LSH: A New Fast Secure Hash Function Family

Dong-Chan Kim, Deukjo Hong, Jung-Keun Lee,
Woo-Hwan Kim, and Daesung Kwon

The attached institute of ETRI, Daejeon, Korea
{dongchan,hongdj,jklee,whkim5,ds_kwon}@ensec.re.kr

Abstract. Since Wang's attacks on the standard hash functions MD5 and SHA-1, design and analysis of hash functions have been studied a lot. NIST selected Keccak as a new hash function standard SHA-3 in 2012 and announced that Keccak was chosen because its design is different from MD5 and SHA-1/2 so that it could be secure against the attacks to them and Keccak's hardware efficiency is quite better than other SHA-3 competition candidates. However, software efficiency of Keccak is somewhat worse than present standards and other candidates. Since software efficiency becomes more important due to increase of kinds and volume of communication/storage data as cloud and big data service spread widely, its software efficiency degradation is not desirable.
In this paper, we present a new fast hash function family LSH, whose software efficiency is above four times faster than SHA-3, and 1.5-2.3 times faster than other SHA-3 finalists. Moreover it is secure against all critical hash function attacks.

Keywords: hash function, Merkle-Damgård mode, wide-pipe structure, PGV model, parallel implementation, SIMD instruction, ARX operations

1 Introduction

1.1 Background and motivation

As critical attacks were found for dedicated hash functions including MD5 and SHA-1 [50, 54, 55], doubts on the security have been continuously raised that SHA-2 may be vulnerable to such attacks due to similar design approach to attacked hash functions. For this reason, NIST has prepared a new US standard hash function SHA-3 based on Keccak, the winner of SHA-3 Cryptographic Hash Algorithm Competition(2007-2012) [8]. NIST said that Keccak is chosen because its design is different from MD5 and SHA-1/2 so that it could be secure against the attacks to them and Keccak's hardware efficiency is quite better than other SHA-3 competition candidates. However Keccak shows relatively low software performance compared to other candidates.

Much more and bigger data needs to be hashed in the era of smart devices, cloud and big data, so the faster hash function is strongly required to prevent any degradation in the performance of cryptographic modules or services. To maximize such performance, implementing cryptographic algorithm at the hardware

level would be a good way. However, the hardware implementation won't be able to have the competitive edge in price to the software one without large quantity production. Even when the hardware implementation costs less, the software implementation has many advantages in terms of management: flexibility, portability, ease of use/upgrade, etc [12]. Upon these, a cryptographic algorithm having good software performance would be more marketable. In accordance with given circumstances and this consideration, we have developed a new hash function family LSH.

1.2 Design approach

We kept the following two principles in mind when designing the hash function family LSH:

- (Security) Adopt a hash structure which is provably secure and has full security bounds in the ideal setting, and design a compression function which has enough security margin to be defended against critical hash function attack.
- (Implementation) Design a compression function which can be easily implemented using parallel processing instructions SSE, AVX2 and NEON provided by pervading processors at present, such that a high-speed implementation can maximize the operational efficiency on platforms of servers and smart devices, as good as it gets.

1.3 Hash function family LSH

This paper presents a new hash function family LSH. Its best feature is that it shows the software performance superior to the other existing hash functions, thanks to the use of parallel processing instructions such as SSE, AVX2 and NEON.

Brief design description The hash function family LSH consists of n-bit hash functions based on w-bit word, $\{\text{LSH-}8w\text{-}n : w = 32 \text{ or } 64,\ 1 \leq n \leq 8w\}$. The hash structure of LSH-$8w$-n is wide-pipe Merkle-Damgård mode with one-zeros padding. After all message blocks are compressed, LSH-$8w$-n returns n-bit hash value by a finalization function. The compression function is designed on the 17^{th} PGV structure [18]. Bit length of a chaining variable is $16w$ and that of a message block is $32w$. The compression of a message block is proceeded by repeating step function operations. The number of step functions is 26 if $w = 32$, or 28 if $w = 64$. Each step function has three layers: (i) Message addition layer, (ii) Mix layer, (iii) Word-permutation layer. The message addition layer is a mere exclusive-or process between a chaining variable and a sub-message generated from a given message block. In mix layer, every two words are mixed independently. This layer is designed for parallel implementation with ARX(modular Addition, bit-Rotation, eXclusive-or) operations. Word-permutation layer plays the role of diffusion. Section 2 shows the specification of LSH-$8w$-n in detail.

Security The structure of LSH-8w-n which is composed of wide-pipe Merkle-Damgård mode and the compression function designed on the 17$^{\text{th}}$ PGV, is essentially proved to have full security under the ideal cipher model proof [18], i.e., the structure has 2^n pre-image and 2nd pre-image resistance, and $2^{n/2}$ collision resistance. We analyzed security of LSH with various cryptanalytic methods, and got the result that LSH-256-256 is secure against all the existing hash function attacks when the number of steps is 13 or more, while LSH-512-512 is secure if the number of steps is 14 or more. Note that the steps which work as security margin are 50% of the compression function. See Section 4.2 for details.

Comparison with SHA-3 finalists Keccak does not have good software performance among SHA-3 finalists. The designs of LSH, Blake, and Skein are based on ARX systems, but LSH is 1.5-2.3 times faster than the others. The best known attack on Keccak is the second-preimage attack on 8 rounds [13]. So, one may doubt that the high speed of LSH stems from relatively short steps by considering that the number of attacked rounds of Keccak is just 1/3 of its underlying permutation. However, he is missing the difference between design and attack in the sense that designers should determine a safe guideline for security and any attack result does not exclude possibility of better one in future. Note that we give the safe boundary for number of steps for all the existing attacks and 12 steps of LSH-256-256 and 13 steps of LSH-512-512 have never been broken by any hash function attack.

Performance The software speed of LSH is measured on various platforms and we compare the speed of LSH-8w-n with SHA-2 and SHA-3 competition finalists because they have been matured in terms of security. At the platform based on Haswell architecture CPU, the speed of LSH-256-n is 3.60 cycles/byte and the speed of LSH-512-n is 2.39 cycles/byte. LSH-8w-n is the fastest one on this platform. LSH-512-n is about 2.3-times faster than the second best one, Skein-512 of 5.58 cycles/byte. At the platform based on Samsung Exynos 5250 ARM Cortex-A15 CPU, the speed of LSH-256-n and LSH-512-n are 11.17 cycles/byte and 8.94 cycles/byte respectively. LSH-8w-n is the fastest one among them as well. Excepting LSH-8w-n, the highest speed is 13.46 cycles/byte of Blake-512, and LSH-512-n is about 1.5 times faster than it. See Section 5.3.

Contents of the paper Section 2 introduces the specification of LSH-8w-n. Section 3 presents the design rationale of LSH-8w-n. Section 4 presents security analysis of LSH-8w-n. Section 5 and Section 6 show software and hardware implementation. Differential characteristics for collision attacks of LSH-8w-n are added at Appendixes A.

2 Specification

The hash function family LSH consists of n-bit hash functions based on w-bit word, $\{\text{LSH-8}w\text{-}n : w = 32 \text{ or } 64,\ 1 \le n \le 8w\}$.

Table 1: Hex digit representation of a 4-bit string

hex	bit string	hex	bit string	hex	bit string	hex	bit string
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

2.1 Definitions, notation and conventions

Glossary of terms and acronyms

- Bit: Value of 0 or 1.
- Byte: 8-bit string.
- Word: w-bit string where w is either 32 or 64. In this paper, w is used as bit length of a word.
- Array: Collection of bytes or words.
- $\mathcal{W}^t$: Set of all t-word arrays ($t \geq 1$). In this paper, let $\mathcal{W}$ denote $\mathcal{W}^1$.
- LSH-8w-n: The n-bit hash function based on w-bit word ($1 \leq n \leq 8w$).

Bit strings and convention The little-endian convention is used when expressing an l-bit string x, i.e., $x = x_0||x_1||\cdots||x_{l-1}$ for all $x_i \in \{0,1\}$. However, for convenience, the big-endian convention is used when expressing an w-bit word, so that within each word, the most significant bit is stored in the left-most bit position, i.e., a word X is written in a bit string $x_{w-1}||x_{w-2}||\cdots||x_1||x_0$ for all $x_i \in \{0,1\}$. A hex digit is the representation of a 4-bit string as Table 1.

Operations We define the following operations:

Operations on bit strings Let x and y be bit strings.

- $x \parallel y$: Concatenation of x and y.
- $x \oplus y$: Bit-wise exclusive-or of x and y.
- $|x|$: Bit length of x.
- $x_{[i:j]} := x_i||x_{i+1}||\cdots||x_j$: Sub-bit string of a l-bit string $x = x_0||x_1||\cdots||x_{l-1}$ for $i \leq j$.

Operations on words Let $X = x_{w-1}||\cdots||x_0$ and Y be words and $Z = \sum_{l=0}^{w-1} z_l \cdot 2^l$ be an integer where $x_l, z_l \in \{0,1\}$ for all l.

- $X^{\lll i}$: i-bit left rotation of X.
- $X^{\ggg i} := X^{\lll w-i}$: i-bit right rotation of X.
- $\textsc{WordToInt}(X) := \sum_{l=0}^{w-1} x_l \cdot 2^l$.
- $\textsc{IntToWord}(Z) := z_{w-1}||\cdots||z_0$.
- $X \boxplus Y := \textsc{IntToWord}(\textsc{WordToInt}(X) + \textsc{WordToInt}(Y) \bmod 2^w)$.
- $X_{[i:j]} := x_i||x_{i-1}||\cdots||x_j$: Sub-bit string of a word X for $i \geq j$.

Data array assignment and conversion Let $\mathsf{X} = (X[0], \ldots, X[s-1])$ and $\mathsf{Y} = (Y[0], \ldots, Y[s-1])$ be s-word arrays, and let $\mathsf{z} = (z[0], z[1], \ldots, z[t-1])$ be a t-byte array, where $t = sw/8$. Let $p = w/8$.

- $\mathsf{X} \leftarrow \mathsf{Y}$: Assign a t-word array Y to X as $X[l] \leftarrow Y[l]$ for all l.
- $\mathsf{X} \leftarrow \mathsf{z}$: Assign a t-byte array z to a s-word array X as (1).

$$X[l] \leftarrow z[pl + (p-1)] \parallel \cdots \parallel z[pl+1] \parallel z[pl], \text{ for } 0 \le l < s. \tag{1}$$

- $\mathsf{z} \leftarrow \mathsf{X}$: Assign a s-word array X to a t-byte array z as (2).

$$z[l] \leftarrow X[\lfloor l/p \rfloor]^{\ggg 8l}_{[7:0]}, \text{ for } 0 \le l < t, \tag{2}$$

  where $\lfloor x \rfloor$ is the largest integer not greater than x.

Algorithm parameters The parameters used in the specification are as follows:

- n: Bit length of a hash value ($1 \le n \le 8w$).
- N_s: Number of step functions used in a compression function.
- $\mathsf{M}^{(i)} := (M^{(i)}[0], \ldots, M^{(i)}[31])$: The i-th 32-word array message block.
- $\mathsf{M}_j^{(i)} := (M_j^{(i)}[0], \ldots, M_j^{(i)}[15])$: The j-th 16-word array sub-message generated from the i-th message block $\mathsf{M}^{(i)}$.
- $\mathsf{IV} := (IV[0], \ldots, IV[15])$: The 16-word array initialization vector.
- $\mathsf{CV}^{(i)} := (CV^{(i)}[0], \ldots, CV^{(i)}[15])$: The i-th 16-word array chaining variable.
- $\mathsf{SC}_j := (SC_j[0], \ldots, SC_j[7])$: The j-th 8-word array step constant.
- $\mathsf{T} := (T[0], \ldots, T[15])$: The 16-word array temporary variable used in a step function.

2.2 Hash structure

The n-bit hash function based on w-bit word, LSH-8w-n has the wide-pipe Merkle-Damgård structure with one-zeros padding. The message hashing process of LSH-8w-n consists of the following three stages.

1. *Initialization*:
   - One-zeros padding of a given bit string message.
   - Conversion to 32-word array message blocks from the padded bit string message.
   - Initialization of a chaining variable with the initialization vector.
2. *Compression*:
   - Updating of chaining variables by iteration of a compression function with message blocks.
3. *Finalization*:
   - Generation of an n-bit hash value from the final chaining variable.

Intialization Let m be a given bit string message. The m is padded by one-zeros, i.e., the bit '1' is appended to the end of m, and the bit '0's are appended until a bit length of a padded message is $32wt$-bit, where $t = \left\lceil \frac{|m|+1}{32w} \right\rceil$ and $\lceil x \rceil$ is the smallest integer not less than x.

Let $m' = m_0||m_1||\cdots||m_{32wt-1}$ be the one-zeros-padded $32wt$-bit string of m. Then m' is considered as a $4wt$-byte array $\mathsf{m} = (m[0], \ldots, m[4wt-1])$, where $m[l] = m_{8l}||m_{8l+1}||\cdots||m_{8l+7}$ for all l. By (1), the $4wt$-byte array m converts into a $32t$-word array $\mathsf{M} = (M[0], \ldots, M[32t-1])$. From the word array M, we define the t 32-word array message blocks $\{\mathsf{M}^{(i)}\}_{i=0}^{t-1}$ by (3).

$$\mathsf{M}^{(i)} \leftarrow (M[32i], M[32i+1], \ldots, M[32i+31]). \tag{3}$$

The 16-word array chaining variable $\mathsf{CV}^{(0)}$ is initialized to the initialization vector IV of LSH-8w-n, shown in Section 2.4, i.e., $\mathsf{CV}^{(0)} \leftarrow \mathsf{IV}$.

Compression In this stage, the t 32-word array message blocks $\{\mathsf{M}^{(i)}\}_{i=0}^{t-1}$, which are generated from a message m, are compressed by iteration of compression functions. The compression function $\textsc{Cf} : \mathcal{W}^{16} \times \mathcal{W}^{32} \to \mathcal{W}^{16}$ has two inputs; the i-th 16-word chaining variable $\mathsf{CV}^{(i)}$ and the i-th 32-word message block $\mathsf{M}^{(i)}$, and returns the $(i+1)$-th 16-word chaining variable $\mathsf{CV}^{(i+1)}$. For the detail process of a compression function $\textsc{Cf}$, see Section 2.3.

Finalization The finalization function $\textsc{Fin}_n$ return n-bit hash value h from the final chaining variable $\mathsf{CV}^{(t)} = (CV^{(t)}[0], \ldots, CV^{(t)}[15])$. Let $\mathsf{h} = (h[0], \ldots, h[w-1])$ be an w-byte array. $\textsc{Fin}_n$ proceeds as (4).

$$\begin{aligned} &\mathsf{h} \leftarrow (CV^{(t)}[0] \oplus CV^{(t)}[8],\ CV^{(t)}[1] \oplus CV^{(t)}[9],\ \ldots, CV^{(t)}[7] \oplus CV^{(t)}[15]), \\ &h \leftarrow (h[0] \parallel \cdots \parallel h[w-1])_{[0:n-1]}. \end{aligned} \tag{4}$$

Algorithm 1 shows the message hashing process of LSH-8w-n.

2.3 Compression function

The i-th 16-word chaining variable $\mathsf{CV}^{(i)}$ and the i-th 32-word message block $\mathsf{M}^{(i)}$ are inputs of a compression function $\textsc{Cf} : \mathcal{W}^{16} \times \mathcal{W}^{32} \to \mathcal{W}^{16}$. The following four functions are used in a compression function:

1. $\textsc{MsgExp} : \mathcal{W}^{32} \to \mathcal{W}^{16(\mathrm{N}_s+1)}$ (Message expansion function),
2. $\textsc{MsgAdd} : \mathcal{W}^{16} \times \mathcal{W}^{16} \to \mathcal{W}^{16}$ (Message addition function),
3. $\textsc{Mix}_j : \mathcal{W}^{16} \to \mathcal{W}^{16}$ (Mix function),
4. $\textsc{WordPerm} : \mathcal{W}^{16} \to \mathcal{W}^{16}$ (Word-permutation function),

where the number N_s is defined in (5) and $0 \le j < \mathrm{N}_s$.

$$\mathrm{N}_s := \begin{cases} 26, & \text{if LSH-256-}n, \\ 28, & \text{if LSH-512-}n. \end{cases} \tag{5}$$

Algorithm 1 Hash function LSH-$8w$-n

Input: Bit string message m

Output: n-bit hash value h of m

1: One-zeros padding of m
2: Generation of t message blocks $\{\mathsf{M}^{(i)}\}_{i=0}^{t-1}$, where $t = \left\lceil \frac{|m|+1}{32w} \right\rceil$ from the padded bit string
3: $\mathsf{CV}^{(0)} \leftarrow \mathsf{IV}$
4: **for** $i = 0$ to $t-1$ **do**
5: $\quad \mathsf{CV}^{(i+1)} \leftarrow \textsc{Cf}(\mathsf{CV}^{(i)}, \mathsf{M}^{(i)})$
6: **end for**
7: $h \leftarrow \textsc{Fin}_n(\mathsf{CV}^{(t)})$
8: **return** h

Here we define the j-th step function $\textsc{Step}_j : \mathcal{W}^{16} \times \mathcal{W}^{16} \to \mathcal{W}^{16}$ by (6).

$$\textsc{Step}_j := \textsc{WordPerm} \circ \textsc{Mix}_j \circ \textsc{MsgAdd}. \tag{6}$$

In a compression function, the message expansion function MsgExp generates $\mathrm{N}_s + 1$ 16-word array sub-messages $\{\mathsf{M}_j^{(i)}\}_{j=0}^{\mathrm{N}_s}$ from given $\mathsf{M}^{(i)}$. Let $\mathsf{T} = (T[0], \ldots, T[15])$ be a temporary 16-word array set to the i-th chaining variable $\mathsf{CV}^{(i)}$. The j-th step function $\textsc{Step}_j$ having two inputs T and $\mathsf{M}_j^{(i)}$ updates T, i.e., $\mathsf{T} \leftarrow \textsc{Step}_j(\mathsf{T}, \mathsf{M}_j^{(i)})$. All step functions are proceeded in order $j = 0, \ldots, \mathrm{N}_s - 1$. Then one more MsgAdd operation by $\mathsf{M}_{\mathrm{N}_s}^{(i)}$ is proceeded, and the $(i+1)$-th chaining variable $\mathsf{CV}^{(i+1)}$ is set to T. Algorithm 2 shows the process of a compression function in detail.

Message expansion function Let $\mathsf{M}^{(i)} = (M^{(i)}[0], \ldots, M^{(i)}[31])$ be the i-th 32-word array message block. The message expansion function MsgExp generates $\mathrm{N}_s + 1$ 16-word array sub-messages $\{\mathsf{M}_j^{(i)}\}_{j=0}^{\mathrm{N}_s}$ from a message block $\mathsf{M}^{(i)}$. The first two sub-messages $\mathsf{M}_0^{(i)} = (M_0^{(i)}[0], \ldots, M_0^{(i)}[15])$ and $\mathsf{M}_1^{(i)} = (M_1^{(i)}[0], \ldots, M_1^{(i)}[15])$ are defined by (7).

$$\mathsf{M}_0^{(i)} \leftarrow (M^{(i)}[0], \ldots, M^{(i)}[15]), \quad \mathsf{M}_1^{(i)} \leftarrow (M^{(i)}[16], \ldots, M^{(i)}[31]). \tag{7}$$

The next sub-messages $\{\mathsf{M}_j^{(i)} = (M_j^{(i)}[0], \ldots, M_j^{(i)}[15])\}_{j=2}^{\mathrm{N}_s}$ are generated by (8).

$$\mathsf{M}_j^{(i)}[l] \leftarrow \mathsf{M}_{j-1}^{(i)}[l] \boxplus \mathsf{M}_{j-2}^{(i)}[\tau(l)], \text{ for } 0 \le l < 16, \tag{8}$$

where τ is the permutation over $\mathbb{Z}_{16}$ defined by Table 3.

Algorithm 2 Compression function Cf

Input: The i-th chaining variable $\mathsf{CV}^{(i)}$ and the i-th message block $\mathsf{M}^{(i)}$
Output: The $(i+1)$-th chaining variable $\mathsf{CV}^{(i+1)}$

1: $\{\mathsf{M}_j^{(i)}\}_{j=0}^{\mathrm{N}_s} \leftarrow \textsc{MsgExp}(\mathsf{M}^{(i)})$
2: $\mathsf{T} \leftarrow \mathsf{CV}^{(i)}$
3: **for** $j = 0$ to $\mathrm{N}_s - 1$ **do**
4: $\mathsf{T} \leftarrow \textsc{Step}_j(\mathsf{T}, \mathsf{M}_j^{(i)})$ $\begin{cases} \mathsf{T} \leftarrow \textsc{MsgAdd}(\mathsf{T}, \mathsf{M}_j^{(i)}) \\ \mathsf{T} \leftarrow \textsc{Mix}_j(\mathsf{T}) \\ \mathsf{T} \leftarrow \textsc{WordPerm}(\mathsf{T}) \end{cases}$
5: **end for**
6: $\mathsf{CV}^{(i+1)} \leftarrow \textsc{MsgAdd}(\mathsf{T}, \mathsf{M}_{\mathrm{N}_s}^{(i)})$
7: **return** $\mathsf{CV}^{(i+1)}$

Message addition function The message addition function $\textsc{MsgAdd} : \mathcal{W}^{16} \times \mathcal{W}^{16} \to \mathcal{W}^{16}$ is defined by (9): for two 16-word arrays $\mathsf{X} = (X[0], \ldots, X[15])$ and $\mathsf{Y} = (Y[0], \ldots, Y[15])$,

$$\textsc{MsgAdd}(\mathsf{X}, \mathsf{Y}) := (X[0] \oplus Y[0], \ldots, X[15] \oplus Y[15]). \tag{9}$$

Mix function The j-th mix function $\textsc{Mix}_j : \mathcal{W}^{16} \to \mathcal{W}^{16}$ updates the 16-word array $\mathsf{T} = (T[0], \ldots, T[15])$ by mixing every two-word pair; $T[l]$ and $T[l+8]$ for $0 \le l < 8$. For $0 \le j \le \mathrm{N}_s - 1$, the mix function $\textsc{Mix}_j$ proceeds (10).

$$(T[l], T[l+8]) \leftarrow \textsc{Mix}_{j,l}(T[l], T[l+8]), \text{ for } 0 \le l < 8, \tag{10}$$

where $\textsc{Mix}_{j,l}$ is a two-word mix function. Let X and Y be words. The two-word mix function $\textsc{Mix}_{j,l} : \mathcal{W}^2 \to \mathcal{W}^2$ is defined by Fig. 1 and (11). Here the bit rotational amounts $\alpha_j, \beta_j, \gamma_l$ used in $\textsc{Mix}_{j,l}$ are shown in Table 2.

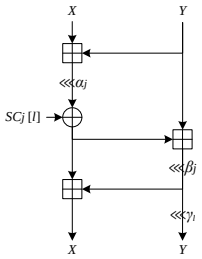


Fig. 1: Two-word mix function $\textsc{Mix}_{j,l}(X, Y)$

$$\begin{aligned} X &\leftarrow X \boxplus Y, \\ X &\leftarrow X^{\lll \alpha_j}, \\ X &\leftarrow X \oplus SC_j[l], \\ Y &\leftarrow X \boxplus Y, \\ Y &\leftarrow Y^{\lll \beta_j}, \\ X &\leftarrow X \boxplus Y, \\ Y &\leftarrow Y^{\lll \gamma_l}. \end{aligned} \tag{11}$$

Table 2: Bit rotation amounts: α_j, β_j and γ_l

Algorithm	j	α_j	β_j	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7
LSH-256-n	even	29	1	0	8	16	24	24	16	8	0
	odd	5	17								
LSH-512-n	even	23	59	0	16	32	48	8	24	40	56
	odd	7	3								

The j-th 8-word array constant $\mathsf{SC}_j = (SC_j[0], \ldots, SC_j[7])\}$ used in $\textsc{Mix}_{j,l}$ for $0 \le l < 8$, is defined as follows: The initial 8-word array constant $\mathsf{SC}_0 = (SC_0[0], \ldots, SC_0[7])$ of LSH-256-n and LSH-512-n are defined by (12) and (13). Let $\delta = 768372$, where 76, 83, 72 are ASCII codes of 'L,' 'S,' and 'H' respectively.

– LSH-256-n: These are the first 256-bit of the fractional parts of $\sqrt{\delta}$.

$$\begin{array}{ll} SC_0[0] = \texttt{917caf90}, & SC_0[1] = \texttt{6c1b10a2}, \\ SC_0[2] = \texttt{6f352943}, & SC_0[3] = \texttt{cf778243}, \\ SC_0[4] = \texttt{2ceb7472}, & SC_0[5] = \texttt{29e96ff2}, \\ SC_0[6] = \texttt{8a9ba428}, & SC_0[7] = \texttt{2eeb2642}. \end{array} \tag{12}$$

– LSH-512-n: These are the first 512-bit of the fractional parts of $\sqrt[3]{\delta}$.

$$\begin{array}{ll} SC_0[0] = \texttt{97884283c938982a}, & SC_0[1] = \texttt{ba1fca93533e2355}, \\ SC_0[2] = \texttt{c519a2e87aeb1c03}, & SC_0[3] = \texttt{9a0fc95462af17b1}, \\ SC_0[4] = \texttt{fc3dda8ab019a82b}, & SC_0[5] = \texttt{02825d079a895407}, \\ SC_0[6] = \texttt{79f2d0a7ee06a6f7}, & SC_0[7] = \texttt{d76d15eed9fdf5fe}. \end{array} \tag{13}$$

For $1 \le j \le \mathrm{N}_s - 1$, the j-th constant $\mathsf{SC}_j = (SC_j[0], \ldots, SC_j[7])$ is generated by (14).

$$SC_j[l] \leftarrow SC_{j-1}[l] \boxplus SC_{j-1}[l]^{\lll 8}, \text{ for } 0 \le l < 8. \tag{14}$$

Word-permutation function Let $\mathsf{X} = (X[0], \ldots, X[15])$ be an 16-word array. The word-permutation function $\textsc{WordPerm} : \mathcal{W}^{16} \to \mathcal{W}^{16}$ is defined by (15).

$$\textsc{WordPerm}(\mathsf{X}) := (X[\sigma(0)], \ldots, X[\sigma(15)]), \tag{15}$$

where σ is the permutation over $\mathbb{Z}_{16}$ defined by Table 3.

Fig. 2 shows the j-th step function $\textsc{Step}_j$ of a compression function.

2.4 Initialization vector generation

The initialization vector $\mathsf{IV} \in \mathcal{W}^{16}$ of LSH-8w-n is defined by $\textsc{Cf}(\mathsf{X}, \mathsf{Y})$, where $\mathsf{X} \in \mathcal{W}^{16}$ such that $X[0] = \textsc{IntToWord}(w)$, $X[1] = \textsc{IntToWord}(n)$, and $X[l]$ are all zero words for $2 \le l \le 15$, and $\mathsf{Y} \in \mathcal{W}^{32}$ such that all $Y[l]$ are all zero words. The initialization vector of LSH-256-256 and LSH-512-512 are (16) and (17) respectively.

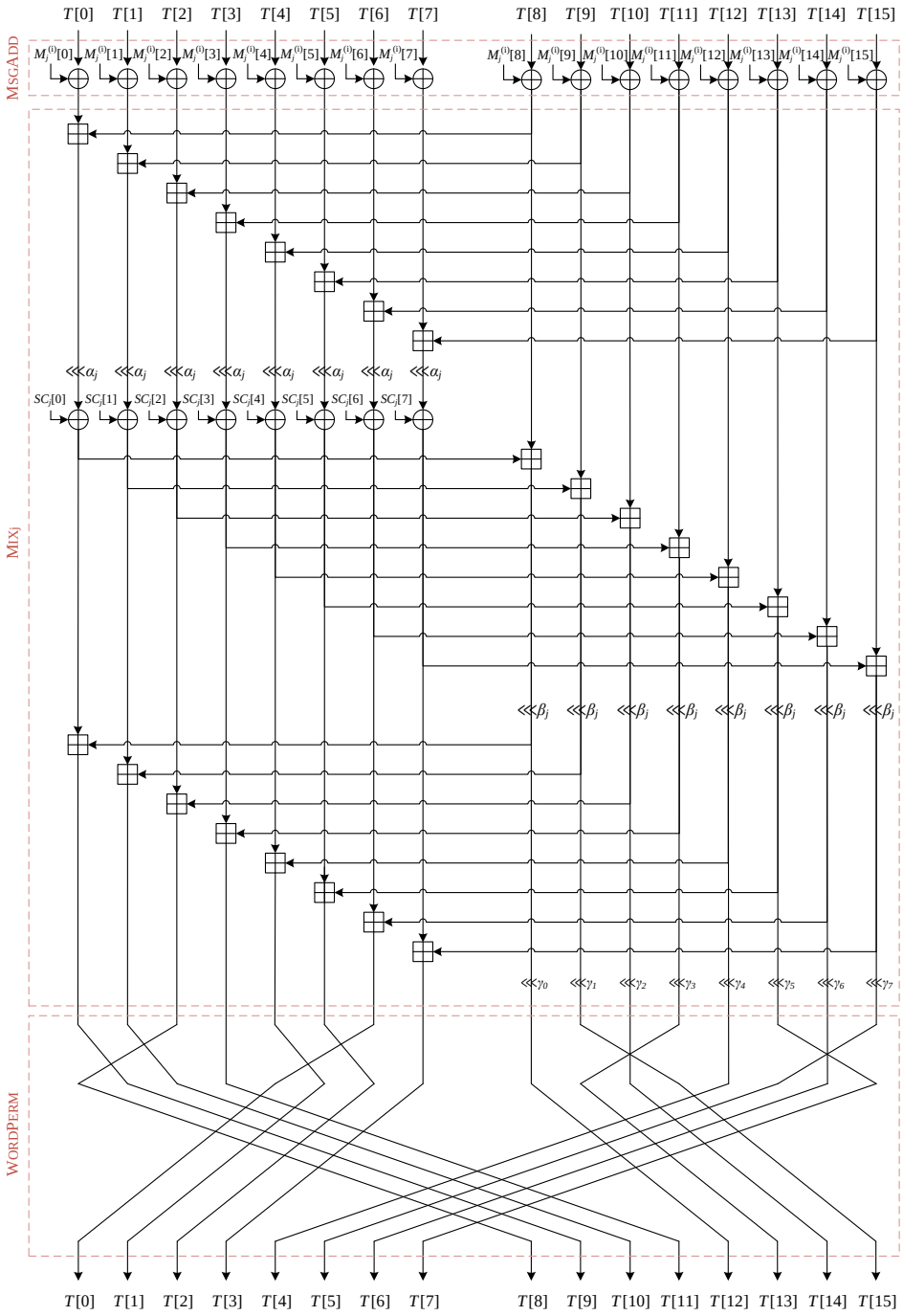


Fig. 2: The j-th step function STEP_j

Table 3: The permutation $\tau, \sigma : \mathbb{Z}_{16} \to \mathbb{Z}_{16}$

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

– LSH-256-256

$$\begin{aligned} IV[0] &= \texttt{46a10f1f}, & IV[1] &= \texttt{fddce486}, \\ IV[2] &= \texttt{b41443a8}, & IV[3] &= \texttt{198e6b9d}, \\ IV[4] &= \texttt{3304388d}, & IV[5] &= \texttt{b0f5a3c7}, \\ IV[6] &= \texttt{b36061c4}, & IV[7] &= \texttt{7adbd553}, \\ IV[8] &= \texttt{105d5378}, & IV[9] &= \texttt{2f74de54}, \\ IV[10] &= \texttt{5c2f2d95}, & IV[11] &= \texttt{f2553fbe}, \\ IV[12] &= \texttt{8051357a}, & IV[13] &= \texttt{138668c8}, \\ IV[14] &= \texttt{47aa4484}, & IV[15] &= \texttt{e01afb41}. \end{aligned} \tag{16}$$

– LSH-512-512

$$\begin{aligned} IV[0] &= \texttt{add50f3c7f07094e}, & IV[1] &= \texttt{e3f3cee8f9418a4f}, \\ IV[2] &= \texttt{b527ecde5b3d0ae9}, & IV[3] &= \texttt{2ef6dec68076f501}, \\ IV[4] &= \texttt{8cb994cae5aca216}, & IV[5] &= \texttt{fbb9eae4bba48cc7}, \\ IV[6] &= \texttt{650a526174725fea}, & IV[7] &= \texttt{1f9a61a73f8d8085}, \\ IV[8] &= \texttt{b6607378173b539b}, & IV[9] &= \texttt{1bc99853b0c0b9ed}, \\ IV[10] &= \texttt{df727fc19b182d47}, & IV[11] &= \texttt{dbef360cf893a457}, \\ IV[12] &= \texttt{4981f5e570147e80}, & IV[13] &= \texttt{d00c4490ca7d3e30}, \\ IV[14] &= \texttt{5d73940c0e4ae1ec}, & IV[15] &= \texttt{894085e2edb2d819}. \end{aligned} \tag{17}$$

3 Design Rationale

3.1 Hash structure

The hash structure of LSH-8w-n is wide-pipe Merkle-Damgård mode, and the compression function is designed on the 17^{th} PGV model [18] which has no feed-forward computation of input. Input feed-forward structure is not efficient in terms of memory use since input value should be reserved till the end of all round or step function computations in a compression function. So we use the memory resource to expand the length of a message block. Even though LSH-8w-n uses the 17^{th} PGV model, it is easily proven that the structure has 2^n preimage and second-preimage resistance, and $2^{n/2}$ collision resistance under the ideal cipher model proof thanks to wide-pipe mode [18,24]. See Section 4.1 for details. LSH-8w-n uses one-zeros padding for implementation efficiency because any padding can be applied to wide-pipe mode [22].

3.2 Compression function

Bit length of a message block of LSH-8w-n is $32w$, whereas $16w$ is that of a chaining variable. The compression function of LSH-8w-n is designed for paral-

lel implementation paying attention to efficient memory resource usage as well as fast software performance, even in short length message case. That is, all functions of a compression function are chosen to implement with 128/256-bit register SIMD(Single Instruction Multiple Data) instructions readily.

Message expansion The message expansion function MsgExp generates the j-th sub-message $\mathsf{M}_j^{(i)}$ with two sub-messages $\mathsf{M}_{j-1}^{(i)}$ and $\mathsf{M}_{j-2}^{(i)}$. This structure is good for efficient memory usage implementation. MsgExp has every 4-word array updating structure. The permutation τ of MsgExp is chosen to have efficient implementation using SIMD instructions.

Mix function As shown in Fig. 2, the j-th mix function $\textsc{Mix}_j$ of the j-th step function $\textsc{Step}_j$ is designed by 8-word parallel operations except the γ_l-bit rotations($0 \leq l < 8$) operation. In order to strengthen security of LSH-8w-n, we needed all different gammas. By the way, all different bit-rotations operation is inefficient in the implementation using SIMD instructions, only except the case that γ_ls are multiples of 8. In that case, the implementation using SIMD instructions get efficient because they support byte shuffle instructions, such as `pshufb`(in SSSE3), `vpshufb`(in AVX2) and `vext.8`(in NEON). Therefore, the γ_ls were searched under the following two conditions:

1. They should be multiples of 8.
2. They should minimize the iterative difference pattern probability derived from the structural issue without reference to α_j and β_j.

After decision of γ_ls satisfying the above, the α_j and β_j are chosen from the group of candidates which minimizes difference probability. See Section 4.2 for details. The exclusive-or with a constant is added to have resistance against rotational attack [28, 34].

Word-permutation function The word-permutation function WordPerm is chosen to satisfy the following feature:

1. Easy to implement using SSE/AVX2/NEON-instructions:
   – First, the permutation σ used in WordPerm permutes every 4-word arrays, and then re-arranges the position of the four 4-word arrays. See Fig. 2. Since word-shuffle instructions are supported in SSE, AVX2 and NEON, WordPerm can be efficiently implemented with low latency. See Section 5.1 for details.
2. The fastest propagation of a word change:
   – Any word value $T[l]$ of the input $\mathsf{T} = (T[0], \ldots, T[15])$ of a step function affects all words of T after five step function processes. $T[l]$ acts on all words after four step function operations for $l \in \{0, 1, 4, 5, 8, 9, 12, 13\}$, while the other ls need five step functions.

One more message addition We need the final message addition process in a compression function because there is no input feed-forward computation. If this process does not exist, the last step function is meaningless.

Number of step functions The number of step functions is determined such that the compression function guarantees at least 50% security margin. More precisely, our security analysis shows that 13 steps of LSH-256-256 and 14 steps of LSH-512-512 are sufficient to defeat all the existing attack methods. As a result, we determined the number of steps as 26 for LSH-256-256 and 28 for LSH-512-512, respectively, by doubling the estimated safe boundaries.

4 Security analysis

We study the security of LSH. Firstly, we introduce secure bounds of number of queries for LSH structure by referring previous research results in ideal setting. Then, we explain how we analyzed security of LSH against hash function attacks. Finally, we also how to translate the distinguishers used for block cipher attacks.

4.1 Security in the ideal cipher model

The compression function of LSH is a kind of permutation family because it is a permutation on a chaining variable for a fixed message block. So, we can model it as an ideal cipher, i.e., it is uniformly chosen from the set of all block ciphers. For $n = 8w$, we denote LSH-$8w$-n by $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ and consider the compression function CF as the 17^{th} PGV scheme which is $\mathrm{CF}(\mathsf{CV}^{(i)}, \mathsf{M}^{(i)}) = E(\mathsf{CV}^{(i)}, \mathsf{M}^{(i)})$, where E is a permutation family. We define Bloc to be the set of all block ciphers with block bit length $2n(= 16w)$ and key bit length $4n(= 32w)$. In the ideal cipher model, we assume that an adversary $\mathcal{A}$ is a probabilistic algorithm which has oracle access to a random cipher $E \xleftarrow{\$} \mathsf{Bloc}$. The advantages of $\mathcal{A}$ finding a collision, a preimage, and a second-preimage are defined as (18)-(20), respectively.

$$\mathcal{A}^{\mathrm{coll}}_{\mathcal{H}}(\mathcal{A}) = \Pr[E \xleftarrow{\$} \mathsf{Bloc}, (M, M') \leftarrow \mathcal{A}^E : M \neq M' \wedge \mathcal{H}(M) = \mathcal{H}(M')], \tag{18}$$

$$\mathcal{A}^{\mathrm{epre}}_{\mathcal{H}}(\mathcal{A}) = \max_{h \in \{0,1\}^n} \Pr[E \xleftarrow{\$} \mathsf{Bloc}, M \leftarrow \mathcal{A}^E(h) : \mathcal{H}(M) = h], \tag{19}$$

$$\mathcal{A}^{\mathrm{esec}[\lambda]}_{\mathcal{H}}(\mathcal{A}) = \max_{M \in \{0,1\}^\lambda} \Pr[E \xleftarrow{\$} \mathsf{Bloc}, M' \leftarrow \mathcal{A}^E(M) : M \neq M' \wedge \mathcal{H}(M) = \mathcal{H}(M')], \tag{20}$$

where λ is the bit length of the target message.

In order to analyze preimage security, we consider the notion of everywhere preimage resistance, which guarantees security on every range point; for the second-preimage security analysis, we consider the notion of everywhere second-preimage resistance, which guarantees security on every domain point [49]. For the above advantages, we define the maximum advantages of any adversary making q queries by $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{H}}(q)$, $\mathrm{Adv}^{\mathrm{epre}}_{\mathcal{H}}(q)$, and $\mathrm{Adv}^{\mathrm{esec}[\lambda]}_{\mathcal{H}}(q)$, respectively.

chopMD is the Merkle-Damgård hash function with a chop function which returns a hash value by truncating a part of the output of the last compression function [22]. It is easily proved that LSH-8w is as secure as chopMD with the 17^{th} PGV compression function in the ideal cipher model and has full security bounds for collision resistance, preimage resistance, and second-preimage resistance, from the previous results of [18, 24]. Lemma 1 summarizes the security analysis of LSH-8w in the ideal cipher model.

Lemma 1. *Let* $L = \lceil \lambda/4n \rceil$. *Then we have the followings: (i)* $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{H}} \leq \frac{q(q+1)}{2^n}$, *(ii)* $\mathrm{Adv}^{\mathrm{epre}}_{\mathcal{H}} \leq \frac{q(q+1)}{2^{2n}} + \frac{q}{2^n}$, *(iii)* $\mathrm{Adv}^{\mathrm{esec}[\lambda]}_{\mathcal{H}} \leq \frac{q(q+L)}{2^{2n}} + \frac{q}{2^n}$.

Lemma 1 implies that LSH-8w is collision-resistant for $q < 2^{n/2}$ and preimage-resistant and second-preimage-resistant for $q < 2^n$ in the ideal cipher model. Roughly, it means that LSH-8w has no weakness as long as its internal permutation family is not attacked. By [43], we can also show that LSH-8w is indifferentiable from a random oracle.

4.2 Collision Security

A colliding message pair makes a differential path. Our concern is focused on how difficult any adversary finds good differential paths. We use differential cryptanalysis [16] used for block ciphers assuming all the modular additions are independent.

Framework of collision security analysis We translate the collision search as finding a message pair making zero output difference for any fixed input difference of the compression function, and extend it to the problem of finding a message pair making an intended and fixed output difference for any fixed input difference of the compression function. So, our concern is moved to high-probability differential characteristics for the compression function. For this, we linearize the compression function by replacing the additions with XORs, search low-weight paths for the linearized structure, and evaluate the probabilities by assuming all additions are independent. Of course, this approach can cause invalid differential characteristics which do not hold, but we prefer finding high-probability characteristics to valid ones.

We consider the threshold of the probability as 2^{-48w}. Note that $48w$ is the bit length of the chaining variable plus the bit length of the message block. The reason why we adopt this threshold is the message-modification paradigm, in which the attacker uses freedom degree of input (chaining variable and message block) for achieving his goal. It is usually applied to collision-finding attack on dedicated hash functions [54, 55]. The basic idea of message modification technique is to control the values of the message blocks instead of choosing a pair of message blocks uniformly at random. A one-bit condition in the differential characteristic can be satisfied probabilistically, or by a message modification which requires at least one bit of a message or a chaining variable to be fixed.

Based on this reasoning, we can make an argument that message modification techniques will be hardly successful if the number of conditions in a differential characteristic significantly exceeds the number of input bits. For LSH-8w, the number of input bits is $48w (= 16w(\text{chaining variable}) + 32w(\text{message block}))$.

Differential characteristic Considering the message modification technique, the remaining work is to examine the differential characteristics. It is very difficult to find the best differential characteristic for ARX structure or to estimate a tight upper bound of the probabilities of differential characteristics. Alternatively, we simplify the problem by linearizing the compression function of LSH-8w such that every modular addition is replaced with exclusive-or.

We tried various strategies and techniques from coding theory for characteristic search. After linearization, the compression function can be considered as a linear code. We can regard a characteristic yielding a collision as a shortening codeword, and a high-probability characteristic is mostly a low-weight codeword. There are a few probabilistic low-weight codeword searching algorithms [21, 48], but it is still difficult to find a good linearized differential path since finding a minimum-weight codeword is known as one of NP-complete problems.

The probabilities of the characteristics are computed with Lipmaa-Moriai formula [42] by assuming that all additions in step functions and message expansion are independent. The hypothesis of the independence between additions can lead to impossible characteristics or misleading probability estimation. Leurent indicated a similar problem for the differential analysis of MD5 and SHA-1/2 and presented the multi-bit constraint technique for improving the differential characteristic search in ARX structures [40]. At the very least, however, a high-weight codeword is seldom a high-probability characteristic, and we are devoted to finding a codeword with as low weight as possible.

The best ones which we found are 12-step characteristic with the probability 2^{-1340} for LSH-256 and 13-step characteristic with the probability 2^{-2562} for LSH-512. Table 4 shows each differential probability of LSH-256 and LSH-512. They are found by starting at low-weight intermediate values of an internal state and sub-messages and computing them in forward and backward directions with the linearized form.

Table 4: Linearized differential characteristics with lowest weights. The probabilities are calculated with Lipmaa-Moriai formula assuming the independency of modular additions.

Number of steps	LSH-256	LSH-512
12	2^{-1340}	2^{-1655}
13	2^{-1858}	2^{-2562}
14	2^{-2396}	2^{-3538}

Iterative characteristic Our characteristic search methods do not find any iterative patterns. So, we take a different approach for iterative differential characteristics.

Lemma 2. *In the linearized setting, we have the followings:* (*i*) *The order of* MsgExp *of LSH-8w divides 12.* (*ii*) *For any positive integer k,* $(3k+1)$ *or* $(3k+2)$*-step iterative differential characteristics require zero message difference.*

Lemma 2 implies that 3, 6, 9, or 12 can be considered as the iteration number. Start at the $2j$-th step function. For 3-step iterative differential characteristic, we obtain the following input difference $(\Delta\mathsf{T}, \Delta\mathsf{M}_{2j}||\Delta\mathsf{M}_{2j+1})$ by making a system of corresponding linear equations and solving it with some conditions:

$$\begin{aligned}
\Delta\mathsf{T} &= (0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0),\\
\Delta\mathsf{M}_{2j} &= (0,\ 0,\ 0,\ 0,\ A,\ A,\ A,\ A,\ 0,\ 0,\ 0,\ 0,\ A,\ A,\ A,\ A),\\
\Delta\mathsf{M}_{2j+1} &= (A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A,\ A),
\end{aligned}$$

where $A \in \mathcal{W}$ such that $A = A^{\lll \alpha_{2j}+\beta_{2j+1}} = A^{\lll \alpha_{2j+1}+\beta_{2j}}$. For convenience, let $(\alpha_{2j}, \beta_{2j}, \alpha_{2j+1}, \beta_{2j+1})$ denote by $(\alpha_0, \beta_0, \alpha_1, \beta_1)$. Since this 3-step iterative differential characteristic starts from zero $\Delta\mathsf{T}$ and ends with a bilateral-symmetric output difference, it can be straightforwardly used as a collision path on step-reduced LSH-8w. However, its probability for LSH-8w is too low to be available for long steps. The condition for A shows how the rotation amounts $(\alpha_0, \beta_0, \alpha_1, \beta_1)$ affect the iterative differential characteristic. For $g = \gcd(w, \alpha_0 + \beta_1, \alpha_1 + \beta_0)$, A must be composed of w/g repetition of least significant g bits. For example, if $(\alpha_0, \beta_0, \alpha_1, \beta_1) = (5, 23, 25, 19)$, then $\alpha_0 + \beta_1 = 24$ and $\alpha_1 + \beta_0 = 48$, and we could take $A =$ `80...80`. For the current version of $(\alpha_0, \beta_0, \alpha_1, \beta_1)$, A should equal either `aa...aa` or `55...55` which produce high-weight codewords. Obviously, the former makes characteristics with much higher probabilities than the latter as well as ones in Table 4.

Differential characteristic with zero message difference As mentioned in Section 4.2, our main concern for the analysis of collision security is a differential characteristic with non-zero message difference. However, in the sense of algorithm design, it is desirable that differential characteristics with zero message difference should also have low probabilities.

Our characteristic search methods did not find any good zero-message-difference characteristic with significantly high probability. However, we found such one by solving a system of equations for an iterative differential characteristic.

Let $\mathsf{X} = (X[0], \ldots, X[15])$ be the difference of the input chaining variable. For 1-step iterative differential characteristic with zero message difference, we obtain the following conditions for X.

$$\begin{aligned}
&X[l] = X[l]^{\lll \alpha_j}, \quad X[l] = X[l]^{\lll \beta_j} \text{ for all } l \in \{0, 1, ..., 15\}, j \in \{0, 1\},\\
&X[0] = X[1] = X[2] = X[3] = X[12] = X[13] = X[14] = X[15],\\
&X[8] = X[9] = X[10], \quad X[5] = X[7].
\end{aligned}$$

Since α_0, α_1, β_0, and β_1 are odd numbers, each $X[l]$ equals either `00...00` or `ff...ff`. The lowest differential characteristics are constructed by setting one of $X[4]$, $X[6]$, and $X[11]$ to `ff...ff` and each of the other 15 words to `00...00`.

We also considered a pseudo-iterative differential characteristic with zero message difference, and its input difference is as follows.

$$X[l] = \begin{cases} A^{\lll \gamma_1}, & \text{if } l \in \{3, 7, 11, 15\}, \\ A, & \text{otherwise.} \end{cases}$$

where $A \in \mathcal{W}$ such that $A = A^{\lll \gamma_0} = A^{\lll \gamma_2} = A^{\lll \gamma_1+\gamma_3}$ and $A^{\lll \gamma_j} = A^{\lll \gamma_{j+4}}$ for $j \in \{0, 1, 2, 3\}$. After k steps, the corresponding output difference Y is computed as $Y[l] = X[l]^{\lll \sum_j \beta_j+\gamma_1}$ for $l \in \{3, 7, 11, 15\}$ and $Y[l] = X[l]^{\lll \sum_j \beta_j}$ for $l \notin \{3, 7, 11, 15\}$. The condition of A shows how the rotation amounts $(\gamma_0, \gamma_1, \ldots, \gamma_7)$ affect the pseudo-iterative differential characteristic. We define g by $g = \gcd(w, \gamma_0, \gamma_2, \gamma_1 + \gamma_3, \gamma_0 - \gamma_4, \gamma_1 - \gamma_5, \gamma_2 - \gamma_6, \gamma_3 - \gamma_7)$, where w is the bit length of a word. A must be composed of w/g repetition of least significant g bits. For current version of $(\gamma_0, \gamma_1, \ldots, \gamma_7)$, we have $g = 8$.

We denote the 1-step iterative differential characteristic by ϕ and the pseudo-iterative differential characteristic by ψ. Table 5 lists the best probabilities of ϕ and ψ. It shows that for same steps ϕ and ψ have much higher probabilities than the differential characteristics with non-zero message difference in Table 4. However, we note that no technique is known for using them to make a hash function attack for LSH-8w.

Table 5: The best probabilities of differential characteristics ϕ and ψ. The probabilities are calculated with Lipmaa-Moriai formula assuming the independency of modular additions.

Number of steps	LSH-256		LSH-512	
	ϕ	ψ	ϕ	ψ
1	2^{-93}	2^{-80}	2^{-189}	2^{-176}
2	2^{-186}	2^{-176}	2^{-378}	2^{-368}
3	2^{-279}	2^{-272}	2^{-567}	2^{-560}
4	2^{-372}	2^{-368}	2^{-756}	2^{-752}
5	2^{-465}	2^{-464}	2^{-945}	2^{-944}
6	2^{-558}	2^{-560}	2^{-1134}	2^{-1136}
7	2^{-651}	2^{-656}	2^{-1323}	2^{-1328}
8	2^{-744}	2^{-744}	2^{-1512}	2^{-1512}
9	2^{-837}	2^{-824}	2^{-1701}	2^{-1688}
10	2^{-930}	2^{-920}	2^{-1890}	2^{-1880}

Nonlinear differential characteristic We also tried to find nonlinear differential characteristics with higher probabilities than linear ones. We can obtain them by optimizing the linearized differential characteristics. We used various techniques for characteristic optimization. For example, we have used a SAT solver. Recently, Mouha and Preneel presented a proof about the security of Salsa20 against differential cryptanalysis [47]. Since the large internal state of the compression function of LSH-8w greatly complicates SAT-solver-based analysis, we had to need some additional assumptions.

Again, the goal of the analysis is to measure the minimum number of step functions making the compression function of LSH-8w secure against a collision attack. However, in spite of hard works for characteristic optimization, the improvement of the probability is not significant such that there is no effect on the result of the secure steps in Table 4. In [40], Leurent presented the multi-bit constraint technique which is useful for searching more plausible characteristics than linearized ones. In [41], he used it to find differential characteristics for Skein [25]. For LSH-8w, search with the multi-bit constraints finds only the differential characteristics with significantly lower probabilities than ones in Tables 4 and 5.

4.3 (Second-)Preimage attacks

In the (second-)preimage attack, the adversary has to find a proper message block mapping the fixed input chaining variable to the fixed hash value. Firstly, we check possibility of the meet-in-the-middle attack. It is trivially applied up to two steps. It can be partially applied to eight $\text{MIX}_{j,l}$ functions of three steps (two in the first step; two in the second step; four in the third step), and requires roughly computational complexity of 2^{12w}. The meet-in-the-middle preimage attack requires large complexity close to 2^{16w} for more than three steps, while the complexity of a typical brute force attack is 2^n for LSH-8w-n. Aoki-Sasaki's meet-in-the-middle attack framework [10, 50] consists of constructing a pseudo-preimage-finding algorithm and converting it to a preimage-finding algorithm [46, Fact 9.99]. We can not use the framework because we can find trivially a pseudo-preimage of LSH but it is not helpful for finding a preimage at all.

Biclique technique is often used for preimage attack [19,36,37]. We can make a biclique of dimension w for 6 steps. However, we have two problems to apply biclique technique to preimage attack on LSH. One is that existing biclique construction methods do not cause a particular output of the compression function of LSH. We need an advanced technique related to the finalization function FIN_n for a proper biclique construction. The other one is that a biclique-based preimage attack requires large complexity near 2^{16w}.

Kelsey-Schneier's generic attack [33] for finding second-preimages is not applicable to wide-pipe structure of LSH because its complexity is not less than that of brute-force attack. We do not exclude possibility of message-modification technique in second-preimage attacks, within 12 steps of LSH-256 and 13 steps of LSH-512.

4.4 Distinguishers and Other Attacks

We investigated distinguishers for the compression function of LSH, as well as differential characteristics.

- We can construct 16-step and 17-step boomerang distinguishers [53] for LSH-256 and LSH-512, respectively, by combining short differential characteristics. Some advanced combination techniques [17, 23] may improve slightly boomerang distinguishers which we found.
- A linear approximation [44] exists up to 7 steps for LSH-256 and 8 steps for LSH-512. We consider the possibility to combine a short differential characteristic and a short linear approximation to construct a differential-linear approximation [15] up to 13 steps for LSH-256 and 15 steps for LSH-512. Multidimensional techniques [30] may slightly improve cryptanalyses based on linear approximations.
- A truncated differential characteristic [38] with the probability 1 exists up to 4 steps in forward direction and 5 steps in backward direction. We can combine them to construct 9-step impossible differential characteristics [14]. We also observe a similar property for linear approximations, and combine 9-step zero-correlation linear approximations [20].
- An integral characteristic [39] exists up to 7.5 steps in forward direction and 8 steps in backward direction. We can combine them to construct a 11-step known-key distinguisher.
- We simulated empirical tests to LSH, which Skein designers did to the block cipher component Threefish of Skein [25]. Specifically, we fix a 1-bit difference and insert it to the same position of the input chaining variable and the first sub-message used in the first step. We observe the biases of the output difference bits by testing 20 to 50 million pairs satisfying this form. The output difference bits with bias > 0.0001 are found up to 6 steps of LSH-256 and LSH-512, while such things are found up to 17 rounds of Threefish.

We can study block cipher attacks for the LSH compression function by regarding the input chaining variable, the output chaining variable, and the message block as the plaintext, the ciphertext, and the key. However, it is not so valuable to discuss high complexity attacks because LSH-8w-n aims to $2^{n/2}$ collision security and 2^n (second-)preimage security. Therefore, we consider only distinguishing and key-recovery attacks which require less than 2^{256} complexity for LSH-256 or 2^{512} complexity for LSH-512, as valid.

In conclusion, any valid distinguishing or key-recovery attacks have not been found for more than 13 steps of LSH-256 and 14 steps of LSH-512. A biclique attack [19] is the only key recovery attack which works for full steps of the compression function of LSH-8w, but it requires a large amount of data and its computational complexity is close to 2^{16w}. We do not think that it causes any weakness. Rotational [34] and rebound [45] cryptanalysis which have been popularly researched during SHA-3 competition are not applicable to LSH. A rotational attack [34] is essentially improper to LSH because of the step constants. So, rotational rebound attack is not applicable, either [35]. A typical rebound

attack [45] does not work well because LSH is based on ARX operations instead of S-boxes.

5 Software implementation

The n-bit hash function based on w-bit word, LSH-8w-n is designed for parallel implementation. SSE and AVX2 are SIMD instruction sets using 128/256-bit register, which are supported on Intel processors [31]. SSE instructions up to SSE4.1 are available on Intel CPUs since 2008, and AVX2 can be used on CPUs based on Haswell architecture which are released in June, 2013. XOP is an extension of the SSE instructions in the x86 and AMD64 instruction set for the Bulldozer processor core released on 2011 [7]. NEON is a SIMD instruction set using 128-bit register for the ARM Cortex-A series CPUs [4]. In this section, we present the implementations of LSH-8w-n using these SIMD instructions.

5.1 Parallelism

Let r-bit register $\mathcal{X}$ define an s-word array $(X[0], \ldots, X[s-1])$, where $X[l] \in \mathcal{W}$ for $0 \le l \le s-1$, $s = r/w$ and r is either 128 or 256, i.e., $\mathcal{X} := (X[0], \ldots, X[s-1])$.

Operations on registers Let $\mathcal{X} = (X[0], \ldots, X[s-1])$ and $\mathcal{Y} = (Y[0], \ldots, Y[s-1])$ be r-bit registers, and let ρ be a permutation over $\mathbb{Z}_s$. We define the following operations on registers:

- $\mathcal{X} \oplus \mathcal{Y} := (X[0] \oplus Y[0], \ldots, X[s-1] \oplus Y[s-1])$.
- $\mathcal{X} \boxplus \mathcal{Y} := (X[0] \boxplus Y[0], \ldots, X[s-1] \boxplus Y[s-1])$.
- $\mathcal{X}^{\lll i} := (X[0]^{\lll i}, \ldots, X[s-1]^{\lll i})$.
- $\mathcal{X}^{\lll i_0, i_1, \ldots, i_{s-1}} := (X[0]^{\lll i_0}, \ldots, X[s-1]^{\lll i_{s-1}})$.
- $\rho(\mathcal{X}) := (X[\rho(0)], \ldots, X[\rho(s-1)])$.

Implementation using SIMD instructions SSE, AVX2 and NEON have instructions for word-wise exclusive-or " $\oplus$ " and word-wise modular addition " $\boxplus$ " of two registers. The word-wise all same bit-rotation " $\lll i$" is supported only in XOP. The operation should be implemented using two "bit-shifts" and one "or" instructions in SSE, AVX2 and NEON.

NEON has the instruction for the different bit-rotation " $\lll i_0, i_1, \ldots, i_{s-1}$", but the others don't. Intel announced that they had a plan to release AVX-512 in 2015 which supports these bit-rotation operations [31]. If the rotational amounts are divided by 8, the operation can be implemented by byte shuffle instructions `pshufb` and `vpshufb`(latency 1 on Haswell) in SSE and AVX2, respectively [2,6], and byte extraction instruction `vext.8` in NEON. Since the rotational amounts $\gamma_0, \ldots, \gamma_7$ used in the mix function $\textsc{Mix}_j$ are multiples of 8, we use these instructions. Note that the instruction `pshufb` is supported in SSSE3 or above which are extensions of SSE3.

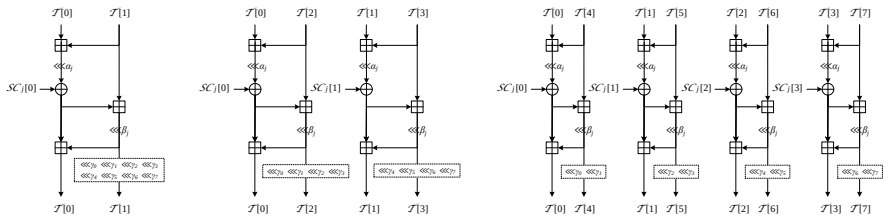
For the word-permutation ρ, SSE and AVX2 have more instructions(for example, `pshufd`, `vpshufd`, `vperm2i128`, `vpermq`) than NEON. However, the word-permutations, τ and σ used in LSH-8w-n are efficiently implemented by a few number of instructions like `vext.32` and `vrev64.32` and so on.

Mix function LSH-8w-n has a 16-word array chaining variable and a 32-word array message block. Each can be converted into a $2t$-register array and a $4t$-register array where $t = 8w/r$, respectively. Let $\mathsf{T} = (T[0], \ldots, T[15]) \in \mathcal{W}^{16}$ be the temporary variable used in the function MIX_j of the j-th step function and let $\mathsf{SC}_j = (SC_j[0], \ldots, SC_j[7]) \in \mathcal{W}^8$ be the constant of that. Let $\widehat{\mathcal{T}} = (\mathcal{T}[0], \ldots, \mathcal{T}[2t-1])$ and $\widehat{\mathcal{SC}_j} = (\mathcal{SC}_j[0], \ldots, \mathcal{SC}_j[t-1])$ be register arrays for T and SC_j defined by (21).

$$\begin{aligned} \mathcal{T}[l] &\leftarrow (T[2tl], T[2tl+1], \ldots, T[2t(l+1)-1]), \text{ for } 0 \le l < 2t, \\ \mathcal{SC}_j[l] &\leftarrow (SC_j[tl], SC_j[tl+1], \ldots, SC_j[t(l+1)-1]), \text{ for } 0 \le l < t. \end{aligned} \tag{21}$$

Then, (22) shows the process of the mix function MIX_j, and Fig. 3 is r-bit register representation of MIX_j depending on t. For $0 \le l < t$,

$$\begin{aligned} \mathcal{T}[l] &\leftarrow ((\mathcal{T}[l] \boxplus \mathcal{T}[l+t])^{\lll \alpha_j}) \oplus \mathcal{SC}_j[l], \\ \mathcal{T}[l+t] &\leftarrow (\mathcal{T}[l] \boxplus \mathcal{T}[l+t])^{\lll \beta_j}, \\ \mathcal{T}[l] &\leftarrow \mathcal{T}[l] \boxplus \mathcal{T}[l+t], \\ \mathcal{T}[l+t] &\leftarrow \mathcal{T}[l+t]^{\lll \gamma_{tl}, \ldots, \gamma_{t(l+1)-1}}. \end{aligned} \tag{22}$$



(a) $t = 1$ (b) $t = 2$ (c) $t = 4$

Fig. 3: r-bit register representation of MIX_j

As we mentioned, all operations of MIX_j can be efficiently implemented using NEON, SSE and AVX2 instructions.

5.2 Performance results on several platforms

Intel/AMD processors Table 6 shows the speed performance of 1MB message hashing of LSH-8w-n at the platforms based on Intel/AMD CPU. LSH-8w-n on

Table 6: 1MB message hashing speed on Intel/AMD/ARM processor (cycles/byte)

Platform	$\Theta 1$	$\Theta 2$	$\Theta 3$	$\Theta 4$	$\Lambda 1$	$\Lambda 2$	$\Lambda 3$	$\Lambda 4$
LSH-256-n	3.60	3.86	5.26	3.89	11.17	15.03	15.28	14.84
LSH-512-n	2.39	5.04	7.76	5.52	8.94	18.76	19.00	18.10

* $\Theta 1$: Intel Core i7-4770K @ 3.5GHz (Haswell), Ubuntu 12.04 64-bit, GCC 4.8.1 with "-m64 -mavx2 -O3"
* $\Theta 2$: Intel Core i7-2600K @ 3.40GHz (Sandy Bridge), Ubuntu 12.04 64-bit, GCC 4.8.1 with "-m64 -msse4 -O3"
* $\Theta 3$: Intel Core 2 Quad Q9550 @ 2.83GHz (Yorkfield), Windows 7 32-bit, Visual studio 2012
* $\Theta 4$: AMD FX-8350 @ 4GHz (Piledriver), Ubuntu 12.04 64-bit, GCC 4.8.1 with "-m64 -mxop -O3"
* $\Lambda 1$: Samsung Exynos 5250 ARM Cortex-A15 @ 1.7GHz dual core (Huins ACHRO 5250), Android 4.1.1
* $\Lambda 2$: Qualcomm Snapdragon 800 Krait 400 @ 2.26GHz quad core (LG G2), Android 4.4.2
* $\Lambda 3$: Qualcomm Snapdragon 800 Krait 400 @ 2.3GHz quad core (Samsung Galaxy S4), Android 4.2.2
* $\Lambda 4$: Qualcomm Snapdragon 400 Krait 300 @ 1.7GHz dual core (Samsung Galaxy S4 mini), Android 4.2.2

the platform $\Theta 1$ is implemented using AVX2 intrinsics. On the platform $\Theta 2$ and $\Theta 3$, LSH-8w-n is implemented using SSE4.1 instrinsic, and LSH-8w-n on the platform $\Theta 4$ is implemented using XOP intrinsic.

Recall that we compare the speed of LSH-8w-n with SHA-2 and SHA-3 competition finalists because they are matured in terms of security. On the platforms $\Theta 1$-$\Theta 4$, LSH-8w-n is the fastest one. The second fastest hash function on the platform $\Theta 1$ is Skein-512 with 5.58 cycles/byte [3]. On the platform $\Theta 2$, Blake-512 is the second fastest with 5.65 cycles/byte [3]. On the platform $\Theta 3$, Blake-256 is the second fastest with 8.48 cycles/byte [3].

Since there is no speed result of other hash functions on the platform $\Theta 4$ in eBash [3], we compare the speed results of LSH-8w-n with that on the platform AMD FX-8150 @ 3.6GHz. Blake-512 with 6.09 cycles/byte is the second fastest. See Table 7 in detail.

Platforms based on ARM We measured the speed performance at the platforms based on Cortex-A15 CPU and similar one which are the mainstream of current smart device market. Table 6 shows the speed performance of 1MB ($=2^{20}$bit) message hashing of LSH-8w-n on the platforms.

LSH-8w-n is implemented using NEON intrinsics, and GCC 4.8.1 is used with option "`-mfpu=neon -mfloat-abi=softfp -O3`".

On the platform $\Lambda 1$, LSH-512-n(8.94 cycles/byte) is the fastest among them. Blake-512(13.46 cycles/byte) is the second fastest one. See Table 8. The speed of LSH-8w-n is even faster than that of SHA-256 at Apple A7 based on ARMv8-A which has a SHA-2(SHA-256 only) instruction. The speed of LSH-256-n and LSH-512-n on the platform $\Lambda 1$ corresponds to about 150MB/sec and 190MB/sec, respectively. In iPhone5S equipped with A7 SoC, the speed of SHA-256 is about 102.2MB/sec(single core setting) [51]. Since there is no speed result from other

hash functions in eBASH [3], we can not compare the speed to others in $\Lambda 2$, $\Lambda 3$, and $\Lambda 4$.

5.3 Comparison with SHA-2 and the SHA-3 competition finalists

Table 7 and Table 8 are speed comparisons with SHA-2 and the SHA-3 finalists, reported in eBASH [3]. All values are the first quartile of many speed measurement. Table 7 is the comparison at the platform based on Haswell, LSH-8w-n is measured on Intel Core i7-4770k @ 3.5GHz quad core platform, and others are measured on Intel Core i5-4570S @ 2.9GHz quad core platform. Table 8 is measured on Samsung Exynos 5250 ARM Cortex-A15 @ 1.7GHz dual core platform. In these tables, Keccak-256 and Keccak-512 mean Keccak[r=1088,c=512] and Keccak[r=576,c=1024], respectively.

Table 7: Speed benchmark of LSH, SHA-2 and the SHA-3 finalists at the platform based on Haswell CPU (cycles/byte)

Algorithm	Message byte length					
	long	4,096	1,536	576	64	8
LSH-256-256	3.60	3.71	3.90	4.08	8.19	65.37
Skein-512-256	5.01(?)	5.58	5.86	6.49	13.12	104.50
Blake-256	6.61(?)	7.63	7.87	9.05	16.58	72.50
Grøstl-256	9.48(?)	10.68	12.18	13.71	37.94(?)	227.50(?)
Keccak-256	10.56	10.52	9.90	11.99	23.38	187.50
SHA-256	10.82(?)	11.91(?)	12.26	13.51	24.88	106.62
JH-256	14.70(?)	15.50(?)	15.94	17.06(?)	31.94	257.00
LSH-512-512	2.39	2.54	2.79	3.31	10.81	85.62
Skein-512-512	4.67(?)	5.51(?)	5.80	6.44	13.59	108.25
Blake-512	4.96(?)	6.17(?)	6.82	7.38	14.81	116.50(?)
SHA-512	7.65(?)	8.24	8.69	9.03	17.22	138.25
Grøstl-512	12.78(?)	15.44	17.30	17.99(?)	51.72	417.38
JH-512	14.25(?)	15.66	16.14(?)	17.34	32.69	261.00
Keccak-512	16.36(?)	17.86	18.46(?)	20.35	21.56(?)	171.88(?)

∗ Question marks mean the measurements with large variance [3], so each value should be re-measured.

6 Hardware implementation

We have implemented LSH with Verilog HDL and synthesized to ASIC. For HDL implementation and verification of our design, we have used Mentor Modelsim 6.5f for RTL simultation and Synopsys Design Compiler Ver. B-2008.09-SP5 for

Table 8: Speed benchmark of LSH, SHA-2 and the SHA-3 finalists at the platform based on Exynos 5250 ARM Cortex-A15 CPU (cycles/byte)

Algorithm	Message byte length					
	long	4,096	1,536	576	64	8
LSH-256-256	11.17	11.53	12.16	12.63	24.42	192.68
Skein-512-256	15.64	16.72	18.33	22.68	75.75	609.25
Blake-256	17.94	19.11	20.88	25.44	83.94	542.38
SHA-256	19.91	21.14	23.03	28.13	90.89	578.50
JH-256	34.66	36.06	38.10	43.51	113.92	924.12
Keccak-256	36.03	38.01	40.54	48.13	125.00	1000.62
Grøstl-256	40.70	42.76	46.03	54.94	167.52	1020.62
LSH-512-512	8.94	9.56	10.55	12.28	38.82	307.98
Blake-512	13.46	14.82	16.88	20.98	77.53	623.62
Skein-512-512	15.61	16.73	18.35	22.56	75.59	612.88
JH-512	34.88	36.26	38.36	44.01	116.41	939.38
SHA-512	44.13	46.41	49.97	54.55	135.59	1088.38
Keccak-512	63.31	64.59	67.85	77.21	121.28	968.00
Grøstl-512	131.35	138.49	150.15	166.54	446.53	3518.00

its synthesis. Our RTL level design result of LSH is synthesized to ASIC with the UMC 0.13μm standard cell library and 100MHz operating frequency.

We compared the hardware implementation results of LSH in the sense of FOM (throughput/area) with SHA-2 and the SHA-3 competition finalists where each throughput is revised for the clock frequency 100MHz. We did not consider lightweight hash functions because their designs are quite far from FOM optimization. Table 9 shows the comparison. We referred webpages of eHash for it [1]. Except Keccak, LSH has as good as FOM efficiency among the hash functions.

For JH and Keccak, we considered only 256-bit hash since they get slower for 512-bit hash than for 256-bit hash even though these have same size of area. For SHA-512, we could not find a good ASIC implementation result for comparison.

References

1. ehash webpage – sha-3 hardware implementations. `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations`.
2. Intel intrinsics guide. `http://software.intel.com/sites/landingpage/IntrinsicsGuide`.
3. Measurements of sha-3 finalists, indexed by machine. `http://bench.cr.yp.to/results-sha3.html`.
4. Neon. `http://www.arm.com/products/processors/technologies/neon.php`.

Table 9: Comparison of hardware implementations of LSH and other hash functions

Algorithm	Area (kGEs)	Throughput† (Mbps)	Tech. (nm)	Max. Freq. (MHz)	FOM (Mbps/GE)
Keccak-256 [9]	10.5	4,251	90	454.5	0.405
LSH-256-256	26.67	3,793	130	100.0	0.142
LSH-512-512	64.22	7,043	130	100.0	0.110
Skein-512-512 [32]	57.93	5,120	32	631.3	0.088
Blake-256 [11]	58.30	3,318	180	114	0.057
Skein-256-256 [52]	53.87	2,561	180	68.8	0.048
Blake-512 [29]	128.00	5,965	90	298.0	0.047
Grøstl-256 [56]	110.11	5,110	130	188	0.046
SHA-256 [26]	71.9	776	65	179.86	0.041
Grøstl-512 [27]	341.00	7,315	180	85.1	0.021
JH-256 [5]	54.6	1,110	90	763.4	0.020

†Throughtput@100KHz

5. Rcis webpage (other asic implementations). `http://staff.aist.go.jp/akashi.satoh/SASEBO/en/sha3/others.html`.
6. x86, x64 instruction latency, memory latency and cpuid dumps. `http://instlatx64.atw.hu`.
7. Amd64 architecture programmers manual volume 6: 128-bit and 256-bit xop, fma4 and cvt16 instructions. Technical report, May 2009.
8. Sha-3 standard: Permutation-based hash and extendable-output functions. May 2014.
9. Abdulkadir Akin, Aydin Aysu, Onur Can Ulusel, and Erkay Savaş. Efficient hardware implementations of high throughput sha-3 candidates keccak, luffa and blue midnight wish for single- and multi-message hashing. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 168–177, New York, NY, USA, 2010. ACM.
10. Kazumaro Aoki and Yu Sasaki. Meet-in-the-middle preimage attacks against reduced sha-0 and sha-1. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 70–89. Springer Berlin Heidelberg, 2009.
11. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Sha-3 proposal blake. Submission to NIST (Round 3), 2010.
12. Elaine B. Barker, William C. Barker, and Annabelle Lee. Guideline for implementing cryptography in the federal government. 2005.
13. Daniel J. Bernstein. Second preimages for 6 (7? (8??)) rounds of keccak? NIST mailing list, 2010.
14. Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials. In *EUROCRYPT*, pages 12–23, 1999.
15. Eli Biham, Orr Dunkelman, and Nathan Keller. Enhancing differential-linear cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology ASIACRYPT*

2002, volume 2501 of *Lecture Notes in Computer Science*, pages 254–266. Springer Berlin Heidelberg, 2002.
16. Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, London, UK, UK, 1993.
17. Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full aes-192 and aes-256. In Mitsuru Matsui, editor, *Advances in Cryptology ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2009.
18. John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In Moti Yung, editor, *Advances in Cryptology CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer Berlin Heidelberg, 2002.
19. Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full aes. In DongHoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer Berlin Heidelberg, 2011.
20. Andrey Bogdanov and Meiqin Wang. Zero correlation linear cryptanalysis with reduced data complexity. In Anne Canteaut, editor, *Fast Software Encryption*, volume 7549 of *Lecture Notes in Computer Science*, pages 29–48. Springer Berlin Heidelberg, 2012.
21. A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to mceliece's cryptosystem and to narrow-sense bch codes of length 511. *Information Theory, IEEE Transactions on*, 44(1):367–378, Jan 1998.
22. Donghoon Chang and Mridul Nandi. Improved indifferentiability security analysis of chopmd hash function. In *FSE*, pages 429–443, 2008.
23. Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time related-key attack on the kasumi cryptosystem used in gsm and 3g telephony. *Journal of Cryptology*, pages 1–26, 2013.
24. Lei Duo and Chao Li. Improved collision and preimage resistance bounds on pgv schemes. Cryptology ePrint Archive, Report 2006/462, 2006. `http://eprint.iacr.org/`.
25. Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Submission to NIST (Round 3), 2010.
26. B. Muheim E. Homsirikamol C. Keller M. Rogawski H. Kaeslin J. Kaps G. Gürkaynak, K. Gaj. Lessons learned from designing a 65nm asic for evaluating third round sha-3 candidates. Third SHA-3 Candidates Conference, 2012. `http://csrc.nist/gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/GURKAYNAK_paper.pdf`.
27. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S. Thomsen. Grøstl – a sha-3 candidate. Submission to NIST (Round 3), 2011.
28. Jian Guo, Pierre Karpman, Ivica Nikolic, Lei Wang, and Shuang Wu. Analysis of blake2. Cryptology ePrint Archive, Report 2013/467, 2013. `http://eprint.iacr.org/`.
29. L. Henzen, J.-P. Aumasson, W. Meier, and R.C.-W. Phan. Vlsi characterization of the cryptographic hash function blake. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(10):1746–1754, Oct 2011.
30. Miia Hermelin and Kaisa Nyberg. Multidimensional linear distinguishing attacks and boolean functions. *Cryptography and Communications*, 4(1):47–64, 2012.

31. Intel. Intel architecture instruction set extensions programming reference. 319433-018, FEBRUARY 2014.
32. S. K. Mathew J. Walker, F. Sheikh and R. Krishnamurthy. A skein-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/`.
33. John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 474–490, Berlin, Heidelberg, 2005. Springer-Verlag.
34. Dmitry Khovratovich and Ivica Nikoli. Rotational cryptanalysis of arx. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer Berlin Heidelberg, 2010.
35. Dmitry Khovratovich, Ivica Nikoli, and Christian Rechberger. Rotational rebound attacks on reduced skein. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2010.
36. Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for preimages: Attacks on skein-512 and the sha-2 family. In Anne Canteaut, editor, *Fast Software Encryption*, volume 7549 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2012.
37. Simon Knellwolf and Dmitry Khovratovich. New preimage attacks against reduced sha-1. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 367–383. Springer Berlin Heidelberg, 2012.
38. Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer Berlin Heidelberg, 1995.
39. Lars R Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin Heidelberg, 2002.
40. Gatan Leurent. Analysis of differential attacks in arx constructions. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 226–243. Springer Berlin Heidelberg, 2012.
41. Gatan Leurent. Construction of differential characteristics in arx designs application to skein. In Ran Canetti and JuanA. Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 241–258. Springer Berlin Heidelberg, 2013.
42. Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In Mitsuru Matsui, editor, *Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer Berlin Heidelberg, 2002.
43. Yiyuan Luo, Zheng Gong, Ming Duan, Bo Zhu, and Xuejia Lai. Revisiting the indifferentiability of pgv hash functions. Cryptology ePrint Archive, Report 2009/265, 2009. `http://eprint.iacr.org/`.
44. Mitsuru Matsui. Linear cryptanalysis method for des cipher. In Tor Helleseth, editor, *Advances in Cryptology EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin Heidelberg, 1994.

45. Florian Mendel, Christian Rechberger, Martin Schlffer, and SrenS. Thomsen. The rebound attack: Cryptanalysis of reduced whirlpool and grøstl. In Orr Dunkelman, editor, *Fast Software Encryption*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer Berlin Heidelberg, 2009.
46. Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.
47. Nicky Mouha and Bart Preneel. Towards finding optimal differential characteristics for arx: Application to salsa20. Cryptology ePrint Archive, Report 2013/328, 2013. `http://eprint.iacr.org/`.
48. Tomislav Nad. The codingtool library, 2010. Presentation.
49. Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer Berlin Heidelberg, 2004.
50. Yu Sasaki and Kazumaro Aoki. Finding preimages in full md5 faster than exhaustive search. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 134–152. Springer Berlin Heidelberg, 2009.
51. Anand Lal Shimpi. The iphone 5s review. `http://www.anandtech.com/show/7335/the-iphone-5s-review/4`. 17 Sep. 2013.
52. Stefan Tillich. Hardware implementation of the sha-3 candidate skein. Cryptology ePrint Archive, Report 2009/159, 2009. `http://eprint.iacr.org/`.
53. David Wagner. The boomerang attack. In Lars R Knudsen, editor, *Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin Heidelberg, 1999.
54. Xiaoyun Wang, YiqunLisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In Victor Shoup, editor, *Advances in Cryptology CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin Heidelberg, 2005.
55. Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *EUROCRYPT*. Springer-Verlag, 2005.
56. Leyla Nazhandali Xu Guo, Sinan Huang and Patrick Schaumont. Fair and comprehensive performance evaluation of 14 second round sha-3 asic implementations. Second SHA-3 Candidate Conference, 2010. `http://csrc.nist/gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/SCHAUMONT_SHA3.pdf`.

A Differential characteristics for collision attack

LSH-256-256 The best 12-step differential characteristic with probability 2^{-1340} (2^{-1317} in step functions, 2^{-23} in message expansion) is as follows: Note that this characteristic starts from the step function STEP_1.

– Difference of a chaining variable: $\Delta\mathsf{T}$

```
7055c502 2762a531 ecb0207c 59a126ed 7d9ea591 0ce1f4ce a4805bfd 91c2e233
5667e004 ae09ec85 f684c37f 058406d2 da80b205 1e76c9a1 00767204 4adc413a
```

– Difference of sub-messages: $\Delta\mathsf{M}_1||\Delta\mathsf{M}_2$

```
80000004 80000004 00000000 00000000 00000000 00000000 00000000 00000000
80000000 00000000 80000000 80000000 00000000 00000000 00000000 00000000
00000000 80000004 80000004 80000004 00000000 00000000 00000000 00000000
00000000 80000000 00000000 00000000 00000000 00000000 00000000 00000000
```

LSH-512-512 The best 13-step differential characteristic with probability 2^{-2562} (2^{-2535} in step functions, 2^{-27} in message expansion) is as follows: Note that this characteristic starts from the step function STEP_1.

– Difference of a chaining variable: $\Delta\mathsf{T}$

```
51102224b4620122 0c29d317cb02ef13 000090414918c242 a046aa4209442500
0c83e503b765044a 13aa3fff2cba2c36 5211940602291c46 7942d68622882342
10100306f1000120 5ca5b115acae2bcd 800091639c90e720 210083200116c000
8082c0829641060e df9637b8ad8f8662 3901976042bed0e3 f42b15854c588246
```

– Difference of sub-messages: $\Delta\mathsf{M}_1||\Delta\mathsf{M}_2$

```
8000010000000000 8000010000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
8000000000000000 0000000000000000 8000000000000000 8000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 8000010000000000 8000010000000000 8000010000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 8000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
```