Preimage Attacks on Step-Reduced RIPEMD-128 and RIPEMD-160

Chiaki Ohtahara[1], Yu Sasaki[2], and Takeshi Shimoyama[3]

[1] Chuo-University
cohtahara@chao.ise.chuo-u.ac.jp
[2] NTT Corporation
sasaki.yu@lab.ntt.co.jp
[3] Fujitsu Laboratories Ltd.
shimo@labs.fujitsu.com

Abstract. This paper presents the first results on the preimage resistance of ISO standard hash functions RIPEMD-128 and RIPEMD-160. They were designed as strengthened versions of RIPEMD. While preimage attacks on the first 33 steps and intermediate 35 steps of RIPEMD (48 steps in total) are known, no preimage attack exists on RIPEMD-128 (64 steps) or RIPEMD-160 (80 steps). This paper shows three variations of attacks on RIPEMD-128; the first 33 steps, intermediate 35 steps, and the last 32 steps. It is interesting that the number of attacked steps for RIPEMD-128 reaches the same level as RIPEMD. We show that our approach can also be applied to RIPEMD-160, and present preimage attacks on the first 30 steps and the last 31 steps.

Keywords: RIPEMD-128, RIPEMD-160, hash, preimage, meet-in-the-middle.

1 Introduction

Cryptographic hash functions are one of the most basic primitives. For symmetric-key primitives, it is quite standard to evaluate their security by demonstrating cryptanalysis on them or weakened versions e.g. step-reduced versions. In fact, analysis on the step-reduced versions is useful to know the security margin.

Preimage resistance is an important security for hash functions. When digests are n-bits, the required security is usually n-bits e.g. the SHA-3 competition [1].

Since the collision resistance of MD5 and SHA-1 have been significantly broken [2,3], many hash functions with various designs such as RIPEMD, Tiger, Whirlpool, and FORK-256 have been pointed out to be vulnerable or non-ideal [4,5,6,7,8]. Meanwhile, no attack is known against (full specifications of) RIPEMD-128 and RIPEMD-160 [9] though they were designed more than ten years ago.

RIPEMD [10] is a double-branch hash function, where the compression function consists of two parallel copies of a compression function. In 1996, Dobbertin *et al.* designed RIPEMD-128 and RIPEMD-160 [9] as strengthened versions of RIPEMD. They are standardized in ISO/IEC 10118-3:2003 [11].

X. Lai, M. Yung, and D. Lin (Eds.): Inscrypt 2010, LNCS 6584, pp. 169–186, 2011.

Table 1. Summary of attack results

Target	Steps	Method	Time for pseudo-preimage	Time for (2nd-)preimage	Mem.	Ref.
RIPEMD-128	first 33	IE	2^{119}	$2^{124.5}$	2^{12}	Ours
	middle 35	LC	2^{112}	2^{121}	2^{16}	Ours
	last 32	IE	$2^{122.4}$	$2^{126.2}$ †	2^{12}	Ours
RIPEMD-160	first 30	IE	2^{148}	2^{155} †	2^{16}	Ours
	last 31	IE	2^{148}	2^{155}	2^{17}	Ours

IE and LC represent the Initial-Exchange and Local-Collision approaches, respectively. Attacks with † can generate second-preimages but cannot generate preimages.

In 2005, Wang *et al.* showed a collision attack on full RIPEMD [4]. They used a property where two compression functions are identical but for the constant value and thus the same differential path can be used for both branches. Because RIPEMD-128 and -160 adopt different message expansion for two branches, the attack cannot be applied to them. For RIPEMD-128 and -160, only pseudo-(near-)collisions against step-reduced and modified versions are known [12].

On the preimage resistance, Wang *et al.* attacked the first 29 steps of RIPEMD [13]. Then, Sasaki *et al.* attacked more steps; the first 33 and intermediate 35 steps [14]. These attacks seem inefficient for RIPEMD-128 and -160 due to the different message expansion between two branches. In fact, as far as we know, no preimage attack exists on RIPEMD-128 and -160 even for step-reduced versions.

Saarinen [8] presented a preimage attack on a 4-branch hash function FORK-256 [15]. It uses several properties particular to FORK-256, and thus the same approach cannot be applied to RIPEMD, RIPEMD-128, or RIPEMD-160.

Our contributions. We present the first results on the preimage resistance of RIPEMD-128 and -160. Our attacks employ the meet-in-the-middle preimage attack [16]. Firstly, we devise *initial-exchange* technique, which exchanges a message-word position located in the first several steps for a branch with the one for the other branch. Secondly, we use a *local-collision* approach, which was first proposed by [14] to attack RIPEMD. The results are summarized in Table 1. Note that the approach of attacking the last few rounds was also taken by [17].

2 Specifications

RIPEMD-128. RIPEMD-128 [9] takes arbitrary and finite length messages as input and outputs 128-bit digest. It follows the Merkle-Damgård hash function mode (with standard length encoding). The input message is padded to be a multiple of 512 bits and is divided into 512-bit blocks M_i. Then, the hash value is computed as follows:

$$H_0 \leftarrow \mathrm{IV}, \qquad H_{i+1} \leftarrow \mathrm{CF}(H_i, M_i) \text{ for } i = 0, 1, \ldots, N-1$$

where IV is the initial value defined in the specification, H_N is the output hash value, and CF: $\{0,1\}^{128} \times \{0,1\}^{512} \rightarrow \{0,1\}^{128}$ is a compression function.

The compression function has a double-branch structure. Two compression functions $\mathrm{CF}^L(H_i, M_i) : \{0,1\}^{128} \times \{0,1\}^{512} \rightarrow \{0,1\}^{128}$ and $\mathrm{CF}^R(H_i, M_i) : \{0,1\}^{128} \times \{0,1\}^{512} \rightarrow \{0,1\}^{128}$ are computed and the output of the compression function is a mixture of (H_i, M_i), $\mathrm{CF}^L(H_i, M_i)$, and $\mathrm{CF}^R(H_i, M_i)$. Let $p_j^L, a_j^L, b_j^L, c_j^L, d_j^L$ be 128-bit, 32-bit, 32-bit, 32-bit, 32-bit variables, respectively, satisfying $p_j^L = a_j^L \| b_j^L \| c_j^L \| d_j^L$. Similarly, we define $p_j^R, a_j^R, b_j^R, c_j^R, d_j^R$. Details of the computation procedure is as follows.

1. M_i is divided into sixteen 32-bit message words m_j $(j = 0, 1, \ldots, 15)$ and H_i is divided into four 32-bit chaining variables $H_i^a \| H_i^b \| H_i^c \| H_i^d$.
2. p_0^L and p_0^R are set to H_i (and thus $p_0^L = p_0^R$).
3. Compute $p_{j+1}^L \leftarrow R_j^L(p_j^L, m_{\pi^L(j)})$ and $p_{j+1}^R \leftarrow R_j^R(p_j^R, m_{\pi^R(j)})$ for $j = 0, 1, \ldots, 63$, where $R_j^L, m_{\pi^L(j)}, R_j^R$, and $m_{\pi^R(j)}$ will be explained later.
4. Compute the output value $H_{i+1} = (H_{i+1}^a \| H_{i+1}^b \| H_{i+1}^c \| H_{i+1}^d)$ as follows, where "+" denotes a 32-bit word-wise addition.

$$H_{i+1}^a = H_i^b + c_{64}^L + d_{64}^R, \qquad H_{i+1}^b = H_i^c + d_{64}^L + a_{64}^R,$$
$$H_{i+1}^c = H_i^d + a_{64}^L + b_{64}^R, \qquad H_{i+1}^d = H_i^a + b_{64}^L + c_{64}^R.$$

R_j^L and R_j^R are the step functions for Step j. $R_j^L(p_j^L, m_{\pi^L(j)})$ is defined as follows:

$$a_{j+1}^L = d_j^L, \qquad b_{j+1}^L = (a_j^L + \Phi_j^L(b_j^L, c_j^L, d_j^L) + m_{\pi^L(j)} + k_j^L) \lll s_j^L,$$
$$c_{j+1}^L = b_j^L, \qquad d_{j+1}^L = c_j^L,$$

where Φ_j^L, k_j^L, and $\lll s_j^L$ are Boolean function, constant, and left rotation defined in Table 2. $\pi^L(j)$ is the message expansion of CF^L. R_j^R is similarly defined.

RIPEMD-160. Each branch of RIPEMD-160 consists of 80 steps using 160-bit state. Let the chaining variables in step j of CF^L be $p_j^L = a_j^L \| b_j^L \| c_j^L \| d_j^L \| e_j^L$. Step function $R_j^L(p_j^L, m_{\pi^L(j)})$ is as follows. ($R_j^R(p_j^R, m_{\pi^R(j)})$ is similarly described.)

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

$\pi(j), \Phi_j$, and $\lll s_j$ are shown in Table 2. Finally, the output value $H_{i+1} = (H_{i+1}^a \| H_{i+1}^b \| H_{i+1}^c \| H_{i+1}^d \| H_{i+1}^e)$ is computed as follows.

$$H_{i+1}^a = H_i^b + c_{80}^L + d_{80}^R, \quad H_{i+1}^b = H_i^c + d_{80}^L + e_{80}^R, \quad H_{i+1}^c = H_i^d + e_{80}^L + a_{80}^R,$$
$$H_{i+1}^d = H_i^e + a_{80}^L + b_{80}^R, \quad H_{i+1}^e = H_i^a + b_{80}^L + c_{80}^R.$$

Table 2. Detailed specifications of RIPEMD-128 and RIPEMD-160

r	$\pi^L(r), \pi^L(r+1), \ldots, \pi^L(r+15)$	$\pi^R(r), \pi^R(r+1), \ldots, \pi^R(r+15)$
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

j	$\Phi_j(X,Y,Z)$	Abbreviation
$0 \le j \le 15$	$X \oplus Y \oplus Z$	Φ_F
$16 \le j \le 31$	$(X \wedge Y) \vee (\neg X \wedge Z)$	Φ_G
$32 \le j \le 47$	$(X \vee \neg Y) \oplus Z$	Φ_H
$48 \le j \le 63$	$(X \wedge Z) \vee (Y \wedge \neg Z)$	Φ_I
$64 \le j \le 79$	$X \oplus (Y \vee \neg Z)$	Φ_J

For RIPEMD-128: $\Phi_j^L = \Phi_j, \Phi_j^R = \Phi_{63-j}$
For RIPEMD-160: $\Phi_j^L = \Phi_j, \Phi_j^R = \Phi_{79-j}$

r	$s_r^L, s_{r+1}^L, \ldots, s_{r+15}^L$	$s_r^R, s_{r+1}^R, \ldots, s_{r+15}^R$
0	11 14 15 12 5 8 7 9 11 13 14 15 6 7 9 8	8 9 9 11 13 15 15 5 7 7 8 11 14 14 12 6
16	7 6 8 13 11 9 7 15 7 12 15 9 11 7 13 12	9 13 15 7 12 8 9 11 7 7 12 7 6 15 13 11
32	11 13 6 7 14 9 13 15 14 8 13 6 5 12 7 5	9 7 15 11 8 6 6 14 12 13 5 14 13 13 7 5
48	11 12 14 15 14 15 9 8 9 14 5 6 8 6 5 12	15 5 8 11 14 14 6 14 6 9 12 9 12 5 15 8
64	9 15 5 11 6 8 13 12 5 12 13 14 11 8 5 6	8 5 12 9 12 5 14 6 8 13 6 5 15 13 11 11

3 Related Work

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_{N-1}) such that $\mathrm{CF}(H_{N-1}, M_{N-1}) = H_N$, and $H_{N-1} \neq \mathrm{IV}$. In n-bit narrow-pipe iterated hash functions, if pseudo-preimages with appropriate padding string can be generated with a complexity of 2^m, where $m < n - 2$, preimages can be generated with a complexity of $2^{\frac{m+n}{2}+1}$ [18, Fact9.99]. Leurent pointed out that constraints of the padding string can be ignored when we generate second preimages [19].

3.2 Meet-in-the-Middle Preimage Attack

Aoki *et al.* proposed a framework of the meet-in-the-middle preimage attack [16]. The attack divides the compression function into two *chunks* of steps so that each chunk includes independent message words, which are called *neutral words*. Then, pseudo-preimages are obtained by performing the meet-in-the-middle attack, namely, computing each chunk independently and matching the partially-computed intermediate chaining variables. The framework is illustrated in Figure 1. Please refer [16] for more details such as terminologies and procedure.

Assume that the first chunk has d_1 free bits and the second chunk has d_2 free bits, where $d_1 \le d_2$. Also assume that each chunk computes d_3 bits of intermediate chaining variables used for the match, where $d_3 \ge min(d_1, d_2) = d_1$. In this framework, an attacker computes d_3 match bits of the first chunk for 2^{d_1} possible values and store the results in a table. The table is sorted with time 2^{d_1} (e.g. Bucket Sort) so that look-up can later be carried out with time 1. Then, for each of 2^{d_2} possible values, compute the d_3 match bits of the second chunk and

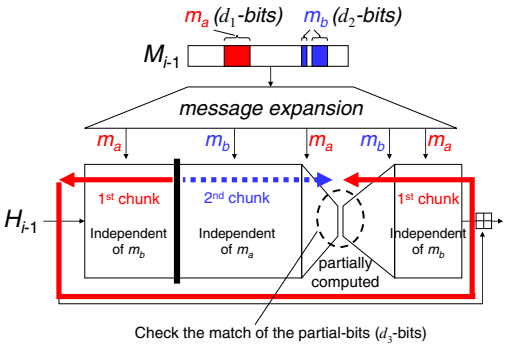

Fig. 1. Framework of the meet-in-the-middle preimage attack

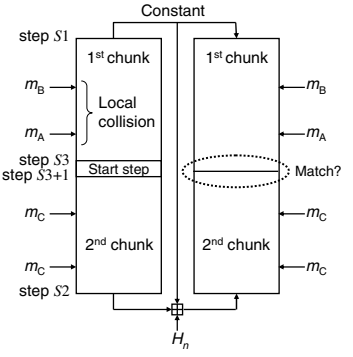

Fig. 2. Local-collision approach by [14] against RIPEMD

check if they exist in the table. If exist, compute and check the match of the other $n - d_3$ bits with the matched message words, where n is the state size. Using the 2^{d_2} computations of the second chunk, $2^{d_1+d_2-d_3}$ pairs whose d_3 bits match are obtained. Finally, by iterating the procedure $2^{n-(d_1+d_2)}$ times, a pseudo-preimage will be obtained. The attack complexity is $(2^{d_1} + 2^{d_2}) \cdot 2^{n-(d_1+d_2)}$ in time, and 2^{d_1} in memory.

The *initial-structure* technique proposed by Sasaki *et al.*[20] is a technique for this attack framework, which exchanges the positions of two message words in neighbouring steps. This can increase the search space of neutral words.

3.3 Preimage Attacks on RIPEMD

Since the internal state size of RIPEMD is double of the digest size, a simple application of the meet-in-the-middle attack for RIPEMD does not give any advantage. Sasaki *et al.* proposed an approach using a local-collision to solve this problem [14], which is depicted in Figure 2. In this approach, the attack target (steps $S1$ to $S2$) is divided into two chunks; $S1$ to $S3$ are the first chunk and $S3 + 1$ to $S2$ are the second chunk. The independent computations start from the middle of CF^L and find a match in the middle of CF^R. However, because of the feed-forward operation, the second chunk cannot be computed independently of the first chunk in a straight-forward manner. To avoid this, [14] used local-collisions. Namely, the first chunk includes two neutral words, and the attacker chooses neutral words so that the impact of changing one neutral word is always cancelled by the other neutral word. This fixes the feed-forward value to a constant, and thus independent computations can be carried out.

4 New Analytic Tool: Initial-Exchange Technique

In this section, we explain the *initial-exchange* technique, which exchanges the message-word positions located in the top of the steps across the right and left

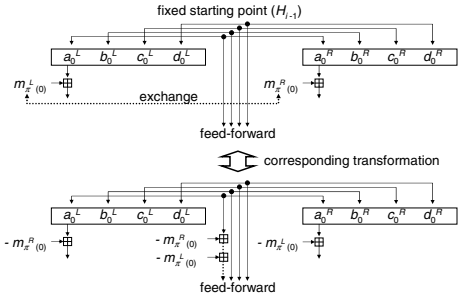

Fig. 3. Basic initial-exchange

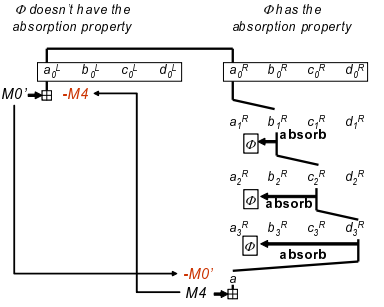

Fig. 4. Extended initial-exchange 1

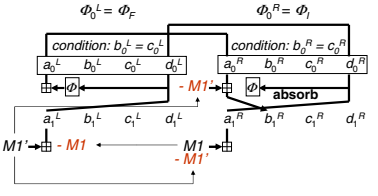

Fig. 5. Extended initial-exchange 2

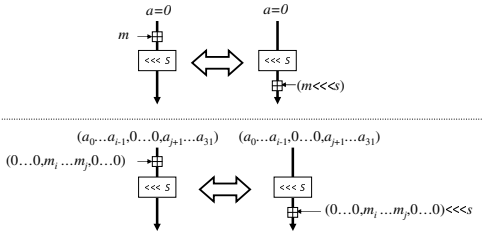

Fig. 6. Exchanging addition positions over a rotation. Top is for the standard initial-exchange. Bottom is for the partial-bit initial-exchange.

branches. This technique can be applied to both of RIPEMD-128 and RIPEMD-160. In this section, we explain the technique based on RIPEMD-128. We introduce the basic concept in Sect. 4.1 and extend the idea in Sect. 4.2.

4.1 Basic Idea of the Initial-Exchange Technique

We explain the basic idea; how to exchange the positions of $m_{\pi^L(0)}$ and $m_{\pi^R(0)}$. The idea is illustrated in Figure 3. In the standard computation, the value of H_{i-1} is fixed, and we compute CF^L by using $m_{\pi^L(0)}$ and CF^R by using $m_{\pi^R(0)}$. We transform this computation by exchanging the order of additions so that the positions of $m_{\pi^L(0)}$ and $m_{\pi^R(0)}$ are exchanged (bottom of Figure 3). This enables us to compute CF^L and CF^R independently for more steps. Note that the value of H_i^a used in the feed-forward operation is affected by both of $m_{\pi^L(0)}$ and $m_{\pi^R(0)}$, and thus, we cannot fix it until we fix $m_{\pi^L(0)}$ and $m_{\pi^R(0)}$. However, we can still partially compute the feed-forward operation for other variables.

4.2 Extension of the Initial-Exchange Technique

The basic idea only exchanges the messages in the first steps. By considering absorption properties of Φ_j^L and Φ_j^R, we can exchange messages in various

Table 3. Summary of the initial-exchange technique for RIPEMD-128

Left (Φ_F)	Right (Φ_I)	FF effect	Conditions for M'_i	Conditions for M_i	comments
M'_0	M_0	H^a	—	—	
M'_0	M_1	H^a,H^d	—	$b_0 = c_0$	
M'_0	M_2	H^a,H^c	$d_1 = 0$	$d_0 = 1, b_1 = c_1$	partial bit condition
M'_0	M_3	H^a,H^b	$d_1 = 0, d_2 = 1$	$d_0 = 0, d_1 = 1, b_2 = c_2$	partial bit condition
M'_0	M_4	H^a	same as right	$d_1 = 0, d_2 = 1, b_3 = c_3$	
M'_0	M_8	H^a	same as right	$d_1 = 0, d_2 = 1, b_3 = c_3$, $d_5 = 0, d_6 = 1, b_7 = c_7$	
M'_1	M_0	H^a,H^d	$b'_0 = c'_0$	—	
M'_1	M_1	H^a,H^d	$b'_0 = c'_0$	$b_0 = c_0$	same condition

On the 1st round (similar on the 4th round)

Left (Φ_H)	Right (Φ_G)	FF effect	Conditions for M'_i	Conditions for M_i	comments
M'_0	M_0	H^a	—	—	
M'_0	M_1	H^a,H^d	—	$b_0 = 1$	
M'_0	M_2	H^a,H^c	$c_1 = d_1$	$b_0 = 0$, $b_1 = 1$	partial bit condition
M'_0	M_3	H^a,H^b	$c_1 = d_1, b_2 = 0$	$c_0 = d_0$, $b_1 = 0$, $b_2 = 1$	partial bit condition
M'_0	M_4	H^a	same as right	$c_1 = d_1$, $d_2 = 0$, $b_3 = 1$	
M'_0	M_8	H^a	same as right	$c_1 = d_1$, $d_2 = 0$, $b_3 = 1$, $c_5 = d_5$, $d_6 = 0$, $b_7 = 1$	
M'_1	M_0	H^a,H^d	$b'_0 = 0$,$c'_0 = 1$	—	
M'_1	M_1	H^a,H^d	$b'_0 = 0$,$c'_0 = 1$	$b_0 = 1$	partial bit condition

On the 3rd round (similar on the 2nd round)

positions. Note that the absorption property is the one where the output of $\Phi(X, Y, Z)$ can be independent of one input variable. For example, the output of $\Phi_I(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$ can be independent of X by fixing Z to 0. It is well-known that Φ_G and Φ_I have absorption properties [2].

The first round of CF^R uses Φ_I, which has the absorption property. In such a case, messages words located in the second or latter step can be exchanged. Figure 4 shows an example where $M4$ and $M0'$ are exchanged. Mi is a message word $m_{\pi^R(i)}$ for a branch with the absorption property and Mi' is $m_{\pi^L(i)}$ for the other branch.

Moreover, even if Φ does not have the absorption property, we may exchange the message words in a few steps from the initial step. For example, we consider exchanging message words of the second steps, which is illustrated in Figure 5. Φ_j^L is Φ_F, which does not have the absorption property. Therefore, the impact of changing the value of $M1'$ always go through Φ_0^L. However, we still can apply the corresponding transformation by fixing Φ_0^L as a simple function. In Figure 5, we guarantee that the output of Φ_0^L is always $-M1'$ by setting the condition $b_0^L = c_0^L$ and $d_0^L = -M1'$. On the other hand, because the corresponding condition $b_0^R = c_0^R$ also makes the absorption property for Φ_0^R, we can apply the corresponding transformation and thus can exchange the positions of these message words.

Note that when we exchange positions of message-word additions over a bit-rotation, we need to set conditions as shown in Figure 6 in order to avoid the uncontrolled carry. We confirmed all of our attacks could satisfy this restriction.

Besides the above two examples, many other extensions of initial-exchange are possible. Some of such extension, for example the case shown in Figure 8 which will be explained later, require more complicated analysis to exchange message words. In several steps of this example, two input variables to Φ are changed depending on different neutral words. It is known that Φ used in RIPEMD-128 cannot absorb independent changes of two different input variables, and thus the initial-exchange cannot be applied directly. To overcome this problem, we adopt a *partial-bit initial-exchange*, where only a part of bits in neutral words are changed so that the active-bit positions of two input chaining variables do not overlap each other. This enables us to use the *cross-absorption property* proposed by Sasaki *et al.*[20] to absorb the changes of two input variables of Φ. Finally, we can exchange message words even in such a complicated case.

Table 3 summarizes the initial-exchange technique applied to each round of RIPEMD-128. The first two columns show the message words where we exchange their positions. The third column shows feed-forward variables which cannot be fixed in advance. We denote by "FF effect" such an effect. The forth and fifth columns list the conditions to set up the absorption properties for Φ_F or Φ_H.

5 Attacks on RIPEMD-128

5.1 Attack on the First 33 Steps of RIPEMD-128

With the initial-exchange technique explained in Sect. 4, we attack the first 33 steps of RIPEMD-128. The chunk separation for the first 33 steps is shown in Figure 7. Note that for all attacks in this paper, we searched for the chunk separations by hand. In this attack, m_2 is a neutral word for computing CF^L and m_0 is for CF^R. The positions of $m_{\pi^L(0)}$ and $m_{\pi^R(5)}$ are exchanged with the initial-exchange technique, and the last 8 steps of CF^L and the last 2 steps of CF^R, in total 10 steps are skipped in the partial-matching phase.

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
	IE	first chunk (depends on m_2)														
index R	5	14	7	(0)	9	(2)	11	4	13	6	15	8	1	10	3	12
						IE	second chunk (depends on m_0)									

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 (depends on m_2)									skip						
index R	6	11	3	7	(0)	13	5	10	14	15	8	12	4	9	1	(2)
	second chunk (depends on m_0)															skip

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

"IE" represents that the message positions will be exchanged with the initial-exchange technique.

Fig. 7. Chunk separation for the first 33 steps of RIPEMD-128

Set up for the initial-exchange technique. Because this attack includes another neutral word $m_{\pi^R(3)}$ in the initial-exchange section, the construction, especially selection of the active-bit positions of neutral words is complicated. The details of the construction is shown in Figure 8. As a result of our by-hand analysis, we determine that 9 bits (bit positions 23–31) of m_2 and 9 bits (bit positions 2–10) of m_0 are active. This avoids the overlap of the active bit positions for Φ_j, and thus we can absorb the impact of changes of these bits with absorption or cross-absorption properties [20]. Please refer to Table 3 for conditions to achieve these properties. With this effort, the positions of $m_{\pi^L(0)} = m_0$ and $m_{\pi^R(5)} = m_2$ are exchanged, hence CF^L and CF^R can be computed independently by using 9 free bits of m_2 and m_0, respectively. Note that the active bits of m_0 make the 9 bits (bit positions 2–10) of a feed-forward value H_a^i unfixed. In other words, 23 bits (bit positions 0–1 and 11–31) of H_a^i are fixed. Similarly, m_2 makes 9 bits (bit positions 14–22) of a feed-forward value H_d^i unfixed and 23 bits (bit positions 0–13 and 23–31) of H_d^i are fixed.

Partial-matching phase. Computation for steps 25 to 32 of CF^L and 31 and 32 of CF^R are performed only partially. Because each neutral word has 9 active bits, we need to match at least 9 bits of results from each chunk. Details of the partial-computations are shown in Figure 9. We compute 23 bits of d_{28}^L, 23 bits of c_{28}^L, and 14 bits of b_{28}^L in the first chunk. We denote these partially computed bits in the first chunk by α. In the second chunk, we compute 4 bits of d_{28}^L, 11 bits of c_{28}^L, and 11 bits of b_{28}^L, which are denoted by β. Note that α and β include 15 bit positions in common (4 bits of d_{28}^L, 5 bits of c_{28}^L, and 6 bits of b_{28}^L).

In the computations of α and β, we often compute the modular additions without knowing the carry from the lower bit positions. For example in Figure 9, to compute the bit positions 0–13 and 23–31 of b_{26}^L, we compute the addition of

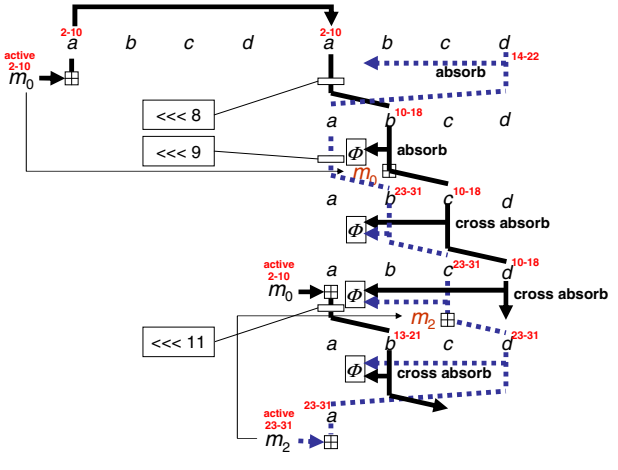

Fig. 8. Construction of the initial-exchange for the first 33 steps of RIPEMD-128

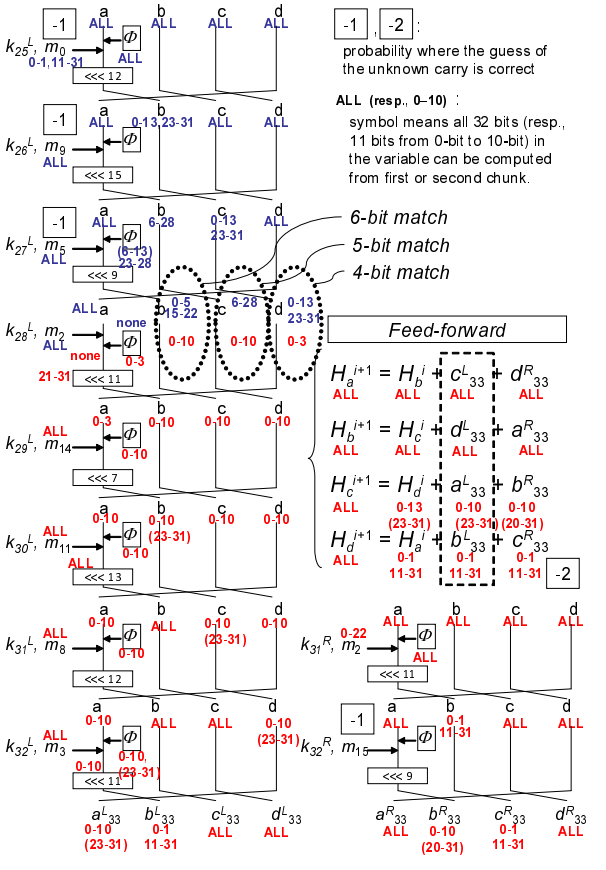

Fig. 9. Partial-match for the first 33 steps of RIPEMD-128

m_0 where only bit positions 0–1 and 11–31 are known. In this case, we obtain two candidates of bit positions 11–31 of the addition results, because we need to consider two carry patterns from bit positions 10 to 11. The computations of α and β require such a trick 3 times and 3 times respectively as marked in Figure 9. Hence, the number of pairs where we check the match will increase 2^6 times. To filter out these wrong candidates efficiently, we need to check the match of additional 6 bits of the results from each chunk. (9 bits for the standard meet-in-the-middle and 6 bits for filtering wrong candidates, in total 15 bits.) Because it can match 15 bits, our attack can filter out wrong candidates efficiently, and has the same efficiency as the standard meet-in-the-middle attack.

Attack procedure

1. Fix message words and chaining variables so that the set up for the initial-exchange technique and the padding for 2-block messages are satisfied. In the followings, every time we are short of freedom degrees, we go back to this step and restart the procedure by changing the values fixed in this step.

2. For 9 active bits (positions 23–31) of m_2, compute the first chunk. For each m_2, we obtain 2^3 candidates of α due to the unknown carry. Hence, we obtain $2^9 \cdot 2^3 = 2^{12}$ candidates of α. Store them in a table, and sort the table.
3. For each of 9 active bits (positions 2–10) of m_0, compute the second chunk to obtain β. For each of m_0, we obtain 2^3 candidates of β due to the unknown carry. For 2^9 values of m_0, we obtain $2^9 \cdot 2^3 = 2^{12}$ candidates of β.
4. For each β, check the match of 15 bits described in Figure 9 with all 'α's stored in the table.
5. If they match, with the corresponding m_0 and m_2, check the correctness of the carry assumptions step by step.
6. If all carry assumptions are correct, compute the remaining $128 - 15 = 113$ bits and check if all 113 bits will match or not.
7. If all bits match, the corresponding p_0 and message words are the pseudo-preimage of the given hash value.

Complexity evaluation. Steps 2 and 3 require 2^9 computations of the half of the compression function. Step 4 matches the 15 bits of $2^{12} \cdot 2^{12} = 2^{24}$ pairs and $2^{24} \cdot 2^{-15} = 2^9$ pairs will remain. In Step 5, a pair satisfies all the carry assumptions with a probability of 2^{-6}, hence 2^3 pairs will remain. So far, we obtain 2^3 pairs whose 15 bits match. Therefore, by iterating Step 6 2^{110} times, we will find a pair where all bits match, namely, a pseudo-preimage is obtained.

The complexity of the pseudo-preimage attack is $2^9 \cdot 2^{110} = 2^{119}$, and we need 2^{12} memory for Step 2. Note that Step 3 can be performed sequentially, and thus we do not need 2^{13} memory. Finally, this pseudo-preimage attack can be converted to a preimage attack with a complexity of $2^{\frac{119+128}{2}+1} = 2^{124.5}$ by using the conversion algorithm explained in Sect. 3.1.

5.2 Attack on Intermediate 35 Steps of RIPEMD-128

To attack intermediate steps, the local-collision approach [14] explained in Sect. 3.3 is more effective than the initial-exchange approach. The chunk separation for the intermediate 35 steps is shown in Figure 10, where neutral words are (m_0, m_6) and m_2. We make local collisions in Steps 21 to 25 of CF^L. For the partial-matching, we only activate bit positions 16 to 31 of m_0 and 5 to 20 of m_2.

Set up for the attack. Fix chaining variables between p_{21}^L and p_{26}^L as shown in Table 4, where $C_0, C_1, \ldots, C_4$ are arbitrary fixed values, **0** denotes 0 (=`0x00000000`), **1** denotes -1 (=`0xffffffff`), and $*$ denotes a variable that changes depending on the values of (m_0, m_6). Compute m_{15}, m_3, and m_{12} with a equation $m_{\pi^L(j)} \leftarrow (b_{j+1}^L \ggg s_j^L) - k_j^L - \Phi_j^L(b_j^L, c_j^L, d_j^L) - a_j^L$, so that the values fixed in Table 4 can be achieved. This equation can be computed without fixing the value of $*$ due to the absorption property. Now, every time we choose the value of m_6, we can make a local collision by adaptively choosing m_0 as follows: $m_0 \leftarrow (b_{26}^L \ggg s_{25}^L) - k_{25}^L - \Phi_{25}^L(b_{25}^L, c_{25}^L, d_{25}^L) - ((a_{21}^L + \Phi_{21}^L(b_{21}^L, c_{21}^L, d_{21}^L) + m_6 +$

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								fix				
index R	5	14	7	(0)	9	(2)	11	4	13	(6)	15	8	1	10	3	12
				excluded					fix			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
			fix				local-collision				fix		second chunk			
index R	(6)	11	3	7	(0)	13	5	10	14	15	8	12	4	9	1	(2)
							first chunk									skip

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

Fig. 10. Chunk separation for intermediate 35 steps of RIPEMD-128

Table 4. Set up for the local-collision

j	$m_{\pi^L(j)}$	a_j^L	b_j^L	c_j^L	d_j^L
21	m_6	C_0	C_1	C_1	C_2
22	m_{15}	C_2	$*$	C_1	C_1
23	m_3	C_1	**0**	$*$	C_1
24	m_{12}	C_1	**1**	**0**	$*$
25	m_0	$*$	C_3	**1**	**0**
26		**0**	C_4	C_3	**1**

$k_{21}^L) \lll s_{21}^L$). With this equation, fix the bit positions 0 to 15 of m_6 and compute the corresponding bits of m_0 so that local-collision can be formed. Then, fix the values of m_j, where $0 \leq j \leq 15, j \notin \{15, 3, 12, 0, 2, 6\}$, to randomly chosen values. Finally, compute the fixed part $(p_7^L, \ldots, p_{20}^L, p_{27}^L, p_{28}^L, p_8^R, p_9^R)$ of the attack target. Store the randomly chosen values and corresponding p_{28}^L and p_9^R.

Attack procedure

1. Carry out the set up procedure.
2. For all active bits (bit positions 16 to 31) of m_6, do as follows.
   (a) Compute the values of m_0 so that the local-collision can be formed.
   (b) Compute $p_{j+1}^R \leftarrow R_j^R(p_j^R, m_{\pi^R(j)})$ for $j = 9, 10, \ldots, 30$.
   (c) Compute the bit positions 0 to 15 of b_{32}^R by using $R_{31}^R(p_{31}^R, m_{\pi^R(31)})$ with fixed bits (bit positions 0 to 4 and 21 to 31) of $m_{\pi^R(31)} = m_2$.
   (d) Store the values of m_6, m_0, p_{31}^R and the lower half bits of b_{32}^R in a table.
3. For all active bits (bit positions 5 to 20) of m_2, do as follows.
   (a) Compute $p_{j+1}^L \leftarrow R_j^L(p_j^L, m_{\pi^L(j)})$ for $j = 28, 29, \ldots, 41$,
   (b) Compute p_{42}^R by using feed-forward equations.
   (c) Compute $p_j^R \leftarrow R_j^{R(-1)}(p_j^R, m_{\pi^R(j)})$ for $j = 41, 40, \text{and}, 39$.

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
	IE	first chunk (depends on m_7)														
index R	15	5	1	(3)	(7)	14	6	9	11	8	12	2	10	0	4	13
					IE	second chunk (depends on m_3)										

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 (depends on m_7)									skip						
index R	8	6	4	1	(3)	11	15	0	5	12	2	13	9	(7)	10	14
	second chunk (depends on m_3)													skip		

Fig. 11. Chunk separation for the last 32 steps of RIPEMD-128

(d) Compute bit positions 0 to 15 of b_{32}^R by using $R_j^{R(-1)}(p_{j+1}^R, m_{\pi^R(j)})$ for $j = 38, 37, \ldots, 32$ with fixed bits (positions 0 to 15) of $m_{\pi^R(38)} = m_6$.
(e) Check if the computed results (bit positions 0 to 15 of b_{32}^R) match the one of the values stored at Step 2d.
(f) If they match, compute all values of p_{32}^R from both chunks with the corresponding m_0, m_6, and m_2.
(g) If all bits match, the corresponding message words and p_7 are the pseudo-preimage of the given output value.

Complexity evaluation. Complexity for the set up part can be ignored because it is less often repeated. Complexity for Steps 2 and 3 are roughly 2^{16} computations of the half compression function, respectively. In the matching part, we check the match of 2^{16} items of 16 bits stored at Step 2d and 2^{16} items of 16 bits computed at Step 3d. Therefore, we obtain 2^{16} pairs where 16 bits of p_{32}^R are matched. Other 112 bits are randomly satisfied. Therefore, by repeating the above procedure $2^{112-16} = 2^{96}$ times, we will find a matched pair. The total complexity is $2^{16} \cdot 2^{96} = 2^{112}$ compression function computations. Note that for Step 2d, we need 2^{16} memory. Finally, this pseudo-preimage attack can be converted to a second-preimage attack with a complexity of $2^{\frac{112+128}{2}+1} = 2^{121}$.

Note that in the set up procedure using Table 4, we can make the freedom degrees for m_{15} instead of C_2. This enables us satisfy the padding string located in m_{13}, m_{14}, and m_{15}. Therefore, this attack can generate preimages.

5.3 Attack on the Last 32 Steps of RIPEMD-128

We use the initial-exchange technique to attack the last 32 steps of RIPEMD-128. The chunk separation is shown in Figure 11.

The form of the initial-exchange used in this attack is exactly the same as the one in Figure 4. Hence we omit the details. Different from the attack for the first 33 steps, we do not have to use the partial-bit initial-exchange. However, due to the large number of skipped steps in the partial-matching phase, many bits of neutral words need to be fixed. In this attack, we make 7 bits (positions 18–24) of m_3 and 9 bits (positions 0–3 and 27–31) of m_7 active. Then, in the partial-matching phase, we can match 14 bits as shown in Figure 12. Note that the partial-computation with unknown carry effect is performed 8 times.

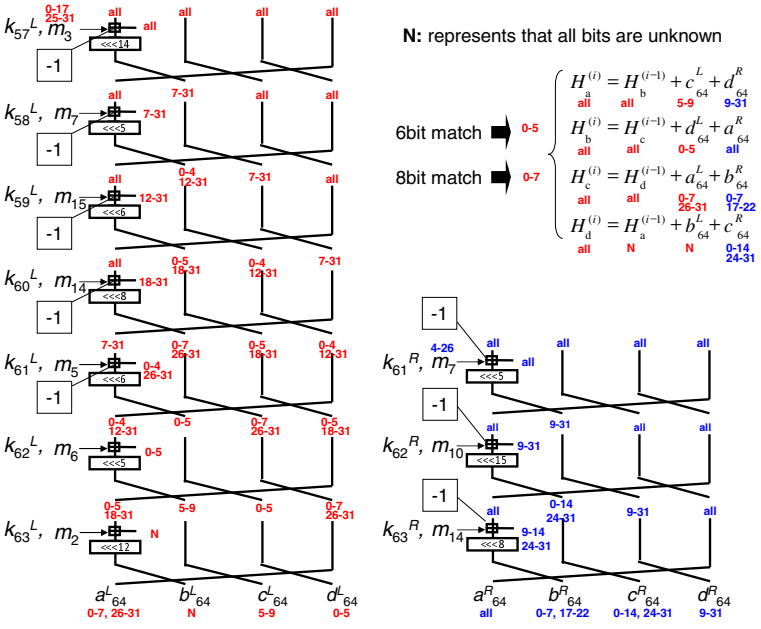

Fig. 12. Partial-match for the last 32 steps of RIPEMD-128

Attack summary. To avoid the redundancy, we omit the details of the attack procedure. In summary, the first and second chunks include 7 and 9 active bits, respectively. During the partial-computation, the number of candidates will increase 2^8 times, and we can match 14 bits of the results from two chunks. As a result, with a complexity of $2^7 + 2^9$ half compression function computations, we will obtain $2^{7+9+8-14} = 2^{10}$ matched pairs. Then, with a complexity of 2^{10}, we can check the correctness of the carry assumption and $2^{10-8} = 2^2$ pairs will remain. By iterating this procedure 2^{112} times, we will find a pseudo-preimage. The attack complexity is $(\frac{1}{2} \cdot 2^7 + \frac{1}{2} \cdot 2^9 + 2^{10}) \cdot 2^{112} \approx 2^{122.39}$ and we use 2^{12} memory (for 2^7 values of m_3, we obtain 2^5 candidates of α). In this attack, the value of m_{15} cannot be controlled because we need to fix it to a certain constant value in order to achieve the absorption property used in the initial-exchange technique. Therefore, with the conversion algorithm explained in Section 3.1, this can only be a second-preimage attack with a complexity of $2^{\frac{122.39+128}{2}+1} \approx 2^{126.20}$.

6 Attacks on RIPEMD-160

RIPEMD-160 follows the same structure as RIPEMD-128. However, several different characteristics give influence to the attack strategy. Specifically, the direct addition of e_j to update b_{j+1} increases the resistance against our attacks. In this section, we first summarize our observations particular to RIPEMD-160.

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
	IE	first chunk (depends on m_{14})														
index R	5	**14**	7	**0**	9	2	11	4	13	6	15	8	1	10	3	12
		IE	second chunk (depends on m_0)													

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 (depends on m_{14})									skip					excluded	
index R	6	11	3	7	**0**	13	5	10	**14**	15	8	12	4	9	1	2
	2nd chunk (depends on m_0)									skip					excluded	

Fig. 13. Chunk separation for the first 30 steps of RIPEMD-160

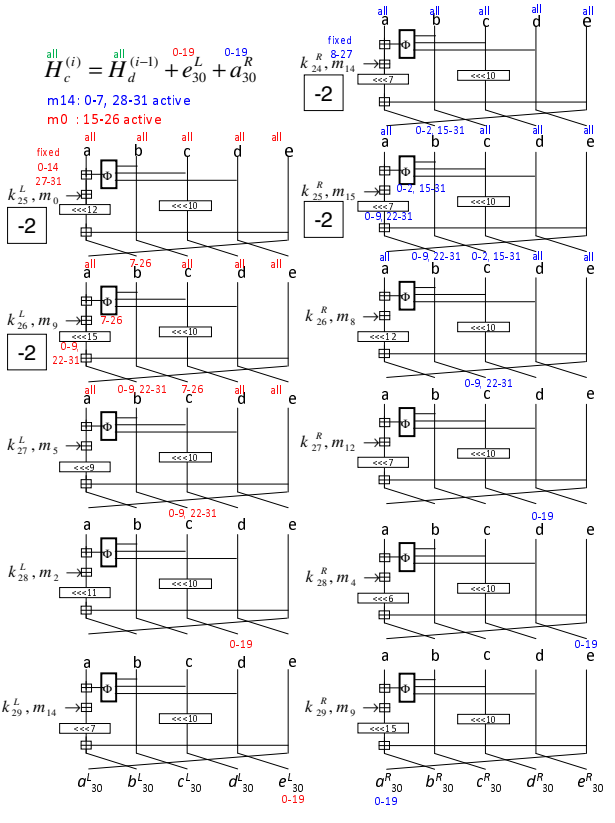

Fig. 14. Partial-match for the first 30 steps of RIPEMD-160

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
excluded			IE			first chunk (depends on m_3)										
index R	8	6	4	1	(3)	(11)	15	0	5	12	2	13	9	7	10	14
excluded					IE	second chunk (depends on m_{11})										

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 (depends on m_3)												skip			
index R	12	15	10	4	1	5	8	7	6	2	13	14	0	(3)	9	(11)
	second chunk (depends on m_{11})													skip		

Fig. 15. Chunk separation for the last 31 steps of RIPEMD-160

- Due to the addition of e_j to b_{j+1}, 3 message words are necessary to form a local-collision. This makes the local-collision approach inefficient.
- Basic strategy of the initial-exchange technique can be applied to RIPEMD-160. However, the direct addition of e_j, which can be regarded as a function without the absorption property, makes its extension very hard.
- The number of chaining variables increases from RIPEMD-128. This enables attackers to skip more steps in the partial-matching phase.
- Φ in the first round do not have the absorption property in both sides. This makes the attack from the first steps harder than RIPEMD-128.

6.1 Attack on the First 30 Steps of RIPEMD-160

The chunk separation for the first 30 steps of RIPEMD-160 is shown in Figure 13. Due to the difficulties of applying the initial-exchange technique in the first round of RIPEMD-160, the positions where we can exchange the message words are limited. On the other hand, we can skip more steps in the partial-matching phase. As a result, the number of attacked steps reaches 30 steps. In this attack, we make 12 bits of m_{14} and 12 bits of m_0 active. In the partial-matching phase, we consider unknown carry effects 8 times and match the results in 20 bits, which results in the same efficiency as the standard meet-in-the-middle attack. Details of the partial-matching phase is described in Figure 14. Finally, the pseudo-preimages can be found with a complexity of 2^{148} and with a memory of 2^{16}. Note that we cannot satisfy the padding because m_{14} is a neutral word. Finally, this attack is converted to a second preimage attack with a complexity of 2^{155}.

6.2 Attack on the Last 31 Steps of RIPEMD-160

The chunk separation for the last 31 steps is shown in Figure 15. We make 12 bits of m_3 and 12 bits of m_{11} active. The initial-exchange construction is depicted in Figure 16. In the partial-matching phase, we consider the match of $15+8=23$ bits in the feed-forward equation with 5 unknown carry effects. Note that increasing match bits is possible by considering more unknown carry effects. However, because this does not impact to the final complexity, we simply match only 23 bits. Details of the partial-matching phase is described in Figure 17.

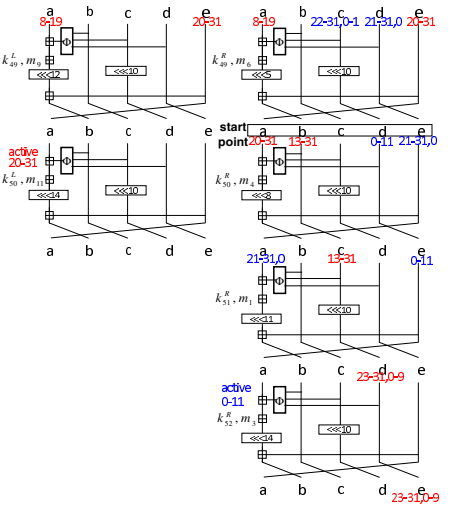

Fig. 16. Initial-exchange for last 31 steps

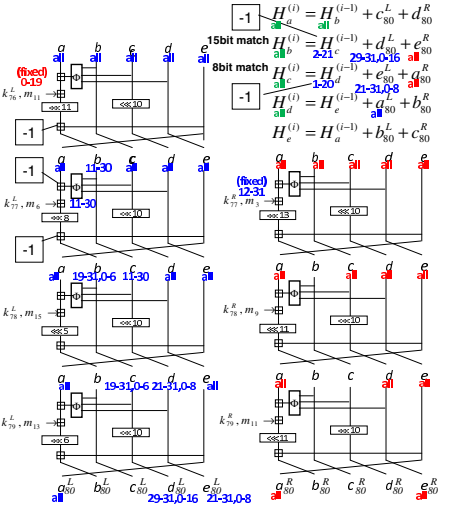

Fig. 17. Partial-match for the last 31 steps

Pseudo-preimages can be found with a complexity of 2^{148} and 2^{17} memory. Finally, the attack is converted to a preimage attack with a complexity of 2^{155}.

7 Concluding Remarks

We presented the first results on preimage resistance of RIPEMD-128 and -160. By using the initial-exchange technique, we discovered the (second) preimage attacks on the first 33, intermediate 35, and the last 32 steps of RIPEMD-128, and the first 30 and the last 31 steps of RIPEMD-160.

RIPEMD-128 and -160, have been believed to be more secure than RIPEMD. This may be true with respect to the collision resistance due to the differences between CF^L and CF^R. However, meet-in-the-middle attacks do not care most of the components except for the message order, and their security could be the same level as RIPEMD with respect to the preimage resistance.

References

1. U.S. Department of Commerce, National Institute of Standards and Technology: Federal Register/vol. 72, No. 212/Friday, November 2, 2007/Notices (2007)
2. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
3. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
4. 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)

5. Mendel, F., Rijmen, V.: Cryptanalysis of the tiger hash function. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 536–550. Springer, Heidelberg (2007)
6. Guo, J., Ling, S., Rechberger, C., Wang, H.: Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016 (2010)
7. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound distinguishers: Results on the full whirlpool compression function. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 126–143. Springer, Heidelberg (2009)
8. 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)
9. 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)
10. RIPE Integrity Primitives: Integrity Primitives for Secure Information Systems, Final RIPE Report of RACE Integrity Primitives Evaluation, RIPE-RACE 1040 (1995)
11. International Organization for Standardization: ISO/IEC 10118-3:2004, Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions (2004)
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. 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)
14. Sasaki, Y., Aoki, K.: Meet-in-the-middle preimage attacks on double-branch hash functions: Application to RIPEMD and others. In: Boyd, C., González Nieto, J. (eds.) ACISP 2009. LNCS, vol. 5594, pp. 214–231. Springer, Heidelberg (2009)
15. Hong, D., Chang, D., Sung, J., Lee, S.-J., Hong, S.H., Lee, J.S., Moon, D., Chee, S.: A new dedicated 256-bit hash function: FORK-256. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 195–209. Springer, Heidelberg (2006)
16. Aoki, K., Sasaki, Y.: Preimage attacks on one-block MD4, 63-step MD5 and more. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 103–119. Springer, Heidelberg (2009)
17. den Boer, B., Bosselaers, A.: An attack on the last two rounds of MD4. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 194–203. Springer, Heidelberg (1992)
18. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC Press, Boca Raton (1997)
19. Leurent, G.: MD4 is not one-way. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 412–428. Springer, Heidelberg (2008)
20. 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, Heidelberg (2009)