KIPS Trans. Comp. and Comm. Sys.
Vol.10, No.8 pp.223~230
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2021.10.8.223

Optimized Implementation of Block Cipher PIPO in Parallel-Way on 64-bit ARM Processors

Si Woo Eum† · Hyeok Dong Kwon†† · Hyun Jun Kim†† · Kyoung Bae Jang†† · Hyun Ji Kim† · Jae Hoon Park† · Gyeung Ju Song† · Min Joo Sim† · Hwa Jeong Seo†††

ABSTRACT

The lightweight block cipher PIPO announced at ICISC'20 has been effectively implemented by applying the bit slice technique. In this paper, we propose a parallel optimal implementation of PIPO for ARM processors. The proposed implementation enables parallel encryption of 8-plaintexts and 16-plaintexts. The implementation targets the A10x fusion processor. On the target processor, the existing reference PIPO code has performance of 34.6 cpb and 44.7 cpb in 64/128 and 64/256 standards. Among the proposed methods, the general implementation has a performance of 12.0 cpb and 15.6 cpb in the 8-plaintexts 64/128 and 64/256 standards, and 6.3 cpb and 8.1 cpb in the 16-plaintexts 64/128 and 64/256 standards. Compared to the existing reference code implementation, the 8-plaintexts parallel implementation for each standard has about 65.3%, 66.4%, and the 16-plaintexts parallel implementation, about 81.8%, and 82.1% better performance. The register minimum alignment implementation shows performance of 8.2 cpb and 10.2 cpb in the 8-plaintexts 64/128 and 64/256 specifications, and 3.9 cpb and 4.8 cpb in the 16-plaintexts 64/128 and 64/256 specifications. Compared to the existing reference code implementation, the 8-plaintexts parallel implementation has improved performance by about 76.3% and 77.2%, and the 16-plaintext parallel implementation is about 88.7% and 89.3% higher for each standard.

Keywords : PIPO Block Cipher, 64-bit ARM Processor, Parallel Optimal Implementation

64-bit ARM 프로세서 상에서의 블록암호 PIPO 병렬 최적 구현

엄 시 우† · 권 혁 동†† · 김 현 준†† · 장 경 배†† · 김 현 지† · 박 재 훈† · 송 경 주† · 심 민 주† · 서 화 정†††

요 약

ICISC'20에서 발표된 경량 블록암호 PIPO는 비트 슬라이스 기법 적용으로 효율적인 구현이 되었으며, 부채널 내성을 지니기에 안전하지 않은 환경에서도 안정적으로 사용 가능한 경량 블록암호이다. 본 논문에서는 ARM 프로세서를 대상으로 PIPO의 병렬 최적 구현을 제안한다. 제안하는 구현물은 8평문, 16평문의 병렬 암호화가 가능하다. 구현에는 최적의 명령어 활용, 레지스터 내부 정렬, 로테이션 연산 최적화 기법을 사용하였다. 또한 레지스터 내부 정렬을 매 라운드마다 진행하는 구현물과, 정렬을 최소화하는 구현물 두 종류로 구분하여 구현한다. 구현은 A10x fusion 프로세서를 대상으로 한다. 대상 프로세서 상에서, 기존 레퍼런스 PIPO 코드는 64/128, 64/256 규격에서 각각 34.6 cpb, 44.7 cpb의 성능을 가지나, 제안하는 기법 중, 일반 구현물은 8평문 64/128, 64/256 규격에서 각각 12.0 cpb, 15.6 cpb, 16평문 64/128, 64/256 규격에서 각각 6.3 cpb, 8.1 cpb의 성능을 보여준다. 이는 기존 대비 각 규격별로 8평문 병렬 구현물은 약 65.3%, 66.4%, 16평문 병렬 구현물은 약 81.8%, 82.1% 더 좋은 성능을 보인다. 레지스터 최소 정렬 구현물은 8평문 64/128, 64/256 규격에서 각각 8.2 cpb, 10.2 cpb, 16평문 64/128, 64/256 규격에서 각각 3.9 cpb, 4.8 cpb의 성능을 보여준다. 이는 기존 레퍼런스 코드 구현물 대비 각 규격별로 8평문 병렬 구현물은 약 76.3%, 77.2%, 16평문 병렬 구현물은 약 88.7% 89.3% 더 향상된 성능을 가진다.

키워드 : PIPO 블록암호, 64-bit ARM 프로세서, 병렬 최적 구현

1. 서 론

사물인터넷 기기와 같은 제한된 저사양 환경(성능, 메모리, 전력 등)을 위해 다양한 경량 암호가 개발되었다. 하지만 대부분의 경량 암호는 암호화가 진행되는 동안의 각종 부차적인 정보(시간, 전력, 소리 등)을 분석하여 암호화에 사용된 키를 부채널 공격(Side-Channel Attack)으로 획득할 수 있다는 취약점이 존재한다[1].

경량 블록암호 PIPO는 부채널 내성을 지니도록 개발 된 암호로서, 내재된 가능성이 각광받는 암호이다. 본 논문에서

※ 이 논문은 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.2018-0-00264, IoT 융합형 블록체인 플랫폼 보안 원천 기술 연구, 50%) 그리고 이 논문은 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.2021-0-00540, GPU/ASIC 기반 암호알고리즘 고속화 설계 및 구현 기술개발, 50%).
※ 이 논문은 2021년 한국정보처리학회 춘계학술발표대회의 우수논문으로 "64-bit ARM 프로세서 상에서의 블록암호 PIPO 병렬 최적 구현"의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 한성대학교 IT융합공학부 석사과정
†† 준 회 원 : 한성대학교 정보컴퓨터공학과 박사과정
††† 종신회원 : 한성대학교 IT융합공학부 조교수
Manuscript Received : June 17, 2021
Accepted : July 14, 2021
* Corresponding Author : Hwa Jeong Seo(hwajeong84@gmail.com)

는 고성능 프로세서 중 하나인 ARM 프로세서 경량 블록암호 PIPO의 병렬 구현을 통한 성능 개선 기법에 대해 제안한다.

이어지는 구성은 다음과 같다. 2장에서는 병렬 구현, PIPO 블록암호, 64-bit ARM 프로세서에 대해서 설명한다. 3장에서는 병렬 최적 구현을 위한 최적 구현 기법에 대해 설명한다. 4장에서는 기존 PIPO 구현물과 제안하는 구현물의 성능을 비교한다. 마지막으로 5장에서는 본 논문의 결론을 내린다.

2. 배 경

2.1 병렬 구현

병렬 구현(Parallel Implementation)은 크게 데이터 병렬(Data parallelism)과 작업 병렬(Task parallelism) 방식으로 나뉜다. 데이터 병렬은 많은 데이터가 있을 때 데이터를 효율적을 나누어 병렬 처리를 하는 방식이다. 작업 병렬은 스케쥴링 기법과 같이 작업을 나누어 병렬 처리를 하는 방식이다. 본 논문에서는 데이터 병렬 방식을 활용하여 다수의 평문을 한번에 암호화하는 것을 제안한다. 기존 방식인 하나의 평문만 암호화 하는 것에 비해 다수의 평문을 한번에 암호화 하기 때문에 같은 시간에 더 많은 암호문을 암호화 할 수 있어 많은 데이터를 암호화할 때 효율적인 기법이다.

2.2 경량 블록암호 PIPO

PIPO[2]는 ICISC'20에서 발표된 경량 블록암호이다. PIPO는 다른 블록암호보다 비선형 연산을 적게 사용하여 구현되었고, 효율적인 고차 마스킹 소프트웨어 구현이 가능하며, 특히 8-bit AVR 상에서 부채널 공격 내성을 지니기에 다른 동일한 입·출력 규격을 지닌 블록암호보다 뛰어난 보안성을 제공한다.

PIPO는 SPN(Substitution Permutation Network) 구조를 채택하였다. 블록 크기는 64-bit를 사용하며, 키의 크기는 128-bit, 256-bit 두 종류를 지원한다. 전체적인 PIPO의 매개변수는 Table 1과 같다.

PIPO의 Round Function은 선형 연산을 하는 R-Layer와 비선형 연산을 하는 S-Layer로 구성된다. 각 Round Function 후 Roundkey와 XOR을 하는 반복된 Feistel 구조로 설계되었다. Fig. 1은 PIPO의 전체적인 구조이다.

라운드 키는 마스터 키 길이에 따라 다음 Table 2와 같이 생성된다.

S-layer는 LUT(Look-Up-Table)과 Bit-slice 두 가지로 구현되어 있다. 본 논문에서는 23개의 선형 비트 연산과 11개의 비선형 비트 연산만 포함하는 효율적인 Bit-slice 구현을 사용한다.

R-layer는 하드웨어 및 소프트웨어에서 효율적인 구현을 보장하기 위해 Byte 단위의 비트 회전만 사용하는 비트 순열로 구현되어 있다. Fig. 2는 R-Layer와 S-Layer을 나타낸다.

Table 1. Parameters PIPO of Block Cipher

Type	Block size	Key size	Rounds
64/128	64-bit	128-bit	13
64/256	64-bit	256-bit	17

Table 2. Pseudocode of PIPO Key Scheduling K^{256}

Input : Key K^{128} or K^{256}
Output : RoundKey RK_i (i = 0 to 13 or 17)

IF K^{128}
  For i = 0 to 13
    $K = K_1 \| K_0$
    $RK_i = K_{i \bmod 2}$
Else if
  For i = 0 to 17
    $K = K_3 \| K_2 \| K_1 \| K_0$
    $RK_i = K_{i \bmod 4}$
Return RK

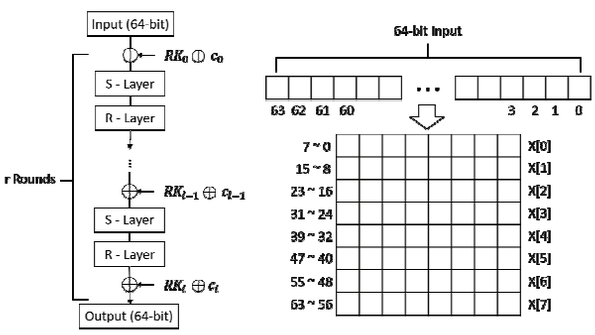

Fig. 1. Structure of PIPO Block Cipher

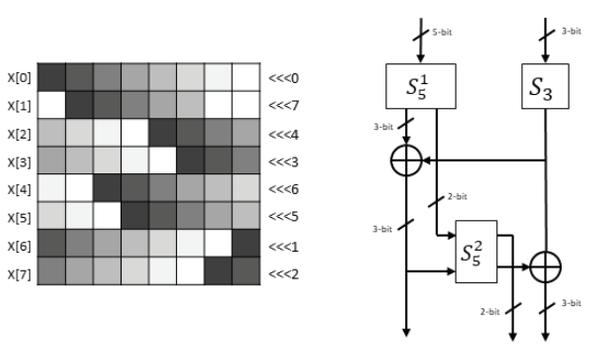

Fig. 2. (Left) R-Layer
(Right) Unbalanced-Bridge Structure S-Layer

2.3 대상 프로세서 64-bit ARM 프로세서

ARM 프로세서는 AVR을 위시한 사물인터넷 환경에서, 고성능을 자랑하는 경량 프로세서 중 하나이다. ARM 프로세서 중, ARMv8-A (ARMv8)은 ARM 프로세서 제품군 중에서 최신 프로세서 중 하나로 32-bit인 AArch32와 64-bit인 AArch64로 나뉘어진다. 본 논문에서는 ARMv8의 두 종류 중 하나인, 64-bit AArch64를 대상으로 한다. AArch64는 스칼라 레지스터(Scalar register), 벡터 레지스터(Vector register) 두 종류의 레지스터가 존재한다[3].

스칼라 레지스터는 31개(X0-X30)의 64-bit 레지스터를 의미한다. 일반적으로 스칼라 레지스터는 64-bit로 운영되나, 다른 레지스터 지시자(W0-W30)을 사용할 경우 32-bit로 사용할 수 있다. 벡터 레지스터는 32개(V0-V31)의 128-bit 레지스터를 의미한다. 벡터 레지스터는 arrangement를 사용하여 레지스터 내에 값을 일정 단위로 저장할 수 있다. 이를 Packing이라 칭한다. Packing 단위로는 8-bit B, 16-bit H, 32-bit S, 그리고 64-bit D가 존재한다. 벡터 레지스터를 사용할 때는 128-bit를 사용하거나, 또는 이것의 절반인 64-bit만 사용한다. 예를 들어, 8-bit Pakcing 단위인 B를 사용한다면, 8B 또는 16B만 사용 가능하다. 이외에도 특수 목적 레지스터로 ZR (Zero Register), PC(Program Counter), SP(Stack Pointer), ELR(Exception Link Register)가 있다. 함수 호출 시 함수 인자는 최대 8개까지 레지스터(X0-X7)을 통해 전달되고, callee saved(X19-X29) 레지스터는 사용하기 전에 레지스터 내부 값을 보존한 후, 함수가 종료되기 전에 값을 복원해야 한다.

AArch64는 산술 연산 시 64-bit 정수 데이터를 1개의 instruction으로 처리할 수 있다. 이전에는 여러 개의 instruction이 필요했던 것에 비해 연산에 필요한 클록 사이클 수가 절약된다.

2.4 관련 연구 동향

ARMv8 프로세서 상에서 구현한 다양한 블록암호에 대해서 확인한다.

Song 등[4]는 ARX 기반의 블록암호 CHAM과 HIGHT를 ARMv8 프로세서 상에서 병렬 구현하는데 집중하였다. 제안하는 기법은 레지스터 사용을 최적화 했으며, 부채널 공격 주 하나인 Fault attack에 내성을 지니는 모델을 제시하였다. Song 등[4]에서 제안한 HIGHT의 경우 기존 대비 50%, CHAM은 64/128, 128/128, 128/256 규격 별로 30%, 80%, 70%의 성능 향상을 보여줬다.

Seo[5]는 ARX 기반의 블록암호 LEA를 ARMv8 프로세서에 속하는 Apple A7, Apple A9 프로세서 상에서 24개의 평문을 병렬 연산하도록 구현하였다. 제안하는 기법으로는 레지스터를 정렬하여 정상적인 병렬 연산이 가능하게 하였으며, 메모리 접근 횟수를 최소화할 수 있도록 모든 레지스터를 활용하였다. 제안하는 기법은 ARMv8 상에서 2.4, 2.2 cpb (Cycle per byte)를 달성하였다.

Song 등[6]은 ARX 기반의 블록암호 LEA, HIGHT 그리고 CHAM을 ARMv8 상에서 구현하였다. Song 등[6]에서는 블록암호 운용모드 중 카운터 모드를 적용하여, 논스를 사용하여 중복되는 연산 부분을 저장하고, 이후 블록의 암호화 시에 이를 호출하여 대량의 명령어를 생략하는 기법을 적용하였다. 제안 기법은 LEA, HIGHT, CHAM 알고리즘 별로 최대 8.76%, 8.62% 그리고 15.87%의 성능 개선을 보였다.

3. 제안기법

제안하는 기법은, 블록암호 PIPO를 ARM 프로세서 상에서 병렬 최적 구현하는 기법을 제시한다. 제안하는 구현물은 명령어 활용, 레지스터 스케쥴링의 기본적인 설정을 확인한다. 이후 레지스터 내부 정렬, 로테이션 연산 최적화, 그리고 레지스터 내부 정렬 최소화의 세 가지 최적 기법을 제시한다.

3.1 명령어 활용

ARM 프로세서에는 병렬 연산을 위한 벡터 명령어(Vector instruction)가 제공된다. 구현을 위해 최소한의 명령어들을 사용하였으며, 구현에 사용된 명령어는 Table 3에서 확인이 가능

Table 3. Summarized Instruction Set for Optimized PIPO in Parallel-way,
Vd: Destination Vector Register, Vn/Vm: Source Vector Register, Vt: Transferred Vector Register, Xn: Source Scalar Register

Instruction	Operand	Description	Operation
AND	Vd, Vn, Vm	Bitwise AND	Vd ← Vn & Vm
EOR	Vd, Vn, Vm	Bitwise Exclusive OR	Vd ← Vn ⊕ Vm
LD1	Vd, (Xn)	Load one single-element	Vd ← (Xn)
LD1	Vd1-4, (Xn)	Load multiple single-element	Vd1-4 ← (Xn)
MOV	Vt, Vn	Move vector	Vt ← Vn
NOT	Vd, Vn	Bitwise NOT	Vd ← !Vn
ORR	Vd, Vn, Vm	Bitwise inclusive OR	Vd ← Vn | Vm
SLI	Vd, Vn, #shift	Shift Left and Insert immediate	Vd ← Vn 〈〈 #shift
SRI	Vd, Vn, #shift	Shift Right and Insert immediate	Vd ← Vn 〉〉 #shift
ST1	Vt1-4, (Xn)	Store multiple single-element	(Xn) ← Vt1-4
TRN1	Vd, Vn, Vm	Transpose vectors primary	Vd ← Vn[even], Vm[even] Vd ← Vn[odd], Vm[odd]
TRN2	Vd, Vn, Vm	Transpose vectors secondary	
UZP1	Vd, Vn, Vm	Unzip vectors primary	
UZP2	Vd, Vn, Vm	Unzip vectors secondary	
ZIP1	Vd, Vn, Vm	Zip vectors primary	
ZIP2	Vd, Vn, Vm	Zip vectors secondary	

Table 4. Register Scheduling

Register	Usage	Remarks
v0-v3	PT0-7	
v4-v7	PT8-15	For 16-PT only
v8-v15	Register alignment	v11-v15 used multiple purpose
v11-v17	S-layer, R-layer	
v29-v31	S-box value	v31 used for multiple purpose
v31	round key	
x0-x2	pointer address	

하다. 해당 명령어들은 모두 벡터 명령어이며 스칼라 명령어(Scalar instruction)은 사용되지 않았다. 표기상 Arrangement는 생략하였다.

3.2 레지스터 스케쥴링

한정된 레지스터를 활용하기 위해서 레지스터 스케쥴링을 진행한다. 기본적으로 벡터 레지스터는 128-bit까지 저장할 수 있으며, PIPO의 평문 크기는 64-bit이므로 최대 두 개 까지 저장할 수 있다. 8평문 구현물의 경우 4개의 벡터 레지스터를 사용하며, 16평문 구현물은 8개의 벡터 레지스터를 사용한다. 또한 두 구현물 공통적으로 레지스터 내부 정렬을 위해 중간 값이 거쳐 가는 레지스터가 8개 필요하다. 이 과정에서 12 또는 16개의 레지스터가 소요된다.

라운드 키는 한번에 64-bit씩 사용하므로 하나의 벡터 레지스터를 할당한다. S-layer에서 5개의 레지스터가 필요하고 R-layer에서 7개의 레지스터가 필요하다. 하지만 모든 레지스터가 동시에 필요하지 않으므로, 일부 레지스터를 다목적으로 사용한다면 레지스터의 실질적 소모량은 8평문의 경우 소요되는 레지스터는 17개이며 16평문은 21개가 필요하다. 실제 사용한 레지스터는 Table 4와 같다.

벡터 레지스터는 12개가 남지만, 더 많은 평문을 암호화할 수 없다. 8평문을 추가한다면 4개의 벡터 레지스터를, 16평문을 추가한다면 8개의 벡터 레지스터를 사용하게 된다. 하지만 Table 4에서 확인 가능하듯이, 평문 외에도 추가로 소요되는 레지스터가 존재하기 때문에, 추가적인 평문의 암호화를 진행할 수 없다.

Table 4에서 사용된 3개의 스칼라 레지스터는 연산중에는 사용하지 않으며, Pointer address 목적으로만 사용한다.

3.3 레지스터 내부 정렬

S-layer는 치환 연산, R-layer는 로테이션 연산을 진행한다. 이때 하나의 레지스터에 동일한 평문 인덱스의 값들만 들어있다면, 하나의 명령어로 다수의 평문이 S-layer를 통과하도록 할 수 있다. 벡터 레지스터는 128-bit를 저장할 수 있으므로, 초기 상태에서는 64-bit의 평문(X[0]-X[7]) 두 개를 저장할 수 있다. 하나의 평문 블록 X는 8-bit이므로, 벡터 레지스터 하나에는 동일한 인덱스의 평문 블록을 최대 16개까지 저장할 수 있다. 즉, 하나의 벡터 레지스터에 첫 번째부터 마지막 평문 16개의 X[0] 블록을 저장할 수 있다. 나머지 블록에 대해서도 동일하게 하나의 벡터 레지스터에 저장할 수 있다. Fig. 3은 본 과정에 사용하는 일부 레지스터를 표현한 것이다. 내부 값의 이동을 시각적으로 확인할 수 있다.

이를 구현하기 위해서 Table 5의 코드를 사용한다. 코드는 16

Table 5. Register Value Alignment Code

Pre alignment code 16-PT	Post alignment code 16-PT
UZP1.4s v8, v0, v1	ZIP1.2d v8, v0, v4
UZP1.4s v9, v2, v3	ZIP1.2d v9, v2, v6
UZP2.4s v10, v0, v1	ZIP1.2d v10, v1, v5
UZP2.4s v11, v2, v3	ZIP1.2d v11, v3, v7
UZP1.4s v12, v4, v5	ZIP2.2d v12, v0, v4
UZP1.4s v13, v6, v7	ZIP2.2d v13, v2, v6
UZP2.4s v14, v4, v5	ZIP2.2d v14, v1, v5
UZP2.4s v15, v6, v7	ZIP2.2d v15, v3, v7
UZP1.8h v0, v8, v9	ZIP1.16b v0, v8, v10
UZP1.8h v1, v10, v11	ZIP1.16b v1, v9, v11
UZP2.8h v2, v8, v9	ZIP2.16b v2, v8, v10
UZP2.8h v3, v10, v11	ZIP2.16b v3, v9, v11
UZP1.8h v4, v12, v13	ZIP1.16b v4, v12, v14
UZP1.8h v5, v14, v15	ZIP1.16b v5, v13, v15
UZP2.8h v6, v12, v13	ZIP2.16b v6, v12, v14
UZP2.8h v7, v14, v15	ZIP2.16b v7, v13, v15
UZP1.16b v8, v0, v1	ZIP1.8h v8, v0, v1
UZP1.16b v9, v2, v3	ZIP1.8h v9, v2, v3
UZP2.16b v10, v0, v1	ZIP2.8h v10, v0, v1
UZP2.16b v11, v2, v3	ZIP2.8h v11, v2, v3
UZP1.16b v12, v4, v5	ZIP1.8h v12, v4, v5
UZP1.16b v13, v6, v7	ZIP1.8h v13, v6, v7
UZP2.16b v14, v4, v5	ZIP2.8h v14, v4, v5
UZP2.16b v15, v6, v7	ZIP2.8h v15, v6, v7
TRN1.2d v0, v8, v12	ZIP1.4s v0, v8, v9
TRN1.2d v1, v10, v14	ZIP2.4s v1, v8, v9
TRN1.2d v2, v9, v13	ZIP1.4s v2, v10, v11
TRN1.2d v3, v11, v15	ZIP2.4s v3, v10, v11
TRN2.2d v4, v8, v12	ZIP1.4s v4, v12, v13
TRN2.2d v5, v10, v14	ZIP2.4s v5, v12, v13
TRN2.2d v6, v9, v13	ZIP1.4s v6, v14, v15
TRN2.2d v7, v11, v15	ZIP2.4s v7, v14, v15

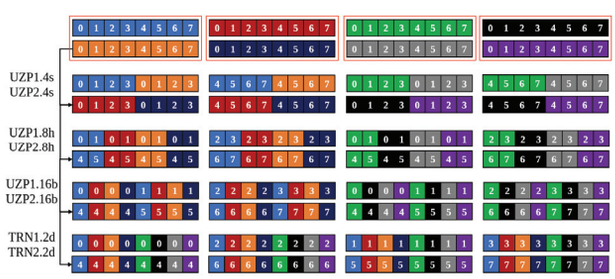

Fig. 3. Register Alignment Structure

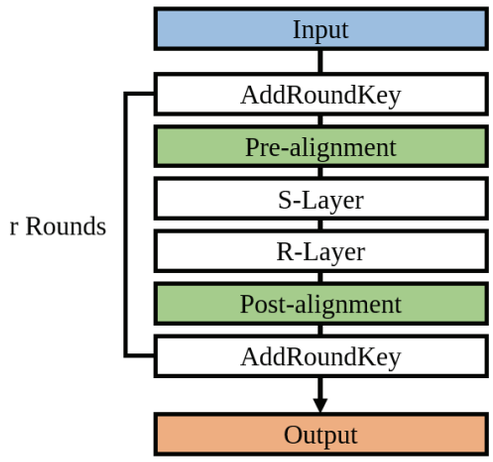

Fig. 4. Revised Algorithm Flow

평문 기준의 코드이며, 8평문은 '4s, 8h'와 같은 arrangement를 변경하는 것으로 동일한 연산을 진행할 수 있다. 코드의 좌측은 S-layer 진입 전 값을 정렬하는 코드이다. S-layer 이후에는 R-layer가 이어지므로 R-layer 진입 시에는 레지스터 정렬을 진행하지 않는다. R-layer 이후로는 라운드 키를 포함시켜야 하므로 레지스터의 배치를 복구해야 한다. 이는 Table 5의 우측 코드를 사용한다.

Fig. 3은 실제로 레지스터 내부 정렬이 어떻게 진행되는지 확인할 수 있도록 일부 레지스터를 표현하였다. 모든 평문의 동일한 인덱스가 하나의 레지스터로 모이는 것을 시각적으로 확인할 수 있다.

최종적으로, 레지스터 내부 정렬을 포함한다면 암호화 과정은 Fig. 4와 같은 형태로 변경된다.

3.4 로테이션 연산 최적화

R-layer에서는 로테이션 연산이 주로 사용된다. 다른 경량 프로세서인 8-bit AVR 프로세서의 경우, 어셈블리 명령어로 ROL, ROR과 같은 로테이션 연산을 자체적으로 지원한다. 하지만 ARM 프로세서의 벡터 명령어 중에서는 로테이션 연산을 완벽하게 지원하는 명령어가 존재하지 않는다. 대신 시프트 명령어를 활용하여 로테이션을 구현할 수 있다.

로테이션 연산 구현에는 Table 3에 나열된 명령어 중에서, SLI, SRI 명령어를 활용한다. SLI 명령어는 벡터 레지스터의 값을 왼쪽 방향으로 시프트 한 이후, 레지스터의 빈 공간에는 기존 레지스터가 가지고 있던 값을 채우게 된다. SRI 명령어는 오른쪽으로 시프트하는 차이점이 있다.

로테이션 연산의 구현은 SLI, SRI 명령어를 하나씩 조합하는 것으로 이루어진다. 이때, 원본 값은 저장할 대상 레지스터와는 다른 레지스터에 존재해야 하므로, 임시 레지스터가 한 개 필요하다. Fig. 5에서는 rotate 구현을 위한 명령어 조합을 단순한 형태로 표현하였다. 내부 값의 이동을 확인해본 결과, SLI, SRI 명령어를 조합하는 것으로 rotate 연산이 완성되는 것을 알 수 있다.

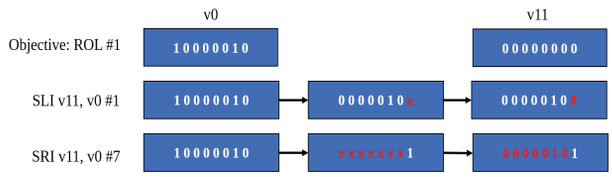

Fig. 5. Implement of Rotate Algorithm with Vector Instructions

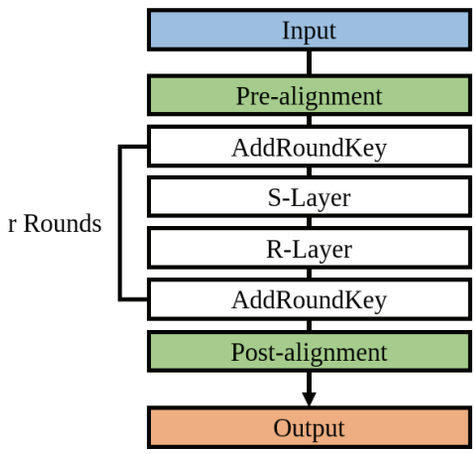

Fig. 6. Revised Algorithm Flow with Least Register-alignment

3.5 레지스터 내부 정렬 최소화

3.3절에서 레지스터 내부 정렬을 소개하였다. 레지스터 내부 정렬을 통해 벡터 레지스터를 활용하여 병렬 연산이 가능하게 되었다. 하지만, 매 라운드마다 레지스터 내부 정렬을 포함하는 관계로 내부 정렬에 많은 시간이 소요된다. 따라서 내부 정렬을 최소화하는 추가적인 기법을 제시한다.

레지스터 내부 정렬이 필요한 시점은 S-layer와 R-layer에 진입하기 직전이며, AddRoundKey는 내부 정렬을 진행하지 않은 원본 상태가 필요하다. 이점에 착안하여, AddRoundKey의 데이터를 레지스터 내부 정렬에 맞도록 변경해준다면 AddRoundKey 단계를 진행하기 위해 정렬된 상태를 복구할 필요가 없으며, 정렬 진행은 알고리즘 첫 단계에서 1회 진행하고, 정렬 복구는 마지막 라운드 종료 이후 암호문 반환 전에 1회 진행한다. 따라서 각 라운드마다 2회씩 있던 정렬을 전체 라운드에 걸쳐 단 2회만 진행하게 된다.

따라서 3.2절에서 제안한 구조는 Fig. 4와 같은 구조를 지니지만, 내부 정렬을 최소화하게 된다면 Fig. 6과 같은 구조로 변경된다.

4. 성능 평가

본 장에서는 구현물의 성능평가를 진행한다. 비교 대상으로는 ICISC'20에서 발표하며 배포된 PIPO 레퍼런스 C 코드 구현물을 사용한다.

Table 6. Comparison Result Table (Unit: cpb), 8: 8-PT in Parallel-way, 16: 16-PT in Parallel-way, l: Least-alignment

Type	Ref. C	This work8	This work16	This work8l	This work16l
64/128	34.6	**12.0**	**6.3**	**8.2**	**3.9**
64/256	44.7	**15.6**	**8.1**	**10.2**	**4.8**

구현은 Xcode 프레임워크를 사용하여 구현하며, 컴파일 옵션 -O2를 사용하여 컴파일한다. 대상 프로세서는 ARM 프로세서 중 하나인 A10X Fusion 프로세서를 대상으로 구현한다. 성능 비교의 단위로는 cpb(cycle per byte)를 사용한다. cpb 단위는 1바이트를 연산하는데 소요되는 클록 사이클의 수를 의미한다.

비교 결과는 Table 6에서 확인 가능하다.

기존 구현물은 64/128은 34.6 cpb, 64/256은 44.7 cpb의 성능을 보인다. 이에 반해, 제안하는 기법의 8평문 병렬 연산은 64/128, 64/256 규격에서 각각 12.0 cpb, 15.6 cpb의 성능을 가진다. 또한 16평문 병렬 연산은 각각의 규격에서 6.3 cpb, 8.1 cpb의 성능을 가진다. 결과적으로 레퍼런스 코드 대비 64/128, 64/256 규격 별로 8평문 병렬 구현물은 각각 65.3%, 66.4%, 16평문 병렬 구현은 각각 81.8%, 82.1% 만큼의 더 높은 성능을 보인다.

또한 최소 정렬 구현물의 경우, 8평문 병렬 연산은 64/128, 64/256 규격에서 각각 8.2 cpb, 10.2 cpb의 성능을 보여준다. 16평문 병렬 연산은 각각의 규격에서 3.9 cpb, 4.8 cpb를 가진다. 이는 기존 구현물 대비 76.3%, 77.2%, 88.7%, 그리고 89.3%의 성능 개선을 보인다.

이러한 성능 격차의 가장 큰 원인은 병렬 구현에 있다. 기존 구현물은 병렬 연산이 불가하므로 1개의 평문만을 암호화 할 수 있는 반면, 제안하는 구현물은 8개 또는 16개의 평문을 동시에 암호화 할 수 있다. 즉, 단위 시간당 처리 가능한 바이트의 수가 늘어났기 때문에 기존 대비 뛰어난 성능을 지닌다.

또한 어셈블리 명령어를 사용한 최적 구현도 성능 개선에 이바지한다. 특히 R-layer의 로테이션 연산을 단 두 개의 명령어를 사용하여 구현하는 부분에서 동작 속도를 크게 개선시킬 수 있다.

하지만 레지스터 내부 정렬을 최소화한 경우, 라운드 키를 복사 및 정렬하는 과정이 추가된다. 즉, 라운드 키가 갱신될 경우 해당 시간이 추가되기 때문에 동작 속도에 영향을 줄 수 있다. Fig. 7은 라운드 키 갱신 주기별로 성능 측정을 한 그래프이다.

Fig. 7에서 X축은 키 갱신 주기이며, Y축은 cpb 수치이다. 키 갱신 주기가 1회라면 암호화를 1회 할 때마다 키가 갱신된다는 의미이다. 레지스터 내부 정렬을 상시 진행하는 모델은 키 갱신 주기에 영향을 받지 않고 항상 같은 성능을 지닌다. 이는 라운드 키를 관리하는 부분이 따로 없기 때문에, 키가 갱신되더라도 해당 부분에 영향을 받지 않는다. 반면 레지스터 최소 정렬 모델은 키 갱신이 이루어질 때, 라운드 키를 복제 및 정렬해야 하기 때문에 해당 시간이 추가적으로 발생한다. 가령 1회 갱신 주기에는 매번 라운드 키를 관리하기 때문에 동작속도가 느려진다. 일반 정렬 모델과 최소 정렬 모델의 동작 속도가 교차하는 구간이 동일한 속도를 보여주는 갱신 주기이며, 최소 정렬 모델이 더 빨라지는 구간은 동작 속도 면에서 이득이 발생하는 갱신 주기이다.

64/128 8평문의 경우, 일반 정렬 모델이 11.97 cpb를 지니고 있다. 최소 정렬 모델은 5회 갱신 주기 때 11.97 cpb를 지니므로, 적어도 6회 갱신 주기 이후로 동작 속도 면에서 이득이 발생한다. 동일한 방법으로 그래프를 분석한다면, 64/128 16평문의 경우 3회 갱신 주기 때, 64/256 8평문은 3회 갱신 주기 때, 64/256 16평문은 2회 갱신 주기 때 일반 정렬 모델보다 더 좋은 성능을 지닌다.

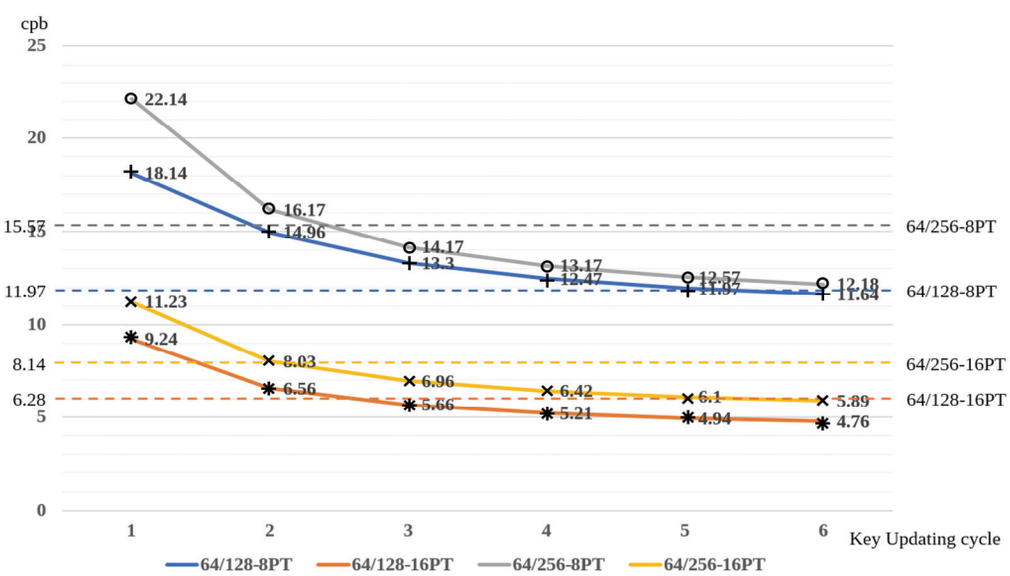

Fig. 7. Performance Measurement Graph

5. 결 론

본 논문에서는 경량 블록 암호 PIPO를 ARM 프로세서 상에서 병렬 최적 구현하는 기법을 제시하였다. 제안하는 기법은 벡터 레지스터와 벡터 명령어을 활용하여 8평문 또는 16평문을 동시에 암호화하는 병렬 구현을 시도하였다. 또한 최적 구현을 위해 효율적인 명령어를 사용하였고 레지스터 정렬을 통해 병렬 연산이 이루어지게 하였다. 레지스터 정렬의 횟수에 따라서 일반 정렬 모델, 최소 정렬 모델 두 가지로 분리하였다. 추가로 로테이션 연산 구현을 두 개의 명령어로 처리하여 최적의 로테이션 연산을 구현하였다.

제안하는 기법을 통해 기존 레퍼런스 코드 대비 각각의 규격에서 8평문 병렬 구현은 65.3%, 66.4%, 76.3%, 77.2%, 그리고 16평문 병렬 구현은 81.8%, 82.1%, 88.7%, 89.3% 만큼의 더 높은 성능 향상을 확인할 수 있었다. 본 논문에서는 ECB 운용모드로 구현한 연구 결과를 제시하였다. 향후 연구 과제로 병렬 연산 최적 구현 기법을 다양한 운용 모드를 활용한 연구를 제시한다.

References

[1] A. Heuser, S. Picek, S. Guilley, and N. Mentens, "Side-channel analysis of lightweight ciphers: Does lightweight equal easy?," *Lecture Notes in Computer Science*, Vol.10155, pp.91-104, 2017.

[2] H. G. Kim, et al., "A new method for designing lightweight S-boxes with high differential and linear branch numbers, and its application," *International Conference on Information Security and Cryptology (ICISC 2020)*, Seoul, Korea, pp.62, 2020.

[3] H. J. Seo, Z. Liu, P. Longa, and Z. Hu, "SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange," *Conference on Cryptographic Hardware and Embedded Systems (CHES 2018)*, Amsterdam, Netherlands, pp.19, 2018.

[4] J. G. Song and S. C. Seo, "Secure and fast implementation of ARX-Based block ciphers using ASIMD instructions in ARMv8 platforms," in *IEEE Access*, Vol.8, pp.193138-193153, 2020.

[5] H. J. Seo, "High speed implementation of LEA on ARMv8," *The Korea Institute of Information and Commucation Engineering*, Vol.21, No.10, pp.1929-1934, 2017.

[6] J. G. Song and S. C. Seo, "Efficient parallel implementation of CTR mode of ARX-Based block ciphers on ARMv8 microcontrollers," in *MDPI Applied Sciences*, Vol.11, No.6, pp.1-28, 2021.

엄 시 우

https://orcid.org/0000-0002-9583-5427
e-mail : shuraatum@gmail.com
2021년 한성대학교 IT융합공학부(학사)
2021년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 암호 구현, 인공지능

권 혁 동

https://orcid.org/0000-0002-9173-512X
e-mail : korlethean@gmail.com
2018년 한성대학교 IT응용시스템공학부 (학사)
2020년 한성대학교 IT융합공학부(석사)
2020년 ~ 현 재 한성대학교 정보컴퓨터공학과 박사과정
관심분야 : 정보보호, 암호 구현

김 현 준

https://orcid.org/0000-0002-6847-6772
e-mail : khj930704@gmail.com
2019년 한성대학교 IT응용시스템공학부 (학사)
2021년 한성대학교 IT융합공학부(석사)
2021년 ~ 현 재 한성대학교 정보컴퓨터공학과 박사과정
관심분야 : 정보보호, 부채널분석, 블록체인

장 경 배

https://orcid.org/0000-0001-5963-7127
e-mail : starj1023@gmail.com
2019년 한성대학교 IT응용시스템공학부 (학사)
2021년 한성대학교 IT융합공학부(석사)
2021년 ~ 현 재 한성대학교 정보컴퓨터공학과 박사과정
관심분야 : 정보보호, IoT, 양자컴퓨터

김 현 지

https://orcid.org/0000-0001-9828-3894
e-mail : khj1594012@gmail.com
2020년 한성대학교 IT융합공학부(학사)
2020년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 인공지능

박 재 훈

https://orcid.org/0000-0003-1725-3621
e-mail : p9595jh@gmail.com
2020년 한성대학교 IT융합공학부(학사)
2020년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 웹 보안, 블록체인

송 경 주

https://orcid.org/0000-0001-4337-1843
e-mail : thdrudwn98@gmail.com
2021년 한성대학교 IT융합공학부(학사)
2021년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 암호, 인공지능

심 민 주

https://orcid.org/0000-0001-5242-214X
e-mail : minjoos9797@gmail.com
2021년 한성대학교 IT융합공학부(학사)
2021년 ~ 현 재 한성대학교 IT융합공학부 석사과정
관심분야 : 정보보호, 부채널분석

서 화 정

https://orcid.org/0000-0003-0069-9061
e-mail : hwajeong84@gmail.com
2010년 부산대학교 컴퓨터공학과(학사)
2012년 부산대학교 컴퓨터공학과(석사)
2012년 ~ 2016년 부산대학교 컴퓨터공학과 (박사)
2016년 ~ 2017년 싱가포르 과학기술청 연구원
2019년 ~ 현 재 한성대학교 IT융학공학부 조교수
관심분야 : 정보보호, 암호화 구현, IoT