Intel CPU 에서 지원하는 SIMD 를 이용한 고속해시함수 LSH 최적화 구현

송행권*, 이옥연*
*국민대학교 금융정보보안학과
e-mail : haeng9769@kookmin.ac.kr

Optimized Implementing A new fast secure hash function LSH using SIMD supported by the Intel CPU

Haeng-Gwon Song*, Ok-yeon Lee**
*Dept. of Financial Information Security, Kookmin University

요 약

해시함수는 사회 전반에 걸쳐 무결성 및 인증을 제공하기 위하여서 사용하는 함수로써 암호학적으로 중요한 함수이다. 본 논문에서는 2014 년 국가보안기술 연구소에서 개발한 해시함수 LSH 를 하드웨어적인 구현이 아닌 소프트웨어적인 구현을 수행하였고 또한 Intel CPU 상에서 동작하는 SIMD 기법인 SSE 를 이용하여 LSH 알고리즘의 최적화 구현을 수행한다. 고속해시함수 LSH 알고리즘에서 사용하는 주 연산은 ARX(Addition Rotation, Xor)연산으로 SIMD 를 적용하기에 용이한 구조로 되어 있다. 본 논문에서는 기존 32 비트 단위의 연산을 수행하는 LSH 알고리즘을 SIMD 를 이용하여 128 비트 단위의 연산을 수행 하도록 개발하였다. 그 결과 Intel Xeon CPU 에서 SIMD 를 적용한 결과 적용하지 않은 LSH 알고리즘보다 최대 2.79 배의 성능의 향상을 확인 할 수 있다.

1. 서론

정보화 사회에 접어들면서 무결성 및 인증 등의 암호학적인 기능을 제공하기 위하여 사회 각처에서 해시함수의 사용이 빈번하게 증가하였다. 해시함수(hash function)는 임의의 길이의 데이터를 입력으로 받아 고정된 길이의 데이터를 출력하는 알고리즘이다. 현재까지 사용되고 있던 해시함수로는 MD5 나 SHA1 과 같은 국제적으로 표준으로 채택된 알고리즘들을 사용하였다. 하지만 최근 MD5 나 SHA-1 알고리즘의 경우 충돌 쌍이 발견되어 안전성 측면에서 안전하지 않은 것으로 알려져 NIST(National Institute of Standards and Technology)에서 2008 년 SHA-1 알고리즘 사용 중지를 선언하였다. NIST 는 SHA-1 을 보강하기 위하여 현재 SHA-2 해시함수를 사용하고 있으나 SHA-2 또한 SHA-1 과 유사한 공격에 대한 취약성이 알려졌다. 이러한 안전성의 문제를 보안하기 위하여 전 세계적으로 SHA-3 알고리즘 공모전을 시행하여 2012 년 SHA-3 알고리즘으로 Kecchack 알고리즘을 선정하였다. Keccack 알고리즘은 기존의 해시 알고리즘과는 다른 구조를 가지고 있으므로 기존 해시함수에 존재하던 취약점들을 보안할 수 있었다. 새로 채택된 SHA-3 알고리즘은 공모전에 제시되었던 다른 알고리즘들에 비하여 하드웨어적으로 구현하였을 경우 효율성이 좋은 것으로 알려져 있으나 소프트웨어적인 구현 시에 효율성이 낮은 것으로 알려져 있다. 하드웨어적인 알고리즘 구현은 한번 설계 후 구현이 완료되면 새로운 기법을 적용하기 어려운 단점을 가지고 있으나 소프트웨어적인 알고리즘 구현은 구현이 완료 된 후에도 새로운 기법의 적용이나 알고리즘의 수정의 용이성이 높다는 장점을 가지고 있다.

본 논문에서는 소프트웨어적으로 구현하기 용이하며 현재 알려진 해시함수 공격에도 안전하게 설계된 고속해시함수 LSH(Lightweight Secure Hash) 알고리즘을 소개하고, 또한 Intel CPU 에서 지원하는 SIMD(Single Instruction Multiple Data)를 이용하여 LSH 알고리즘에 최적화 구현 결과를 제시한다.

본 논문의 구성은 2 장에서는 LSH 알고리즘을 설명하고 3 장에서는 SIMD 의 개념 및 Intel CPU 에서 지원하는 SIMD 인 SSE 에 대해 설명하고 4 장에서는 실험 환경 명세 및 LSH 알고리즘의 병렬화 적용 방법에 대해 설명하고 5 장에서는 SIMD 를 적용한 결과 및 향후 연구 방향을 제시한다.

2. LSH 알고리즘

본 논문에서는 해시함수 LSH 에 대해서 간략히 소개한다. 해시함수는 암호학적으로 암호학적으로 사

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (NO. A2015-0067, 다양한 IoT 서비스 개발을 위한 경량 암호/인증 보안 라이브러리 개발

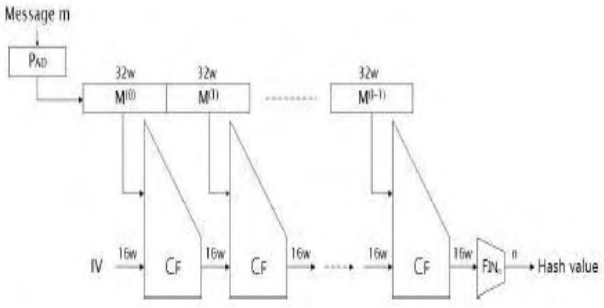
용되기 위해서 요구되는 세가지 성질로는 역상 저항성, 제 2 역상 저항성 그리고 충돌 저항성이 있다. 제 2 역상 저항성과 충돌 저항성은 충돌 쌍을 찾는 문제의 어려움에 기반한다. 이러한 성질을 만족하는 해시함수는 사용자 및 메시지 정보 변조 방지를 위한 무결성 검증을 위하여 사용된다. 또한 해시함수는 전자서명, 메시지 인증코드, 암호키 유도, 의사난수 생성 등에 있어서도 핵심기능을 담당하고 있다.

LSH(Lightweight Secure Hash)는 w 비트 워드 단위로 동작하며 n 비트 출력 값을 가지는 해시함수 LSH-8w-n 으로 구성된 해시함수 군(Hash Function Family)이다. 여기서 w 는 32, 64 비트이고, n 은 1 에서 8w 사이의 정수이다. 해시함수 LSH-8w-n 은 상기 암호학적 해시함수의 성질을 모두 제공 할 수 있도록 설계되었다.

2.1 LSH 전체구조

LSH 를 구성하는 해시함수는 (그림 1)과 같은 전체 구조를 가지며, 입력 메시지에 대해 다음의 세가지 단계를 거쳐 해시 값을 출력한다.

- 초기화 (Initialization) : 입력 메시지를 메시지 블록 비트 길이의 배수가 되도록 패딩을 한 후, 이를 메시지 블록 단위로 분할한다.
- 압축(Compression) : 32 워드 배열 메시지 블록을 압축 함수의 입력으로 하여 얻은 출력 값으로 연결 변수를 갱신하여, 이를 마지막 메시지 블록을 처리 할 때까지 반복하여 메시지를 압축한다.
- 완료(Finalization) : 압축 과정을 통해 연결 변수에 최종 저장된 값으로부터 n 비트 길이의 출력 값을 생성한다.



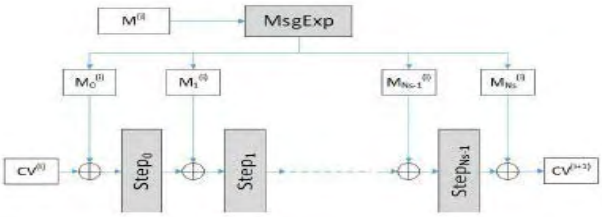
(그림 1) LSH 전체구조

2.1.1 초기화 함수

초기화 함수는 입력 메시지를 m 이라고 하면, 메시지의 길이가 32wt 비트가 될 때까지 패딩을 수행하는 함수이다. 패딩은 m 의 끝에 비트 '1'을 덧 붙인 후, 길이가 32wt 비트가 될 때까지 비트 '0'을 덧붙인다.

2.1.2 압축 함수

압축 함수는 다시 4 가지의 함수로 나뉜다. 메시지 확장 함수, 메시지 덧셈 함수, 섞음 함수, 워드 단위 순환 함수로 구분된다. 압축함수는 (그림 2)와 같은 구조를 가진다.



(그림 2) 압축함수 구조

2.1.2.1 메시지 확장함수

메시지 확장함수는 32 워드 입력을 통하여 라운드 수+1 만큼의 16 워드 배열로 확장하는 함수이다. 생성 방법은 다음 식과 같다.

$$M_0^{(i)} \leftarrow (M^{(i)}[0], M^{(i)}[1], \cdots, M^{(i)}[15]),$$

$$M_1^{(i)} \leftarrow (M^{(i)}[16], M^{(i)}[17], \cdots, M^{(i)}[31]),$$

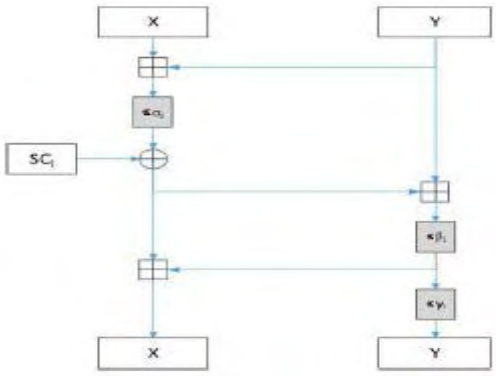
$$M_j^{(i)}[l] \leftarrow M_{j-1}^{(i)}[l] \boxplus M_{j-2}^{(i)}[\tau(l)] \qquad (0 \le l \le 15,\ 2 \le j \le N_s).$$

2.1.2.2 메시지 덧셈 함수

메시지 덧셈함수는 두 개의 16 워드 배열의 덧셈 함수를 의미한다.

2.1.2.3 섞음함수

메시지 섞음함수는 16 워드 배열을 두개의 워드 T[i], T[i+8] (0 ≤ i ≤ 7)을 쌍으로 구성한 후 (그림 3)과 같이 섞어주는 방식으로 T 를 갱신한다.



(그림 3) LSH 섞음함수 구성도

2.1.2.3 워드단위 순환 함수

워드 단위 순환 함수는 16 워드 입력을 Z_{16} 상의 정

수환 상에서의 치환 과정을 통하여 16 워드 배열을 출력한다. 다음 그림은 치환 과정 중에 사용되는 함수이다.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ(i)	6	4	5	7	12	15	14	13	2	0	1	3	8	11	10	9

(그림 4) Z_{16} 정수 상 상에서의 치환

2.1.3 완료 함수

완료 함수는 마지막 압축함수의 출력 16 를 출력하고자 하는 n 비트의 값으로 출력하는 함수이다.

3. SIMD(Single Instruction Multiple Data)

SIMD 는 병렬 프로세서의 한 종류로, 하나의 명령어로 여러 개의 값을 병렬로 계산하는 방식이다. 비디오 게임 콘솔이나 그래픽 카드와 같은 멀티미디어 분야에서 주로 사용되는 기법이다. SIMD 는 사용되는 CPU 에 따라서 다양한 종류를 지원하고 있다. Intel CPU 에서 지원하는 SIMD 는 MMX, SSE, AVX 를 지원하며 Arm CPU 에서는 NEON, Thumb 등을 지원하고 Power CPU 에서는 VMX, Altivec 과 같은 SIMD 를 지원한다. SIMD 는 각 CPU 에서 지원하는 레지스터 크기에 따라서 한 번에 처리할 수 있는 데이터의 양을 결정할 수 있다. SIMD 는 보통 64, 128, 256, 512bit 단위의 레지스터를 이용하여 각각의 데이터를 처리한다.
본 논문에서는 Intel CPU 에서 지원하는 SSE2(Streaming SIMD Extension version2)을 이용하였다. SSE2 는 128 비트 레지스터 (XMM0 ~ XMM7)을 이용하여 데이터를 한 번에 처리 할 수 있다. SSE2 명령어로는 산술, 비교, 데이터 셔플 등의 명령어 등을 가지고 있으며 70 여가지의 instruction set 을 가지고 있다.

4. 실험 환경 및 실험 방법

본 실험에서는 Intel CPU 두 종류에서 실험을 진행하였다. 첫 번째 실험환경은 데스크탑 환경에서 주로 사용되는 Intel i7 CPU 와 서버 급 환경에서 주로 사용되는 Intel Xeon CPU 에서 실험을 진행하였다.

<표 1> 실험환경 명세

	운영체제	CPU
1	Windows 7	Intel®Core i7-4790@3.60GHz
2	Windows Server 2008	Intel®Xeon®CPU E3-1220 V2 @3.10GHz

SIMD 는 어셈블리 언어 에서 적용 할 수 있는 함수들과 C 언어에 적용할 수 있는 Intrinsic SIMD 가 존재한다. 본 논문에서는 Intel CPU 에서 지원하는 Intrinsic SIMD 인 SSE2 를 이용하여 고속 해시 함수의 초기화 함수와 압축함수에 최적화 구현을 시도하였다. 초기화 함수와 압축함수에서 사용되는 연산은 ARX(Addition, Rotation, Xor)연산으로 SIMD 를 이용하여 하나의 명령어로 다중 블록을 처리하기에 용이한 구조이다. SSE2 를 이용하여 기존에 32 비트 단위로 연산이 구성 된 LSH 에 128 비트 단위로 처리할 수 있는 연산을 적용하였다.

메시지 확장함수와 워드 단위의 순환함수에서는 Z_{16} 정수 환 상에서의 치환 한 값을 이용하여 메시지의 확장 및 치환 과정을 이루고 있다. 두 치환함수는 각각 다른 함수로 구성되어 있지만 두 함수모두 치환 후의 결과 값인 인접한 4 개의 w 에서 치환이 수행 된 것을 확인 할 수 있다. 즉 128 비트 레지스터 안에서 셔플을 통하여 값을 변경할 수 있음을 의미한다. SSE2 에서 지원하는 shuffle 함수를 이용하여 Z_{16}정수 환상에서의 치환 함수를 수행 할 수 있다. 또한 각각의 함수들에서 수행 되는 덧셈과 Xor 연산 또한 인접한 4 개의 w 를 128 비트 하나의 레지스터에서 처리할 수 있도록 변경하였다.

다음 표들은 압축 함수 내의 각 함수에서 SIMD 적용과 미적용 시에 수행되는 연산의 수를 측정한 비교 결과를 제시한다.

<표 2>메시지 확장함수의 연산 수

W=32 기준 (처리 단위 /times)	Load	Add	Shuffle
미적용 LSH	32bit/432	32bit/400	32bit/0
SIMD LSH	128bit/208	128bit/100	128bit/100

<표 3> 메시지 덧셈함수의 연산 수

W=32 기준 (처리 단위 /times)	Load	Add
미적용 LSH	32bit/16	32bit/16
SIMD LSH	128bit/4	128bit/4

<표 4> 섞음 함수의 연산 수

W=32 기준 (처리 단위 /times)	Load	Xor	Add	Rotate
미적용 LSH	32bit/32	32bit/8	32bit/24	32bit/24
SIMD LSH	128bit/12	128bit/2	128bit/6	128bit/10

<표 5> 순환 함수의 연산 수

W=32 기준 (처리 단위 /times)	Load	Shuffle
미적용 LSH	32bit/16	32bit/0
SIMD LSH	128bit/4	128bit/4

고속해시함수 LSH 알고리즘에 SIMD 를 적용하여 위의 표에서 제시한 바와 같이 연산의 수를 줄일 수 있

었다. SIMD 를 적용한 LSH 알고리즘과 기존의 LSH 알고리즘을 비교하기 위하여 CPB(Cycle Per Byte)를 이용하여 연산의 성능을 측정하였다. 또한 LSH 알고리즘의 우수성을 증명하기 위하여 다양한 환경에서 사용되고 있는 Openssl SHA2 알고리즘과의 비교 결과를 제시한다.

<표 6> Intel i7 CPU 에서의 최적화(단위 : CPB)

입력/출력	SHA2	Standard LSH	SIMD LSH
64/256	168.70	28.97	18.75
4096/256	147.49	12.59	8.07
Long/256	147.33	12.79	7.59

<표 7> Intel Xeon CPU 에서의 최적화(단위 : CPB)

입력/출력	SHA2	Standard LSH	SIMD LSH
64/256	175.75	46.50	18.72
4096/256	150.59	21.75	7.77
Long/256	150.23	16.37	7.02

5. 결론

현재 사회 곳곳의 다양한 환경에서 전자 서명이나 서명 생성 등의 기능을 제공하기 위하여 해시함수가 사용되고 있다. 본 논문에서는 경량 고속해시함수인 LSH 알고리즘을 Intel CPU 에서 지원하는 SSE 를 이용하여 최적화를 진행하였다. LSH 알고리즘의 압축 함수에서 사용되는 ARX 연산 부분에 SIMD 를 적용하여 해시함수의 최적화를 진행하였다. LSH 알고리즘 중 압축함수에 최적화를 진행한 결과 Intel i7 CPU 에서는 SIMD 를 적용하지 않는 LSH 알고리즘 보다 최대 1.68 배 빠른 것을 증명하였고, Intel Xeon CPU 에서는 최대 2.79 배의 속도 향상을 확인 할 수 있었다. 또한 기본적인 LSH 알고리즘과 Openssl 에서 사용하는 SHA2 알고리즘을 비교한 결과는 i7 CPU 에서 5.8 배 빠른 것으로 나타났고 SIMD 를 적용한 LSH 와는 8.99 배의 속도 차이를 확인 할 수 있었다.

본 논문에서는 SHA2 알고리즘과 LSH 알고리즘을 비교한 결과를 제시함으로써 LSH 알고리즘의 효율성을 증명하였고 또한 LSH 알고리즘에 SIMD 를 적용하여 최적화된 LSH 알고리즘을 개발하였다. 본 실험에서는 압축함수에 국한하여 SIMD 를 적용하였기 때문에 초기화 함수나 종료 함수에도 SIMD 를 적용한다면 더 높은 효율을 기대할 수 있을 것이고 또한 ARM, PowerPC 등과 같은 다른 CPU 에서 지원하는 SIMD 를 이용하여 최적화를 진행 할 수 있을 것이다.

참고문헌

[1] Dong-Chan Kim, LSH: A New Fast Secure Hash Function Family, Information Security and Cryptology - ICISC 2014 Lecture Notes in Computer Science Vloume 8949, 2015, pp 256~313

[2] Intel intrinsics guide. http://software.intel.com/sites/landingpage/IntrinsicsGuide.

[3] Intel. Intel architecture instruction set extensions programming reference. 319433-018, FEBRUARY 2014.

[4] S. K. Mathew J. Walker, F. Sheikh and R. Krishnamurthy. Askein-512 hardware implementation. Second SHA-3 Candidate Conference,2010. http://csrc.nist/gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/WALKER_skein-intel-hwd.pdf/.