석사학위논문

ARMv8 프로세서 상에서의 해시 함수 LSH 최적 구현

2023년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

심 민 주

석 사 학 위 논 문
지도교수 서화정

ARMv8 프로세서 상에서의 해시 함수 LSH 최적 구현

Optimized Implementation of LSH Hash Function on ARMv8 Processors

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

심 민 주

석 사 학 위 논 문
지도교수 서화정

ARMv8 프로세서 상에서의 해시함수 LSH 최적화 구현

Optimized Implementation of LSH Hash Function on ARMv8 Processors

위 논문을 공학 석사학위 논문으로 제출함

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

심 민 주

심민주의 공학 석사학위 논문을 인준함

2022년 12월 일

심사위원장 최 원 석 (인)

심 사 위 원 구 동 영 (인)

심 사 위 원 서 화 정 (인)

국 문 초 록

ARMv8 프로세서 상에서의 해시 함수 LSH 최적화 구현

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
심 민 주

본 논문에서는 64-bit ARMv8 프로세서의 벡터 레지스터를 활용한 해시함수 LSH 최적화 구현을 제안한다. 해시함수 LSH 전체 연산 과정에 대해 벡터 레지스터를 활용하여 병렬 연산을 효율적으로 수행한다. 제안 기법은 LSH-256과 LSH-512에 대한 싱글 메시지 최적화 구현과 LSH-256에 대한 멀티 메시지 최적화 병렬 구현을 제안한다. LSH의 Wordperm 함수와 메시지 확장 함수에서 수행되는 순열을 구현하기 위해서는 많은 벡터 명령어가 필요하다. 따라서, 싱글 메시지 최적화 구현에서는 Wordperm과 메시지 확장함수에 대해 최적화를 수행한다. 제안 기법은 순열 P를 LSH 연산 전에 수행한다. P가 적용되어 수정된 LSH를 수행한 후에는 P의 역순열인 P^{-1}을 수행하여 연산을 마무리한다. 이 과정에서 LSH-256과 LSH-512에 각각에 적합한 P를 찾아 벡터 명령어를 사용하여 효율적으로 구현한다. 결과적으로, P가 적용되어 수정된 Wordperm 함수와 메시지 확장 함수는 P를 적용하기 전보다 더 적은 벡터 명령어를 사용하여 구현하였다. LSH의 주요 연산은

ARX(Addition, Rotation, eXclusive OR)이다. 따라서, 수정된 LSH에서 수행되는 ARX 연산에 대한 최적화를 수행한다. 멀티 메시지에 대한 최적화 병렬 구현은 레지스터 내부 정렬을 통해 서로 다른 2개의 메시지에 대해 LSH-256을 병렬 구현하였다. 이 과정에서 메시지 확장 함수에서 수행되는 덧셈 연산을 기존의 절반으로 생략하여 최적화한다. 결과적으로, LSH-256 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 한국인터넷진흥원(KISA)에서 제공한 레퍼런스 코드보다 각각 3.94배, 4.32배 성능 향상을 확인하였다. LSH-512 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스 코드보다 각각 2.34배, 1.78배 성능 향상을 확인하였다. 멀티 메시지에 대한 LSH-256 병렬 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스 코드보다 각각 5.26배, 5.61배 성능 향상을 확인하였다. 그리고 1,024-bit와 2,048-bit 메시지에 대해 LSH-256 최적화 구현 대비 각각 1.33배, 1.30배 성능 향상을 확인하였다.

【주요어】 LSH 해시함수, 소프트웨어 구현, ARMv8 프로세서, 병렬 연산

목 차

표 목 차

알고리즘 목 차

수식 목 차

그 림 목 차

제 1 장 서론

제 1 절 연구 목적

사물인터넷(Internet of Things, IoT)의 급속한 발전과 함께 IoT에 대한 보안도 매우 중요하게 고려되어야 한다. 이러한 이유로 IoT에 사용되는 임베디드 프로세서의 보안에 대한 연구가 활발히 진행되고 있다. 임베디드 프로세서는 일반 컴퓨터에 비해 메모리가 제한적이다. 따라서, 임베디드 프로세서에서 암호화 알고리즘이 효율적으로 동작하기 위한 최적화 구현 연구가 활발히 진행되고 있다. 2014년 NSR(National Security Research)에서 메시지 인증, 디지털 서명 등 다양한 암호화 응용에서 사용할 수 있는 해시 함수인 LSH(Lightweight Secure Hash)가 개발되었다.

본 논문에서는 64비트 ARMv8 프로세서에서 해시 함수 LSH 최적화 구현을 제안한다. LSH-256과 LSH-512에 대한 싱글 메시지 최적화 구현과 LSH-256에 대한 멀티 메시지 최적화 병렬 구현을 제안한다. 이때, 싱글 메시지 최적화 구현은 더 적은 벡터 명령어를 사용하기 위해 순열 P가 적용되어 수정된 LSH를 수행하고 얻은 결과 값에 P의 역순열인 P^{-1} 을 적용하는 기법을 활용한다. 따라서, ARMv8 프로세서 상에서의 효율적인 P를 찾아 LSH-256과 LSH-512에 대해 서로 다른 P를 적용하여 각각에 대해 효율적으로 구현한다. 멀티 메시지 최적화 병렬 구현은 워드 사이즈가 32인 LSH-256의 특징을 활용하여 하나의 레지스터에 서로 다른 메시지가 위치하게 레지스터 내부 정렬을 수행하여 최적화 병렬 구현을 한다.

본 논문의 구성은 다음과 같다. 2장에서는 해시 함수 LSH와 ARMv8 프로세서와 선행 연구인 Intel SIMD를 활용한 LSH 고속 구현 연구에 대해 설명한다. 3장에서는 제안 기법으로 LSH-256과 LSH-512에 대한 싱글 메시지 최적화 구현과 LSH-256에 대한 멀티 메시지 최적화 병렬 구현에 대해 설명한다. 4장에서는 제안 기법을 활용한 LSH-256과 LSH-512에 대한 싱글 메시지 최적 구현 성능과 LSH-256에 대한 멀티 메시지 최적화 병렬 구현에

대한 성능을 평가한다. 마지막으로 5장에서 결론과 향후 연구에 대해 설명하며 본 논문을 마무리 짓는다.

제 2 절 연구 기여

본 논문의 연구 기여는 다음과 같다.

첫 번째, 임베디드 프로세서인 8-bit AVR, 16-bit MSP, 32-bit ARM과 인텔 SIMD를 활용한 LSH 최적화 연구가 수행되었다. 하지만, 64-bit ARMv8 프로세서 상에서의 LSH 최적화 연구는 수행되지 않았다. 따라서, 본 논문에서는 ARM-NEON을 사용한 LSH 최적화 연구를 제안한다.

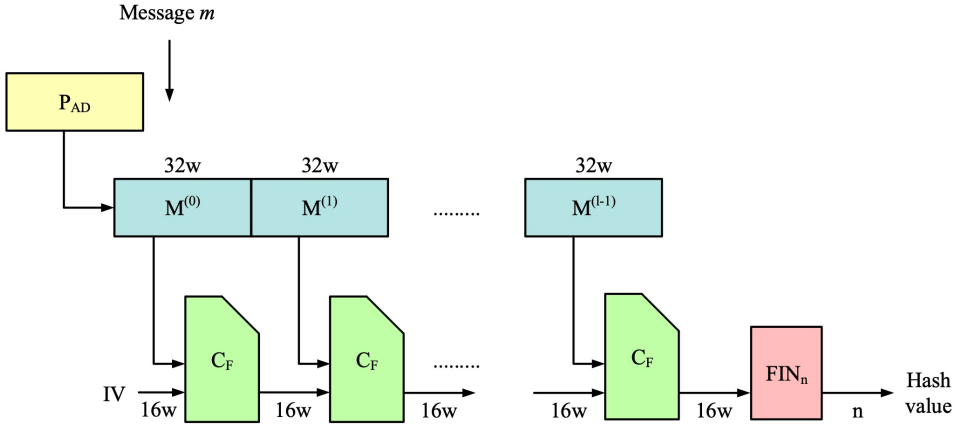
두 번째, LSH의 Wordperm 함수와 메시지 확장 함수에서 수행되는 순열 연산을 더 적은 벡터 명령어를 사용하여 구현하기 위해 LSH-256과 LSH-512에 대해 각각 ARMv8에서 제공하는 명령어를 효율적으로 사용하기에 적합한 최적의 순열 P를 찾아 이를 적용하였다. P가 적용되어 수정된 LSH는 P를 적용하기 전보다 더 적은 벡터 명령어를 사용하여 Wordperm 함수와 메시지 확장 함수에서 수행되는 순열 연산을 구현하였다.

세 번째, LSH-256에 대한 멀티 메시지 최적화 병렬 구현하였다. ARMv8에서 제공하는 벡터 레지스터는 128비트이다. 따라서, 워드 사이즈가 32인 LSH-256에 대해 하나의 레지스터에 2개의 서로 다른 메시지가 위치하게 레지스터 내부 정렬을 수행하여 최적화를 하였다. 레지스터 내부 정렬 통해 기존의 메시지 확장 함수에서 수행되던 덧셈 연산을 절반으로 생략하였다.

제 2 장 관련 연구

제 1 절 해시 함수 LSH

LSH(Lightweight Secure Hash)는 2014년 NSR(National Security Research)에서 개발한 해시 함수로, ICISC'14에서 발표되었다. LSH는 SHA-3보다 4배 빠르고, 다른 SHA-3 finalist에 진출한 암호보다 1.5~2.3배 빠르다는 특징을 갖고 있다. LSH는 총 6가지(LSH-224, LSH-256, LSH-512-224, LSH-512-256, LSH-384, LSH-512)의 규격을 제공한다. 각 규격의 파라미터는 [표 2-1]과 같다.



[그림 2-1] 해시함수 LSH

LSH는 n-bit 해시 값을 생성하기 위해 초기화 단계, 압축 단계, 종료 단계가 수행된다. LSH의 전체 구조는 [그림 2-1]과 같다.

- 초기화(Initialization) 단계 : 수신된 메시지 m에 대해 메시지 블록 비트 길이의 배수가 되도록 one-zero padding을 수행한다. 패딩이 수행된 비트 문자열 메시지를 32w 배열 메시지 블록으로 변환된다. 마지막으로 초기화 벡터

인 IV로 연결 변수를 초기화한다.

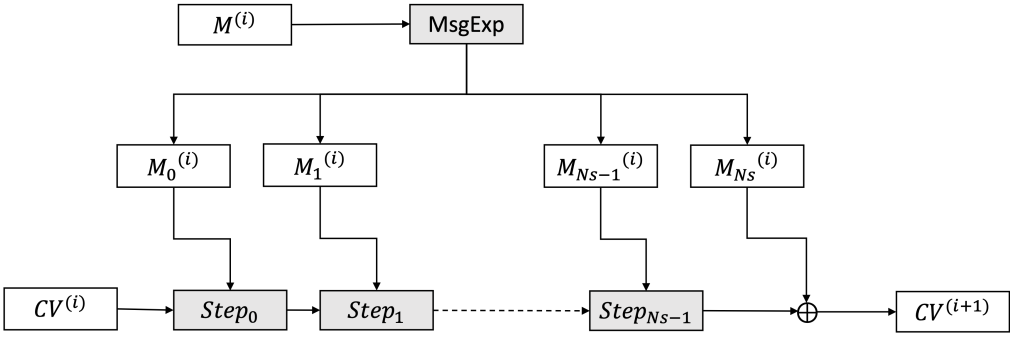
- 압축(Compression) 단계 : 연결 변수는 메시지 블록과 함께 압축 함수 C_F의 입력으로 사용된다. 이렇게 얻은 출력값으로 연결 변수 갱신하고, 마지막 메시지 블록을 처리할 때까지 반복하여 업데이트된다.

- 완료(Finalization) 단계 : 압축 단계를 통해 마지막 연결 변수에 저장된 출력값을 통해 n-bit 해시 값이 생성된다.

Type	n	N_s	연쇄 변수 비트 길이	w	메시지 블록 비트 길이
LSH-224	224	26	512	32	1,024
LSH-256	256				
LSH-512-224	224	28	1,024	64	2,048
LSH-512-256	256				
LSH-384	384				
LSH-512	512				

[표 2-1] 해시함수 LSH 규격

압축 단계에서 수행되는 압축 함수 C_F는 [그림 2-2]와 같이 수행된다.



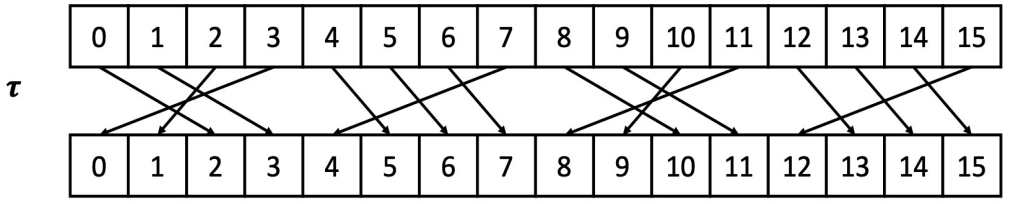
[그림 2-2] 압축 함수

압축 함수 C_F는 메시지 확장 함수(Message Expansion Function)인 MsgExp 함수를 거친 후, 단계 함수인 Step 함수가 순차적으로 수행된다. MsgExp의 식은 [수식 2-1]과 같다.

$$For\ j = 2, 3, \cdots, N_s (N_s : 26 \text{ or } 28)$$
$$M_0^{(i)} \leftarrow M^{(i)}[0],\ M^{(i)}[1],\ \cdots, M^{(i)}[15]$$
$$M_1^{(i)} \leftarrow M^{(i)}[16],\ M^{(i)}[17],\ \cdots, M^{(i)}[31]$$
$$M_j^{(i)} \leftarrow ADD(M_{j-1}^{(i)},\ M_{j-2}^{(i)}[\tau(l)]) \quad (0 \le l \le 16)$$

[수식 2-1] LSH의 메시지 확장(MspExp) 함수

MsgExp 내부에서는 [그림 2-4]와 같은 τ 순열(Permutation)이 수행된다. Step 함수는 [표 2-1]에 명시되어 있는 것과 같이 워드 값에 따라 26번 혹은 28번 수행된다.

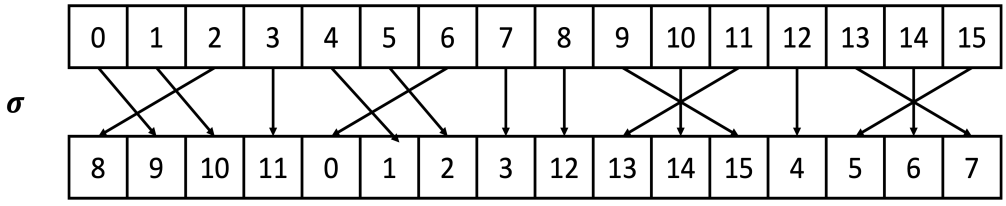


[그림 2-3] 메시지 확장 함수에서 수행되는 순열 τ

Step 함수의 식은 [수식 2-2]와 같다. Step 함수의 내부 함수는 Wordperm , Mix, MsgAdd로 이뤄져 있다. 워드 단위 순환인 Wordperm 함수는 σ 순열이 수행된다. σ 순열은 [그림 2-4]와 같이 수행된다.

$$Step_j := Wordperm \circ Mix_j \circ MsgAdd \quad (0 \le j \le (N_s - 1))$$

[수식 2-2] LSH의 Step 함수



[그림 2-4] 워드 단위 순환 함수 Wordperm의 순열 σ

Mix 함수는 두 개의 워드에 대해 ARX(Addition, Rotation, eXclusive OR)이 수행된다.

Mix Function	
Input : Word X, Y	
Output : Word X, Y	
1: $X \leftarrow X \boxplus Y$	5: $Y \leftarrow Y^{\lll \beta_j}$
2: $X \leftarrow X^{\lll \alpha_j}$	6: $X \leftarrow X \boxplus Y$
3: $X \leftarrow X \oplus SC_j[l]$	7: $Y \leftarrow Y^{\lll \gamma_i}$
4: $Y \leftarrow X \boxplus Y$	

[알고리즘 2-1] Mix 함수

[알고리즘 2-1]은 Mix 함수의 연산을 나타낸 것이다. Mix 함수에서 수행되는 비트 순환량 α_j, β_j, γ_i 은 [표 2-2]를 따른다. 비트 순환량 α_j, β_j는 홀수 라운드와 짝수 라운드에서 각각 다른 값이 적용된다.

w	j	α_j	β_j	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7
32	even	29	1	0	8	16	24	24	16	8	0
	odd	5	17								
64	even	23	59	0	16	32	48	8	24	40	56
	odd	7	3								

[표 2-2] Mix 함수에서 수행되는 비트 순환량 (α_j, β_j, γ_i)

MsgAdd 함수는 메시지 덧셈 함수로 두 개의 16w 배열에 대해 덧셈 연산이 수행된다. 덧셈 연산은 [수식 2-3]과 같이 수행된다.

$$MsgAdd(X, Y) := (X[0] \oplus Y[0], \cdots, X[15] \oplus Y[15])$$

[수식 2-3] 메시지 덧셈 함수(MsgAdd)

제 2 절 64-bit ARMv8 Processor

ARM 프로세서는 자원이 제한된 사물인터넷 환경에서 고성능을 제공하는 저사양 프로세서 중 하나이다. 최신 ARMv8-A는 32-bit AArch32(A32)와 AArch64(A64)를 제공한다. 그 중 64-bit ARMv8 프로세서는 31개의 64-bit 범용 레지스터(스칼라 레지스터)와 32개의 128-bit 벡터 레지스터를 제공한다. 병렬 구현을 지원하는 특징을 가진 벡터 레지스터는 저장된 데이터를 특정 단위로 패킹하여 처리할 수 있다. ARMv8 프로세서에서 지원하는 데이터 패킹 종류는 [표 2-3]과 같다.

[표 2-4]는 해시 함수 LSH 최적화 구현하기 위해 사용된 64-bit ARMv8 명령어 셋을 요약한 것이다. Xd는 범용 레지스터의 destination 레지스터이고, Vd는 벡터 레지스터의 destination 레지스터이다.

Type	Unit	Specifier	Data Quantity
Byte	8-bit	16b or 8b	16 or 8
Half-word	16-bit	8h or 4h	8 or 4
Single-word	32-bit	4s or 2s	4 or 2
Double-word	64-bit	2d or 1d	2 or 1

[표 2-3] ARMv8 벡터 레지스터의 데이터 패킹 종류

Xn은 범용 레지스터의 source 레지스터이고, Vn과 Vm은 벡터 레지스터의 source 레지스터이다. Vt는 벡터 레지스터의 transferred 레지스터이다. T는 [표 2-3]의 데이터 패킹을 위한 Specifier이다.

asm	Operands	Descript
ADD	Vd.T, Vn.T, Vm.T	Add
AND	Vd.T, Vn.T, Vm.T	Bitwise AND
EOR	Vd.T, Vn.T, Vm.T	Bitwise Exclusive OR
LD1	Vt.T, [Xn]	Load multiple single-element structures to one, two, three, or four registers
MOV	Vd.T, Vn.T	Move(vector)
MOV	Vd.Ts[index1], Vn.Ts[index2]	Move vector element to another vector element
MOVI	Vt.T, #imm	Move immediate(vector)
RET	{Xn}	Return from subroutine
SHL	Vd.T, Vn.T, #shift	Shift Left immediate(vector)
SRI	Vd.T, Vn.T, #shift	Shift Right and immediate(vector)
ST1	Vt.T, [Xn]	Store multiple single-element structures from one, two, three, or four registers
SUB	Xd, Xn, #imm	Subtract immediate
REV32	Vd.T, Vn.T	Reverse elements in 32-bit words
REV64	Vd.T, Vn.T	Reverse elements in 64-bit doublewords
ZIP1	Vd.T, Vn.T, Vm.T	Zip vectors primary
ZIP2	Vd.T, Vn.T, Vm.T	Zip vectors secondary
UZP1	Vd.T, Vn.T, Vm.T	Unzip vectors primary
UZP2	Vd.T, Vn.T, Vm.T	Unzip vectors secondary
TRN1	Vd.T, Vn.T, Vm.T	Transpose vectors primary
TRN2	Vd.T, Vn.T, Vm.T	Transpose vectors secondary

[표 2-4] LSH 최적화 구현에 사용된 벡터 레지스터의 명령어셋

제 3 절 SIMD를 활용한 LSH 고속화 구현 연구 동향

2019년 SIMD를 사용하여 LSH를 구현하기 위해 워드 단위 순열 기반 구현 기법이 제안되었다. 해당 기법은 순열 P와 P의 역순열인 P^{-1}를 사용한다. [수식 2-4]를 증명하여 이를 활용한 기법을 제안하였다.

$$LSH = P \circ LSH' \circ P^{-1}$$

[수식 2-4] 순열 P와 LSH

$$MsgExp' = P \circ MsgExp \circ P^{-1}$$
$$\tau' = P \circ \tau \circ P^{-1}$$

[수식 2-5] 순열 P와 메시지 확장함수

이를 통해, Intel SIMD를 사용하여 효율적으로 더 적은 SIMD 명령어를 사용하여 LSH를 최적화 구현하였다. LSH-256과 LSH-512에 대해 각각 명령어를 적게 사용할 수 있는 최적의 순열 P를 구하여 P를 적용한 기법을 제안하였다. 이때, LSH-256과 LSH-512는 서로 다른 순열 P이 적용된다.

$$Step = P \circ Step' \circ P^{-1}$$
$$Step' = wordperm' \circ Mix_j \circ MsgAdd'$$
$$Step' = \sigma' \circ Mix_j \circ MsgAdd'$$

[수식 2-6] 순열 P와 Step 함수

$$MsgAdd = MsgAdd'$$

[수식 2-7] 순열 P와 MsgAdd 함수

LSH를 수행하기 전에 [수식 2-4]에 따라 P를 적용한다. P가 적용되어 수정된 LSH를 수행한 후, 다시 P의 역순열인 P^{-1}를 연산하여 LSH 연산을 마무리한다. [수식 2-4]에 의해 [수식 2-5], [수식 2-6], [수식 2-7]과 같이 기

존 LSH 내부 함수들도 수정된다.

본 논문에서는 워드 단위 순열 기반 LSH 최적화 구현 연구에서 제안한 $LSH = P \circ LSH' \circ P^{-1}$ 기법을 활용한다.

제 3 장 제안 기법

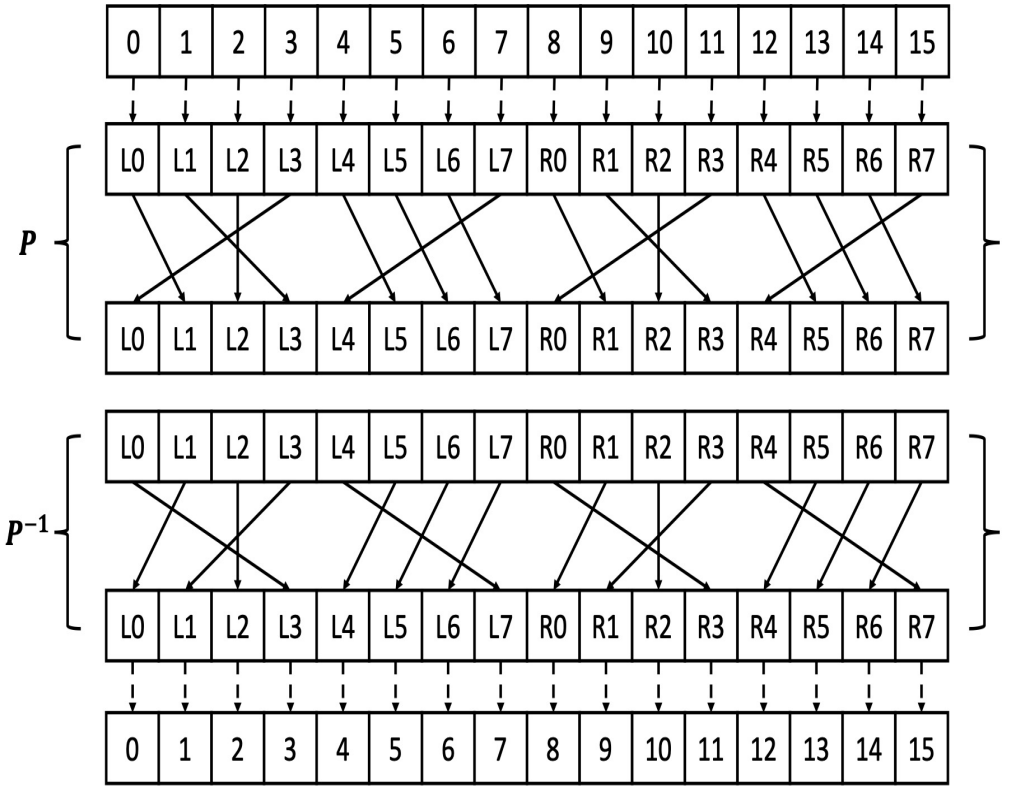
본 장에서는 64-bit ARMv8 상에서의 해시 함수 LSH 최적화 구현 기법에 대해 서술한다. 1절에서는 싱글 메시지에 대한 LSH 최적화 구현에 대해서 서술한다. 싱글 메시지에 대한 LSH 최적화 구현을 위해 사용되는 구현 기법은 더 적은 벡터 명령어를 사용하기 위해 수행되는 최적의 순열 P를 활용한 최적화와 ARX 연산 최적화 그리고 Wordperm 함수와 메시지 확장 함수에서 수행되는 순열 연산 최적화이다. 2절에서는 멀티 메시지에 대한 LSH-256 최적화 병렬 구현에 대해서 서술한다. 멀티 메시지에 LSH를 연산하기 위해서는 LSH를 수행하기 전에 레지스터 내부 정렬을 통해 서로 다른 두 개의 메시지를 하나의 레지스터에 위치하게 레지스터 내부 정렬을 수행하였다. 이를 통해, 기존 메시지 확장함수에서 수행하는 덧셈 연산은 기존 덧셈 연산 대비 절반으로 생략이 가능하였다.

제 1 절 싱글 메시지에 대한 LSH 최적화 구현

본 절에서는 제안하는 ARMv8 상에서 벡터 명령어를 효율적으로 사용하기 위해 LSH-256과 LSH-512에 적용될 최적의 순열 P에 대해 살펴본다.

1) LSH-256의 최적화를 위한 최적의 순열 P

[그림 3-1]은 ARMv8 상에서의 LSH-256을 효율적으로 구현하기 위해 적용할 최적의 순열 P를 나타낸 것이다. LSH-256이 수행되기 전에 [그림 3-1]의 P 순열을 적용하여 수정된 LSH-256에 대한 연산을 수행한다.



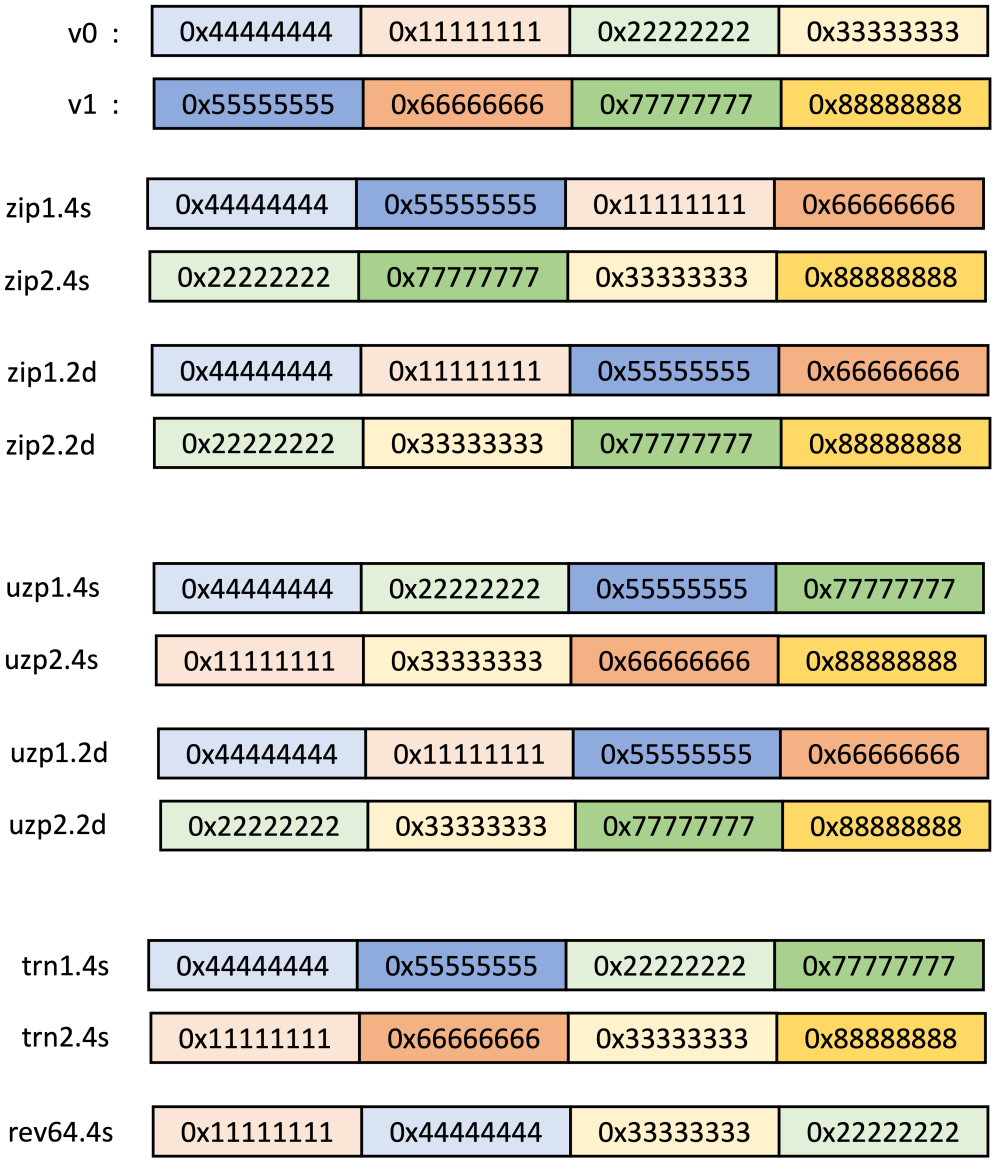
[그림 3-1] LSH-256의 순열 P와 P의 역순열 P^{-1}

[알고리즘 3-1]은 LSH-256을 위한 최적의 순열 P를 구현한 것이다. \s0 ~\s3은 입력 레지스터이고, v9, v10, v20, v21은 temp 레지스터로 사용된다. [그림 3-2]는 벡터 명령어인 ZIP1, ZIP2, UZP1, UZP2, TRN1, TRN2, REV 명령어를 수행하였을 때의 레지스터 내의 데이터 값에 대한 변화를 표현한 것이다.

Permutation P optimized for LSH-256; \s0, \s1, \s2, \s3 : input registers, v9, v10, v20, v21: temporary registers.	
Input : s0, s1, s2, s3	
Output : s0, s1, s2, s3	
1: uzp1.2d v20, \s0, \s2	8: rev64.4s v20, v20
2: uzp2.2d v21, \s0, \s2	9: trn1.4s v9, v20, v21
3: rev64.4s v21, v21	10: trn2.4s v10, v20, v21
4: zip1.4s \s0, v21, v20	11: rev64.4s v10, v10
5: zip2.4s \s2, v21, v20	12: zip1.2d \s1, v10, v9
6: uzp1.2d v20, \s1, \s3	13: zip2.2d \s3, v10, v9
7: uzp2.2d v21, \s1, \s3	

[알고리즘 3-1] LSH-256을 위한 최적의 순열 P

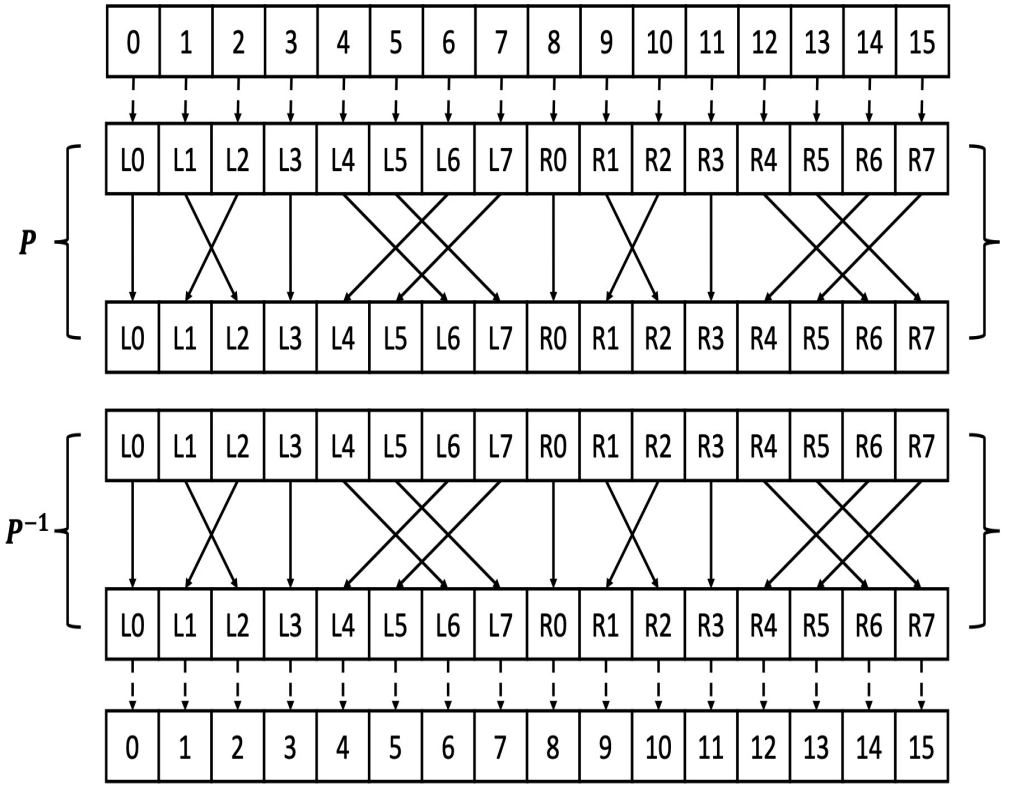
128-bit 벡터 레지스터 내부의 데이터를 각각 32-bit로 나눠서 보면 [그림 3-2]와 같이 표현이 가능하다. 이에 대해 각각 데이터 패킹을 4s, 2d로 설정하여 [그림 3-1]과 같은 배열 값을 갖도록 [알고리즘 3-1]과 같이 구현하여 최적의 순열 P를 적용하였다.



[그림 3-2] ZIP, UZP, TRN, REV 명령어 수행 결과

2) LSH-512의 최적화를 위한 최적의 순열 P

[그림 3-3]은 ARMv8 상에서의 LSH-512를 효율적으로 구현하기 위해 적용할 최적의 순열 P를 나타낸 것이다. LSH-512가 수행되기 전에 [그림 3-3]의 P 순열을 적용하여 수정된 LSH-512에 대한 연산을 수행한다.



[그림 3-3] LSH-512의 순열 P와 P의 역순열 P^{-1}

Permutation P optimized for LSH-512: \s0, \s1, \s2, \s3: input registers, v1, v2: temporary registers	
Input : s0, s1, s2, s3	
Output : s0, s1, s2, s3	
1: zip1.2d v2, \s0, \s1	5: mov.2d v2, \s2
2: zip2.2d v1, \s0, \s1	6: mov.2d \s2, \s3
3: mov.2d \s0, v2	7: mov.2d \s3, v2
4: mov.2d \s1, v1	

[알고리즘 3-2] LSH-512를 위한 최적의 순열 P

[알고리즘 3-2]는 LSH-512를 위한 최적의 순열 P를 구현한 것이다. \s0~\s3은 입력 레지스터이고, v1과 v2는 temp 레지스터로 사용된다. 워드 사이즈가 64-bit인 LSH-512의 구조에 따라 ZIP 명령어와 MOV 명령어를 활용하여 효율적으로 구현하였다.

3) ARX 연산 최적화

[알고리즘 2-1]은 압축 함수의 Mix 함수를 나타낸다. Mix 함수는 ARX 연산을 수행한다. 메시지 블록 M_j 의 값인 X, Y 워드에 대해 덧셈 연산을 수행하기 때문에 덧셈 연산을 위해 따로 P를 적용할 필요가 없다.

하지만, XOR 연산을 수행하기 위해서는 P를 적용해야 한다. [알고리즘 2-1]의 XOR 연산에서 P가 적용된 워드 값인 SC_j와 P가 적용되지 않은 j 번째 8-word 배열 Step 함수의 상수가 연산되는 것을 볼 수 있다. 이때,

SC_j는 테이블로 선언된 값이기 때문에 메모리에서 값을 로드하여 사용한다. 그렇기 때문에, SC_j를 포함한 연산을 수행하기 전에 SC_j에 P를 적용해야 한다. 하지만, SC_j을 포함하는 연산이 수행될 때 마다 반복적으로 P를 적용하는 것은 비효율적이다. 따라서, 제안 기법에서는 LSH 연산을 수행하기 전 SC_j가 저장되는 테이블에 P를 직접 적용하여 사용하였다. 이를 통해 SC_j가 메모리에서 로드될 때마다 P를 적용하기 위한 동작이 생략되었다.

Parameter	$\gamma[0]$	$\gamma[1]$	$\gamma[2]$	$\gamma[3]$	$\gamma[4]$	$\gamma[5]$	$\gamma[6]$	$\gamma[7]$
γ of LSH-256	0	8	16	24	24	16	8	0
γ' of LSH-256	24	0	16	8	0	24	16	8

[표 3-1] 순열 P가 적용된 LSH-256의 γ'

[알고리즘 2-1]의 왼쪽 로테이션 연산은 [표 2-2]를 따른다. 홀수 라운드와 짝수 라운드마다 비트 순환량 α_j, β_j는 각각 다른 값이 적용된다. 따라서, 제안 기법에서는 홀수 라운드와 짝수 라운드를 구별하여 Mix 함수를 구현하였다. 비트 순환량 γ 는 라운드와 관계없이 고정된 값이다. γ 상수는 블록마다 서로 다른 값을 갖는 특징을 갖고 있다. 따라서, LSH 암호화 연산 이전에 메시지 블록 M_j에는 P가 적용된 상태이다. 따라서, γ 상수에도 P가 적용되어야 한다. 하지만, P를 직접 적용해주는 것보다 각각의 블록에 따라 고정된 γ 상수가 반복적으로 사용되기 때문에 직접 γ 상수에 P를 적용한 γ'을 사용하였다. 제안 기법에서 사용된 LSH-256의 γ'와 LSH-512의 γ'는 각각 [표 3-1]과 [표 3-2]와 같다.

Parameter	$\gamma[0]$	$\gamma[1]$	$\gamma[2]$	$\gamma[3]$	$\gamma[4]$	$\gamma[5]$	$\gamma[6]$	$\gamma[7]$
γ of LSH-512	0	16	32	48	8	24	40	56
γ' of LSH-512	0	32	16	48	40	56	8	24

[표 3-2] 순열 P가 적용된 LSH-512의 γ'

[알고리즘 3-3]은 LSH-256의 γ'을 나타낸다. 벡터 명령어 중에서 로테이션 연산을 지원하는 명령어는 없다. 그러므로 로테이션 연산을 벡터 명령어를 활용하여 구현하기 위해서는 시프트 명령어를 사용하여 직접 구현이 가능하다. 따라서, [표 2-4]에 나열된 명령어 중 왼쪽 시프트 명령어인 SHL과 오른쪽 시프트 명령어인 SRI 명령어를 사용하여 로테이션 연산을 구현하였다. SHL 명령어는 벡터 레지스터의 왼쪽으로 비트를 이동시키고, SRI 명령어는 오른쪽으로 비트를 이동시킨 후, 레지스터의 빈 공간을 기존 레지스터의 값으로 채운다는 특징을 갖는다.

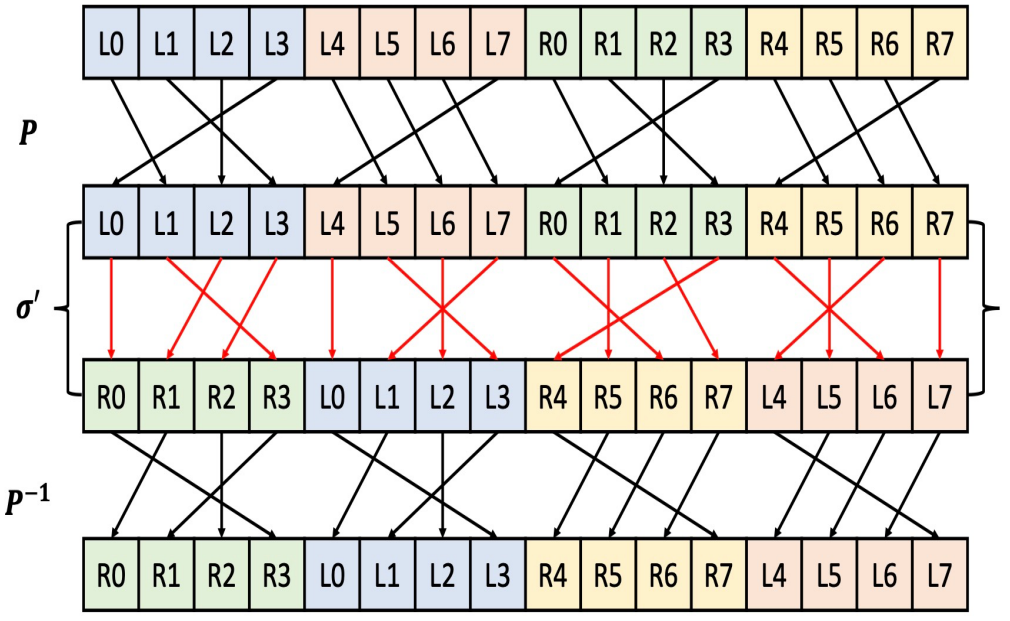
γ 상수가 0인 경우, 별도의 로테이션 연산을 수행할 필요가 없으므로 해당 부분의 로테이션 연산을 생략한다. 그리고 γ 상수가 16일 경우, REV32.8h 명령어를 사용하는 것이 효율적이다. REV32 명령어는 32비트 워드의 값을 반전시키는 명령어이다. 따라서, REV32.8h 명령어를 사용하여 32비트 워드의 상위 16비트와 하위 16비트를 반전하여 구현하였다. 그리고 이외의 다른 γ 상수는 시프트 명령어를 활용하여 구현하였다.

Optimized γ' for LSH-256; v7, v8: input registers, v20, v21, v26, v27: temporary registers	
Input : v7, v8	
Output : v7, v8	
1: mov v26.s[0], v7.s[3]	11: mov v7.s[3], v26.s[0]
2: mov v26.s[1], v8.s[3]	12: mov v8.s[3], v26.s[1]
3: mov v27.s[0], v7.s[0]	13: mov v7.s[0], v27.s[0]
4: mov v27.s[1], v7.s[1]	14: mov v8.s[1], v27.s[1]
5: shl.4s v20, v26, #8	15: mov v26.s[0], v7.s[2]
6: sri.4s v20, v26, #26	16: mov v26.s[1], v8.s[2]
7: mov.4s v26, v20	17: rev32 v26.8h, v26.8h
8: shl.4s v21, v27, #24	18: mov v7.s[2], v26.s[0]
9: sri.4s v21, v27, #8	19: mov v8.s[2], v26.s[1]
10: mov.4s v27, v21	

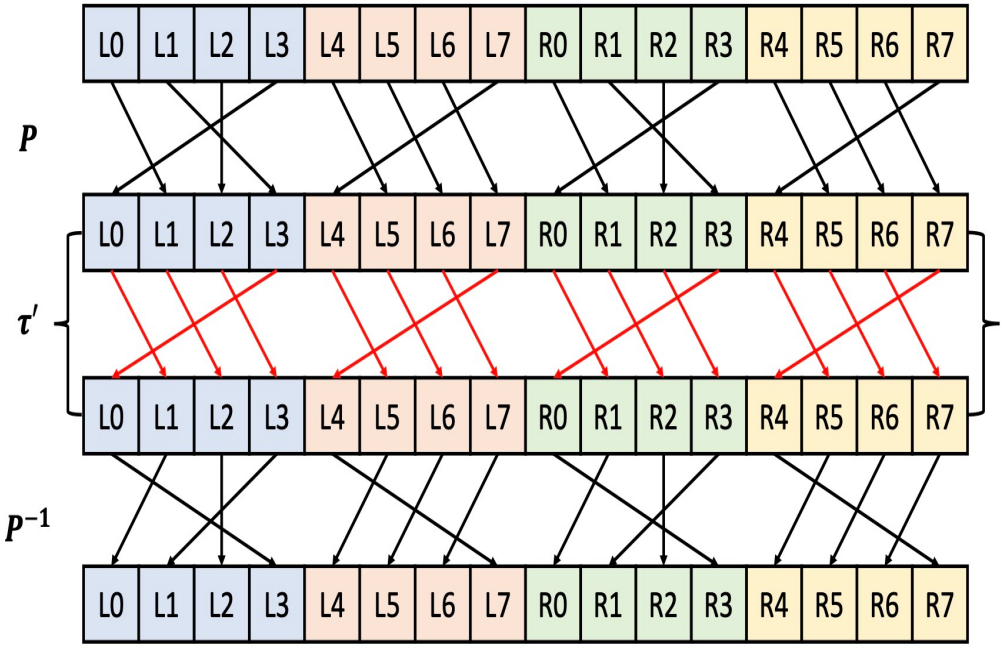
[알고리즘 3-3] LSH-256의 γ'

4) Wordperm 함수와 메시지 확장 함수의 순열 최적화

LSH-256의 경우, 기존 σ와 τ를 활용하여 연산을 수행하였을 때와 제안한 σ'와 τ'를 활용하여 연산을 수행하였을 때를 비교하면, 기존 대비 37번의 연산이 생략되었다. 이는 기존에 없던 P와 P^{-1}를 추가했음에도 불구하고, 수정된 σ'와 τ'에 의해 많은 연산이 감소하였음을 의미한다. [그림 3-4]는 LSH-256을 효율적으로 구현하기 위한 P와 P^{-1}가 적용되어 수정된 Wordperm 함수의 순열 σ'과 메시지 확장 함수의 순열 τ' 나타낸 것이다. [그림 2-4]와 [그림 2-3]과 비교하였을 때, 빨간색 선과 같이 수정된 것을 확인할 수 있다.



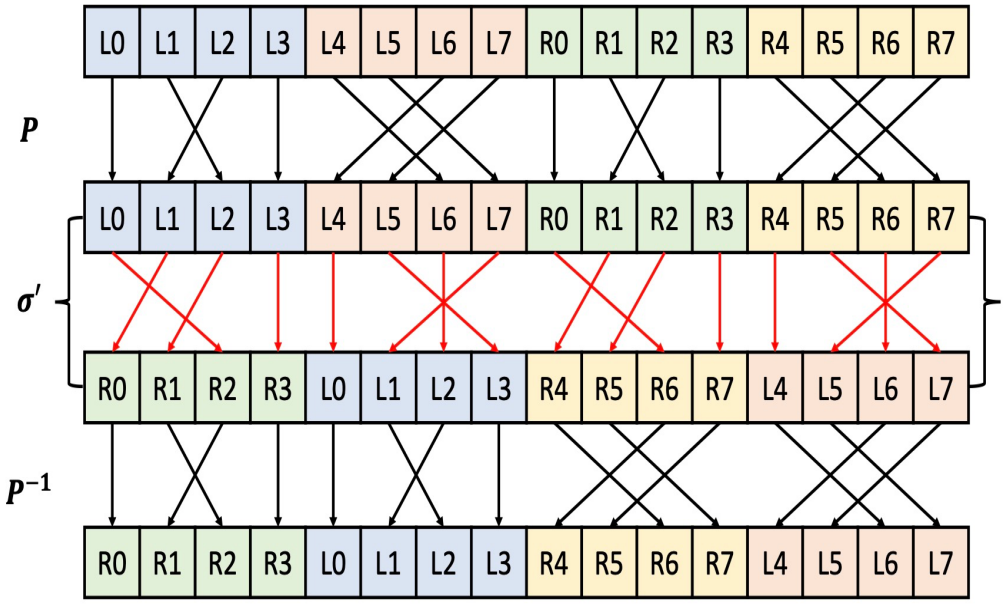
[그림 3-4] LSH-256의 σ'



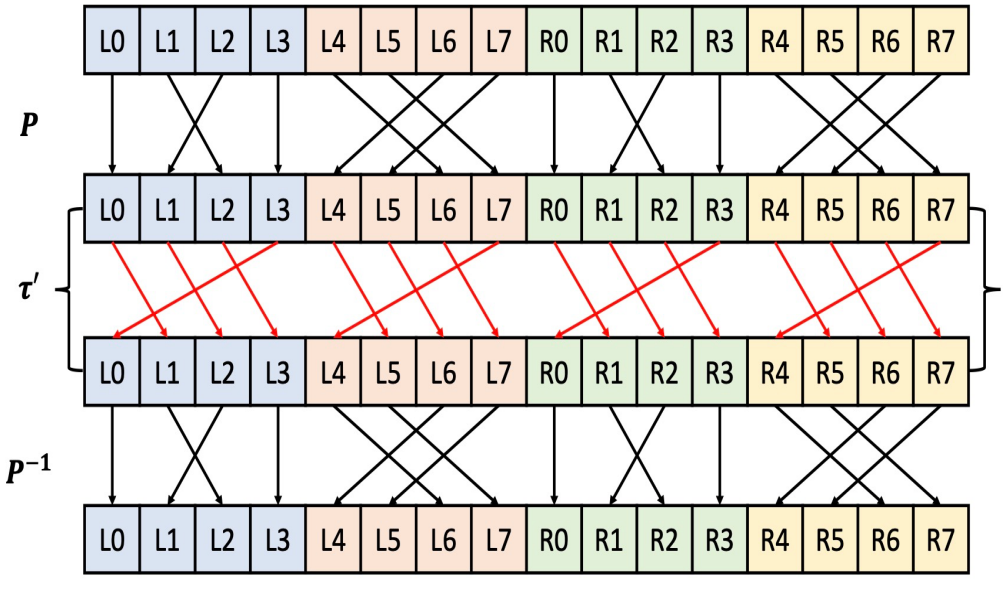
[그림 3-5] LSH-256의 τ'

[그림 3-6]과 [그림 3-7]은 LSH-512를 효율적으로 구현하기 위한 P와 P^{-1}가 적용되어 수정된 Wordperm 함수의 순열 σ'와 메시지 확장 함수의 순열 τ' 나타낸 것이다. [그림 2-4]와 [그림 2-3]과 비교하였을 때, 빨간색 선과 같이 수정된 것을 확인할 수 있다.

LSH-512의 경우, 기존 σ와 τ를 활용하여 연산을 수행하였을 때와 제안한 σ'와 τ'를 활용하여 연산을 수행하였을 때를 비교하면, 기존 대비 83번의 연산이 생략되었다. 이는 기존에 없던 P와 P^{-1}를 추가했음에도 불구하고, 수정된 σ'와 τ'에 의해 많은 연산이 감소하였음을 의미한다.



[그림 3-6] LSH-512의 σ'



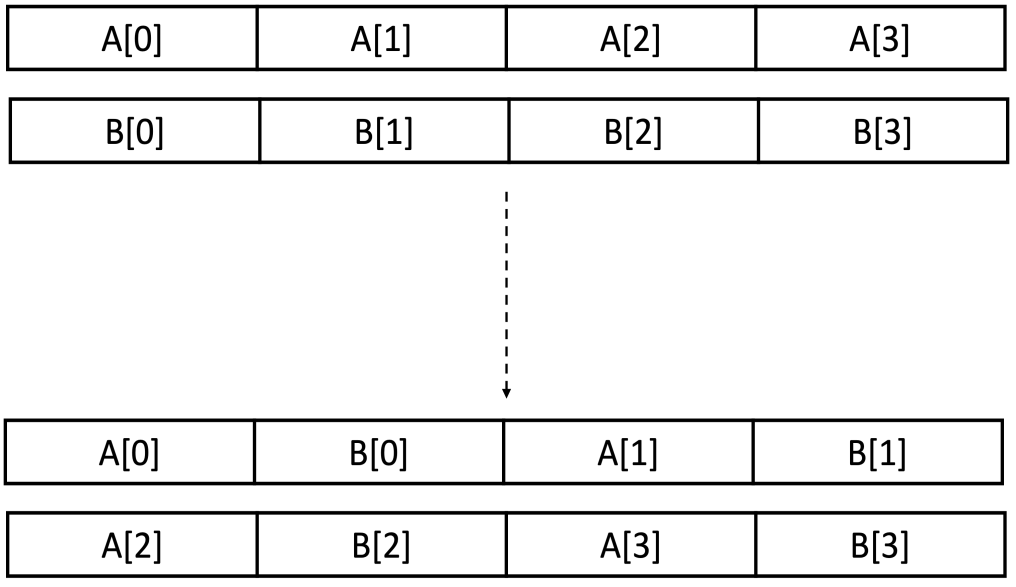
[그림 3-7] LSH-512의 τ'

제 2 절 멀티 메시지에 대한 LSH-256 병렬 구현

본 절에서는 ARMv8 상에서 벡터 레지스터를 효율적으로 사용하기 위해 멀티 메시지에 대한 LSH-256 병렬 구현에 대해서 서술한다.

1) 레지스터 내부 정렬

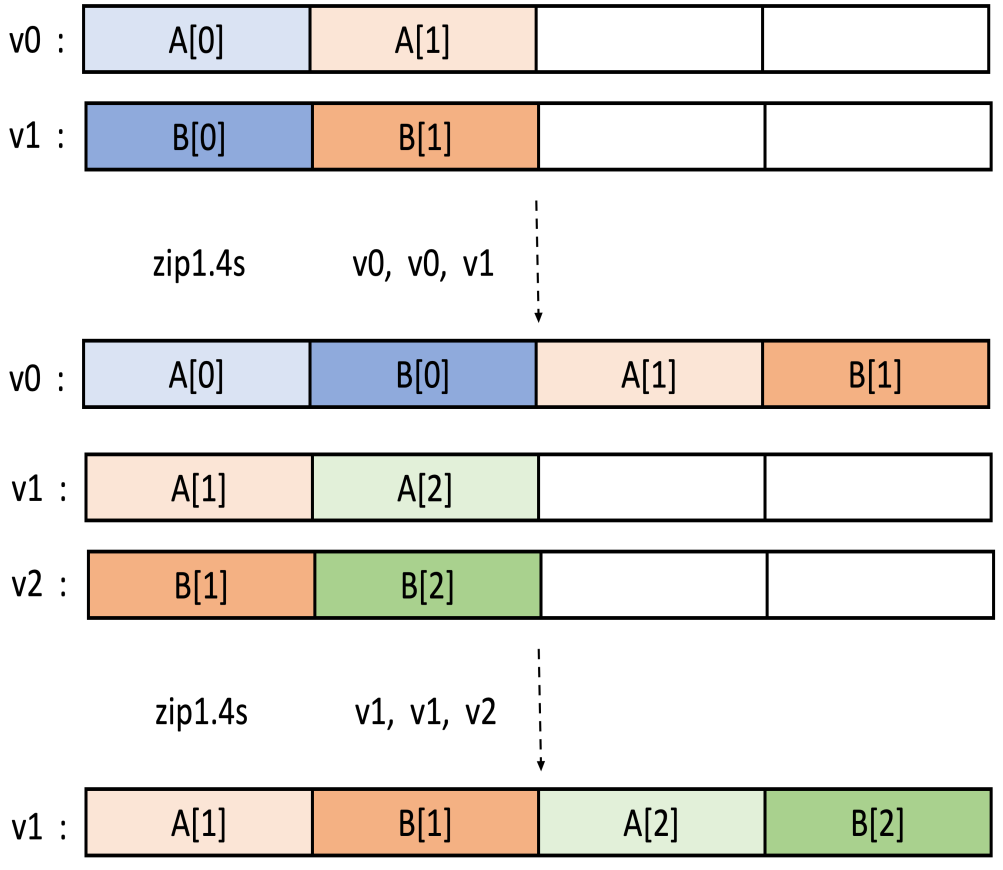
두 개의 서로 다른 레지스터에 각각 섞이지 않아야 하는 2개의 메시지가 로드 되어있다. 하지만, 병렬 연산을 하기 위해서는 하나의 레지스터에 동일한 연산이 수행되는 레지스터들이 위치해야한다. 이를 위해 LSH 연산을 수행하기 전에 레지스터 내부 정렬이 필요하다. 따라서, LSH-256 병렬 구현을 위해 동일한 연산이 수행되는 메시지 블록을 하나의 레지스터에 위치하게 레지스터 내부 정렬을 수행하였다.



[그림 3-8] 병렬 구현을 위한 레지스터 내부 정렬

[그림 3-8]은 서로 다른 메시지 값을 갖고 있는 두 개의 레지스터에 대해

값이 서로 섞이지 않고, 동일한 연산이 수행이 가능하게 레지스터를 내부 정렬을 나타낸 것이다. 레지스터 내부 정렬을 수행하기 전에는 2개의 레지스터(v0, v1)에 각각 다른 메시지가 로드 되어있다. v0 레지스터에는 메시지 A[0]~A[3]의 값이 차례대로 저장되어 있고 v1 레지스터에는 메시지 B[0]~B[3]의 값이 차례대로 저장되어 있다. 하지만, 병렬 구현하기 위해서는 동일한 연산이 수행되는 메시지 배열에 해당되는 값을 [그림 3-8]과 같이 하나의 레지스터에 위치 시켜야 한다. [그림 3-8]과 같이 레지스터 내부 정렬을 해주기 위해 [그림 3-9]와 같이 ZIP.4s 명령어를 활용하여 레지스터 내부 정렬을 수행하였다.



[그림 3-9] ZIP1 명령어를 활용한 레지스터 내부 정렬

2) 메시지 확장함수 최적화

Part of message extension function for LSH-256 multi-message; v16, v25, v26, v29, v30, v31:temporary registers.	
Input : v2, v3, v10, v11	
Output : v2, v3	
1: uzp1.2d v29, v2, v3	12: add.4s v29, v25, v30
2: uzp2.2d v30, v2, v3	
3: mov.2d v31, v29	13: mov v16.d[1], v30,d[0]
	14: mov v30.d[0], v30,d[1]
4: mov v16.d[1], v30,d[0]	15: mov v30.d[1], v16,d[1]
5: mov v30.d[0], v30,d[1]	
6: mov v30.d[1], v16,d[1]	16: mov v16.d[1], v26.d[0]
	17: mov v26,d[0], v26.d[1]
7: uzp1.2d v25, v10, v11	18: mov v26.d[1], v16.d[1]
8: uzp2.2d v26, v10, v11	
	19: add.4s v30, v26, v31
9: mov v16.d[1], v26,d[0]	
10: mov v26.d[0], v26,d[1]	20: uzp1.2d v2, v29, v30
11: mov v26.d[1], v16,d[1]	21: uzp2.2d v3, v29, v30

[알고리즘 3-4] LSH-256 멀티 메시지에 대한 메시지 확장함수의 일부

메시지 확장 함수 내부에는 τ 순열에 따라 값이 치환된다. 그리고 τ 순열에 의해 치환된 값과 치환되지 않은 값에 대해 16개의 배열 2개에 대한 덧셈 연산이 수행된다. 하지만, 제안하는 병렬 구현은 LSH 연산 전 레지스터 내부 정렬로 인해 하나의 레지스터에 서로 다른 2개의 메시지 값이 병렬로 저장되어 되어 있다. 이와 같은 특징으로 인해 16개의 배열 2개에 대한 하나의 ADD명령어만으로 서로 다른 배열에 대한 덧셈 연산이 2번 수행된다. 따라서, 기존 단일 평문 대비 덧셈 연산을 수행하는 과정이 절반으로 줄어든다.

[알고리즘 3-4]는 멀티 메시지에 대한 메시지 확장 함수의 일부를 나타낸 것이다. τ 순열에 따른 치환 연산과 덧셈 연산을 동시에 수행한다. 이 과정에서 τ 순열을 적용하면서 하나의 ADD 명령어를 활용하여 2번의 덧셈 연산을 하기 위해 ZIP, UZP, MOV 명령어를 사용하여 효율적으로 구현이 가능하였다.

제 4 장 성능 평가

제 1 절 실험 환경

본 논문에서는 ARMv8 아키텍처로 설계된 Apple M1 chip이 탑재된 Macbook Pro'13에서 성능 평가를 진행한다. 구현 및 성능 측정은 Xcode Framework에서 수행하였다. Apple M1 프로세서는 최신 ARM 프로세서 중 하나이다. M1은 Apple에서 개발한 프로세서로 Mac이나 iPad 등에서 사용하도록 설계되었다. M1 프로세서는 멀티 코어 CPU, GPU, DSP, 뉴럴엔진을 하나의 칩에 탑재한 일종의 시스템 온 칩(System on Chip, SoC)로, 5mm 공정으로 생산되며, 약 160개의 트랜지스터로 구성된다.

ARMv8 상에서의 LSH에 대한 선행 연구가 없기 때문에 KISA에서 제공하는 레퍼런스 코드와 구현 기법이 적용되지 않은 구현과 성능 평가를 진행한다. 성능 비교 단위로는 cpb(cycle per byte)를 사용한다.

제 2 절 성능 측정

LSH-256과 LSH-512에 대한 싱글 메시지에 대한 최적화 구현과 LSH-256에 대한 멀티 메시지에 대한 최적화 병렬 구현에 대한 성능 측정 결과는 [표 4-1]과 같다. Previous work는 KISA에서 제공하는 레퍼런스 코드를 의미하고, This work는 싱글 메시지에 대한 LSH 최적화 기법이 적용되지 않는 어셈블리 구현을 의미한다. 그리고 This work*는 싱글 메시지에 대한 LSH 최적화 기법이 적용된 어셈블리 구현을 의미하고, This work**는 멀티 메시지에 대한 LSH 최적화 기법이 적용된 어셈블리 구현을 의미한다.

Hash Function		size of message	
		1,024-bit	2,048-bit
LSH-256	Previous work	1,183.797	1,740.734
	This work	350.659	469.792
	This work*	**299.392**	**403.235**
	This work**	**450.485**	**620.934**
LSH-512	Previous work	1,261.794563	628.273531
	This work	586.121	383.570
	This work*	**538.325**	**353.072**

[표 4-1] ARMv8 상에서의 성능 측정 비교
(**P*를 적용한 싱글 메시지 최적화 기법 적용, **멀티 메시지 최적화 병렬 구현)

LSH-256 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스 코드보다 각각 3.94배, 4.32배 성능 향상을 확인하였다. LSH-512 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스 코드보다 각각 2.34배, 1.78배 성능 향상을 확인하였다. 멀티 메시지에 대한 LSH-256 병렬 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스 코드보다 각각 5.26배, 5.61배 성능 향상을 확인하였다 그리고 1,024-bit와 2,048-bit 메시지에 대해 LSH-256 최적화 구현 대비 각각 1.33배, 1.30배 성능 향상을 확인하였다.

제 5 장 결론 및 향후 연구 방안

본 논문에서는 64비트 ARMv8 프로세서에서 해시 함수 LSH 최적화 구현을 제안하였다. LSH-256과 LSH-512에 대한 싱글 메시지 최적화 구현과 LSH-256에 대한 멀티 메시지 최적화 병렬 구현을 제안하였다. 싱글 메시지 최적화 구현은 더 적은 벡터 명령어를 사용하기 위해 순열 P를 적용된 LSH를 수행하는 기법을 활용하여 ARMv8 상에서의 효율적인 순열 P를 찾아 LSH-256과 LSH-512에 대해 각각 다른 순열 P를 적용하였다. 이 과정에서 순열 P에 대해 효율적으로 벡터 명령어를 사용하여 구현하였다. 이로 인해, 메시지 확장 함수와 Wordperm 함수에서 수행되는 순열 연산도 기존보다 적은 명령어를 사용하여 구현하였다. 결과적으로, 순열 P와 역순열 P^{-1}에 대한 연산이 추가되었음에도 불구하고, 수정된 LSH-256은 37번 그리고 수정된 LSH-512는 83번의 연산 생략이 가능하였다. 이외에 LSH의 주요 연산인 ARX 연산에 대해서도 효율적으로 구현하였다. 특히, γ 로테이션 연산은 일부 연산에 대한 생략과 시프트 명령어를 활용하여 로테이션 연산을 구현하였고, 일부 γ값에 대해 REV 명령어를 활용하여 효율적으로 로테이션 구현이 가능하였다. 그리고 멀티 메시지 최적화 병렬 구현은 워드 사이즈가 32인 LSH-256의 특징을 활용하여 하나의 레지스터에 서로 다른 메시지가 위치하게 레지스터 내부 정렬을 수행하여 최적화 구현을 하였다. 이러한 레지스터 내부 정렬로 인해, 기존 메시지 확장함수에서 수행되던 덧셈 연산을 기존 대비 절반으로 생략하였다.

결과적으로, 본 논문에서 제안한 구현 기법을 적용한 성능은 다음과 같다. LSH-256 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스보다 각각 3.94배, 4.32배 성능 향상을 확인하였다. LSH-512 최적화 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스보다 각각 2.34배, 1.78배 성능 향상을 확인하였다. 멀티 메시지에 대한 LSH-256 병렬 구현은 1,024-bit와 2,048-bit 메시지에 대해 KISA에서 제공한 레퍼런스보다 각각 5.26배, 5.61배 성능 향상을 확인하였다 그리고

1,024-bit와 2,048-bit 메시지에 대해 LSH-256 최적화 구현 대비 각각 1.33배, 1.30배 성능 향상을 확인하였다.

본 논문에서는 다양한 임베디드 프로세서 상에서의 LSH 최적 구현 연구가 수행됨을 확인하였다. 하지만, 32-bit RISC-V 상에서 LSH 최적 구현 연구는 아직 진행되지 않았음을 확인하였다. 따라서, 향후 연구로 본 논문에서 제안한 기법을 적용한 RISC-V 상에서의 LSH 최적 구현 연구를 제안한다.

참 고 문 헌

1. 국내문헌

Song, H.; Lee, O. Optimized Implementing A new fast secure hash function LSH using SIMD supported by the Intel CPU. In Proceedings of the Korea Information Processing Society Conference Korea Information Processing Society, pp. 701-704, 2015.

Pack, C.; Kim, H.; Hong, D.; Seo, C. Parallel Implementation of LSH Using SSE and AVX. *Journal of the Korea Institute of Information Security & Cryptology*, *26*(1), 31-39, 2016.

2. 국외문헌

Wang, X.; Yao A.; Yao F. Cryptanalysis on SHA-1. CRYPTOGRAPHIC HASH WORKSHOP. 2005, Oct.

Wang, X.; Yin, Y. L.; Yu, H. Finding collisions in the full SHA-1. In Annual international cryptology conference, Springer, Berlin, Heidelberg, pp. 17-36, 2005, Aug.

Seo, H.; Liu, Z.; Choi, J.; Park, T.; Kim, H. Compact implementations of LEA block cipher for low-end microprocessors; in Revised Selected Papers of the 16th International Workshop on Information Security Applications; Volume 9503, pp. 28-40, 2015.

Seo, H.; Liu, Z.; Longa, P.; Hu, Z. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. IACR Transactions on Cryptographic Hardware and Embedded Systems, 1-20. 2018.

Song, J.; Seo, S. Secure and fast implementation of ARX-based block ciphers using ASIMD instructions in ARMv8 platforms. IEEE Access, 8, 193138-193153, 2020.

Seo, H.; Sanal, P.; Azarderakhsh, R. SIKE in 32-bit ARM processors based on redundant number system for NIST level-II. ACM Transactions on Embedded Computing Systems (TECS), 20(3), 1-23, 2021.

Chen, M; Chou, T; Krausz, M. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. Cryptology ePrint Archive, 2021.

Chen, M; Chou, T. Classic mceliece on the ARM cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems, 125-148, 2021.

Kim, Y; Song, J; Seo, S; Accelerating Falcon on ARMv8. IEEE

Access, 10, 44446-44460, 2022.

Kim, D; Hong, D; Lee, J; Kim, W; Kwon, D. LSH: A new fast secure hash function family. In International Conference on Information Security and Cryptology (pp. 286-313). Springer, Cham, December, 2014.

Park, T; Seo, H; Liu, Z' Choi, J' Kim, H. Compact implementations of LSH. In International Workshop on Information Security Applications (pp. 41-53). Springer, Cham, August, 2015.

Kim, D; Jung, Y; Ju, Y; Song, J. Fast implementation of LSH with SIMD. IEEE Access, 7, 107016-107024, 2019,

Arm® A64 Instruction Set Architecture: Armv8, for Armv8-A Architecture Profile. Available online: https://developer.arm.com/docs/ddi0596/c/simd-and-floating-point-instructions-alphabetic-or er (accessed on 29 July 2022).

Choi, H; Seo, S. Optimization of PBKDF2 using HMAC-SHA2 and HMAC-LSH families in CPU environment. IEEE Access, 9, 40165-40177, 2021.

ABSTRACT

Optimized Implementation of LSH Hash Function on ARMv8 Processors

Sim, Min-Joo

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

In this paper, we propose an LSH optimization implementation of hash function using vector registers of 64-bit ARMv8 processor. Hash function LSH Performs parallel operation efficiently by utilizing vector registers for the entire operation process. The proposed method proposes a single message optimization implementation for LSH-256 and LSH-512 and a multi-message optimization parallel implementation for LSH-256. Many vector instructions are needed to implement the permutations performed by LSH's Wordperm function and message expansion function. Therefore, in the single message optimization implementation, Wordperm and message extension functions are optimized. The proposed method performs permutation P before LSH operation. After P is applied and the modified LSH is performed, the operation is completed by performing P, which is the reverse permutation of P. In this process, P suitable for

LSH-256 and LSH-512 is found and implemented efficiently using vector instructions. As a result, the Wordperm function and the message extension function modified by applying P were implemented using fewer vector instructions than before applying P. The main operation of LSH is ARX (Addition, Rotation, eXclusive OR). Therefore, optimization of the ARX operation performed in the modified LSH is performed. Optimized parallel implementation for multi-message parallel implementation of LSH-256 for two different messages through register internal alignment. In this process, the addition operation performed in the message extension function is omitted to half of the previous one and optimized. As a result, the LSH-256 optimized implementation confirmed 3.94 times and 4.32 times performance improvement over the reference code provided by the Korea Internet & Security Agency (KISA) for 1,024-bit and 2,048-bit messages, respectively. The LSH-512 optimized implementation confirmed 2.34 times and 1.78 times performance improvement over the reference code provided by KISA for 1,024-bit and 2,048-bit messages, respectively. Parallel implementation of LSH-256 for multi-message confirmed performance improvement of 5.26 times and 5.61 times, respectively, compared to the reference code provided by KISA for 1,024-bit and 2,048-bit messages. Also, for 1,024-bit and 2,048-bit messages, performance improvements were confirmed by 1.33 times and 1.30 times, respectively, compared to LSH-256 optimized implementation.

【KEYWORD】LSH Hash Function, Software Implementation, ARMv8 Processors, Parallel Computation