Meet-in-the-Middle Preimage Attacks on Double-Branch Hash Functions: Application to RIPEMD and Others

Yu Sasaki and Kazumaro Aoki

NTT Information Sharing Platform Laboratories, NTT Corporation
3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 Japan
sasaki.yu@lab.ntt.co.jp

Abstract. We describe preimage attacks on several double-branch hash functions. We first present meet-in-the-middle preimage attacks on RIPEMD, whose output length is 128 bits and internal state size is 256 bits. With this internal state size, a straightforward application of the meet-in-the-middle attack will cost the complexity of at least 2^{128}, which gives no advantage compared to the brute force attack. We show two attacks on RIPEMD. The first attack finds pseudo-preimages and preimages of the first 33 steps with complexities of 2^{121} and $2^{125.5}$, respectively. The second attack finds pseudo-preimages and preimages of the intermediate 35 steps with complexities of 2^{96} and 2^{113}, respectively. We next present meet-in-the-middle preimage attacks on full Extended MD4, reduced RIPEMD-256, and reduced RIPEMD-320. The best known attack for these is the brute force attack. We show how to find preimages more efficiently on these hash functions.

Keywords: RIPEMD, double branch, preimage, meet-in-the-middle.

1 Introduction

Hash functions are cryptographic primitives used for various purposes. They are required to satisfy several security properties: preimage resistance, second preimage resistance, collision resistance, and so on. Usually, if the length of the hash is n-bit, the required security for these properties is $2^n, 2^n$, and $2^{n/2}$, respectively. Note that in the SHA-3 competition [22] conducted by NIST, 2^n security is required for the preimage resistance.

Various hash functions have been designed. A list of hash function types is shown in Fig. 1. The most widely used hash functions, e.g., MD5 [18], SHA-1, and SHA-2 [23], have a structure where the initial value, whose length is the same as the hash value, is iteratively updated by using messages. Hereafter, we call such a structure "single-path." In contrast, some hash functions update two copies of the initial value in parallel, merge each result, and finally output the merged value as the hash value of the message. Hereafter, we call such a structure "double-branch." For example, RIPEMD [16], RIPEMD-128, and RIPEMD-160 [7], are double-branch hash functions.

C. Boyd and J. González Nieto (Eds.): ACISP 2009, LNCS 5594, pp. 214–231, 2009.

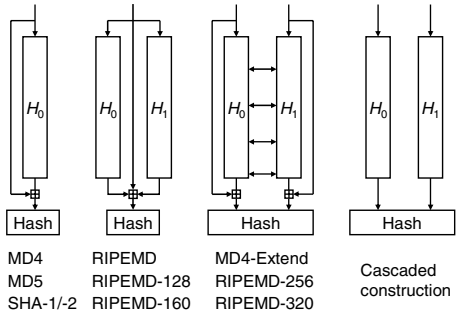


Fig. 1. Types of hash function structures

Hash functions are sometimes required to output longer hash values. For this purpose, some hash functions define an extension to output the double-length hash value, e.g., MD4 [17], RIPEMD-128, and RIPEMD-160. For a given input message, two hash values are computed by using different initial values and round constants. Finally, the concatenated value of two hash values, which is the double-length of the original hash value, is output. Such extensions of MD4, RIPEMD-128, and RIPEMD-160 are called Extended MD4[1], RIPEMD-256, and RIPEMD-320, respectively. Efforts to strengthen the security are made in these extensions. Intermediate chaining values after each round are swapped between computations of two hash values so that a stronger interaction between two computations can be achieved.

A cascaded construction, which was analyzed by Joux [10], produces a long hash value from two short hash values. It computes two hash functions and outputs the concatenated value.

The security of the double-branch hash functions is unclear. Intuitively, if two hash functions are ideal, the security will be a product of two hash functions. However, if two hash computations are similar, some attacks might be performed because of unwanted dependencies. The designers of RIPEMD-256 and -320 consider this situation. Although the known best preimage attack on RIPEMD-256 and -320 is the brute force attack, which costs 2^{256} and 2^{320}, their security is claimed to be 2^{128} and 2^{160}, respectively. Similarly, the security of Extended MD4 is not described by its designer[2]. Reference [14, Fact 9.27] claims the security of the cascaded construction is a product of each hash function; however, Joux showed the security is damaged if iterated hash functions are instantiated [10].

1.1 Attack History

Several papers have been published on finding collisions or variants of collisions on RIPEMD, RIPEMD-128, and RIPEMD-160 [6,5,4,25,12]. However, the

[1] Rivest, the designer of MD4, did not name this extension. Dobbertin, the first cryptanalyst of this extension, called it "Extended MD4."

[2] Dobbertin, in his analysis paper [5], introduced Extended MD4, which was proposed for highest security requirements.

preimage resistance of double-branch hash functions has not been studied much. Since 2008, several meet-in-the-middle preimage attacks on hash functions whose structures are similar to MD4 have been proposed [11,2,1,20,21]. The targets of these attacks are single-path hash functions, hence the attack techniques cannot be applied to double-branch hash functions directly. Mendel presented preimage attacks [13] on HAS-V [15], which is a double-branch hash function with a swapping function. The attack exploits a weakness of the HAS-V step-function, which cannot be applied to other hash functions. Saarinen [19] presented a preimage attack on FORK-256 [8], which is a 4-branch hash function. The attack has some similarity with ours; however, its success lies in the small number of steps in each branch and the weak message schedule of FORK-256. At ISPEC 2009, a preimage attack on 29 steps of RIPEMD with a complexity of $2^{115.2}$ was presented by Wang et al. [24]. Note that our work is independent of Ref. [24].

At CRYPTO'04, Joux analyzed the cascaded construction [10]. He showed that the cascaded construction does not provide the security of each product if an iterated hash function is used. Joux also explained that his technique cannot be applied to RIPEMD-256 and -320 due to the swapping function of intermediate values. This shows that the swapping function strengthens the security at some point. However, whether or not it can prevent other attacks is unclear.

1.2 Our Contribution

We present preimage attacks on several double-branch hash functions. We first present preimage attacks on step-reduced RIPEMD and then show how to find preimages of Extended MD4, RIPEMD-256, and RIPEMD-320 faster than the brute force attack does. The second result shows that using the swapping function does not provide the ideal security for double-branch hash functions. Details of each result are as follows.

1. We describe two preimage attacks on RIPEMD. The first attack, with a complexity of 2^{121}, provides pseudo-preimages of the *first* 33 out of 48 steps of RIPEMD. This can be converted to a preimage attack with a complexity of $2^{125.5}$. Our attack is based on the meet-in-the-middle attack on MD5 and MD4 [1]. However, because RIPEMD runs two MD4 computations, the size of the internal state is also doubled. Therefore, the straightforward application of the meet-in-the-middle attack does not give any advantage. We focus on the differentials of two MD4 computations, and efficiently perform the meet-in-the-middle attack.

   The second approach, with a complexity of 2^{96}, provides pseudo-preimages of the *intermediate* 35 steps of RIPEMD. This can be converted to a preimage attack with a complexity of 2^{113}. Technically, we use the meet-in-the-middle attack and the idea of local collision. This strategy is partially similar to the preimage attack on 1-block MD4 [1].
2. Extended MD4 provides 256-bit hash values. Our preimage attack on full Extended MD4 finds pseudo-preimages and preimages with complexities of 2^{229} and $2^{243.5}$, respectively. We also show that pseudo-preimages and preimages

of the *first* 62 out of 64 steps of RIPEMD-256 are found with complexities of 2^{240} and 2^{249}, respectively, and pseudo-preimages and preimages of the *intermediate* 64 out of 80 steps of RIPEMD-320 are found with complexities of 2^{304} and 2^{313}, respectively.

From a technical viewpoint, we show how to avoid the swapping functions. In Extended MD4, the message schedules of both sides are identical. We show that using swapping functions in such a structure does not prevent our attack. In RIPEMD-256 and -320, message schedules are different on each side. However, the attacker might be able to attack even if the swapping function is used. Then, by combining this idea with the splice-and-cut, partial-matching, and partial-fixing techniques proposed in Ref. [1,21], we attack those hash functions.

Although results in this paper do not contradict the security claims of these hash functions, the known best attack on these hash functions is the brute force attack, and no one knows whether or not better attacks exist. Therefore, we believe that our analyses contribute to better understanding of the security of double-branch hash functions.

2 Description of Hash Functions

2.1 MD4

MD4 takes arbitrary length messages as input and outputs 128-bit hash values. MD4 was proposed in 1990 by Rivest [17] and is a basic component of RIPEMD and Extended MD4. MD4 has the Merkle-Damgård structure. The input message is padded to be multiples of 512-bit. First, a single bit '1' is appended, then bit '0's are appended until the message length becomes 448 mod 512. Finally, the binary expression of the input message length is appended to the last 64 bits. The message is divided into 512-bit message blocks M_i. Then, the hash value is computed as follows:

$$\begin{cases} H_0 \leftarrow IV, \\ H_{i+1} \leftarrow \mathrm{md4}(H_i, M_i) \qquad \text{for } i = 0, 1, \ldots, n-1, \end{cases}$$

where IV is the initial value defined in the specification, H_n is the output hash value, and md4: $\{0,1\}^{128} \times \{0,1\}^{512} \rightarrow \{0,1\}^{128}$ is the compression function of MD4 computed as follows.

1. M_i is divided into 32-bit message words m_j $(j = 0, 1, \ldots, 15)$.
2. p_0 is set to H_i.
3. Compute the following: $p_{j+1} \leftarrow R_j(p_j, m_{\pi(j)})$ for $j = 0, 1, \ldots, 47$.
4. H_{i+1} $(= p_{48} + H_i)$ is output, where "+" denotes 32-bit word-wise addition. In this paper, we similarly use "−" to denote 32-bit word-wise subtraction.

R_j is the step function for Step j. Let a_j, b_j, c_j, and d_j be 32-bit values that satisfy $p_j = (a_j, b_j, c_j, d_j)$. $R_j(p_j, m_{\pi(j)})$ is defined as follows:

$$\begin{aligned} a_{j+1} &= d_j, & b_{j+1} &= (a_j + \Phi_j(b_j, c_j, d_j) + m_{\pi(j)} + k_j) \lll s_j, \\ c_{j+1} &= b_j, & d_{j+1} &= c_j, \end{aligned}$$

where Φ_j, k_j, and $\lll s_j$ are the bitwise Boolean function, constant value, and left rotation defined in the specification, respectively. $\pi(j)$ is the MD4 message schedule. Note that $R_j^{-1}(p_{j+1}, m_{\pi(j)})$ can be computed at almost the same complexity as that of R_j.

2.2 RIPEMD

RIPEMD [16] is an extension of MD4, whose compression function consists of two parallel copies of MD4's compression function. These functions are identical but for the constant number in each step. We describe chaining variables for one side $p_j = (a_j, b_j, c_j, d_j)$ and for the other side $p'_j = (a'_j, b'_j, c'_j, d'_j)$. The message schedule $\pi(j)$, the constant numbers k_j and k'_j, and the rotation number s_j are different from those for MD4. These values are shown in Table 1. Finally, the output of RIPEMD's compression function $H_{n+1} = (H_a, H_b, H_c, H_d)$ is computed by using $H_n = (IV_a, IV_b, IV_c, IV_d)$, p_{48}, and p'_{48} as follows.

Table 1. Message schedule, constants, and rotation numbers of RIPEMD

$\pi(0), \pi(1), \ldots, \pi(15)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(16), \pi(17), \ldots, \pi(31)$	7	4	13	1	10	6	15	3	12	0	9	5	14	2	11	8
$\pi(32), \pi(33), \ldots, \pi(47)$	3	10	2	4	9	15	8	1	14	7	0	6	11	13	5	12

	$0 \le i \le 15$	$16 \le i \le 31$	$32 \le i \le 47$
k_i	0x00000000	0x5a827999	0x6ed9eba1
k'_i	0x50a28be6	0x00000000	0x5c4dd124

$s_0, s_1, \ldots, s_{15}$	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
$s_{16}, s_{17}, \ldots, s_{31}$	7	6	8	13	11	9	7	15	7	12	15	9	7	11	13	12
$s_{32}, s_{33}, \ldots, s_{47}$	11	13	14	7	14	9	13	15	6	8	13	6	12	5	7	5

$$H_a = IV_b + c_{48} + d'_{48}, \qquad H_b = IV_c + d_{48} + a'_{48},$$
$$H_c = IV_d + a_{48} + b'_{48}, \qquad H_d = IV_a + b_{48} + c'_{48}.$$

2.3 RIPEMD-128, RIPEMD-160

RIPEMD-128 and RIPEMD-160, which output 128-bit and 160-bit hash values respectively, were proposed by Dobbertin et al. in 1996 [7]. They have been standardized by the International Organization for Standardization (ISO) [9].

A branch of RIPEMD-160 uses five 32-bit chaining variables. It computes 80 steps to produce the output value. Let the chaining variables in step j be $p_j = (a_j, b_j, c_j, d_j, e_j)$. Step function $R_j(p_j, m_\pi(j))$ is as follows.

$$a_{j+1} = e_j, \quad c_{j+1} = b_j, \quad d_{j+1} = c_j \lll 10, \quad e_{j+1} = d_j,$$
$$b_{j+1} = ((a_j + \Phi_j(b_j, c_j, d_j) + m_{\pi(j)} + k_j) \lll s_j) + e_j.$$

Table 2. Message schedules of RIPEMD-160

r	$\pi(r), \pi(r+1), \ldots, \pi(r+15)$	$\pi'(r), \pi'(r+1), \ldots, \pi'(r+15)$
0	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	5 14 7 0 9 2 11 4 13 6 15 8 1 10 3 12
16	7 4 13 1 10 6 15 3 12 0 9 5 2 14 11 8	6 11 3 7 0 13 5 10 14 15 8 12 4 9 1 2
32	3 10 14 4 9 15 8 1 2 7 0 6 13 11 5 12	15 5 1 3 7 14 6 9 11 8 12 2 10 0 4 13
48	1 9 11 10 0 8 12 4 13 3 7 15 14 5 6 2	8 6 4 1 3 11 15 0 5 12 2 13 9 7 10 14
64	4 0 5 9 7 12 2 10 14 1 3 8 11 6 15 13	12 15 10 4 1 5 8 7 6 2 13 14 0 3 9 11

Between two copies of the compression function, the order of the Boolean functions, message schedules, constants, and rotation numbers are different. The message schedules are shown in Table 2. The output of RIPEMD-160 is computed in the same manner as that of RIPEMD.

A branch of RIPEMD-128 uses 4 chaining variables and consists of 64 steps. The step function is the same as that of MD4 and RIPEMD. The Boolean functions and rotation numbers used in RIPEMD-128 are the same as those of the first 64 steps for RIPEMD-160, but the order is different. The message schedule for RIPEMD-128 is the same as that of the first 64 steps for RIPEMD-160, which is shown in Table 2. Computation for the final output is also the same as that of RIPEMD.

2.4 Extended MD4, RIPEMD-256, and RIPEMD-320

Extended MD4 is an optional extension of MD4 proposed by Rivest to obtain 256-bit hash values [17]. Two copies of MD4 are run in parallel over the input. The first copy is the same as MD4. The second copy is computed with different IV and constants. To strengthen the data dependency between two copies, a swapping function is introduced, which exchanges the values of a_{16} and a'_{16}, a_{32} and a'_{32}, and a_{48} and a'_{48}. The final output is obtained by concatenating both results.

RIPEMD-256 and RIPEMD-320 are extensions of RIPEMD-128 and RIPEMD-160 for obtaining the double length hash values without needing a higher security level [7]. The output is achieved by computing the feedforward of IV in each branch and concatenating the results at the end of every application of the compression function. Interaction between two copies is introduced by a swapping function, which exchanges the values of a_{16} and a'_{16}, b_{32} and b'_{32}, etc.

3 Related Works

3.1 Converting Pseudo-preimage Attack to Preimage Attack

Given a hash value H_n, a pseudo-preimage is a pair of (H_{n-1}, M) such that $\mathrm{Hash}(H_{n-1}, M) = H_n$. In x-bit iterated hash functions, a pseudo-preimage attack whose complexity is $2^y, y < x - 2$ can be converted to a preimage attack with a complexity of $2^{\frac{x+y}{2}+1}$ [14, Fact9.99].

3.2 Meet-in-the-Middle Preimage Attack

Aoki and Sasaki proposed a preimage attack based on the meet-in-the-middle attack [1]. They proposed three techniques named *splice-and-cut*, *partial-matching*, and *partial-fixing*.

The splice-and-cut technique considers the first and last steps of the compression function as consecutive steps. Then, the compression function is divided into two *chunks* of steps so that each chunk includes independent message words, which are called *neutral words*. Then, a pseudo-preimage is computed by the meet-in-the-middle attack.

The partial-matching technique can skip messages in several steps when checking the match in the meet-in-the-middle attack. It focuses on the property where not all the chaining variables are updated in each step. With this idea, we can partially compare two results, even if values of message words in several steps are not known.

The partial-fixing technique enables an attacker to skip more steps. The idea is to fix a part of the neutral words so that an attacker can partially compute a chunk even if a neutral word for the other chunk appears. For example, consider the inversion of MD4: $a_j = (b_{j+1} \ggg s_j) - \Phi_j(c_{j+1}, d_{j+1}, a_{j+1}) - m_{\pi(j)} - k_j$. If the lower n bits of $m_{\pi(j)}$ are fixed and thus known to the attacker and if other variables are fully known, the lower n bits of a_j can be computed independently of the higher $32 - n$ bits.

Since the internal state size of RIPEMD is double the output size, the straightforward application of the meet-in-the-middle attack does not give any advantage.

3.3 Analysis on Double-Branch Hashes and Cascaded Construction

At CRYPTO 2004, Joux proposed attacks on cascaded construction [10]. Joux showed how to generate multi-collisions of iterated hash functions and how to find collisions and preimages of the cascaded construction by using multi-collisions. The success of Joux's attack lies in the independency of the two hash functions in the cascaded construction, namely, multi-collisions of the iterated hash functions can be generated independently of the others. Joux explained that his technique would not be applied to RIPEMD-256 and -320 due to the dependency of the two compression functions caused by swapping functions. In conclusion, if swapping functions are used, no attack is known to break the preimage resistance of double-branch hash functions.

4 Preimage Attacks on RIPEMD

We present here two preimage attacks on RIPEMD. The first attack targets the first 33 steps and the second attack targets the intermediate 35 steps.

4.1 Attacks on First 33 Steps

Outline of Attack. Our attack is based on the meet-in-the-middle attack introduced in Section 3.2. However, since the internal state size is double the

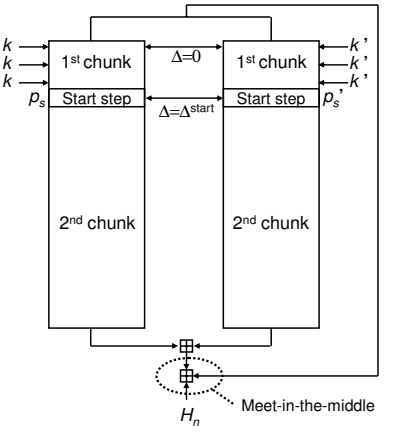


Fig. 2. Outline of strategy 1

output size, a direct application of the meet-in-the-middle attack cannot exceed the brute force attack. Our strategy to solve this problem is shown in Fig. 2. We separate the attack target so that one chunk is located in the first several steps and the other chunk is located in the last relatively long steps, and then we compare the results of the two chunks at the last feedforward operation which is performed in 128 bits.

Assume m_a and m_b are neutral words, where m_a is included in the first chunk but is not included in the second chunk, and m_b is vice versa. Remember that message schedules for both sides are identical, hence if one side can be separated into two chunks, the other side can always be separated in the same manner. First, we fix all message words but m_a and m_b and fix chaining variables at the border of two chunks, e.g., fix p_s and p'_s to compute the first and second chunks independently. We then inversely compute the first chunk with $R_j^{-1}(p_{j+1}, m_j)$ and ${R'_j}^{-1}(p'_{j+1}, m_j)$ for $j = s-1, s-2, \ldots, 0$ by trying all values of m_a and store the results in a table. Finally, we compute the second chunk with $R_j(p_j, m_j)$ and $R'_j(p'_j, m_j)$ for $j = s, s+1, \ldots, 32$ by trying all values of m_b and then check whether the result matches items in the table.

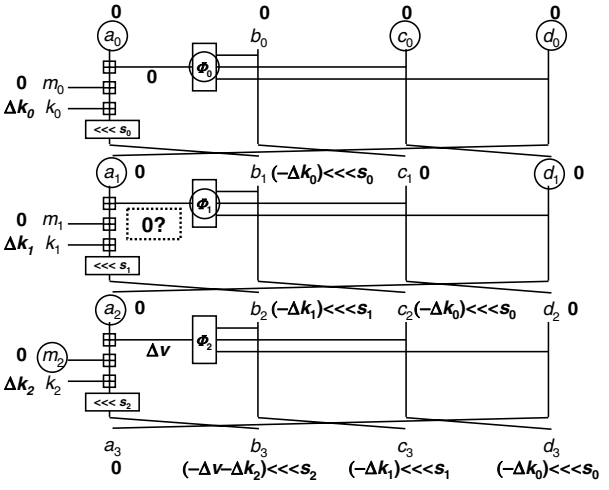
For consistency with the specification of RIPEMD, the values of p_0 and p'_0 computed in the backward computation must be identical, because they are originally computed from the same IV. The differences of the computations for both sides are differences of the constant Δk only. Therefore, we fix intermediate chaining variables p_s and p'_s to have a specific difference Δ^{start} so that Δ^{start} and Δk can be cancelled out in the backward computation.

Set Up of Attack. We separate the 33 steps into 2 chunks as shown in Fig. 3. The border of the two chunks is between Steps 2 and 3; we therefore fix p_3 and p'_3 so that their difference Δ^{start} can cancel Δk in Steps 0–2. Now we trace the differentials in Steps 0–2 of both sides to determine the appropriate Δ^{start}. The analysis is shown in Fig. 4.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index	0	1	**(2)**	3	4	5	6	7	8	9	10	11	**(12)**	13	14	15
	first chunk			second chunk												
Step	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
index	7	4	13	1	10	6	15	3	**(12)**	0	9	5	14	**(2)**	11	8
	second chunk													skip		
Step	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
index	3	10	**(2)**	4	9	15	8	1	14	7	0	6	11	13	5	**(12)**
	skip	Excluded from the attack target														

In RIPEMD, the message schedules of the two compression functions are identical. We separate them into two chunks in the same manner.

Fig. 3. Chunks for first 33 steps of RIPEMD



Bold font represents the value of differences.

Fig. 4. Differences propagation in first three steps

The difference of a variable X is defined as $\Delta X = X' - X$. The goal is determining $\Delta a_3, \Delta b_3, \Delta c_3$, and Δd_3 so that $\Delta a_0, \Delta b_0, \Delta c_0$, and Δd_0 become 0. Since message schedules for both sides are identical, $\Delta m_{\pi(j)}$ is always 0. In the first chunk, m_2 is the neutral word. Therefore, in every trial of the first chunk, the values of m_2 and corresponding chaining variables are changed. In Fig. 4, we circled such variables. The analysis is as follows.

$\Delta b_0 = 0$**:** This can be easily achieved by setting $\Delta a_3 = 0$.

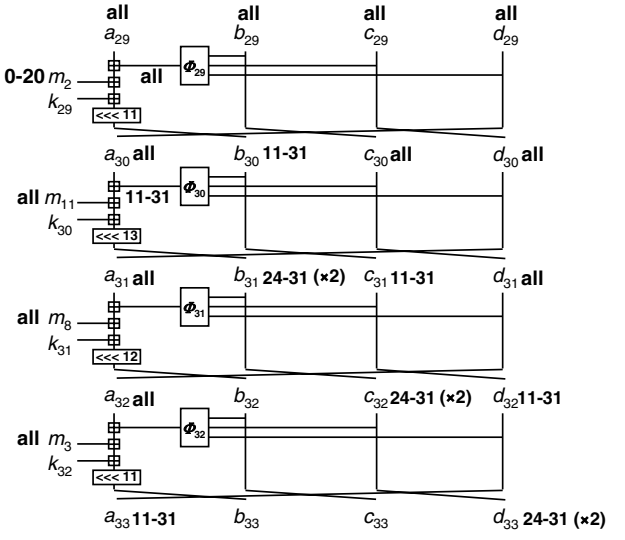
$\Delta a_0 = 0$**:** Assume we can achieve $\Delta b_0 = \Delta c_0 = \Delta d_0 = 0$. Then, $\Delta a_0 = 0$ can be achieved by setting $\Delta d_3 = (-\Delta k_0) \lll s_0$.

$\Delta c_0 = 0$**:** Assume we have finished fixing the values of $a_3, a'_3, c_3, c'_3, d_3$, and d'_3. Then, the value and difference of the output of Φ_2 are fixed. Let this difference be Δv. Finally, $\Delta c_0 = 0$ is achieved by setting $\Delta b_3 = (-\Delta v - \Delta k_2) \lll s_2$.

$\Delta d_0 = 0$**:** Achieving $\Delta d_0 = 0$ is complicated. We want to guarantee $\Delta a_1 = 0$ regardless of the value of the neutral variable m_2. To achieve this, we need to fix the value of Φ_1 independently of m_2. Since the function Φ_1 is $\Phi_1(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$ and $\Delta c_1 = \Delta d_1 = 0$, $\Delta\Phi_1$ can be fixed to 0 by setting $b_1 = b'_1$. However, this is impossible since b_1 and b'_1 must have differences. Consequently, we search for the exact value of b_1 and b'_1 to minimize the Hamming weight of $(b_1 \oplus b'_1)$ so that the probability of $\Delta\Phi_1 = 0$ is maximized. Remember that for each bit where $b_1 \oplus b'_1 = 1$, the equation $\Delta\Phi_1 = 0$ is satisfied with a probability of 1/2.

As we will explain later, we fix the lower 21 bits of m_2 for the partial-fixing technique. Consequently, the lower 21 bits of $\Delta\Phi_1$ are fixed. Therefore, minimizing the Hamming weight of the upper 11 bits (HW^{11}) of $(b_1 \oplus b'_1)$ is enough. We tried 2^{32} values of b_1 and confirmed that no value achieves $HW^{11}(b_1 \oplus b'_1) \leq 3$ and many values achieve $HW^{11}(b_1 \oplus b'_1) = 4$. For example, $b_1 =$ `0xffffffff`, $b'_1 =$ `0x50a28be5` is the case. Finally, by choosing the value of $b_1 (= d_3)$ to minimize the Hamming weight, the probability of $\Delta\Phi_1 = 0$ is 2^{-4}.

Partial-Matching and Partial-Fixing Techniques. As a result of computing the second chunk, we obtain p_{29} and p'_{29}. By fixing the lower 21 bits of m_2, we can perform the meet-in-the-middle attack on a further 4 steps in forward computation, namely, up to Step 32. The equation we use for matching is $H_b = IV_c + d_{33} + a'_{33}$, where H_b is given and the upper 11 bits of IV_c are



Bold font represents the bit positions of known bits.

Fig. 5. Partial-matching and partial-fixing techniques

produced from the first chunk. Therefore, we need to compute d_{33} from p_{29} and a'_{33} from p'_{29}. How we compute d_{33} and a'_{33} is shown in Fig. 5. Since m_2 is identical on both sides, the fixed bit positions in m_2 are also identical on both sides.

In Fig. 5, since the lower 21 bits of m_2 are fixed, we can uniquely compute bits 11–31 of b_{30}. This produces bits 11–31 of Φ_{30}. In the addition of Φ_{30}, we cannot determine the carried number from bit position 10 to 11. Therefore, we consider both carried number patterns and obtain two candidates for bits 24–31 of b_{31} after the $s_{30}(= 13)$-bit left rotation. Then, we compute the upper 8 bits of $d_{33} + a'_{33}$ with consideration of the two candidates of carried number patterns from bit position 23 to 24. In conclusion, the second chunk produces 4 candidates for the upper 8 bits of $d_{33} + a'_{33}$ for given p_{29} and p'_{29}, and thus, we can perform the 8-bit matching in 33-step RIPEMD.

Remark: From a designer's view, when results of two compression functions are merged, adding two values from different registers seems to be a good strategy against our attack. In fact, with only the partial-matching technique, attackers can skip only two steps in RIPEMD, whereas attackers can skip three steps in MD4. This is because the attacker needs to know the values of two different registers to compute each word of the output of RIPEMD.

Attack Procedure. Our attack first finds pseudo-preimages and converts them to preimages. Therefore, our attack finds a 2-block preimage. Hence, we fix m_{13}, m_{14}, and m_{15} to satisfy padding for 2-block messages. Given a hash value $H_2 = (H_a, H_b, H_c, H_d)$, the attack procedure is as follows.

1. Fix m_i $(i \notin \{2, 12, 13, 14, 15\})$ and the lower 21 bits of m_2 to randomly chosen values.
2. Fix a_3, b_3, c_3 to randomly chosen values and d_3 to `0xffffffff`. Then, compute a'_3, b'_3, c'_3, and d'_3 to make Δ^{start}, shown in Fig. 4.
3. (a) For all upper 11 bits of m_2, compute $R_j^{-1}(p_{j+1}, m_{\pi(j)})$ and $R_j'^{-1}(p'_{j+1}, m_{\pi(j)})$ for $j = 2, 1, 0$.
   (b) If $\Delta p_0 = 0$, compute $H_b - c_0$ and store $(m_2, p_0, H_b - c_0)$s in a table.
4. Compute $R_j(p_j, m_{\pi(j)})$ and $R'_j(p_{j'}, m_{\pi(j)})$ for $j = 3, 4, \ldots, 11$, and store p_{12} and p'_{12}.
5. (a) For all m_{12}, compute $R_j(p_j, m_{\pi(j)})$ and $R'_j(p'_j, m_{\pi(j)})$ for $j = 12, 13, \ldots 28$.
   (b) Compute bit positions 11 to 31 of b_{30} and b'_{30}, then compute bit positions 24 to 31 of b_{31} for both carried number patterns as shown in Fig. 5. Then, compute bits 24–31 of $d_{33} + a'_{33}$ by considering both carried number patterns from bit 23 to 24.
   (c) For each item in the table, check whether bits 24–31, in total 8 bits, of $d_{33} + a'_{33}$ are matched with $H_b - c_0$.
   (d) If matched, compute p_{30} to p_{33} by the corresponding m_i, and check whether or not all values are matched.
   (e) If all bits are matched, the corresponding message and p_0 is a pseudo-preimage.

Complexity Evaluation. Assume the complexity for computing 1 step is 1/33 33-step RIPEMD computations. The computational complexity of the above procedure is as follows. Step 3a takes $2^{11} \cdot \frac{3}{33}$. Since the success probability of 3b is 2^{-4}, $2^7 (= 2^{11} \cdot 2^{-4})$ items are stored in the table. Step 4 is negligible. Steps 5a and 5b take $2^{32} \cdot (\frac{17}{33} + \frac{3}{33})$. In 5b, $2^{34} (= 2^{32} \cdot 2 \cdot 2)$ items are produced. Therefore, in 5c, $2^{41} (= 2^{34} \cdot 2^7)$ pairs are compared, and after 8-bit matching for both carried number patterns, $2^{33} (= 2^{41} \cdot 2^{-8})$ pairs will remain. In 5d, we compute p_{30} and p_{31} at the complexity of $2^{28} (= 2^{33} \cdot \frac{2}{64})$, and by applying additional 56-bit match and checking the correctness of the guess for the carried number patterns, $2^{-25} (= 2^{33} \cdot 2^{-56} \cdot 2^{-2})$ pair will remain. Furthermore, by computing p_{31} and p_{32} at negligible complexity, we obtain $2^{-89} (= 2^{-25} \cdot 2^{-64})$ pair that is matched with 128 bits. The dominant complexity so far is 2^{32} of 5a and 5b. Therefore, by repeating the attack 2^{89} times, we obtain a pseudo-preimage at the complexity of $2^{121} (= 2^{32} \times 2^{89})$. Finally, by applying the technique in Section 3.1[3], this pseudo-preimage attack is converted to the preimage attack with a complexity of $2^{125.5}$. In the above procedure, a memory is used to store 2^7 $(m_2, p_0, H_b - c_0)$s at step 3b. Therefore, the memory complexity of this attack is approximately $2^7 \times 6$ words.

4.2 Attack on Intermediate 35 Steps

Similar to the attack on the first 33 steps, this attack is a meet-in-the-middle attack, but the approach is different. The strategy is shown in Fig. 6. In this approach, we start the meet-in-the-middle attack from an intermediate step of either two copies of the compression functions. Let us start from the left side. We separate the compression function so that one chunk includes two neutral messages that can form a local collision. This strategy was first used for the 1-block preimage attack on MD4 [1]. Due to the property of the local collision, the value of a pseudo-preimage is always fixed to a constant value. Therefore, we can consider the feedforward as constant addition and can compute the second chunk independently of the first chunk. Finally, we perform the meet-in-the-middle attack at the right side. The chunk we use is shown in Fig. 7.

The attack procedure is similar to Aoki and Sasaki's one-block MD4 preimage attack [1], and how to construct a local collision in the second round is explained in Leurent's MD4 preimage attack [11]. Therefore, because of the limited space, we omit the detailed attack procedure. Since both of the first and second chunks have 2^{32} free bits, the complexity of finding the pseudo-preimage is 2^{96}, and this attack, at the complexity of 2^{113}, is converted to the preimage attack. The memory complexity is approximately $2^{32} \times 5$ words.

[3] Several techniques converting partial-pseudo-preimages to preimages have been proposed [11,3]. However, since our attack does not find partial-pseudo-preimages efficiently, these techniques cannot be applied.

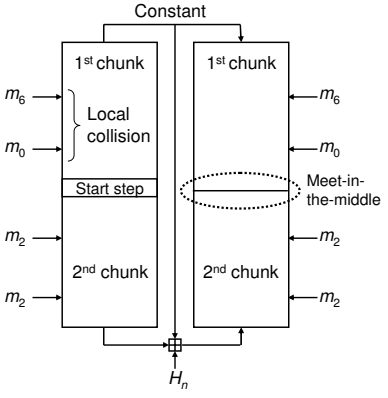


Fig. 6. Outline of strategy 2

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index	**0**	1	**2**	3	4	5	**6**	7	8	9	10	11	12	13	14	15
	excluded							first chunk								
Step	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
index	7	4	13	1	10	**6**	15	3	12	**0**	9	5	14	**2**	11	8
	first chunk													2nd chunk		
Step	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
index	3	10	**2**	4	9	15	8	1	14	3	**0**	**6**	11	13	5	12
	second chunk										excluded					

Chunk separations are identical on both sides.

Fig. 7. Chunks for intermediate 35 steps of RIPEMD

5 Cryptanalyses on Double-Branch Hash Functions

In this section, we analyze the preimage resistance of double length parallel hash functions. Specifically we give a study of relations of the splice-and-cut technique and swapping functions.

5.1 Extended MD4

In Extended MD4, two copies of MD4 with different IV and constants are computed. The swapping function of Extended MD4 exchanges the values of a_{16} and a'_{16}, a_{32} and a'_{32}, and a_{48} and a'_{48}.

MD4 has already been broken by using the splice-and-cut technique [1]. In this research, we found that the swapping function of Extended MD4 cannot prevent the splice-and-cut technique, namely, preimages are generated by almost the same approach as MD4. This is caused by the fact that the message schedules of two MD4 computations are exactly the same as original MD4. Due to this

fact, when we compute chunks in one side, we can also compute the value for the other side. Since we know the values for both sides, we can exchange them according to the swapping function. This means the swapping functions do not contribute to prevent our attack.

The chunk separation is the same as that shown in Ref. [1, Fig.5]. However, the partial-fixing technique can be improved for Extended MD4. In this attack, since we compare the results of two chunks on both compression functions, the number of bits matched by the meet-in-the-middle attack can increase. This enables us to reduce the fixed bits in neutral words, hence free bits in neutral words increase and the meet-in-the-middle attack becomes efficient.

By the partial-fixing technique, we fix the lower 5 bits of the neutral word and examine the 30-bit matching (= 5 bits $\times$ 6 words). This results in the pseudo-preimage attack[4] with a complexity of 2^{229}. This, with a complexity of $2^{243.5}$, is converted to a preimage attack with the algorithm explained in Section 3.1. The memory complexity is approximately $2^{27} \times 11$ words.

5.2 RIPEMD-256 and RIPEMD-320

In RIPEMD-256 and -320, the message orders of two copies of the compression functions are different. Therefore, different from Extended MD4, the attack cannot be applied in a straightforward manner. Note that the swapping function exchanges the value of a_{16} and a'_{16}, b_{32} and b'_{32}, c_{48} and c'_{48}, and so on.

To attack RIPEMD-256, we first search for a pair of neutral words that can attack as many steps as possible on either side. Then, on the other side, we check if we can divide the steps into two chunks so that the intermediate chaining variables that are used in the swapping function can also be computed. Selected neutral words and chunks are shown in Fig. 8.

As shown in Fig. 8, we skip eight steps when we attack the right side of MD4 by using the partial-matching and partial-fixing techniques. As introduced in Ref. [1], the partial-fixing technique, which increases the matching candidate twice, enables us to partially compute four steps in backward computation and one step in forward computation. The partial-matching technique enables us to skip three steps. Finally, eight steps can be skipped.

Avoid Swapping Function. In this attack, we assume that a_{16} and a'_{16}, b_{32} and b'_{32}, and c_{48} and c'_{48} are exchanged. d_{64} and d'_{64} are not exchanged since Step 63 is excluded from the attack target.

As you can see in Fig 8, b_{32} and b'_{32} are included in the second chunk and c_{48} and c'_{48} are included in the first chunk. Therefore, by computing both sides simultaneously, we can compute the values that follow the swapping function. a_{16} and a'_{16} are included in the skipped steps. When we check the matching of the results of both chunks, we do not use the values of a_{16} and a'_{16}. Therefore, swapping a_{16} and a'_{16} does not affect the attack complexity.

[4] If the previous attack is applied without improving the partial-fixing technique, this complexity would be 2^{241}.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index L	(0)	1	2	3	4	5	6	7	8	9	(10)	11	12	13	14	15
index R	5	14	7	(0)	9	2	11	4	13	6	15	8	1	(10)	3	12
	first chunk													skip		

Step	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
index L	7	4	13	1	(10)	6	15	3	12	(0)	9	5	2	14	11	8
											second chunk					
index R	6	11	3	7	(0)	13	5	(10)	14	15	8	12	4	9	1	2
	skip					second chunk										

Step	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
index L	3	(10)	14	4	9	15	8	1	2	7	(0)	6	13	11	5	12
	second chunk							first chunk								
index R	15	5	1	3	7	14	6	9	11	8	12	2	(10)	(0)	4	13
	second chunk													first chunk		

Step	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
index L	1	9	11	(10)	(0)	8	12	4	13	3	7	15	14	5	6	2
	first chunk														excluded	
index R	8	6	4	1	3	11	15	(0)	5	12	2	13	9	7	(10)	14
	first chunk														excluded	

Fig. 8. Chunks for first 62 steps of RIPEMD-256

Outline of Attack Procedure. Fix messages $m_j, j \notin \{0, 10\}$, p_{39}, and p'_{45}, where p_j is a variable for the left side and p'_j is for the right side. In the first chunk, compute $R_j(p_j, m_{\pi(j)})$ and $R'_j(p'_j, m_{\pi(j)})$ to obtain p_{48} and p'_{48} and swap c_{48} and c'_{48} to follow the swapping function. Then, we compute $R'_j(p'_j, m_{\pi(j)})$ until we obtain p'_{13} and store the results in a table. In the second chunk, compute $R_j^{-1}(p_j, m_{\pi(j)})$ and $R_j'^{-1}(p'_j, m_{\pi(j)})$ to obtain p_{32} and p'_{32} and swap b_{32} and b'_{32} to follow the swapping function. Then, we compute $R_j'^{-1}(p'_j, m_{\pi(j)})$ until we obtain p'_{21} and check whether the result matches items in the table by using the partial-matching and partial-fixing techniques. Hence, a pseudo-preimage for the right side is found efficiently, and by repeating this attack 2^{128} times, one of the resulting pseudo-preimages will also be the pseudo-preimage for the left side.

Complexity Estimation. When we attack the right side, we use the splice-and-cut technique. Since the partial-fixing technique is used, the complexity to find a pseudo-preimage of the right side is 2^{112}. If a pseudo-preimage of the right side is found, we check whether the message is also a pseudo-preimage of the left side. This occurs with a probability of 2^{-128}. Therefore, with a complexity of 2^{240}, we obtain a pseudo-preimage, and this, with a complexity of 2^{249}, is converted to a preimage. The memory complexity is approximately $2^{16} \times 9$ words.

Preimage Attack on Intermediate 64 Steps of RIPEMD-320. With the same strategy as the attack on RIPEMD-256, the intermediate 64 steps of RIPEMD-320 can be attacked. From Steps 12–75 are our attack target, and we select m_{10} and m_{11} as neutral words. Since the attack strategy is the same as that of RIPEMD-256, we show details of the chunks in Appendix A, Fig. 9.

Since the partial-fixing technique is necessary, the complexity of the pseudo-preimage attack is 2^{304}, and this, with a complexity of 2^{313}, is converted to a preimage attack. The memory complexity is approximately $2^{16} \times 7$ words.

6 Conclusion

We first described preimage attacks on RIPEMD. The first attack focuses on differentials of two copies of the compression function and attacks the first 33 steps. The second attack uses local collision and attacks the intermediate 35 steps. We next analyzed the preimage resistance of double-length hash functions. Our attacks find preimages of full Extended MD4, the first 62 steps of RIPEMD-256, and the intermediate 64 steps of RIPEMD-320 faster than the brute force attack does. We believe that analyses presented in this paper will contribute to greater understanding of the security of double-branch hash functions.

References

1. Aoki, K., Sasaki, Y.: Preimage attacks on one-block MD4, 63-step MD5 and more. In: Workshop Records of SAC 2008, Sackville, Canada, pp. 82–98 (2008)
2. Aumasson, J.-P., Meier, W., Mendel, F.: Preimage attacks on 3-pass HAVAL and step-reduced MD5. In: Workshop Records of SAC 2008, Sackville, Canada, pp. 99–114 (2008); ePrint version is available at IACR Cryptology ePrint Archive: Report 2008/183, `http://eprint.iacr.org/2008/183.pdf`
3. Canniére, C.D., Rechberger, C.: Preimages for reduced SHA-0 and SHA-1. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 179–202. Springer, Heidelberg (2008); (slides on preliminary results were appeared at ESC 2008 seminar `http://wiki.uni.lu/esc/`)
4. Debaert, C., Gilbert, H.: The RIPEMD^L and RIPEMD^R improved variants of MD4 are not collision free. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 65–74. Springer, Heidelberg (2002)
5. Dobbertin, H.: Cryptanalysis of MD4. Journal of Cryptology 11(4), 253–272 (1997); First result was announced at FSE 1996
6. Dobbertin, H.: RIPEMD with two-round compress function is not collision-free. Journal of Cryptology 10(1), 51–69 (1997)
7. Dobbertin, H., Bosselaers, A., Preneel, B.: RIPEMD-160: A strengthened version of RIPEMD. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 71–82. Springer, Heidelberg (1996)
8. Hong, D., Chang, D., Sung, J., Lee, S., Hong, S., Lee, J., Moon, D., Chee, S.: A new dedicated 256-bit hash function: FORK-256. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, pp. 195–209. Springer, Heidelberg (2006)
9. International Organization for Standardization. ISO/IEC 10118-3:2004, Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions (2004)

10. Joux, A.: Multicollisions in iterated hash functions. Application to cascaded constructions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
11. Leurent, G.: MD4 is not one-way. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 412–428. Springer, Heidelberg (2008)
12. Mendel, F., Pramstaller, N., Rechberger, C., Rijmen, V.: On the collision resistance of RIPEMD-160. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 101–116. Springer, Heidelberg (2006)
13. Mendel, F., Rijmen, V.: Weaknesses in the HAS-V compression function. In: Nam, K.-H., Rhee, G. (eds.) ICISC 2007. LNCS, vol. 4817, pp. 335–345. Springer, Heidelberg (2007)
14. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC Press, Boca Raton (1997)
15. Park, N.K., Hwang, J.H., Lee, P.J.: HAS-V: A New Hash Function with Variable Output Length. In: Stinson, D.R., Tavares, S. (eds.) SAC 2000. LNCS, vol. 2012, pp. 202–216. Springer, Heidelberg (2001)
16. RIPE Integrity Primitives, Berlin, Heidelberg, New York. Integrity Primitives for Secure Information Systems, Final RIPE Report of RACE Integrity Primitives Evaluation, RIPE-RACE 1040 (1995)
17. Rivest, R.L.: The MD4 message digest algorithm. In: Menezes, A.J., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 303–311. Springer, Heidelberg (1991); Also appeared in RFC 1320, `http://www.ietf.org/rfc/rfc1320.txt`
18. Ronald, L.R.: Request for Comments 1321: The MD5 Message Digest Algorithm. The Internet Engineering Task Force (1992), `http://www.ietf.org/rfc/rfc1321.txt`
19. Saarinen, M.-J.O.: A meet-in-the-middle collision attack against the new FORK-256. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 10–17. Springer, Heidelberg (2007)
20. Sasaki, Y., Aoki, K.: Preimage attacks on 3, 4, and 5-pass HAVAL. In: Pieprzyk, J.P. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 253–271. Springer, Heidelberg (2008)
21. Sasaki, Y., Aoki, K.: Finding preimages in full MD5 faster than exhaustive search. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 134–152. Springer, New York (2009)
22. U.S. Department of Commerce, National Institute of Standards and Technology. Federal Register 72(212) (November 2, 2007), `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`
23. U.S. Department of Commerce, National Institute of Standards and Technology. Secure Hash Standard (SHS) (Federal Information Processing Standards Publication 180-3) (2008), `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`
24. Wang, G., Wang, S.: Preimage attack on hash function RIPEMD. In: Bao, F., Li, H., Wang, G. (eds.) ISPEC 2009. LNCS, vol. 5451, pp. 274–284. Springer, Heidelberg (2009)
25. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the hash functions MD4 and RIPEMD. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)

A Chunks for Intermediate 64-Step RIPEMD-320

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index L	0	1	2	3	4	5	6	7	8	9	**10**	**11**	12	13	14	15
						excluded								first chunk		
index R	5	14	7	0	9	2	**11**	4	13	6	15	8	1	**10**	3	12
						excluded								first chunk		

Step	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
index L	7	4	13	1	**10**	6	15	3	12	0	9	5	2	14	**11**	8
							first chunk								skip	
index R	6	**11**	3	7	0	13	5	**10**	14	15	8	12	4	9	1	2

Step	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
index L	3	**10**	14	4	9	15	8	1	2	7	0	6	13	**11**	5	12
	skip							second chunk								
index R	15	5	1	3	7	14	6	9	**11**	8	12	2	**10**	0	4	13
														second chunk		

Step	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
index L	1	9	**11**	**10**	0	8	12	4	13	3	7	15	14	5	6	2
	2nd chunk								first chunk							
index R	8	6	4	1	3	**11**	15	0	5	12	2	13	9	7	**10**	14
						second chunk									first chunk	

Step	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
index L	4	0	5	9	7	12	2	**10**	14	1	3	8	**11**	6	15	13
						first chunk								excluded		
index R	12	15	**10**	4	1	5	8	7	6	2	13	14	0	3	9	**11**
			first chunk											excluded		

Fig. 9. Chunks for intermediate 64 steps of RIPEMD-320