fig. 4.과 같이 구성하면 되며 이를 위해서는 isomorphic mapping function이 필요하다.

$$a_h[0] = a_4 \oplus a_6 \oplus a_5$$
$$a_h[1] = a_1 \oplus a_7 \oplus a_4 \oplus a_6$$
$$a_h[2] = a_2 \oplus a_3 \oplus a_5 \oplus a_7$$
$$a_h[3] = a_1 \oplus a_7$$
$$a_l[0] = a_0 \oplus a_5 \oplus a_4 \oplus a_6$$
$$a_l[1] = a_1 \oplus a_2$$
$$a_l[2] = a_1 \oplus a_7$$
$$a_l[3] = a_2 \oplus a_4$$

위와 같이 isomorphic map function을 이용하여 GF(2^4)의 공간으로 변형하면 된다. map function 또한 다시 GF(2^8) 공간으로 변형해주어야 하므로 Inverse map function도 필요하다.

$$a = map^{-1}(a_{hp} x + a_l),$$
$$a_h, a_l \in GF(2^4), a \in GF(2^8)$$
$$a[0] = a_{lp}[0] \oplus a_{hp}[0]$$
$$a[1] = a_{hp}[0] \oplus a_{hp}[1] \oplus a_{hp}[3]$$
$$a[2] = a_{lp}[1] \oplus a_{hp}[3] \oplus a_{hp}[0] \oplus a_{hp}[1]$$
$$a[3] = a_{hp}[0] \oplus a_{hp}[1] \oplus a_{lp}[1] \oplus a_{hp}[2]$$
$$a[4] = a_{lp}[1] \oplus a_{hp}[3] \oplus a_{hp}[0] \oplus a_{hp}[1] \oplus a_{lp}[3]$$
$$a[5] = a_{hp}[0] \oplus a_{hp}[1] \oplus a_{lp}[2]$$
$$a[6] = a_{lp}[1] \oplus a_{hp}[3] \oplus a_{lp}[2] \oplus a_{lp}[3] \oplus a_{hp}[0]$$
$$a[7] = a_{hp}[0] \oplus a_{hp}[1] \oplus a_{lp}[2] \oplus a_{hp}[3]$$

이외에 Multiplication은 bit에서 AND 연산을 이용하고, Addition의 경우에는 XOR 연산을 이용한다. 연산이 길어진 경우 카르노맵을 이용하여 연산수를 최적화 한다.

4.1.1 arias1 명령어

arias1 명령어는 Fig. 5.와 같이 s-box를 이용한 치환계층(유형 1)의 연산을 수행한다. 치환계층(유형1) 연산 모듈의 입력으로는 $S_1, S_2, S_1^{-1}, S_2^{-1}$ 각 연산의 입력으로 들어갈 8bit 값을 연접한 값이 사용된다. 출력 값은 $S_1, S_2, S_1^{-1}, S_2^{-1}$ 각 연산의 출력으로 나온 8bit 출력값들을 연접하여 사용하므로 한 번에 32bit의 치환(유형 1) 연산을 수행할 수 있다.

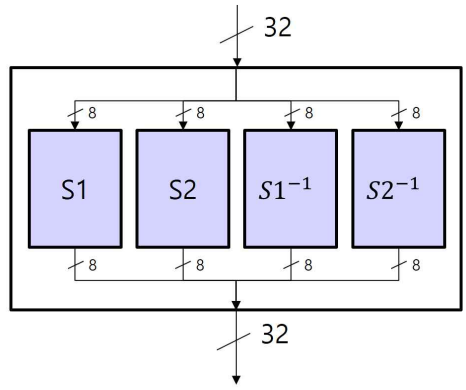

Fig. 5. arias1 operation architecture

4.1.2 arias2 명령어

arias2 명령어는 Fig. 6.과 같이 s-box를 이용한 치환계층(유형 2)의 연산을 수행한다. 치환계층(유형2) 연산 모듈의 입력으로는 $S_1^{-1}, S_2^{-1}, S_1, S_2$ 각 연산의 입력으로 들어갈 8bit 값을 연접한 값이 사용된다.

출력 값은 $S_1^{-1}, S_2^{-1}, S_1, S_2$ 각 연산의 출력으로 나온 8bit 출력값들을 연접하여 사용하므로 한 번에 32bit의 치환(유형 2) 연산을 수행할 수 있다.

Fig. 7.과 같이 두 종류의 substitution layer 명령어는 하드웨어 구조를 가지며, 명령어 수행 시 입력과 출력의 상위 16-bit와 하위 16-bit의 순서를 바꾸어주는 것으로 두 종류의 명령어를 모두 지원하는 것이 가능하다.

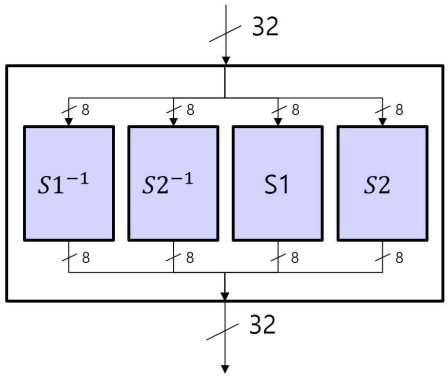

Fig. 6. arias2 operation architecture

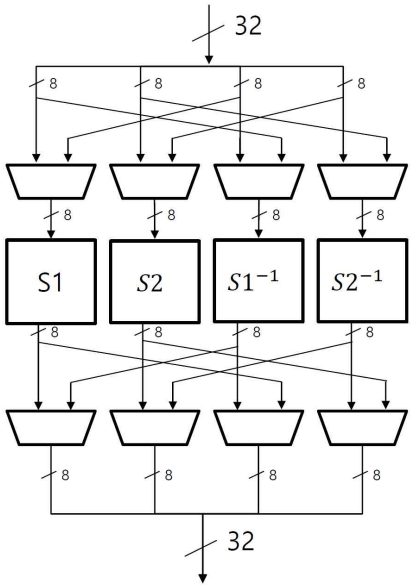

Fig. 7. hardware architecture of substitution layer

4.2 ARIA Diffusion Layer 명령어

ARIA에서 Diffusion Layer는 16*16 Involution 이진 행렬을 사용한 byte 간의 확산 함수로 구성되어 있다.

Fig. 8.의 (b)를 보면 2x2 swap이 2개 ,4x4 swap 1개를 이용하면 4라운드 후에는 처음 자신의 값으로 돌아오게 된다. 처음 들어온 32bit의 입력은 0,1,2,3의 모양으로 정렬을 하게 되고 다음 라운드의 32bit 입력에 의한 128bit 연산 결과값은 현재 라운드의 32bit 입력에 의한 128bit 연산 결과값과 xor 되어 누적된다. 처음 들어온 32bit 값은 4라운드 후에는 다시 원래의 행렬 모양인 0,1,2,3을 가지고 있게 된다. 이를 위해서는 4*4 행렬의 내부 원소 전체를 swap을 해야 하는 기능과 내부 원소 2*4 행렬끼리 swap하는 기능이 추가로 필요하게 된다.

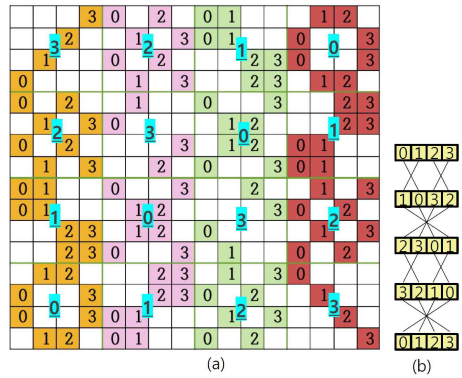

Fig. 8. Diffusion Layer operation

1,3 라운드에서 swap을 odd, 2,4 라운드에서 swap을 even으로 명칭 할 때, 1 라운드에서는 odd 동작을 수행해야 하므로 4*4 행렬의 swap을 수행 후 4*4 행렬의 내부 4*2 row끼리 swap을 진행함으로써 Fig. 9.과 같은 결과를 얻는다.

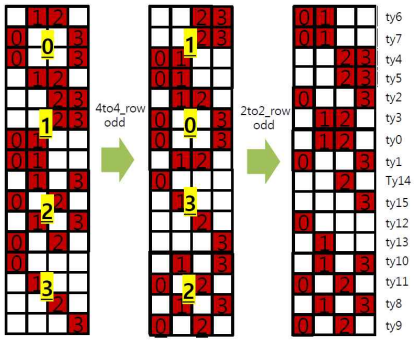

Fig. 9. Round 1 process

2라운드는 even 동작을 수행하게 된다. 2라운드 32bit 입력에 의한 128bit 연산 결과값과 1라운드 연산 결과값을 xor 하여 값을 쌓아가며 even에서는 전체 행렬을 swap 한다.

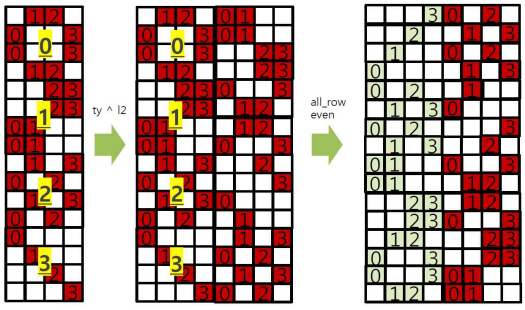

Fig. 10. Round 2 process

3라운드도 마찬가지로 2라운드 연산 결과값과 32bit 입력에 의한 128bit 연산 결과값의 xor을 수행하여 결과를 쌓아가며 4*4 행렬의 swap과 4*4 행렬 내부의 4*2 row의 swap을 수행한다.

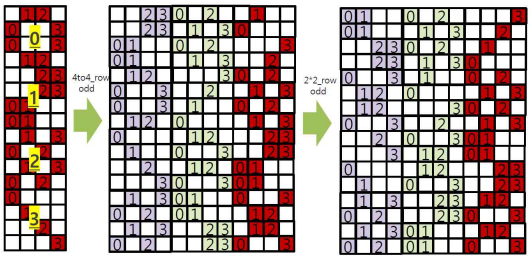

Fig. 11. Round 3 process

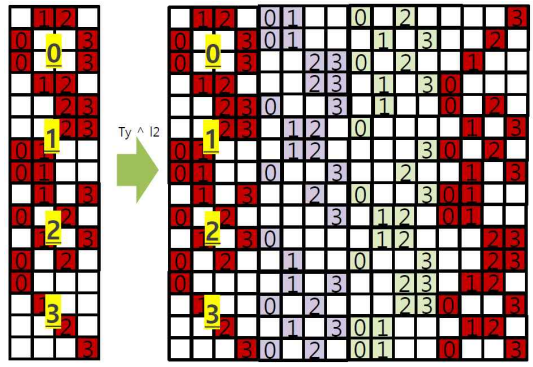

Fig. 12. Round 4-1 process

4라운드는 3라운드 연산 결과값과 32bit 입력값에 의한 128bit 연산 결과값을 xor하여 값을 쌓는다. 이후에는 모든 행렬의 모든 row를 swap 함으로써 결과를 얻을 수 있다.

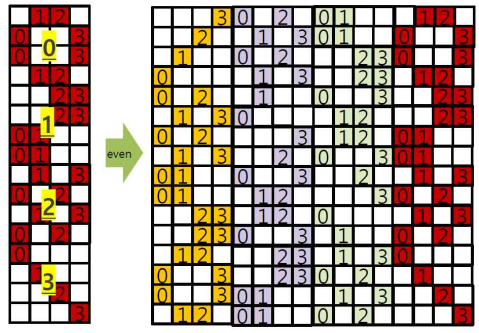

Fig. 13. Round 4-2 process

4라운드로 수행한 Diffusion Layer의 결과를 비교해보면 같은 결과를 얻을 수 있게 된다. 또한 이러한 규칙을 이용함으로써 gate의 재사용성을 높여 16*16 involution 행렬을 ROM에 구현하는 것보

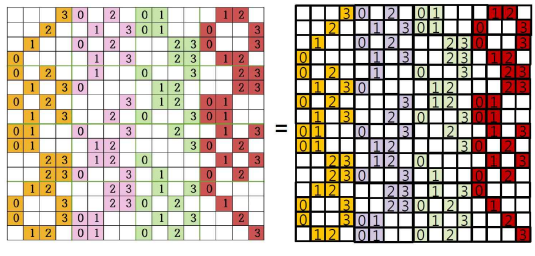

Fig. 14. Diffusion layer result comparison

다 13개의 xor과 2개의 mux만 가지고 구현이 가능하므로 면적에서 높은 이득을 얻을 수 있게 된다.

본 연구에서는 128bit 연산을 4개의 32bit 연산으로 분할하여 명령어를 구성하였으며 4개 연산의 결과값을 합쳐서 128bit의 연산 결과값을 얻을 수 있다.

4.2.1 ariad1_odd 명령어

ariad1_odd 명령어는 앞서 설명한 diffusion layer 연산 중에서 홀수 라운드에서 128bit 결과값의 〔127:96〕 bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx〔31:0〕, l2〔127:96〕
output : diff1〔31:0〕
tx0 = tx 〉〉 24 & 0x000000FF
tx1 = tx 〉〉 16 & 0x000000FF
tx2 = tx 〉〉 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx1^tx2
ty1 = tx0^tx3
ty2 = tx0^tx3
ty3 = tx1^tx2

ty= {ty0, ty1, ty2, ty3}

tz〔127:96〕 = ty ^ l2〔127:96〕

diff1 = {tz〔79:64〕,tz〔95:80〕}
```

Fig. 15. ariad1_odd pseudo code

4.2.2 ariad2_odd 명령어

ariad2_odd 명령어는 diffusion layer 연산 중 홀수 라운드에서 128bit 결과값의 〔95:64〕 bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx〔31:0〕, l2〔95:64〕
output : diff2〔31:0〕
tx0 = tx 〉〉 24 & 0x000000FF
tx1 = tx 〉〉 16 & 0x000000FF
tx2 = tx 〉〉 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx2^tx3
ty1 = tx2^tx3
ty2 = tx0^tx1
ty3 = tx0^tx1

ty= {ty0, ty1, ty2, ty3}

tz〔95:64〕 = ty ^ l2〔95:64〕

diff2 = {tz〔111:96〕,tz〔127:112〕}
```

Fig. 16. ariad2_odd pseudo code

4.2.3 ariad3_odd 명령어

ariad3_odd 명령어는 diffusion layer 연산 중 홀수 라운드에서 128bit 결과값의 〔63:32〕 bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx〔31:0〕, l2〔63:32〕
output : diff3〔31:0〕
tx0 = tx 〉〉 24 & 0x000000FF
tx1 = tx 〉〉 16 & 0x000000FF
tx2 = tx 〉〉 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx1^tx3
ty1 = tx0^tx2
ty2 = tx1^tx3
ty3 = tx0^tx2

ty= {ty0, ty1, ty2, ty3}

tz〔63:32〕 = ty ^ l2〔63:32〕

diff3 = {tz〔15:0〕,tz〔31:16〕}
```

Fig. 17. ariad3_odd pseudo code

4.2.4 ariad4_odd 명령어

ariad4_odd 명령어는 diffusion layer 연산 중 홀수 라운드에서 128bit 결과값의 〔31:0〕 bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx[31:0], l2[31:0]
output : diff4[31:0]
tx0 = tx >> 24 & 0x000000FF
tx1 = tx >> 16 & 0x000000FF
tx2 = tx >> 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx0
ty1 = tx1
ty2 = tx2
ty3 = tx3

ty= {ty0, ty1, ty2, ty3}

tz[31:0] = ty ^ l2[31:0]

diff4 = {tz[47:32],tz[63:48]}
```

Fig. 18. ariad4_odd pseudo code

4.2.5 ariad1_even 명령어

ariad1_even 명령어는 diffusion layer 연산 중 짝수 라운드에서128bit 결과값의 [127:96] bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx[31:0], l2[127:96]
output : diff1[31:0]
tx0 = tx >> 24 & 0x000000FF
tx1 = tx >> 16 & 0x000000FF
tx2 = tx >> 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx1^tx2
ty1 = tx0^tx3
ty2 = tx0^tx3
ty3 = tx1^tx2

ty= {ty0, ty1, ty2, ty3 }

tz[127:96] = ty ^ l2[127:96]

diff1 =
{tz[7:0],tz[15:8],tz[23:16],tz[31:24]}
```

Fig. 19. ariad1_even pseudo code

4.2.6 ariad2_even 명령어

ariad2_even 명령어는 diffusion layer 연산 중 짝수 라운드에서 128bit 결과값의 [95:64] bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx[31:0], l2[95:64]
output : diff2[31:0]
tx0 = tx >> 24 & 0x000000FF
tx1 = tx >> 16 & 0x000000FF
tx2 = tx >> 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx2^tx3
ty1 = tx2^tx3
ty2 = tx0^tx1
ty3 = tx0^tx1

ty= {ty0, ty1, ty2, ty3}

tz[95:64] = ty ^ l2[95:64]

diff2 =
{tz[39:32],tz[47:40],tz[55:48],tz[63:56]}
```

Fig. 20. ariad2_even pseudo code

4.2.7 ariad3_even 명령어

ariad3_even 명령어는 diffusion layer 연산 중 짝수 라운드에서 128bit 결과값의 [63:32] bit를 연산하는 부분이며 다음 알고리즘을 따른다.

```
input : tx[31:0], l2[63:32]
output : diff3[31:0]
tx0 = tx >> 24 & 0x000000FF
tx1 = tx >> 16 & 0x000000FF
tx2 = tx >> 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0  = tx1^tx3
ty1  = tx0^tx2
ty2 = tx1^tx3
ty3 = tx0^tx2

ty= {ty0, ty1, ty2, ty3}

tz[63:32] = ty ^ l2[63:32]

diff3 =
{tz[71:64],tz[79:72],tz[87:80],tz[95:88]}
```

Fig. 21. ariad3_even pseudo code

4.2.8 ariad4_even 명령어

ariad4_even 명령어는 diffusion layer 연산 중 짝수 라운드에서 128bit 결과값의 [31:0] bit를

```
input : tx[31:0], l2[31:0]
output : diff4[31:0]
tx0 = tx >> 24 & 0x000000FF
tx1 = tx >> 16 & 0x000000FF
tx2 = tx >> 8  & 0x000000FF
tx3 = tx       & 0x000000FF

ty0 = tx0
ty1 = tx1
ty2 = tx2
ty3 = tx3

ty= {ty0, ty1, ty2, ty3}

tz[31:0] = ty ^ l2[31:0]

diff4 =
{tz[103:96],tz[111:104],tz[119:112],tz[127:120]}
```

Fig. 22. aria4_even pseudo code

연산하는 부분이며 다음 알고리즘을 따른다.

8종류의 diffusion layer 명령어는 하드웨어상에서는 하나의 모듈로 구현이 되며, 서브 라운드에 따라 기존 연산 결과값에 xor 연산이 되는 값이 바뀌는 구조를 가지기 때문에, 하나의 서브 라운드에서 odd 명령어나 even 명령어 4개가 모두 실행이 되어야 하며 두 종류의 명령어는 하나의 라운드 내에서 함께 사용될 수 없다.

V. 구 현

본 절에서는 RISC-V 프로세서를 시뮬레이션 할 수 있는 SPIKE 시뮬레이터상에 ARIA 확장 명령어를 추가하는 과정과 추가한 명령어의 검증 과정에 대하여 기술한다.

5.1 custom 명령어

본 연구에서는 ARIA 확장 명령어를 추가하고 검증하기 위해서 RISC-V ISA를 소프트웨어로 시뮬레이션이 가능한 32bit system으로 설정된 SPIKE 시뮬레이터와 LLVM[13] 컴파일러를 사용한다.

Fig. 23.에서 볼 수 있듯이, SPIKE 시뮬레이터에 custom 명령어를 추가하기 위해서 사용한 opcode 영역은 custom0와 custom1 영역이며 substitution layer 명령어와 diffusion layer

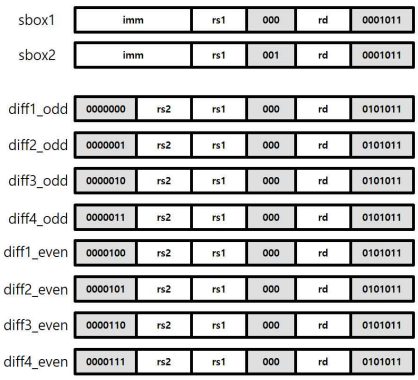

Fig. 23. custom instructions

명령어는 각각 '0001011'과 '0101011'의 7bit opcode를 사용한다.

substitution layer 명령어는 RISC-V ISA의 단일 레지스터와 12bit immediate 값을 입력으로 받는 I 타입 명령어 포맷을 사용한다.

$S_1, S_2, S_1^{-1}, S_2^{-1}$의 순서를 가지는 arias1 명령어의 경우 function3 영역에 '000'을 사용하며, $S_1^{-1}, S_2^{-1}, S_1, S_2$ 의 순서를 가지는 arias2 명령어의 경우 function3 영역에 '001'을 가진다.

diffusion layer odd 명령어는 RISC-V ISA의 두 개의 레지스터 입력을 가지는 R 타입 명령어 포맷을 사용하고 모든 하위 명령어가 function3 영역에 '000'을 가지고, ariad1 명령어의 경우 function7 영역에 '0000000'을, ariad2 명령어의 경우 '0000001'을, ariad3 명령어의 경우 '0000010'을, ariad4 명령어의 경우 '0000011'의 값을 가진다.

even 명령어의 경우 odd 명령어와 유사하게 function3 영역을 '00000100'부터 '00000111' 까지 순서대로 1씩 증가한 값을 가진다.

정해진 명령어의 opcode로 디코더를 위한 mask와 match 값을 생성하여 SPIKE 시뮬레이터의 명령어 인코딩 파일에 정의하고 각 명령어의 동작 방식을 명령어별 헤더 파일로 정의한다.

추가된 명령어의 실행을 위해서 인라인 어셈블리 코드를 바이너리 형태로 컴파일 할 수 있어야 한다. 이를 위해 LLVM의 RISC-V ISA 명령어 인코딩 파일에 추가된 10개의 명령어를 정의해준다.

5.2 검증

추가된 명령어의 검증은 KISA(Korea Internet & Security Agency)에서 제공하는 ARIA 레퍼런스 코드[9]를 이용해 진행하였다. 기존의 sbox 연산부와 diffusion layer 연산부의 코드를 추가한 명령어를 사용하는 인라인 어셈블리 명령어로 치환하여 사용한다.

각 명령어는 프로세서의 종류에 따라 소모되는 clock이 다르며, 본 연구에서는 SPIKE 시뮬레이터를 사용하여 결과를 측정하기 때문에 정확한 clock cycle의 측정이 아닌 상대적 비교를 위해 SPIKE 시뮬레이터의 내부 상태 레지스터를 사용한다.

RISC-V ISA는 프로세서 제어와 상태를 저장하기 위해 CSR(Control and Status Register)이라는 특수 목적 레지스터를 사용한다. 명령어를 실행하는데 걸리는 clock cycle의 측정은 프로세서의 clock cycle 정보를 저장하는 cycle[10]이라는 CSR을 읽어서 수행된다. 명령어 시작 시에 cycle CSR을 읽고 명령어 종료 시에 다시 cycle CSR을 읽어 두 cycle 값의 차를 연산이 실행되는데 필요한 총 clock cycle로 이용하였다.

5.3 결과

검증 결과는 Table 2.와 같으며 ARIA의 한 라운드에서의 128bit sbox 연산은 KISA의 레퍼런스 코드의 경우 581 cycle이 걸렸으며 추가한 arias1, arias2 명령어의 경우 1 cycle이 걸리지만 128bit 연산을 위해 총 4번 실행이 되므로 총 4 cycle이 걸렸다. ARIA의 한 라운드에서의 128bit diffusion layer 연산은 KISA의 레퍼런스 코드의 경우 196 cycle이 걸렸으며 추가한 홀수 라운드, 짝수 라운드의 ariad1, ariad2, ariad3, ariad4 명령어는 홀수 라운드, 짝수 라운드에 맞게 한 라운드에 4개의 명령어가 모두 4번씩 실행되어야 하므로 16 cycle이 걸렸다.

추가된 Substitution과 Diffusion layer 연산 결과를 KISA의 레퍼런스 코드와 비교했을 경우에 각각 약 135배, 13배의 성능 향상을 얻을 수 있었다. 제안된 명령어는 연산 과정 중에 메모리 참조가 없기 때문에 보다 빠른 연산 속도를 얻을 수 있었으며 특히, 치환계층 연산의 경우 reference 연산이 byte 단위 치환 연산을 수행하기 때문에 제안된 명령어와의 속도 차이가 더욱 크게 나타났다.

Table 2. cycles per operations

operations	reference substitution layer	arias1/arias2 * 4
cycles	543	4
operations	reference diffusion layer	{ariad1,ariad2 ,ariad3,ariad4}
cycles	211	16
operations	reference round key generation (encryption)	suggested round key generation (encryption)
cycles	24,913	22,073
operations	reference round key generation (decryption)	suggested round key generation (decryption)
cycles	30,172	25,182
operations	reference encryption with 128-bit key, 128-bit message	suggested encryption with 128-bit key, 128-bit message
cycles	9314	319
operations	reference decryption with 128-bit key, 128-bit message	suggested decryption with 128-bit key, 128-bit message
cycles	9312	317

주로 라운드 함수가 많이 사용되는 암/복호화 연산에 비해 라운드키 생성 연산은 라운드 함수의 사용이 적기 때문에, 암/복호화 연산은 소모 cycle이 크게 줄어들었고 라운드키 생성 연산은 비교적 소모 cycle의 감소가 적었다.

VI. 결 론

본 연구에서는 국내에서 개발된 블록암호 알고리즘 ARIA의 RISC-V 프로세서상에서의 고속 연산을 위한 확장 명령어 셋을 추가하였다. 단순히 ARIA의 알고리즘을 그대로 확장 명령어로 구현한 것이 아닌 하드웨어상에서의 효율적인 구조로 구현하여 32bit 프로세서에서 동작하기에 효과적인 확장 명령어 셋을 구현할 수 있었다. 기존의 소프트웨어 암호화 연산과 비교하여 큰 성능 개선이 이루어졌음을 알 수 있다.

References

[1] D. Canright. "A very compact S-box for AES," International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, pp. 441-455, Sep. 2005.

[2] J. Wolkerstorfer, E. Oswald, and M. Lamberger, "An ASIC implementation of the AES SBoxes," Cryptographers' Track at the RSA Conference. Springer, Berlin, Heidelberg, pp. 67-78, Feb. 2002.

[3] Akashi Satoh, Sumio Morioka, and et al, "A compact Rijndael hardware architecture with S-box optimization," International Conference on the Theory and Application of Cryptology and Information Security, Springer, Berlin, Heidelberg, vol. 2248, pp. 239-254, Nov. 2001.

[4] Hodjat, Alireza, et al. "A 3.84 Gbits/s AES Crypto Coprocessor with Modes of Operation in a 0.18-μm CMOS Technology," Proceedings of the 15th ACM Great Lakes symposium on VLSI, pp. 60-63, Apr. 2005.

[5] Tehrani, Etienne, et al. "RISC-V Extension for Lightweight Cryptography," 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 222-228, Aug. 2020.

[6] AKDEMIR, Kahraman, and et al. "Breakthrough AES performance with intel AES new instructions White paper," Jun. 2010.

[7] STOFFELEN, Ko. "Efficient cryptography on the RISC-V architecture," International Conference on Cryptology and Information Security in Latin America, Springer, Cham, pp. 323-340, Sep. 2019.

[8] T. Fritzmann, G. Sigl, J. Sepúlveda, "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography," ACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2020, pp. 239-280, Aug. 2020.

[9] KISA ,"block cipher algorithm ARIA specification," https://seed.kisa.or.kr/kisa/Board/19/detailView.do, Feb. 2021.

[10] RISC-V ,RISC-V ISA Specifications, https://riscv.org/technical/specifications/, Feb. 2021.

[11] Sang-Woo Lee, Sang-Jae Moon, and Jeong-Nyeo Kim, "High-Speed Hardware Architectures for ARIA with Composite Field Arithmetic and Area-Throughput Trade-Offs," ETRI journal, vol. 30, pp. 707-717, Oct. 2008.

[12] SPIKE, RISC-V ISA Simulator, https://github.com/riscv/riscv-isa-sim, Feb. 2021.

[13] LLVM ,LLVM compiler, https://llvm.org, Feb. 2021.

[14] R. Atri, P. K. Dubey, and et al, "Efficient Rijndael encryption implementation with composite field arithmetic," In CHES2001, vol. 2162, pp. 171-184, Sep. 2001.

[15] E. omer, D.A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," Journal of Cryptology, vol. 23, no. 1, pp. 37-71, Jul. 2010.

〈저자소개〉

이 진 재 (Jin-jae Lee) 학생회원
2020년 2월: 부산대학교 정보컴퓨터공학과 졸업
2020년 3월~현재: 부산대학교 정보융합공학과 석사과정
〈관심분야〉 정보보호, 디지털 회로 설계, 인공지능 가속기

박 종 욱 (Jong-uk Park) 정회원
2021년 2월: 부산대학교 정보컴퓨터공학과 수료
2021년 3월~현재: 부산대학교 사물인터넷 연구센터 연구원
〈관심분야〉 정보보호, 디지털 회로 설계, 인공지능 가속기

김 민 재 (Min-jae Kim) 정회원
2020년 8월: 부경대학교 제어계측공학과 졸업
2020년 9월~현재: 부산대학교 정보융합공학과 석사과정
〈관심분야〉 정보보호, 디지털 회로 설계, 부채널 분석

김 호 원 (Ho-won Kim) 종신회원
1993년 2월: 경북대학교 전자공학과 학사 졸업
1995년 2월: 포항공과대학교 전자전기공학과 석사 졸업
1999년 2월: 포항공과대학교 전자전기공학과 박사 졸업
2008년 2월: 한국전자통신연구원 정보보호연구단 선임연구원/팀장
2008년 3월~현재: 부산대학교 전기컴퓨터공학부 정교수
〈관심분야〉 암호 칩, 보안모듈, 블록체인 플랫폼, AI 보안