*Journal of The Korea Institute of Information Security & Cryptology* ISSN 1598-3986(Print)
VOL.26, NO.1, Feb. 2016 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2016.26.1.31

# SSE와 AVX를 활용한 LSH의 병렬 최적 구현*

박 철 희,† 김 현 일, 홍 도 원,‡ 서 창 호
공주대학교

# Parallel Implementation of LSH Using SSE and AVX*

Cheolhee Pack,† Hyun-il Kim, Dowon Hong,‡ Changho Seo
Kongju National University

## 요 약

해시함수는 인증, 서명, 데이터 무결성 등을 수행하기 위해 반드시 필요한 암호학적 프리미티브이다. 2005년 중국의 Wang교수에 의해 MD5와 SHA-1과 같은 구조에 대해 충돌쌍 공격이 제기됨으로써 NIST는 SHA-3 프로젝트를 진행하여 Keccak을 새로운 표준 해시함수로 선정하였다. 또한 국내의 경우 국가보안기술연구소에서는 높은 안전성과 효율성을 제공하는 해시함수 LSH를 개발했다. LSH는 초기화, 압축, 완료함수로 이루어지며 함수 내에서 mod $2^W$상에서의 덧셈, 비트단위 순환, 워드 단위 순환 및 xor연산을 수행한다. 이러한 연산은 동시에 독립적으로 수행될 수 있으며 병렬화가 가능하다. 본 논문에서는 LSH를 분석하여 구조적인 측면에서 속도를 개선할 수 있는 방법을 제안하고 SIMD의 SSE와 AVX를 활용하여 LSH 함수를 병렬 구현한다.

## ABSTRACT

Hash function is a cryptographic primitive which conduct authentication, signature and data integrity. Recently, Wang et al. found collision of standard hash function such as MD5, SHA-1. For that reason, National Security Research Institute in Korea suggests a secure structure and efficient hash function, LSH. LSH consists of three steps, initialization, compression, finalization and computes hash value using addition in modulo $2^W$, bit-wise substitution, word-wise substitution and bit-wise XOR. These operation is parallelizable because each step is independently conducted at the same time. In this paper, we analyse LSH structure and implement it over SIMD-SSE, AVX and demonstrate the superiority of LSH.

**Keywords:** Hash function, LSH, parallelization, SIMD

## I. 서 론

해시 함수는 인증, 서명, 데이터 무결성 등 여러 가지 암호학적 기능을 수행하기 위해 반드시 필요한 암호 프리미티브이다. 2005년 중국의 Wang교수에 의해 MD5와 SHA-1과 같은 구조에 대한 충돌쌍 공격[1][2]이 제기됨으로써 SHA-1과 유사한 구조를 갖는 SHA-2 또한 구조적으로 위협이 되었다. 그에 따라 NIST는 SHA-3 프로젝트[3] 진행하여 Keccak[4]을 새로운 해시함수로 선정했다[5]. 국내의 경우 국가보안기술연구소는 높은 안전성을 보장하고 소프트웨어 측면에서 SHA-3[6] 보다 좋은 효율성을 제공하는 해시함수 LSH[7]를 개발했다. LSH는 초기화, 압축, 완료함수로 이루어지며 함수 내에서 mod $2^w$상에서의 덧셈, 비트단위 순환, 워드 단위 순환 및 xor연산을 수행한다. 이러한 연산은 각각의 블록에 대해 독립적으로 수행되며 병렬화가 가능하다. 본 논문에서는 LSH를 분석하고 구조적인 측면에서 속도를 개선할 수 있는 방법을 제안한다.

Received(07. 31. 2015), Modified(1st: 09. 21. 2015, 2nd: 12. 24. 2015), Accepted(01. 04. 2016)
* 본 연구는 2015년 공주대학교 학술연구지원사업의 연구지원에 의하여 연구되었음.
† 주저자, newpch89@kongju.ac.kr
‡ 교신저자, dwhong@kongju.ac.kr(Corresponding author)

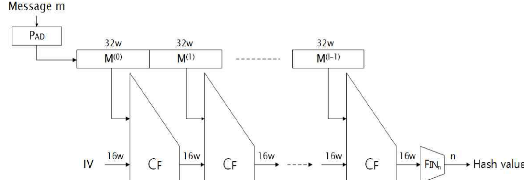


Fig. 1. Overall structure of LSH

또한 SIMD의 SSE와 AVX를 활용하여 병렬 구현하고 SHA-3 및 LSH 참조 코드와의 속도를 비교한다.

## II. 해시함수 LSH

Fig. 1.은 해시함수 LSH의 전체적인 구조를 나타내며 Table. 1.은 그에 대한 도식을 나타낸다. LSH는 크게 초기화, 압축, 완료함수로 구성되며 임의 길이의 메시지를 입력받아 224, 256, 384, 512 크기의 비트열을 출력한다. 또한 LSH는 워드 단위로 연산이 이루어지고, n 비트의 출력을 가지는 LSH함수를 LSH-8w-n으로 나타낸다. 이때 워드 단위의 크기 $w$는 32 또는 64 비트이다.

### 2.1 초기화 함수

LSH의 초기화 함수는 입력된 메시지의 패딩, 분할 및 연결 변수의 초기화를 수행 한다.

LSH의 입력 메시지를 $msg$라 한다면 $msg$는 우선적으로 패딩 과정을 거친다. 패딩 과정은 $msg$의 끝에 비트 '1'을 덧붙이고 길이가 $32wt$ 비트가 될 때까지 비트 '0'을 연접한다. 이때 t는 $\lceil (|msg|+1)/(32w) \rceil$ 이다.

메시지 패딩을 거친 메시지를 $msg_p = m_0 \| m_1 \| \cdots \| m_{32wt-1}$ 라 하면 $msg_p$를 $4wt$개의 바이트 배열 $msg_a = (m[0], \cdots, m[4wt-1])$로 나타낼 수 있으며 이때 $m[k] = m_{8k} \| m_{8k+1} \| \cdots \| m_{8k+7} (0 \le k \le (4wk-1))$이다. 또한 바이트 배열 $msg_a$를 워드 배열 $M = (M[0], \cdots, M[32t-1])$로 변환할 수 있으며 이때 $M[k] = m[ws/8 + (w/8-1)] \| \cdots \| m[ws/8+1] \| m[ws/8] (0 \le k \le (32t-1))$이다. 이어서 워드 배열 $M$은 $t$개의 메시지 블록 $M^{(0)}, M^{(1)}, \cdots, M^{(t-1)}$로 분할할 수 있으며 이때 $M^{(k)} = (M[32i], M[32i+1], \cdots, M[32i+32]) (0 \le k \le (t-1))$을 나타낸다.

연결 변수 $CV^{(0)}$는 LSH의 초기화 상수 배열을 이용하여 각 배열의 색인 별로 연결 변수의 값을 할당한다. 즉, $CV^{(0)}[i] \leftarrow IV[i] (0 \le i \le 15)$이다.

Table 1. Overall process of LSH

| | |
|---|---|
| $Input : message\ m$ | |
| $Process :$ $m_p \leftarrow pad(m)$ | *Initialization* |
| $generate\ (M^{(0)}, M^{(1)}, \cdots, M^{(t-1)})\ from\ m_p$ | |
| $CV^{(0)} \leftarrow IV$ | |
| $for\ i = 0\ to\ (t-1)\ do$ | *Compression* |
| $CV^{(i+1)} \leftarrow CF(CV^{(i)}, M^{(i)})$ | |
| $end\ for$ | |
| $h \leftarrow FIN_n(CV^{(t)})$ | *Finalization* |
| $Output : h \in \{0,1\}^n$ | |

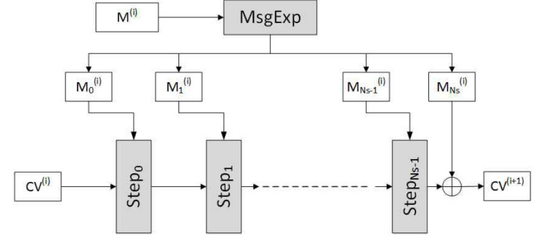


Fig. 2. Compression function of LSH

### 2.2 압축 함수

입력 메시지 $msg$는 초기화 함수를 거친 후 $CV^{(0)}$와 함께 압축함수의 입력으로 전달된다. Fig. 2.는 압축함수의 전반적인 과정을 나타낸다. 압축함수는 메시지확장, 단계, 워드 단위 순환 함수로 이루어지며 $w$가 32 비트인 경우 단계함수의 라운드 수는 26이며 $w$가 64인 경우의 라운드 수는 28이다.

#### 2.2.1 메시지 확장 함수

압축 함수 내의 메시지 확장 함수는 패딩 과정을 거친 메시지 $M^{(k)}$ $(0 \le k \le (t-1))$로부터 순차적으로 32개의 워드를 입력받아 연속적으로 16개의 워드를 생성하며 최종적으로 16×(단계 함수의 라운드 수+1)개의 워드로 확장된다. 초기에 입력된 32개의 워드에 대하여 처음 16개의 워드를 $M_0$이라 하고 그 다음 16개의 워드를 $M_1$이라 한다면 $M_0$ 내의 워드들을 치환한 후 $M_1$과 순차적으로 더하여 $M_2$를 생

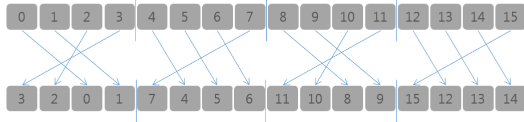

Fig. 3. Permutation in the message expansion function

성한다. Fig. 3은 해당 치환을 나타내며 덧셈 연산은 워드 단위로 수행된다. 이 과정을 연속적으로 진행하여 메시지를 확장 한다.

### 2.2.2 단계 함수

단계 함수는 메시지 덧셈, 섞음 함수 그리고 워드 단위 치환 함수로 이루어진다. 메시지 덧셈 과정은 연결 변수 $CV^{(i)}$에 확장된 메시지 블록 $M_i$를 Xor 시키는 과정이며 이때 $i$는 단계함수의 라운드를 의미한다.

메시지 덧셈 과정을 거친 16개의 워드에 대하여 상위 8개의 워드를 X라 하고 하위 8개의 워드를 Y라 한다면 섞음 함수는 Fig. 4.과 같은 과정을 거친다.

연결 변수의 Y를 X에 순차적으로 더한 후 X의 모든 워드를 비트단위 순환인 $\alpha$ 순환 하여 단계 함수의 라운드 상수인 $SC_i$를 Xor시킨다. 이때 $w$가 32인 경우 짝수라운드의 $\alpha$는 왼쪽으로 29비트 순환이며 홀수라운드의 $\alpha$는 왼쪽으로 5비트 순환이다.

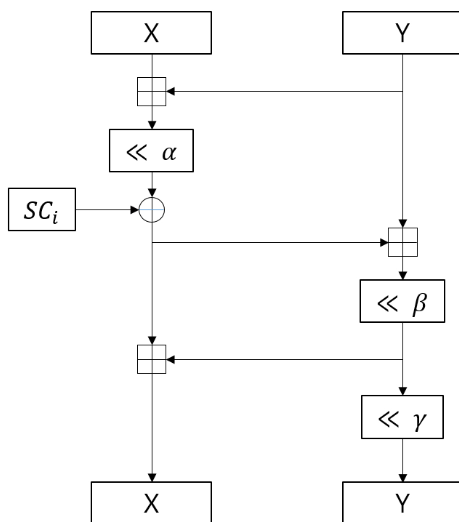


Fig. 4. Mix function of compression function

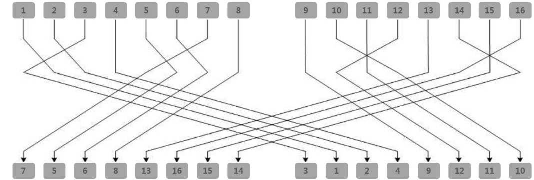

Fig. 5. Permutation in the step function

또한 $w$가 64인 경우 짝수라운드의 $\alpha$는 왼쪽으로 23비트 순환이며 홀수라운드의 $\alpha$는 왼쪽으로 7비트 순환이다.

계속적으로 X를 Y에 순차적으로 더한 후 Y의 모든 워드를 비트단위 순환인 $\beta$ 순환한다. 이때 $w$가 32인 경우 짝수라운드의 $\beta$는 왼쪽으로 1비트 순환이며 홀수라운드의 $\beta$는 왼쪽으로 17비트 순환이다. 또한 $w$가 64인 경우 짝수라운드의 $\beta$는 왼쪽으로 59비트 순환이며 홀수라운드의 $\beta$는 왼쪽으로 3비트 순환이다. 마지막으로 최종적인 X는 X에 Y를 순차적으로 더한 값이고, 최종적인 Y는 비트단위 $\gamma$ 순환한 값이다. 이때 비트단위 순환 $\gamma$는 모두 왼쪽 순환이며 순환양은 Y내 8개의 워드 순서대로 $w$가 32인 경우 {0,8,16,24,24,16,8,0}이고 $w$가 64인 경우 {0,16,32,48,8,24,40,56}이며 모든 라운드에서 같은 양을 순환한다.

섞음 함수가 완료된 16개의 워드는 워드 단위로 각각의 위치를 치환하게 된다. Fig. 5.는 해당 치환의 과정을 나타낸다.

위의 과정을 라운드 수에 맞게 수행한 뒤 생성되는 16개의 워드들을 (라운드수+1)번째 메시지 블록에 순차적으로 xor하여 완료함수의 입력으로 전달한다.

### 2.3 완료 함수

완료 함수는 압축 함수의 최종 출력으로부터 n 비트 길이의 해시 값을 생성한다. 압축 함수의 최종 출력으로부터 상위 8개의 워드에 하위 8개의 워드를 순차적으로 xor하여 8개의 워드로 구성된 배열 $H$를 생성하고 $H$ 내의 각 워드들을 Fig. 6.와 같이 8비트 크기로 분할한 뒤 역순으로 재배치한다. 모든 워드들이 재배치된 배열을 $H'$이라고 한다면 완료 함수의 최종출력 $h$는 $(H'[0]\|\cdots\|H'[7])_{[0:n-1]}$이다.

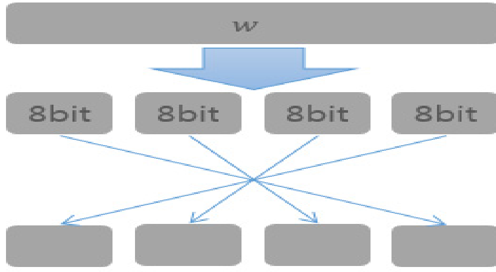


Fig. 6. Relocation of word in the finalization function

## III. LSH의 구조 분석 및 최적 병렬 구현

LSH는 mod $2^w$ 상에서의 덧셈, 비트단위 순환, 워드 단위 순환 및 xor연산을 수행한다. 모든 연산은 워드 단위로 수행되며 병렬을 적용하여 한 번의 연산으로 여러 개의 워드를 처리할 수 있다. 따라서 본 논문에서는 SIMD(single instruction multiple data)의 SSE(streaming SIMD extension)와 AVX(advanced vector extension)를 활용하여 순차적인 LSH의 함수를 병렬로 구현하였다. SIMD는 병렬 프로세서로써 CPU의 레지스터 변수를 활용하여 한 번의 명령어로 여러 개의 데이터를 동시에 처리하는 방식이다. SSE의 경우 최대 128 비트 크기의 레지스터를 활용할 수 있고 AVX의 경우 최대 256 비트 크기의 레지스터를 활용할 수 있다. 따라서 워드 단위로 연산을 수행하는 LSH는 $w$가 32 비트 일 때, SSE의 경우 4개의 워드를, AVX의 경우 8개의 워드를 동시에 처리할 수 있다. 또한 $w$가 64 비트 일 때, SSE의 경우 2개의 워드를, AVX의 경우 4개의 워드를 동시에 처리할 수 있다(기존의 방식은 한 번의 연산으로 한 개의 워드를 처리하는 방식이다.).

본 논문에서의 구현 환경은 Intel Core i7-4790@3.6GHz의 CPU를 이용하였고 운영체제는 Window 7 32bit이다. 또한 Visual Studio 2013 상에서 LSH를 병렬 구현하여 속도를 측정하였다.

### 3.1 SSE를 활용한 LSH의 병렬 구현

128 비트 크기의 CPU 레지스터를 활용하는 SSE의 SSSE3버전으로 LSH를 병렬 구현한다.

병렬 구현된 모든 연산에 대하여 $w$가 32 비트인 경우 한 번의 명령으로 4개의 워드를, $w$가 64비트인 경우 한 번의 명령으로 2개의 워드를 동시에 처리한다.

LSH 내의 함수에 대하여, 압축 함수의 메시지 확장, 덧셈 및 단계함수와 완료함수의 $H$생성 과정은 한 번에 여러 개의 데이터를 동시에 처리하여 진행할 수 있다. Table. 2.는 SSE 병렬구현에 사용된 어셈블리 함수를 나타낸다.

Table 2. Used SSE assembly func.

| SSE func. | | explanation |
|---|---|---|
| SSE2 | movdqu | Substitute 128-bit memory to 128bit register and vice versa |
| | paddd | Add four each 32-bit integers |
| | pxor | xor 128-bit |
| | pslld | Left shift four each 32-bit integers |
| | psrld | Right shift four each 32-bit integers |
| | pshufd | Location change of four 32-bit integers |
| SSSE3 | pshufb | Location change of sixteen 8-bit integers |

#### 3.1.1 압축 함수의 병렬구현

압축함수는 메시지 확장, 덧셈, 단계 함수로 이루어진다. 첫 번째로 메시지 확장과정은 두 개의 $16w$ 메시지 배열로부터 치환 및 덧셈으로 새로운 $16w$ 메시지 배열을 생성한다. 이때 치환 되는 메시지를 $M_0$이라 하면, $M_0[0] \sim M_0[3]$와 $M_0[8] \sim M_0[11]$은 같은 치환을 하고 $M_0[4] \sim M_0[7]$와 $M_0[12] \sim M_0[15]$ 또한 같은 치환을 하게 된다(Fig. 3. 참고). 메시지 확장 과정을 병렬로 구현할 경우 워드 단위의 shuffle함수를 통해 배열의 위치를 치환 하고 다른 메시지 배열에 더하여 새로운 메시지 배열을 생성한다. 이를 $w$가 32 비트인 경우의 의사 코드로 나타내면 다음 식(1)과 같다.

$$pshufd\ xmm0,\ xmm0,\ 01001011b;$$
$$paddd\ xmm0,\ xmm1; \quad (1)$$

이때 레지스터 $xmm0$은 치환될 4개의 워드를 담고 있으며 $xmm1$은 더해질 4개의 워드를 담고 있다고 가정한다. 또한 $01001011b$는 0~3(4~7)번째 워드들의 치환 값을 나타내며 8~11(12~15)번째 워드들의 치환 값은 $10010011b$이다. 이 과정을 단계 함수의 라운드 수에 맞게 반복한다.

두 번째로 메시지 덧셈 과정은 연결 변수 배열인 $CV$에 메시지 블록을 xor하게 된다. 이 과정은 워드 단위의 $pxor$함수를 통해 병렬 구현하였으며 이를 의사 코드로 나타내면 다음 식 (2)와 같다.

$$pxor\ xmm0,\ xmm1; \quad (2)$$

마지막으로 단계 함수는 섞음 과정과 워드 단위 순환으로 이루어져 있다. 섞음 과정에서는 덧셈, 비트 단위 순환 및 xor 연산을 수행한다. 덧셈과 xor 연산은 워드 단위의 $paddd$함수와 $pxor$함수를 통해 병렬 구현하였다. 또한 비트 단위 순환의 경우 한 라운드 내에서 $\alpha$ 순환과 $\beta$ 순환은 모든 워드가 같은 양의 순환을 하게 된다. 그에 따라 $\alpha$ 순환과 $\beta$ 순환의 경우 순환 크기에 따라서 일괄적으로 $\alpha$, $\beta$ 크기의 왼쪽 shift결과와 $w-\alpha$, $w-\beta$ 크기의 오른쪽 shift결과를 xor하여 비트 단위 순환을 병렬화 하였다. 이를 의사 코드로 나타내면 다음 식 (3)과 같다.

$$pslld\ xmm0,\ x_1;$$
$$psrld\ xmm1,\ x_2; \quad (3)$$
$$pxor\ xmm0,\ xmm1;$$

이때 $pslld$, $psrld$는 레지스터 $xmm0$, $xmm1$ 내의 모든 워드에 대하여 왼쪽, 오른쪽 shift를 $x_1$, $x_2(=w-x_1)$ 만큼 수행한다.

$\gamma$ 순환의 경우 한 라운드 내에서 각 워드의 순환 양이 모두 같지 않기 때문에 일괄적인 shift 함수로 구현하는 것이 불가능하다. 따라서 $\gamma$ 순환의 경우 순환 양이 모두 8의 배수인 것을 이용하여 8 비트 크기의 shuffle함수로 $\gamma$ 순환을 병렬 구현하였다. Fig. 7.은 $w$가 32인 경우 비트 단위 순환인 $\gamma$ 순환을 8 비트 크기의 shuffle로 구현한 그림을 나타낸다. 이를 의사 코드로 나타내면 다음 식 (4)와 같다.

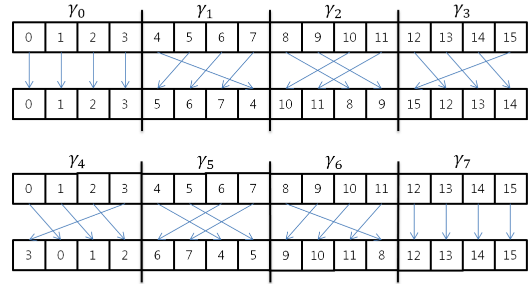


Fig. 7. 8-bit shuffle for $\gamma$-shift(w=32)

$$pshufb\ xmm0,\ xmm1; \quad (4)$$

이때 $xmm0$은 4개의 워드를 담고 있으며($w$가 64인 경우 2개의 워드) $xmm1$은 각 워드에 대한 $\gamma$ 순환의 크기를 담고 있다.

마지막으로 워드 단위 치환 함수는 워드 단위의 shuffle 함수를 통해 레지스터 내에서 치환하고 레지스터를 서로 교환하는 방식으로 구현하였다. 이를 의사 코드로 나타내면 다음 식 (5)와 같다.

$$pshufd\ xmm0,\ xmm2,\ 11010010b; \quad (5)$$

식 (5)는 $xmm2$에 있는 워드들을 $11010010b$와 같이 치환 하고 $xmm0$에 그 결과를 대입하는 연산을 나타낸다.

#### 3.1.2 완료 함수의 병렬구현

완료 함수는 압축 함수의 최종 출력으로부터 상위 8개의 워드에 하위 8개의 워드를 순차적으로 xor하여 8개의 워드로 이루어진 배열 $H$를 생성한다. 이 과정은 워드 단위의 xor연산을 통해 동시에 여러 개의 워드를 처리 하도록 병렬 구현 하였다. 이에 대한 의사 코드는 식 (2)와 유사하다.

### 3.2 AVX를 활용한 LSH의 병렬 구현

256 비트 크기의 CPU 레지스터를 활용하는 AVX의 AVX2버전으로 LSH를 병렬 구현한다. AVX는 SSE 보다 2배 크기의 레지스터를 활용하기 때문에 $w$가 32 비트인 경우 한 번의 명령으로 8개의 워드를, $w$가 64 비트인 경우 한 번의 명령으로 4개의 워드를 동시에 처리한다. Table. 3.은 AVX

Table 3. Used AVX func.

| AVX func. | | explanation |
|---|---|---|
| AVX | vmovdqu | Substitute 128-bit memory to 128bit register and vice versa |
| AVX2 | vpaddd | Add four each 32-bit integers |
| | vpxor | xor 128-bit |
| | vpslld | Left shift four each 32-bit integers |
| | vpsrld | Right shift four each 32-bit integers |
| | vpshufd | Location change of four 32-bit integers |
| | vpshufb | Location change of sixteen 8-bit integers |
| | vperm2i128 | 128-bit size permutation |

Table 4. Used Intrinsic func.

| assembly func. | Intrinsic func. |
|---|---|
| movdqu | _mm_loadu_si128 |
| | _mm_storeu_si128 |
| paddd | _mm_add_epi32 |
| pxor | _mm_xor_si128 |
| pslld | _mm_slli_epi32 |
| psrld | _mm_srli_epi32 |
| shufd | _mm_shuffle_epi32 |
| pshufb | _mm_shuffle_epi8 |
| vmovdqu | _mm256_loadu_si256 |
| | _mm256_storeu_si256 |
| vpaddd | _mm256_add_epi32 |
| vpxor | _mm256_xor_si256 |
| vpslld | _mm256_slli_epi32 |
| vpsrld | _mm256_srli_epi32 |
| vpshufd | _mm256_shuffle_epi32 |
| vpshufb | _mm256_shuffle_epi8 |
| vperm2i128 | _mm256_permute2x128_si256 |

병렬구현에 사용된 어셈블리 함수를 나타낸다. 병렬 구현의 방식은 SSE의 구현과 유사하지만 $w$가 32비트인 경우 한 번에 8개의 워드를 처리하기 때문에 압축 함수의 워드 단위 순환을 구현하기 위하여 레지스터 내의 구조를 변경하는 $vperm2i128$함수를 추가적으로 사용하였다. Fig. 8.은 위의 경우를 구현한 소스코드 이다.

### 3.3 LSH의 구조적 측면에서의 최적화된 병렬 구현

LSH는 초기화, 압축, 완료 함수로 이루어진다. 이중 압축 함수는 또 다시 메시지 확장, 덧셈, 단계

```
//wordperm
vpshufd ymm0, ymm0, 11010010b;
vpshufd ymm1, ymm1, 01101100b;
vmovdqu ymm2, ymm0;
vmovdqu ymm3, ymm1;

vperm2i128 ymm0, ymm3, ymm2, 19;
vperm2i128 ymm1, ymm2, ymm3, 32;
```

Fig. 8. LSH AVX word permutation(w=32)

함수로 이루어진다. 이때 메시지 확장 과정은 두 개의 $16w$ 메시지 배열로부터 연속적으로 새로운 $16w$ 메시지 배열을 생성한다. 이때 연속적으로 새롭게 생성되는 메시지 블록은 할당된 메모리에 저장되며 단계 함수 내에서 메모리로부터 호출되어 연결 변수와 더해진다. 이 과정에서 레지스터로부터 메모리로, 메모리로부터 레지스터로 메시지 블록이 이동하기 때문에 레지스터 활용이 비효율적일 수 있다. 따라서 본 논문의 구현에서는 초기 2개의 메시지 배열을 특정 레지스터 변수에 대입하고 라운드에 따라 해당 레지스터를 계속적으로 갱신・유지하여 배열의 호출 및 저장과정을 제거하였다. 즉 첫 번째 메시지 블록과 두 번째 메시지 블록을 레지스터에 대입하여 바로 첫 번째, 두 번째의 단계 함수 라운드를 진행하고, 반복 loop를 수행할 때 loop 내에서 메시지 블록을 담은 고정된 레지스터를 갱신(메시지 확장)하여 불필요한 오버헤드를 제거하였다. 그러나 이 과정은 메시지 블록을 담은 레지스터를 계속 유지해야 하므로 레지스터 사용에 제한이 된다. 따라서 레지스터 변수 사용의 자율성을 위해 Intel에서 제공하는 Intel Intrinsic 함수[8]를 활용하여 메시지 확장 과정과 단계 함수를 동시 진행 할 수 있도록 구현하였다. 또한 메시지 확장 과정에서 워드 단위 치환을 원활하게

Table 5. Result of implementation and comparing

| w | algorithm \ message length | | long message | 4096 Byte | 64 Byte |
|---|---|---|---|---|---|
| 32bit | SSE | LSH-256 : SSE-assembly | 4.61 | 4.79 | 11.77 |
| | | LSH-256 : SSE-Intrinsic | 3.39 | 3.63 | 10.08 |
| | AVX | LSH-256 : AVX-assembly | 3.31 | 3.55 | 9.52 |
| | comparing code | LSH-256(ref. code) | 12.01 | 12.52 | 26.81 |
| | | SHA3-256 | 12.50 | 12.97 | 34.56 |
| 64bit | SSE | LSH-512 : SSE-assembly | 6.32 | 6.83 | 32.63 |
| | AVX | LSH-512 : AVX-assembly | 3.12 | 3.60 | 22.41 |
| | | LSH-512 : AVX-Intrinsic | 1.98 | 2.26 | 16.13 |
| | comparing code | LSH-512(ref. code) | 17.14 | 19.18 | 80.34 |
| | | SHA3-512 | 24.07 | 24.16 | 29.81 |

수행하기 위해 한 개의 레지스터에 4개의 워드를 동시에 처리할 수 있는 부분에만 적용하였다. 즉, $w$가 32 비트인 경우에서의 SSE와 $w$가 64 비트인 경우에서의 AVX상에서만 구현하였다. 즉 특정 레지스터를 고정하여 식 (1)의 과정을 단계 함수를 진행하는 loop 내에서 수행 되도록 구현하였다.

병렬 구현 시 사용된 Intrinsic 함수는 어셈블리 버전의 명령어와 상응하는 Intrinsic 함수를 사용하였다. Table. 4.는 병렬구현 시 사용된 Intrinsic 함수를 나타낸다. 또한 AVX의 assembly구현에서 $w$가 32 비트인 경우 한 개의 레지스터에 8개의 워드를 담을 수 있기 때문에 여분의 레지스터 활용이 가능하다. 따라서 이 경우도 마찬가지로 메시지 확장 과정과 단계 함수의 동시진행으로 구현하였다.

## IV. LSH 병렬화의 성능 측정 결과

Table. 5.는 병렬 구현된 LSH 코드와 참조 코드 및 SHA-3와의 성능을 비교한 결과이다. 참조 코드는 국가보안기술연구소와 한국 암호 포럼[9]에 게재된 LSH 참조 코드 이며 SHA-3의 성능 결과는 eBASH[10]에서 본 논문의 구현 환경과 가장 유사한 환경(Intel Core i7-4770)의 최고 성능 결과를 참고하였다. 본 연구에서 LSH의 구현 환경은 Intel Core i7-4790 @3.6GHz CPU를 사용하였고 운영체제는 Window 7 32bit이다. 또한 속도 측정 시 Visual studio 2013을 활용하였으며 속도 최적화 옵션을 적용하였다. 측정된 성능의 단위는 cycles/byte이고 해당 길이의 데이터 1000개를 측정하는 실험 20회를 실시하여 얻어진 값 중 최솟값을 기록하였다.

측정된 결과를 long message 기준으로 비교할 때, $w$가 32 비트인 경우 LSH AVX-assembly 구현이 3.31(cycles/byte)로 가장 빠르게 측정되었으며 참조 코드 대비 약 3.62배 정도 개선하였다. 또한 $w$가 64 비트인 경우 LSH AVX-Intrinsic 구현이 1.98(cycles/byte)로 가장 빠르게 측정되었으며 참조 코드 대비 약 8.65배 정도 개선하였다.

## V. 결 론

본 논문에서는 안전하고 효율적인 해시함수 LSH를 분석하고 CPU성능에 맞게 병렬 적용하여 구현하였다. 또한 구조적인 측면에서의 속도 개선 방안을 제안하고 참조코드 및 국제 표준 해시함수인 SHA-3와 비교하였다. LSH를 구조적 최적화 및 병렬 구현한 결과 참조코드 대비 월등한 효율성을 보였으며 국제 표준 해시 함수인 SHA-3보다도 월등한

소프트웨어적 효율성을 확인하였다.

LSH는 알고리즘의 구조상 병렬화 작업이 수월하고 다양한 프로세서에서 최적화가 가능하다. 따라서 ARM의 NEON 및 GPGPU등을 활용하여 다양한 임베디드 시스템에서의 LSH 알고리즘 최적화 연구가 진행 되어야 한다.

## References


〔1〕 X. Wang, A. C. Yao and F. Yao, "Cryptanalysis on SHA-1," CRYPTOGRAPHIC HASH WORKSHOP, October 2005.

〔2〕 X. 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〕 Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G, "Keccak sponge function family main document," Submission to NIST (Round 2), 3, 30. 2009.

〔5〕 SHA-3 STANDARDIZATION, http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standardization.html

〔6〕 NIST, "DRAFT FIPS PUB 202," May 2014.

〔7〕 Kim, D. C., Hong, D., Lee, J. K., Kim, W. H., & Kwon, D, "Lsh: A new fast secure hash function family," In Information Security and Cryptology-ICISC 2014, Springer International Publishing, pp. 286-313, 2014.

〔8〕 Intel Intrinsics Guide, http://software.intel.com/sites/landingpage/IntrinsicsGuide/

〔9〕 Korea cryptographic forum. http://kcryptoforum.or.kr/

〔10〕 eBASH, http://bench.cr.yp.to/ebash.html

## 〈저자소개〉

박 철 희 (Cheolhee Park) 학생회원
2014년 2월: 공주대학교 응용수학과 학사 졸업
2014년 9월~현재: 공주대학교 수학과 석사 재학
〈관심분야〉 암호모듈 구현, 데이터 보호 기술

김 현 일 (Hyun-il Kim) 학생회원
2014년 2월: 공주대학교 응용수학과 학사 졸업
2014년 3월~현재: 공주대학교 융합과학과 석사 재학
〈관심분야〉 암호모듈 구현, 데이터 보호 기술

홍 도 원 (Dowon Hong) 종신회원
1994년 2월: 고려대학교 수학과 학사
2000년 2월: 고려대학교 수학과 박사
2000년 4월~2012년 2월 : 한국전자통신연구원 팀장, 책임연구원
2012년 3월~현재: 공주대학교 응용수학과 교수
〈관심분야〉 암호기술, 프라이버시 보호기술

서 창 호 (Changho Seo) 종신회원
1990년: 고려대학교 수학과 학사
1992년: 고려대학교 수학과 이학석사
1996년: 고려대학교 수학과 이학박사
1996년~1996년: 국방과학연구소 선임연구원
1996년~2000년: 한국전자통신연구원 선임연구원, 팀장
2000년~현재: 공주대학교 응용수학과 교수
〈관심분야〉 암호알고리즘, PKI, 무선인터넷 보안 등