Intel SIMD를 활용한 해시 함수 LSH 최적 구현 연구 동향

심민주*, 권혁동*, 김현준*, 서화정*†

*한성대학교 IT융합공학부(대학원생)

*†한성대학교 IT융합공학부 (교수)

Trend of Research on Optimal Implementation Hash Function LSH using Intel SIMD

Min-Joo Sim*, Hyeok-Dong Kwon*, Hyun-Jun Kim*, Hwa-Jeong Seo*†

*Dept. of IT Convergence Engineering, Hansung University
(Graduate student)

*†Dept. of IT Convergence Engineering, Hansung University
(Professor)

요 약

해시 함수는 데이터 무결성, 인증, 서명 등을 제공하기 위해 사용하는 함수로, 임의의 길이의 비트열을 입력으로 받아 고정된 길이의 비트열을 출력하는 함수이다. ICISC'14에서 새로운 해시함수인 LSH가 발표되었다. LSH 알고리즘에서 사용되는 주요 연산은 ARX(Addition, Rotation, eXclusive-OR) 연산이다. ARX 구조는 SIMD를 활용하기 좋은 구조이다. 이에 따라, LSH에 대한 최적 구현 연구는 다양한 플랫폼 상에서 활발히 진행되고 있다. 그 중 본 논문에서는 Intel SIMD를 활용한 LSH에 대한 최적 구현 연구 동향을 살펴본다.

I. 서론

해시 함수는 임의의 길이의 비트열을 입력으로 받아 고정된 길이의 비트열을 출력하는 함수이다. 해시 함수는 데이터 무결성, 인증, 서명 등을 제공하기 위해 주로 사용된다. 2005년 SHA-1과 MD5와 같은 구조에 대한 충돌 쌍 공격[1, 2]이 제기되어 더 이상 안전하지 않다고 알려졌다. 이에 따라, NIST에서는 SHA-3 공모전[3]을 시행하였다. 국내에서는 SHA-3보다 우수한 효율성을 갖는 LSH[4]를 개발하였다. LSH의 주요 연산은 ARX 연산으로 되어 있기 때문에 SIMD(Single Instruction Multiple Data)를 활용하기 좋은 구조이다. 이에 따라, 다양한 플랫폼 상에서의 LSH 최적 구현 연구가 활발히 진행되고 있다.

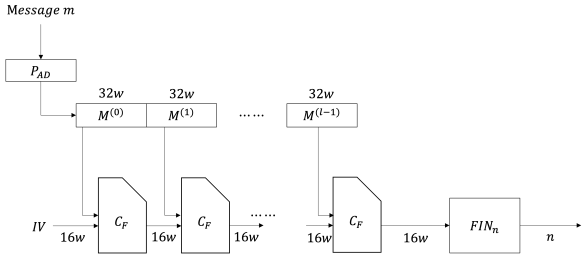
본 논문에서는 Intel SIMD를 활용한 LSH 최적 구현 연구 동향을 살펴본다. 2장에서는 해시 함수 LSH에 대해 알아본다. 3장에서는 Intel SIMD를 활용한 LSH 최적 구현 연구 동향에 대해서 살펴본다. 마지막으로 4장에서 본 논문의 결론을 내린다.

II. LSH

LSH(Lightweight Secure Hash)는 2014년 NSR(National Security Research)에서 개발한 해시함수로, ICISC'14에서 발표되었다[4]. LSH는 SHA-3보다 4배 빠르고, 다른 SHA-3 finalist에 진출한 암호보다 1.5~2.3배 빠르다는 특징을 갖고 있다.

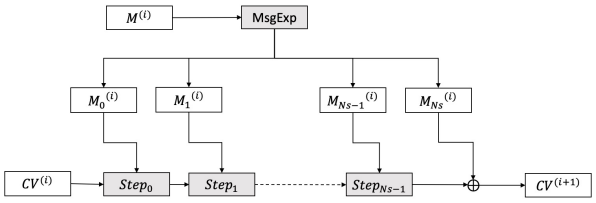
LSH의 전체 구조는 [그림 1]과 같다. LSH는 Message m과 초기화 벡터 IV를 사용한 초기화

단계, 압축함수 CF가 반복적으로 호출되는 압축 단계, 해시 값이 생성되는 종료 단계(FIN_n)로 구분할 수 있다.



[그림 1] Structure of hash function LSH.

압축 함수 CF은 [그림 2]와 같다. 압축 함수는 메시지 확장 함수인 MsgExp를 거친 후, 단계함수인 Step가 수행된다.



[그림 2] Structure of Compression Function.

Step 함수의 주요 연산은 32비트와 64비트에 대한 덧셈 연산, 256비트와 512비트에 대한 xor 연산과 로테이션 연산이다. 그 중 로테이션 연산은 [그림 3]과 같이 w 값에 따라 다르게 수행된다. Step 함수는 j 값이 짝수와 홀수에 따라 각각 다르게 로테이션 되는 비트 값(α_j, β_j)이 다르다.

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

[그림 3] The number of bits in rotation (α_j, β_j, γ_i).

III. Intel SIMD를 활용한 LSH 최적 구현 동향

본 장에서는 Intel SIMD를 활용한 LSH 최적 구현 동향에 대해서 살펴본다.

3.1 SSE2를 활용한 LSH 최적 구현

[5]는 Intel SIMD의 한 종류인 SSE2 (Streaming SIMD Extension version2)를 이용하여 128비트 단위 연산을 수행하도록 최적 구현하였다. SSE2는 128비트 레지스터를 가지고 있으며, 대략 70가지의 명령어를 지원한다.

[5]는 LSH의 초기화 함수와 압축 함수에 대해 최적 구현을 하였다. 초기화 함수와 압축 함수에는 ARX 연산이 진행되는데 이를 SSE2를 이용하여 32비트 단위로 연산이 구성된 LSH를 128비트로 처리할 수 있는 연산을 적용하였다. 압축 함수 내부에 존재하는 치환함수는 인접한 4개의 w를 대상으로 연산이 수행되는데 이를 하나의 레지스터에 위치시켜 shuffle 함수를 사용하여 치환 함수를 구현하였다. 그리고 덧셈 연산과 xor 연산은 4개의 w를 하나의 레지스터에서 연산할 수 있도록 구현하였다.

w가 32비트일 경우에 대해 위와 같은 기법을 적용한 성능 측정 결과는 다음과 같다. 구현 기법이 적용되기 전보다 연산의 수를 줄었고, 기존 대비 최대 2.79배 성능 향상을 보였다. 그리고 Openssl SHA-2와 비교하였을 때, 최대 8.99배 성능 향상을 보였다.

3.2 SSE와 AVX를 활용한 LSH 최적 구현

[6]은 SIMD의 SSE와 AVX(Advanced Vector Extension)를 활용하여 LSH 병렬 최적 구현을 하였다. AVX는 최대 256비트 레지스터를 활용할 수 있다.

SSE를 활용한 구현은 모든 연산에 대해 w가 32비트인 경우, 4개의 워드에 대해 동시 처리가 가능하다. w가 64비트인 경우, 2개의 워드에 대해 동시 처리가 가능하다. 덧셈 연산은 paddd 명령어를 활용하여 구현하였고, xor 연산은 pxor 명령어를 통해 병렬 구현하였다. 로테이션 연산의 경우, 한 라운드 내에서 동일한 비트 (α_j, β_j)에 대해 로테이션 연산이 수행된다. 따라서, 32비트 왼쪽 쉬프트를 수행하는 pslld 명령어를 통해 연산된 값과 w에서 동일한 비트 (α_j, β_j)만큼 뺀 만큼 32비트 오른쪽 쉬프트를

수행하는 psrld 명령어를 통해 연산된 값을 서로 xor 연산하여 로테이션 연산을 병렬 구현하였다. γ 로테이션의 경우, 하나의 라운드마다 동일한 값이 아니기 때문에 [그림 3]에서 확인할 수 있듯이 γ가 8의 배수의 값을 갖는 것을 활용하여 8비트 shuffle 명령어인 pshufb 명령어를 활용하여 구현하였다. 워드 단위 치환 함수도 32비트 shuffle 명령어인 pshufd 명령어를 사용하여 레지스터 내에서 치환 후, 레지스터를 서로 교환하는 방식으로 구현하였다.

AVX를 활용한 병렬 구현은 모든 연산에 대해 w가 32비트인 경우, 8개의 워드에 대해 동시 처리가 가능하고, w가 64비트인 경우, 4개의 워드에 대해 동시 처리가 가능하다. SSE를 활용한 구현과 유사하다. 하지만, 압축 함수의 워드 단위 로테이션을 구현하기 위해 레지스터 내부 구조를 변경을 하기 위해 128비트 permutation 명령어인 vperm2i128를 추가적으로 사용되었다.

압축 함수의 메시지 확장 함수는 두 개의 16w 메시지 배열로부터 연속적으로 새로운 16w 배열을 생성한다. 이 과정에서 메모리 할당과 호출 과정이 수행되어 레지스터와 메모리 사이에 메시지 블록 이동 과정으로 인해 레지스터 활용이 비효율적일 수 있다. 따라서, [6]은 메시지 배열에 대한 호출과 저장 과정을 생략하였다. 생략 과정은 다음과 같다. 첫 번째 메시지 블록과 두 번째 메시지 블록을 레지스터에 저장한다. 그리고 첫 번째와 두 번째 step 함수 연산을 진행한다. 이후, 반복 loop를 수행할 때 메시지 블록을 담은 고정된 레지스터를 갱신하는 기법을 사용하였다. 이때, 레지스터 변수의 자율성을 위해 Intel의 Intrinsic 함수를 사용하여 하나의 레지스터에 4개의 워드를 동시에 처리할 수 있는 w가 32비트 SSE와 w가 64인 AVX에 대해 메시지 확장과 step 함수를 구현하였다.

long message를 기준으로, 위와 같은 기법을 적용한 성능 측정 결과는 다음과 같다. w가 32비트인 경우, AVX-assembly 구현이 가장 빨랐고, 레퍼런스 코드 대비 약 3.62배 성능 향상을 보였다. 그리고 w가 64비트인 경우, AVX-Intrinsic 구현이 가장 빨랐고, 레퍼런스 코드 대비 약 8.65배 성능 향상을 보였다.

3.3 순열 P를 활용한 LSH 최적 구현

2019년 SIMD를 사용하여 LSH를 효율적으로 구현하기 위해 wordwise 순열 기반 구현 기법이 발표되었다[7]. 해당 기법은 순열 P와 P의 역순열인 P^{-1}를 통해 $LSH' = P \circ LSH \circ P^{-1}$를 증명하였다. 이를 통해, SIMD의 SSE2, SSSE3, XOP를 사용하여 효율적으로 더 적은 SIMD 명령어를 사용하였다. LSH-256과 LSH-512에 대해 각각 명령어를 적게 사용할 수 있는 최적의 P를 구하여 찾은 P에 대해 최적 구현을 진행하였다. 이때, LSH-256과 LSH-512는 서로 다른 순열이 적용된다.

l	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma(l)$	6	4	5	7	12	15	14	13	2	0	1	3	8	11	10	9
l	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\tau(l)$	3	2	0	1	7	4	5	6	11	10	8	9	15	12	13	14

[그림 4] Permutation of σ, τ.

이 기법을 사용하기 위해 LSH가 수행되기 전, 순열 P를 적용하였다. γ 로테이션의 경우에도 순열 P에 영향을 받기 때문에 순열 P를 적용한 γ'를 사용하여 로테이션을 수행한다. wordwise 순열 연산을 동시에 수행하기 위해 각각의 순열을 적절하게 변경하였다. 따라서, LSH의 내부 순열인 Wordperm 함수의 σ 순열과 MspExp 함수의 τ 순열에 대해 순열 P를 적용하여 σ'과 τ' 순열로 변경하였다. 기존 σ 순열과 τ순열은 [그림 4] 과 같다.

이를 통해, Wordperm‘ 함수와 MsgExp’ 함수가 SIMD를 활용하여 효율적으로 구현되었다. 그리고 이 과정에서 순열에 대한 TYPE을 5가지로 분류하였다. 그리고 이렇게 5종류로 분류된 순열은 다른 SIMD에서 활용하기 편리하게 사용이 가능하도록 이를 부록으로 제공하였다.

위와 같은 기법을 적용한 성능 측정 결과는 순열 P를 적용하였을 때, 평균 5% 성능 향상을 보였다. 그리고 SIMD 명령어를 사용한 최적화까지 적용하였을 때, 추가적으로 5% 성능 향상을 보였다.

IV. 결론

본 논문에서는 Intel CPU에서 지원하는 SIMD를 활용한 LSH 최적 구현 동향을 살펴보았다. SIMD의 한 종류인 SSE, AVX 등을 각각에 대해 지원하는 명령어를 효율적으로 사용하는 최적화 연구가 활발히 진행되었다. 그리고 기존 연구보다 적은 명령어를 사용하기 위해 순열 P를 사용하여 LSH를 효율적으로 구현한 최적 기법도 확인하였다. 향후 연구로, 다른 여러 암호에 대해 SIMD를 활용한 병렬 최적 구현 연구를 제안한다.

V. Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services, 100%).

[참고문헌]

[1] Wang, A. C. Yao and F. Yao, "Cryptanalysis on SHA-1," CRYPTOGRAPHIC HASH WORKSHOP, October 2005.

[2] Wang, Y. L. Yin and H. Yu, "Finding Collisions in the Full SHA-1," In Advances in Cryptology-CRYPTO 2005, pp. 17-36, August. 2005.

[3] CRYPTOGRAPHIC HASH AND SHA-3 STANDARD DEVELOPMENT, http://csrc.nist.gov/groups/ST/hash/index.html

[4] D.-C. Kim, D. Hong, J.-K. Lee, W.-H. Kim, and D. Kwon, “LSH: A new fastsecure hash function family,” in International Conference on Information Securityand Cryptology, pp. 286-313, Springer, 2014.

[5] Song, Haeng-Gwon, and Ok-yeon Lee. "Optimized Implementing A new fast secure hash function LSH using SIMD supported by the Intel CPU." Proceedings of the Korea Information Processing Society Conference. Korea Information Processing Society, 2015

[6] C. Park, et al. "Parallel Implementation of LSH Using SSE and AVX ." Journal of The Korea Institute of Information Security & Cryptology 26.1 (2016).

[7] D. Kim, Y. Jung, Y. Ju, and J. Song, “Fast implementation of LSH with SIMD,”IEEE Access, vol. 7, pp. 107016-107024, 2019.