Cryptanalysis of Full RIPEMD-128

Franck Landelle[1] and Thomas Peyrin[2,⋆]

[1] DGA MI, France
[2] Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore
landelle.franck@laposte.net thomas.peyrin@gmail.com

Abstract. In this article we propose a new cryptanalysis method for double-branch hash functions that we apply on the standard RIPEMD-128, greatly improving over know results. Namely, we were able to build a very good differential path by placing one non-linear differential part in each computation branch of the RIPEMD-128 compression function, but not necessarily in the early steps. In order to handle the low differential probability induced by the non-linear part located in later steps, we propose a new method for using the freedom degrees, by attacking each branch separately and then merging them with free message blocks. Overall, we present the first collision attack on the full RIPEMD-128 compression function as well as the first distinguisher on the full RIPEMD-128 hash function. Experiments on reduced number of rounds were conducted, confirming our reasoning and complexity analysis. Our results show that 16 years old RIPEMD-128, one of the last unbroken primitives belonging to the MD-SHA family, might not be as secure as originally thought.

Key words: RIPEMD-128, collision, distinguisher, hash function.

1 Introduction

Recent impressive progresses in hash function cryptanalysis [25, 28, 29, 27] led to the fall of most standardized primitives, such as MD4, MD5, SHA-0 and SHA-1. All these algorithms share the same design rationale for their compression functions (i.e. they incorporate additions, rotations, xors and boolean functions in an unbalanced Feistel network), and we usually refer to them as the MD-SHA family. As of today, among this family only SHA-2, RIPEMD-128 and RIPEMD-160 remain unbroken.

The notation RIPEMD represents several distinct hash functions related to the MD-SHA-family, the first representative being RIPEMD-0 [2] that was recommended in 1992 by the European *RACE Integrity Primitives Evaluation* (RIPE) consortium. Its compression function basically consists in two MD4-like [20] functions computed in parallel (but with different constant additions for the two

⋆ Supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

branches), with 48 steps in total. Early cryptanalysis by Dobbertin on a reduced version of the compression function [9] seemed to indicate that `RIPEMD-0` was a weak function and this was fully confirmed much later by Wang *et al.* [27] who showed that one can find a collision for the full `RIPEMD-0` hash function with as few as 2^{16} computations.

However, in 1996, due to the cryptanalysis advances on `MD4` and on the compression function of `RIPEMD-0`, the original `RIPEMD-0` was reinforced by Dobbertin, Bosselaers and Preneel [10] to create two stronger primitives `RIPEMD-128` and `RIPEMD-160`, with 128/160-bit output and 64/80 steps respectively (two other less known 256 and 320-bit output variants `RIPEMD-256` and `RIPEMD-320` were also proposed, but with a claimed security level equivalent to an ideal hash function with a twice smaller output size). The main novelty compared to `RIPEMD-0` is that the two computation branches were made much more distinct by using not only different constants, but also different rotation values and boolean functions, which greatly hardens the attacker's task in finding good differential paths for both branches at a time. The security seems to have indeed increased since as of today no attack is known on the full `RIPEMD-128` or `RIPEMD-160` compression/hash functions and the two primitives are worldwide ISO/IEC standards [12].

Even though no result is known on the full `RIPEMD-128` and `RIPEMD-160` compression/hash functions yet, many analysis were conducted in the recent years. In [17], a preliminary study checked up to what extent can the known attacks [27] on `RIPEMD-0` apply to `RIPEMD-128` and `RIPEMD-160`. Then, following the extensive work on preimage attacks for `MD-SHA` family, [21, 19, 26] describe high complexity preimage attacks on up to 36 steps of `RIPEMD-128` and 31 steps of `RIPEMD-160`. Collision attacks were considered in [16] for `RIPEMD-128` and in [15] for `RIPEMD-160`, with 48 and 36 steps broken respectively. Finally, distinguishers based on non-random properties such as second-order collisions are given in [16, 22, 15], reaching about 50 steps with a very high complexity.

Our contributions. In this article, we introduce a new type of differential path for `RIPEMD-128` using one non-linear differential trail for both left and right branches and, in contrary to previous work, not necessarily located in the early steps (Section 3). The important differential complexity cost of these two parts is mostly avoided by using the freedom degrees in a novel way: some message words are used to handle the non-linear parts in both branches and the remaining ones are used to merge the internal states of the two branches (Section 4). Overall, we obtain the first cryptanalysis of the full 64-round `RIPEMD-128` hash and compression functions. Namely, we provide a distinguisher based on a differential property for both the full 64-round `RIPEMD-128` compression function and hash function (Section 5). Previously best-known results for non-randomness properties only applied to 52 steps of the compression function, 48 steps of the hash function. More importantly, we also derive a semi-free-start (SFS) collision attack on the full `RIPEMD-128` compression function (Section 5), significantly improving the previous free-start (FS) collision attack on 48 steps. Any further improvement of our techniques is likely to provide a practical SFS collision attack

Table 1. Summary of known and new results on RIPEMD-128 hash function

Function	Size	Key Setting	Target	#Steps	Complexity	Ref.
RIPEMD-128	128	comp. function	preimage	35	2^{112}	[19]
RIPEMD-128	128	hash function	preimage	35	2^{121}	[19]
RIPEMD-128	128	hash function	preimage	36	$2^{126.5}$	[26]
RIPEMD-128	128	comp. function	collision	48	2^{40}	[16]
RIPEMD-128	**128**	**comp. function**	**collision**	**60**	$2^{57.57}$	**new**
RIPEMD-128	**128**	**comp. function**	**collision**	**63**	$2^{59.91}$	**new**
RIPEMD-128	**128**	**comp. function**	**collision**	**Full**	$2^{61.57}$	**new**
RIPEMD-128	128	hash function	collision	38	2^{14}	[16]
RIPEMD-128	128	comp. function	non-rand.	52	2^{107}	[22]
RIPEMD-128	**128**	**comp. function**	**non-rand.**	**Full**	$2^{59.57}$	**new**
RIPEMD-128	128	hash function	non-rand.	48	2^{70}	[16]
RIPEMD-128	**128**	**hash. function**	**non-rand.**	**Full**	$2^{105.40}$	**new**

on the RIPEMD-128 compression function. In order to increase the confidence in our reasoning, we implemented independently the two main parts of the attack (the merge and the probabilistic part) and the observed complexity matched our predictions. Our results and previous works complexities are given in Table 1 for comparison.

2 Description of RIPEMD-128

RIPEMD-128 [10] is a 128-bit hash function that uses the Merkle-Damgård construction as domain extension algorithm: the hash function H is built by iterating a 128-bit compression function h that takes as input a 512-bit message block m_i and a 128-bit chaining variable cv_i: $cv_{i+1} = h(cv_i, m_i)$, where the message m to hash is padded beforehand to a multiple of 512 bits[3] and the first chaining variable is set to a predetermined initial value $cv_0 = IV$.

We refer to [10] for a complete description of RIPEMD-128. In the rest of this article, we denote by $[Z]_i$ the i-th bit of a word Z, starting the counting from 0. $\boxplus$ and $\boxminus$ represent the modular addition and subtraction on 32 bits, and $\oplus$, $\vee$, $\wedge$, the bitwise "exclusive or", the bitwise "or", and the bitwise "and" function respectively.

2.1 RIPEMD-128 compression function

The RIPEMD-128 compression function is based on MD4, with the particularity that it uses two parallel instances of it. We differentiate these two computation branches by left and right branch and we denote by X_i (resp. Y_i) the 32-bit word of left branch (resp. right branch) that will be updated during step i of the compression function. The process is composed of 64 steps divided into 4 rounds of 16 steps each in both branches.

Initialization. The 128-bit input chaining variable cv_i is divided into 4 words h_i of 32 bits each, that will be used to initialize the left and right branch 128-bit

[3] The padding is the same as for MD4: a "1" is first appended to the message, then x "0" bits (with $x = 512 - (|m| + 1 + 64 \pmod{512}))$) are added, and finally the message length $|m|$ coded on 64 bits is appended as well.

internal state: $X_{-3} = h_0$, $X_{-2} = h_1$, $X_{-1} = h_2$, $X_0 = h_3$, $Y_{-3} = h_0$, $Y_{-2} = h_1$, $Y_{-1} = h_2$, $Y_0 = h_3$

The message expansion. The 512-bit input message block is divided into 16 words M_i of 32 bits each. Each word M_i will be used once in every round in a permuted order (similarly to `MD4`) and for both branches. We denote by W_i^l (resp. W_i^r) the 32-bit expanded message word that will be used to update the left branch (resp. right branch) during step i. We have for $0 \leq j \leq 3$ and $0 \leq k \leq 15$: $W^l_{j\cdot 16+k} = M_{\pi^l_j(k)}$ and $W^r_{j\cdot 16+k} = M_{\pi^r_j(k)}$, where π^l_j and π^r_j are permutations.

The step function. At every step i, the registers X_{i+1} and Y_{i+1} are updated with functions f^l_j and f^r_j that depends on the round j in which i belongs:

$$X_{i+1} = (X_{i-3} \boxplus \Phi^l_j(X_i, X_{i-1}, X_{i-2}) \boxplus W^l_i \boxplus K^l_j)^{\lll s^l_i},$$
$$Y_{i+1} = (Y_{i-3} \boxplus \Phi^r_j(Y_i, Y_{i-1}, Y_{i-2}) \boxplus W^r_i \boxplus K^r_j)^{\lll s^r_i},$$

where K^l_j, K^r_j are 32-bit constants defined for every round j and every branch, s^l_i, s^r_i are rotation constants defined for every step i and every branch, Φ^l_j, Φ^r_j are 32-bit boolean functions defined for every round j and every branch. All these constants and functions, as well as the IV and the permutations π^l_j and π^r_j can be found in the original `RIPEMD-128` documentation [10].

The finalization. A finalization and a feed-forward is applied when all 64 steps have been computed in both branches. The four 32-bit words h'_i composing the output chaining variable are finally obtained by: $h'_0 = X_{63} \boxplus Y_{62} \boxplus h_1$, $h'_1 = X_{62} \boxplus Y_{61} \boxplus h_2$, $h'_2 = X_{61} \boxplus Y_{64} \boxplus h_3$, $h'_3 = X_{64} \boxplus Y_{63} \boxplus h_0$

3 A new family of differential paths for `RIPEMD-128`

3.1 The general strategy

The first task for an attacker looking for collisions in some compression function is to set a good differential path. In the case of `RIPEMD` and more generally double or multi-branches compression functions, this can be quite a difficult task because the attacker has to find a good path for all branches at the same time. This is exactly what multi-branches functions designers are hoping: it is unlikely that good differential paths exist in both branches at the same time when the branches are made distinct enough (note that the weakness of `RIPEMD-0` is that both branches are almost identical and the same differential path can be used for the two branches at the same time).

Differential paths in recent collision attacks on `MD-SHA` family are composed of two parts: a low probability non-linear part in the first steps and a high probability linear part in the remaining ones. Only the latter will be handled probabilistically and impact the overall complexity of the collision finding algorithm, since during the first steps the attacker can choose message words independently.

This strategy proved to be very effective because it allows to find much better linear parts than before by relaxing many constraints on them. The previous approaches for attacking `RIPEMD-128` [17, 16] are based on the same strategy, building good linear paths for both branches, but without including the first round (i.e. the first 16 steps). The first round in each branch will be covered by a non-linear differential path and this is depicted left in Figure 1. The collision search is then composed of two subparts, the first handling the low-probability non-linear paths with the message blocks (step ① and then the remaining steps in both branches are verified probabilistically (step ②).

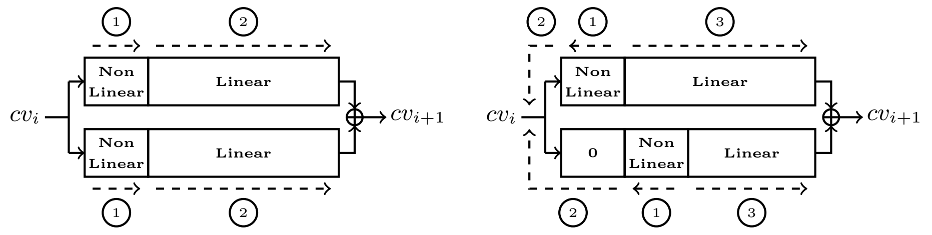

Fig. 1. The previous (left-hand side) and new (right-hand side) approach for collision search on double-branch compression functions.

This differential path search strategy is natural when one will handle the non-linear parts in a classic way (i.e. computing only forward) during the collision search, but in Section 4 we will describe a new approach for using the available freedom degrees provided by the message words in double-branch compression functions (see right in Figure 1): instead of handling the first rounds of both branches at the same time during the collision search, we will satisfy them independently (step ①), then use some remaining free message words to merge the two branches (step ②) and finally handle the remaining steps in both branches probabilistically (step ③). This new approach broadens the search area of good linear differential parts, and provides us better candidates in the case of `RIPEMD-128`.

3.2 Finding a good linear part

Since any active bit in a linear differential path (i.e. a bit containing a difference) is likely to cause many conditions in order to control its spread, most successful collision searches start with a low-weight linear differential path, therefore reducing the complexity as much as possible. `RIPEMD-128` is no exception, and because every message word is used once in every round of every branch in `RIPEMD-128`, the best would be to insert only a single-bit difference in one of them. This was considered in [16], but the authors concluded that none of all single-word differences leads to a good choice and they eventually had to utilize one active bit in two message words instead, therefore doubling the amount of differences inserted during the compression function computation and reducing the overall number of steps they could attack. By relaxing the constraint that both non-linear parts must necessarily be located in the first round, we show that a single-word difference in M_{14} is actually a very good choice.

Boolean functions. Analyzing the various boolean functions in `RIPEMD-128` rounds is very important. Indeed, there are three distinct functions: `XOR`, `ONX` and `IF`, with all very distinct behavior. The function `IF` is non-linear and can absorb differences (one difference on one of its input can be blocked from spreading to the output by setting some appropriate bit value conditions). In other words, one bit difference in the internal state during an `IF` round can be forced to create only a single bit difference 4 steps later, thus providing no diffusion at all. In the contrary, `XOR` is arguably to most problematic function in our situation because it can not absorb any difference. Thus, one bit difference in the internal state during an `XOR` round will double the number of bit differences every step and quickly lead to an unmanageable amount of conditions. Moreover, the linearity of the `XOR` function makes it problematic when using the non-linear part search tool that strongly leverages non-linear behavior to obtain a solution. In between, the `ONX` function is non-linear for two inputs and can absorb difference up to some extent. We can easily conclude that the goal for the attacker will be to locate the biggest proportion of differences in the `IF` or if needed in the `ONX` functions, and try to avoid the `XOR` parts as much as possible.

Choosing a message word. We would like to find the best choice for the single-message word difference insertion. The `XOR` function located in the 4^{th} round of right branch must be avoided, so we are looking for a message word that is incorporated either very early (so we can propagate the difference backward) or very late (so we can propagate the difference forward) in this round. Similarly, the `XOR` function located in the 1^{st} round of left branch must be avoided, so we are looking for a message word that is incorporated either very early (for a FS collision attack) or very late (for a SFS collision attack) in this round as well. It is easy to check that M_{14} is a perfect candidate, being the last inserted in 4^{th} round of right branch and the second-to-last in 1^{st} round of left branch.

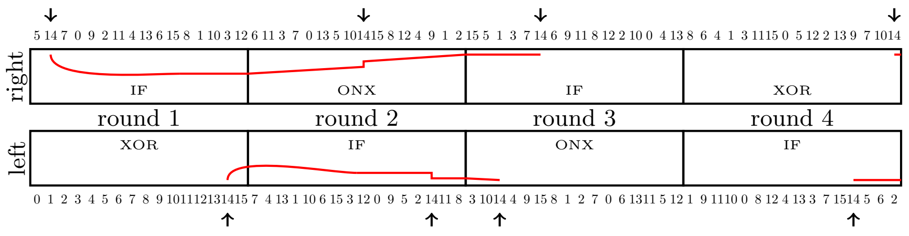

Fig. 2. The shape of our differential path for `RIPEMD-128`. The numbers are the message words inserted at each step and the red curves represent the rough amount differences in the internal state during each steps. The arrows show where the bit differences are injected with M_{14}.

Building the linear part. Once we chose that the only message difference will be a single bit in M_{14}, we need to build the whole linear part of the differential path in the internal state. By linear we mean that all modular additions will be modeled as a bitwise `XOR` function. Moreover, when a difference is input of a boolean function, it is absorbed when possible in order to remain as low weight as possible (though, for a few special bit positions it might be more interesting to not absorb the difference if it can erase another difference in later steps). We give the rough skeleton of our differential path in Figure 2. Both differences inserted in the 4^{th} round of left and right branches are simply propagated forward for a few steps and we are very lucky that this linear propagation leads to two final internal states whose difference can be mutually erased after application of the compression function finalization and feed-forward. All differences inserted in the 3^{rd} and 2^{nd} rounds of left and right branches are propagated linearly backward and will be later connected to the bit difference inserted in the 1^{st} round by the non-linear part. Note that since a non-linear part usually has a low differential probability, we will try to make it as thin as possible. No difference will be present in the input chaining variable, so the trail is well suited for a SFS collision attack. We had to choose the bit position for the message M_{14} difference insertion and among the 32 possible choices, the most significant bit was selected because it is the one maximizing the differential probability of the linear part we just built (this finds an explanation by the fact that at the most significant bit position many conditions due to carry control in modular additions are avoided).

3.3 The non-linear differential part search tool

Finding non-linear differential path is a very complex task, but we implemented a tool similar to [4] for `SHA-1` in order to perform this task in an automated way. Since `RIPEMD-128` also belongs to the `MD-SHA` family, the original technique works well, in particular when used in a round with a non-linear boolean function such as `IF`.

We have to find one non-linear part in each branch and note that they can be handled independently. We included the special constraint that the non-linear parts should be as thin as possible (i.e. spreading on the fewer possible amount of steps), so as to later reduce the overall complexity (linear parts have higher differential probability than non-linear ones).

3.4 The final differential path skeleton

Applying our non-linear part search tool and reusing notations from [4], we obtain the differential path in Figure 3, for which we provide at each step i the differential probability $\mathrm{P}^l[i]$ and $\mathrm{P}^r[i]$ of left and right branch respectively. Also, we give for each step i the accumulated probability $\mathrm{P}[i]$ starting from last step, i.e. $\mathrm{P}[i] = \prod_{j=63}^{j=i}(\mathrm{P}^r[j] \cdot \mathrm{P}^l[j])$.

One can check that the trail has differential probability $2^{-85.09}$ (that is $\prod_{i=0}^{63} \mathrm{P}^l[i] = 2^{-85.09}$) in the left branch and 2^{-145} (i.e. $\prod_{i=0}^{63} \mathrm{P}^r[i] = 2^{-145}$)

```
Step              Xi                Π^l_i  P^l[i]               Yi                 Π^r_i  P^r[i]   P[i]

 -3: --------------------------------
 -2: --------------------------------
 -1: --------------------------------
 00: -----------------------0-------- |  0   0.00 | -----------------------0-------- |  5  -1.00 | -230.09
 01: -------------------------------- |  1   0.00 | -----------------------1-------- | 14  -1.00 | -229.09
 02: -------------------------------- |  2   0.00 | -----------------------n-------- |  7   0.00 | -228.09
 03: -------------------------------- |  3   0.00 | -------------------------------- |  0  -7.00 | -228.09
 04: -------------------------------- |  4   0.00 | --0000000----------------------- |  9  -8.00 | -221.09
 05: -------------------------------- |  5   0.00 | --1111111----------------------- |  2  -7.00 | -213.09
 06: -------------------------------- |  6   0.00 | --nuuuuuu----------------------- | 11  -6.00 | -206.09
 07: -------------------------------- |  7   0.00 | --01-----------------------0-000 |  4  -5.00 | -200.09
 08: -------------------------------- |  8   0.00 | -01------------------------0-011 | 13 -14.00 | -195.09
 09: -------------------------------- |  9   0.00 | -1----------------10-0-----n-nnn |  6 -11.00 | -181.09
 10: -------------------------------- | 10   0.00 | 1n010000----------11-1---------- | 15 -14.00 | -170.09
 11: -------------------------------- | 11   0.00 | 00111111-----00--0nu-n---------- |  8 -17.00 | -156.09
 12: -------------------------------- | 12   0.00 | nuuuuuuu-----11--11--0---------- |  1  -6.00 | -139.09
 13: -------------------------------- | 13   0.00 | -------1-----nn--un--u---------- | 10  -5.00 | -133.09
 14: -------------------------------- | 14  -1.00 | -------1----01----u------------- |  3 -11.00 | -128.09
 15: -----------------------n-------- | 15  -7.00 | -------u----10----0------------- | 12  -6.00 | -116.09
 16: -----------unnnn-------0-------- |  7 -12.09 | -----0-u----u------------------- |  6  -3.00 | -103.09
 17: -------n---00000-------1-------- |  4  -7.00 | -----u-0----u------------------- | 11  -2.00 | -88.00
 18: -------0---01111---------------- | 13  -4.00 | -----u------0------------------- |  3  -2.00 | -79.00
 19: ---u---1-------n------------1--- |  1  -4.00 | 0----0-------------------------- |  7  -1.00 | -73.00
 20: ---0-----------0------------0--- | 10  -3.00 | u------------------------------- |  0  -2.00 | -68.00
 21: ---1-----------1------------n--- |  6  -6.00 | u----------------------------0-- | 13  -2.00 | -63.00
 22: ---------------unnnn--------0--- | 15 -10.00 | 0----------------------------u-- |  5  -1.00 | -55.00
 23: ---------------00000--------u--- |  3  -7.00 | -----------------------------1-- | 10  -2.00 | -44.00
 24: -------------n-11101--------1--- | 12  -4.00 | ----------------------0------0-- | 14  -1.00 | -35.00
 25: -----------n-0--------------1--- |  0  -4.00 | ----------------------u--------- | 15  -1.00 | -30.00
 26: -------u---0-1------------------ |  9  -5.00 | ----------------------u--------- |  8  -1.00 | -25.00
 27: 1------0---1-u------------------ |  5  -3.00 | ----------------------0--------- | 12   0.00 | -19.00
 28: 0------1-----0------------------ |  2  -2.00 | -------------------------------- |  4  -1.00 | -16.00
 29: n------------1------------------ | 14  -1.00 | -------0------------------------ |  9  -1.00 | -13.00
 30: u------------------------------- | 11  -1.00 | -------u------------------------ |  1  -1.00 | -11.00
 31: u------------------------------- |  8  -1.00 | -------1------------------------ |  2  -1.00 |  -9.00
```

Fig. 3. The differential path for `RIPEMD-128`, after the non-linear parts search. The notations are the same as in [4] and $\mathrm{P}^l[i]$, $\mathrm{P}^r[i]$ and $P[i]$ are given in $\log_2()$.

in the right branch. Its overall differential probability is $2^{-230.09}$ and since we have 511 bits of message with unspecified value (one bit of M_4 is already set to "`1`"), plus 127 unrestricted bits of chaining variable (one bit of $X_0 = Y_0 = h_3$ is already set to "`0`"), we expect many solutions to exist (about $2^{407.91}$).

In order for the path to provide a collision, the bit difference in X_{61} must erase the one in Y_{64} during the finalization phase of the compression function: $h'_2 = X_{61} \boxplus Y_{64} \boxplus h_3$. Since the signs of these two bit differences are not specified, this happens with probability 2^{-1} and the overall probability to follow our differential path and to obtain a collision for a randomly chosen input is $2^{-231.09}$.

4 Utilization of the freedom degrees

In the differential path from Figure 3, the difference mask is already entirely set, but almost all message bits and chaining variable bits have no constraint with regards to their value. All these freedom degrees can be used to reduce the complexity of the straightforward collision search (i.e. choosing random 512-bit message values) that requires about $2^{231.09}$ `RIPEMD-128` step computations. We will utilize these freedom degrees in three phases:

- **Phase 1**: we first fix some internal state and message bits in order to prepare the attack. This will allow us to handle in advance some conditions in the differential path as well as facilitating the merging phase. This preparation phase is done once for all.

- **Phase 2**: we will fix iteratively the internal state words X_{21}, X_{22}, X_{23}, X_{24} from left branch, and Y_{11}, Y_{12}, Y_{13},Y_{14} from right branch, as well as message words M_{12}, M_3, M_{10}, M_1, M_8, M_{15}, M_6, M_{13}, M_4, M_{11} and M_7 (the ordering is important). This will provide us a starting point for the merging phase and due to a lack of freedom degrees, we will need to perform this phase several times in order to get enough starting points to eventually find a solution for the entire differential path.
- **Phase 3**: we use the remaining unrestricted message words M_0, M_2, M_5, M_9 and M_{14} to efficiently merge the internal states of the left and right branches.

4.1 Phase 1: preparation

Before starting to fix a lot of message and internal state bit values, we need to prepare the differential path from Figure 3 so that the merge can later be done efficiently and so that the probabilistic part will not be too costly. Understanding these constraints requires a deep insight of the differences propagation and conditions fulfillment inside the `RIPEMD-128` step function. Therefore, the reader not interested in the details of the differential path construction is advised to skip this subsection.

The first constraint that we set is $Y_3 = Y_4$. The effect is that the `IF` function at step 4 of the right branch, $\texttt{IF}(Y_2, Y_4, Y_3) = (Y_2 \wedge Y_3) \oplus (\overline{Y_2} \wedge Y_4) = Y_3 = Y_4$, will not depend on Y_2 anymore. We will see in Section 4.3 that this constraint is crucial in order for the merge to be performed efficiently.

The second constraint is $X_{24} = X_{25}$ (except the two bit positions of X_{24} and X_{25} that contain differences), and the effect is that the `IF` function at step 26 of the left branch (when computing X_{27}), $\texttt{IF}(X_{26}, X_{25}, X_{24}) = (X_{26} \wedge X_{25}) \oplus (\overline{X_{26}} \wedge X_{24}) = X_{24} = X_{25}$, will not depend on X_{26} anymore. Before the final merging phase starts, we will not know M_0, and having this $X_{24} = X_{25}$ constraint will allow us to directly fix the conditions located on X_{27} without knowing M_0 (since X_{26} directly depends on M_0). Moreover, we fix the 12 first bits of X_{23} and X_{24} to "`01000100u001`" and "`01000011110`" respectively because this choice is among the few that minimizes the number of bits of M_9 that needs to be set in order to verify many of the conditions located on X_{27}.

The third constraint consists in setting the bits 18 to 30 of Y_{20} to zero. The effect is that for these 13 bit positions, the `ONX` function at step 21 of the right branch (when computing Y_{22}), $\texttt{ONX}(Y_{21}, Y_{20}, Y_{19}) = (Y_{21} \vee \overline{Y_{20}}) \oplus Y_{19}$, will not depend on the 13 corresponding bits of Y_{21} anymore. Again, because we will not know M_0 before the merging phase starts, this constraint will allow us to directly fix the conditions on Y_{22} without knowing M_0 (since Y_{21} directly depends on M_0).

Finally, the last constraint that we enforce is that the first two bits of Y_{22} are set to "`10`" and the first three bits of M_{14} are set to "`011`". This particular choice of bit values is among the ones that reduces the most the spectrum of possible carries during the addition of step 24 (when computing Y_{25}) and we obtain a probability improvement to reach "`u`" in Y_{25} from 2^{-1} to $2^{-0.25}$.

We observe that all the constraints set in this subsection consume in total $32+51+13+5=101$ bits of freedom degrees, and a huge amount of solutions (about $2^{306.91}$) are still expected to exist.

4.2 Phase 2: generating a starting point

Once the differential path properly prepared in phase 1, we would like to utilize the huge amount of freedom degrees available to fulfill directly as many conditions as possible. Our approach is to fix the value of the internal states in both the left and right branches (they can be handled independently), exactly in the middle of the non-linear parts where the number of conditions is important. Then, we will fix the message words one by one following a particular scheduling, and propagating the bit values forward and backward from the middle of the non-linear parts in both branches.

Fixing the internal state. We chose to start by setting the values of X_{21}, X_{22}, X_{23}, X_{24} in the left branch, and Y_{11}, Y_{12}, Y_{13}, Y_{14} in the right branch, because they are located right in the middle of the non-linear parts. We take the first word X_{21} and randomly set all of its unrestricted "-" bits to "0" or "1" and check if any direct inconsistency is created with this choice. If that is the case, we simply pick another candidate until no direct inconsistency is deduced. Otherwise, we can go to the next word X_{22}, etc. If too many tries are failing for a particular internal state word, we can backtrack and pick another choice for the previous word. Finally, if no solution is found after a certain amount of time, we just restart the whole process, so as to avoid being blocked in a particularly bad subspace with no solution.

Fixing the message words. Similarly to the internal state words, we randomly fix the value of message words M_{12}, M_3, M_{10}, M_1, M_8, M_{15}, M_6, M_{13}, M_4, M_{11} and M_7 (following this particular ordering that facilitates the convergence towards a solution). The difference here is that the left and right branch computations are no more independent since the message words are used in both of them. However, this does not change anything to our algorithm and the very same process is applied: for each new message word randomly fixed, we compute forward and backward from the known internal state values and check for any inconsistency, using backtracking and reset if needed.

Overall, finding one new solution for this entire phase 2 takes about 5 minutes of computation on a recent PC with a naive implementation[4]. However, when one starting point is found, we can generate many for a very cheap cost by randomizing message words M_4, M_{11} and M_7 since the most difficult part is to fix the 8 first message words of the schedule. For example, once a solution is found, one can directly generate 2^{18} new starting points by randomizing a

[4] Our message word fixing approach is certainly not optimal, but this phase is not the bottleneck of our attack and we preferred to aim for simplicity when possible. In case a very fast implementation is needed, a more efficient but more complex strategy would be to find a bit per bit scheduling instead of a word-wise one.

certain portion of M_7 (because M_7 has no impact on the validity of the non-linear part in the left branch, while in the right branch one has only to ensure that the last 14 bits of Y_{20} are set to "`u0000000000000`") and this was verified experimentally.

We give an example of such a starting point in Figure 4 and we emphasize that by "solution" or "starting point" we mean a differential path instance with **exactly** the same probability profile as this one. The 3 constrained bit values in M_{14} are coming from the preparation in phase 1, and the 3 constrained bit values in M_9 are necessary conditions in order to fulfill step 26 when computing X_{27}. It is also important to remark that whatever instance found during this second phase, the position of these 3 constrained bit values will always be the same thanks to our preparation in phase 1.

The probabilities displayed in Figure 4 for early steps (steps 0 to 14) are not meaningful here since they assume an attacker only computing forward, while in our case we will compute backwards from the non-linear parts to the early steps. However, we can see that the uncontrolled accumulated probability (i.e. step ③ in right side of Figure 1) is now improved to $2^{-29.32}$, or $2^{-30.32}$ if we add the extra condition for the collision to happen at the end of the `RIPEMD-128` compression function.

```
Step              Xi                               Wli                 Πli               Yi                                Wri                Πri]  P[i]

 -3: --------------------------------
 -2: --------------------------------
 -1: --------------------------------
 00: -----------------------0-------- | --------------------------------  0 | -----------------------0-------- | --------------------------------  5 | -287.32
 01: -------------------------------- | 00000101111011100000110011000111  1 | ----------------------01-------- | x----------------------------011 14 | -285.32
 02: -------------------------------- | --------------------------------  2 | -----------------------n-------- | 01000010101100100011001110010110  7 | -284.32
 03: -------------------------------- | 00101100100000110100001001011110  3 | 00000000001100101010101011000000 | --------------------------------  0 | -252.32
 04: -------------------------------- | 11110000101100100000101111111100  4 | 00000000001100101010101011000000 | ----------------------0---1-1---  9 | -220.32
 05: -------------------------------- | --------------------------------  5 | 10111111101001001001010100111100 | --------------------------------  2 | -189.32
 06: -------------------------------- | 00100101011001000111000001010101  6 | 00nuuuuuu1100011011101100110010 | 10111001010001001100100111001100 11 | -157.32
 07: -------------------------------- | 01000010101100100011001110010110  7 | 00011011111101110110010011100000 | 11110000101100100000101111111100  4 | -157.32
 08: -------------------------------- | 00111100101111111010001110110000  8 | 10101101110101010010000001001011 | 01100011101010100010110001110011 13 | -157.32
 09: -------------------------------- | ----------------------0---1-1---  9 | 111000110011011100101010110n0nnn | 00100101011001000111000001010101  6 | -157.32
 10: -------------------------------- | 10001010101010011100001100111101 10 | 1n010000110010011011010100011110 | 00000110110000101001110101001010 15 | -157.32
 11: -------------------------------- | 10111001010001001100100111001100 11 | 0011111110111000010nu1n100011011 | 00111100101111111010001110110000  8 | -157.32
 12: 00111010101011111111101110101000 | 01101001001010010010111011101100 12 | nuuuuuuu011101111111110110111100 | 00000101111011100000110011000111  1 | -125.32
 13: 01110011001001011011001011011110 | 01100011101010100010110001110011 13 | 0101111110101nn10un11u1001001110 | 10001010101010011100001100111101 10 | -93.32
 14: 11110100011110100101101111011100 | x----------------------------011 14 | 01011111111001000u01001100000001 | 00101100100000110100001001011110  3 | -61.32
 15: 0110101010111100010111n100110110 | 00000110110000101001110101001010 15 | 1010100u111110000000100011100110 | 01101001001010010010111011101100 12 | -29.32
 16: 01010110010unnnn0010011000101111 | 01000010101100100011001110010110  7 | 1100101u1111u0011110011000010000 | 00100101011001000111000001010101  6 | -29.32
 17: 0100101n011000000000000111111001 | 11110000101100100000101111111100  4 | 11101u101111u0011111001011000010 | 10111001010001001100100111001100 11 | -29.32
 18: 10100010110011111110100000101000 | 01100011101010100010110001110011 13 | 11010u11000000010100110011000111 | 00101100100000110100001001011110  3 | -29.32
 19: 001u10010000101n0111000101111111 | 00000101111011100000110011000111  1 | 01010000011111101010011111100100 | 01000010101100100011001110010110  7 | -29.32
 20: 01100001101001101110001100100101 | 10001010101010011100001100111101 10 | u0000000000000000000000000000000 | --------------------------------  0 | -29.32
 21: 101111010011111111100110100 1n110 | 00100101011001000111000001010101  6 | u----------------------------0-- | 01100011101010100010110001110011 13 | -27.32
 22: 101110111000101unnnn011010000111 | 00000110110000101001110101001010 15 | 0111101111101111010000010100 0u10 | --------------------------------  5 | -27.32
 23: 1011111110111000000000100 0100u001 | 00101100100000110100001001011110  3 | -----------------------------1-- | 10001010101010011100001100111101 10 | -26.32
 24: 0100001011011n011101001000011110 | 01101001001010010010111011101100 12 | ---------------------10------0-- | x----------------------------011 14 | -24.32
 25: 01000010110n10011101001000011110 | --------------------------------  0 | ----------------------u--------- | 00000110110000101001110101001010 15 | -24.08
 26: -------u---0-1------------------ | ----------------------0---1-1---  9 | ----------------------u--------- | 00111100101111111010001110110000  8 | -21.08
 27: 1------0---1-u------------------ | --------------------------------  5 | ----------------------0--------- | 01101001001010010010111011101100 12 | -19.08
 28: 0------1-----0------------------ | --------------------------------  2 | -------------------------------- | 11110000101100100000101111111100  4 | -16.08
 29: n------------1------------------ | x----------------------------011 14 | -------0------------------------ | ----------------------0---1-1---  9 | -13.08
 30: u------------------------------- | 10111001010001001100100111001100 11 | -------u------------------------ | 00000101111011100000110011000111  1 | -11.00
 31: u------------------------------- | 00111100101111111010001110110000  8 | -------1------------------------ | --------------------------------  2 |  -9.00
 32: 1------------------------------- | 00101100100000110100001001011110  3 | -------1------------------------ | 00000110110000101001110101001010 15 |  -7.00
 33: -------------------------------- | 10001010101010011100001100111101 10 | -------------------------------- | --------------------------------  5 |  -7.00
 34: -------------------------------- | x----------------------------011 14 | u------------------------------- | 00000101111011100000110011000111  1 |  -6.00
 35: -------------------------------- | 11110000101100100000101111111100  4 | 0------------------------------- | 00101100100000110100001001011110  3 |  -4.00
 36: -------------------------------- | ----------------------0---1-1---  9 | 1------------------------------- | 01000010101100100011001110010110  7 |  -3.00
 37: -------------------------------- | 00000110110000101001110101001010 15 | -------------------------------- | x----------------------------011 14 |  -3.00
 38: -------------------------------- | 00111100101111111010001110110000  8 | -------------------------------- | 00100101011001000111000001010101  6 |  -3.00
 39: -------------------------------- | 00000101111011100000110011000111  1 | -------------------------------- | ----------------------0---1-1---  9 |  -3.00
 40: -------------------------------- | --------------------------------  2 | -------------------------------- | 10111001010001001100100111001100 11 |  -3.00
 41: -------------------------------- | 01000010101100100011001110010110  7 | -------------------------------- | 00111100101111111010001110110000  8 |  -3.00
 42: -------------------------------- | --------------------------------  0 | -------------------------------- | 01101001001010010010111011101100 12 |  -3.00
 43: -------------------------------- | 00100101011001000111000001010101  6 | -------------------------------- | --------------------------------  2 |  -3.00
 44: -------------------------------- | 01100011101010100010110001110011 13 | -------------------------------- | 10001010101010011100001100111101 10 |  -3.00
 45: -------------------------------- | 10111001010001001100100111001100 11 | -------------------------------- | --------------------------------  0 |  -3.00
 46: -------------------------------- | --------------------------------  5 | -------------------------------- | 11110000101100100000101111111100  4 |  -3.00
 47: -------------------------------- | 01101001001010010010111011101100 12 | -------------------------------- | 01100011101010100010110001110011 13 |  -3.00
 48: -------------------------------- | 00000101111011100000110011000111  1 | -------------------------------- | 00111100101111111010001110110000  8 |  -3.00
 49: -------------------------------- | ----------------------0---1-1---  9 | -------------------------------- | 00100101011001000111000001010101  6 |  -3.00
 50: -------------------------------- | 10111001010001001100100111001100 11 | -------------------------------- | 11110000101100100000101111111100  4 |  -3.00
 51: -------------------------------- | 10001010101010011100001100111101 10 | -------------------------------- | 00000101111011100000110011000111  1 |  -3.00
 52: -------------------------------- | --------------------------------  0 | -------------------------------- | 00101100100000110100001001011110  3 |  -3.00
 53: -------------------------------- | 00111100101111111010001110110000  8 | -------------------------------- | 10111001010001001100100111001100 11 |  -3.00
 54: -------------------------------- | 01101001001010010010111011101100 12 | -------------------------------- | 00000110110000101001110101001010 15 |  -3.00
 55: -------------------------------- | 11110000101100100000101111111100  4 | -------------------------------- | --------------------------------  0 |  -3.00
 56: -------------------------------- | 01100011101010100010110001110011 13 | -------------------------------- | --------------------------------  5 |  -3.00
 57: -------------------------------- | 00101100100000110100001001011110  3 | -------------------------------- | 01101001001010010010111011101100 12 |  -3.00
 58: -------------------------------- | 01000010101100100011001110010110  7 | -------------------------------- | --------------------------------  2 |  -3.00
 59: ------------------------0------- | 00000110110000101001110101001010 15 | -------------------------------- | 01100011101010100010110001110011 13 |  -2.00
 60: ------------------------1------- | x----------------------------011 14 | -------------------------------- | ----------------------0---1-1---  9 |  -1.00
 61: ------------------------x------- | --------------------------------  5 | -------------------------------- | 01000010101100100011001110010110  7 |  -1.00
 62: -------------------------------- | 00100101011001000111000001010101  6 | -------------------------------- | 10001010101010011100001100111101 10 |  -1.00
 63: -------------------------------- | --------------------------------  2 | -------------------------------- | x----------------------------011 14 |  -1.00
 64: -------------------------------- |                                      | ------------------------x-------
```

Fig. 4. The differential path for `RIPEMD-128`, after the second phase of the freedom degree utilization. The notations are the same as in [4] and $P[i]$ is given in $\log_2()$.

4.3 Phase 3: merging left and right branches

At the end of the second phase, we have several starting points equivalent to the one from Figure 4, with many conditions already verified and an uncontrolled accumulated probability of $2^{-30.32}$. Our goal for this third phase is now to use remaining free message words M_0, M_2, M_5, M_9, M_{14} and make sure that both left and right branches start with the same chaining variable.

We recall that during the first phase we enforced that $Y_3 = Y_4$, and for the merge we will require an extra constraint $X_5^{\ggg 5} \boxminus M_4 = \texttt{0xffffffff}$. The message words M_{14} and M_9 will be utilized to fulfill this constraint, and message words M_0, M_2 and M_5 will be used to perform the merge of the two branches only with a few operations, and with a success probability of 2^{-34}.

Handling the extra constraint with M_{14} and M_9. First, let us deal with the constraint $X_5^{\ggg 5} \boxminus M_4 = \texttt{0xffffffff}$, which can be rewritten as $X_5 = (\texttt{0xffffffff} \boxplus M_4)^{\lll 5}$ and then $(\texttt{0xffffffff} \boxplus M_4)^{\lll 5} = X_9^{\ggg 11} \boxminus (X_8 \oplus X_7 \oplus X_6) \boxminus M_8 \boxminus K_0^l$ by replacing M_5 using update formula of step 8 in left branch. Finally, isolating and replacing X_6 using update formula of step 9 in left branch:

$$M_9 = X_{10}^{\ggg 13} \boxminus ((X_9^{\ggg 11} \boxminus M_8 \boxminus K_0^l \boxminus (\texttt{0xffffffff} \boxplus M_4)^{\lll 5}) \oplus X_8 \oplus X_7) \boxminus K_0^l \boxminus (X_9 \oplus X_8 \oplus X_7). \quad (1)$$

All values on the right side of this equation are known if M_{14} is fixed. Therefore, so as to fulfill our extra constraint, what we could do is to simply pick a random value for M_{14}, and then directly deduce the value of M_9 thanks to equation (1). However, one can see in Figure 4 that 3 bits are already fixed in M_9 (the last one being the 10^{th} bit of M_9) and thus a valid solution would be found only with probability 2^{-3}. In order to avoid this extra complexity factor, we will first randomly fix the first 24 bits of M_{14} and this will allow us to directly deduce the first 10 bits of M_9 that fulfill our extra constraint up to the 10^{th} bit (because knowing the first 24 bits of M_{14} will lead to the first 24 bits of X_{11}, X_{10}, X_9, X_8 and the first 10 bits of X_7, which is exactly what we need according to equation (1)). Once a solution is found after 2^3 tries on average, we can randomize the remaining M_{14} unrestricted bits (the 8 most significant bits) and eventually deduce the 22 most significant bits of M_9 with equation (1). With this method, we completely remove the extra 2^3 factor, because the cost is amortized by the final randomization of the 8 most significant bits of M_{14}.

Merging the branches with M_0, M_2 and M_5. Once M_9 and M_{14} fixed, we still have message words M_0, M_2 and M_5 to determine for the merging. One can see that with only these three message words undetermined, all internal state values except X_2, X_1, X_0, X_{-1}, X_{-2}, X_{-3} and Y_2, Y_1, Y_0, Y_{-1}, Y_{-2}, Y_{-3} are fully known when computing backwards from the non-linear parts in each branch.

This is where our first constraint $Y_3 = Y_4$ comes into play. Indeed, when writing Y_1 from the equation from step 4 in right branch, we have:

$$Y_1 = Y_5^{\ggg 13} \boxminus (Y_4 \wedge Y_2 \oplus Y_3 \wedge \overline{Y_2}) \boxminus M_9 \boxminus K_0^r = Y_5^{\ggg 13} \boxminus Y_3 \boxminus M_9 \boxminus K_0^r$$

which means that Y_1 is already completely determined at this point (the bit condition present in Y_1 in Figure 4 is actually handled for free when fixing M_{14}

and M_9, since it requires to know the 9 first bits of M_9). In other words, the constraint $Y_3 = Y_4$ allowed Y_1 to not depend on Y_2 which is currently undetermined. Another effect of this constraint can be seen when writing Y_2 from the equation from step 5 in right branch:

$$Y_2 = Y_6^{\ggg 15} \boxminus (Y_5 \wedge Y_3 \oplus Y_4 \wedge \overline{Y_3}) \boxminus M_2 \boxminus K_0^r = Y_6^{\ggg 15} \boxminus (Y_5 \wedge Y_3) \boxminus M_2 \boxminus K_0^r = C_0 \boxminus M_2$$

where $C_0 = Y_6^{\ggg 15} \boxminus (Y_5 \wedge Y_3) \boxminus K_0^r$ is a constant.

Our second constraint $X_5^{\ggg 5} \boxminus M_4 = \texttt{0xffffffff}$ is useful when writing X_1 and X_2 from the equations from step 4 and 5 in left branch

$$\begin{aligned} X_2 &= X_6^{\ggg 8} \boxminus (X_5 \oplus X_4 \oplus X_3) \boxminus M_5 = C_1 \boxminus M_5 \\ X_1 &= X_5^{\ggg 5} \boxminus (X_4 \oplus X_3 \oplus X_2) \boxminus M_4 = \overline{X_4} \oplus X_3 \oplus X_2 = \overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5) \end{aligned}$$

where $C_1 = X_6^{\ggg 8} \boxminus (X_5 \oplus X_4 \oplus X_3)$ is a constant.

Finally, our ultimate goal for the merge is to ensure that $X_{-3} = Y_{-3}$, $X_{-2} = Y_{-2}$, $X_{-1} = Y_{-1}$ and $X_0 = Y_0$, knowing that all other internal states are determined when computing backwards from the non-linear parts in each branch, except $Y_2 = C_0 \boxminus M_2$, $X_2 = C_1 \boxminus M_5$ and $X_1 = \overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5)$. We therefore write the equations relating these eight internal state words:

$$\begin{aligned} X_0 &= X_4^{\ggg 12} \boxminus (X_3 \oplus X_2 \oplus X_1) \boxminus M_3 = X_4^{\ggg 12} \boxminus \overline{X_4} \boxminus M_3 \\ = Y_0 &= Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus Y_2 \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r = Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus (C_0 \boxminus M_2) \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r \\[1ex] X_{-1} &= X_3^{\ggg 15} \boxminus (X_2 \oplus X_1 \oplus X_0) \boxminus M_2 = X_3^{\ggg 15} \boxminus (\overline{X_4} \oplus X_3 \oplus X_0) \boxminus M_2 \\ = Y_{-1} &= Y_3^{\ggg 9} \boxminus (Y_2 \wedge Y_0 \oplus Y_1 \wedge \overline{Y_0}) \boxminus M_7 \boxminus K_0^r = Y_3^{\ggg 9} \boxminus ((C_0 \boxminus M_2) \wedge X_0 \oplus Y_1 \wedge \overline{X_0}) \boxminus M_7 \boxminus K_0^r \\[1ex] X_{-2} &= X_2^{\ggg 14} \boxminus (X_1 \oplus X_0 \oplus X_{-1}) \boxminus M_1 = (C_1 \boxminus M_5)^{\ggg 14} \boxminus (\overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5) \oplus X_0 \oplus X_{-1}) \boxminus M_1 \\ = Y_{-2} &= Y_2^{\ggg 9} \boxminus (Y_1 \wedge Y_{-1} \oplus Y_0 \wedge \overline{Y_{-1}}) \boxminus M_{14} \boxminus K_0^r = (C_0 \boxminus M_2)^{\ggg 9} \boxminus (Y_1 \wedge X_{-1} \oplus X_0 \wedge \overline{X_{-1}}) \boxminus M_{14} \boxminus K_0^r \\[1ex] X_{-3} &= X_1^{\ggg 11} \boxminus (X_0 \oplus X_{-1} \oplus X_{-2}) \boxminus M_0 = (\overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5))^{\ggg 11} \boxminus (X_0 \oplus X_{-1} \oplus X_{-2}) \boxminus M_0 \\ = Y_{-3} &= Y_1^{\ggg 8} \boxminus (Y_0 \wedge Y_{-2} \oplus Y_{-1} \wedge \overline{Y_{-2}}) \boxminus M_5 \boxminus K_0^r = Y_1^{\ggg 8} \boxminus (X_0 \wedge X_{-2} \oplus X_{-1} \wedge \overline{X_{-2}}) \boxminus M_5 \boxminus K_0^r \end{aligned}$$

If these four equations are verified, then we have merged left and right branch to the same input chaining variable. We first remark that X_0 is already fully determined and thus the second equation $X_{-1} = Y_{-1}$ only depends on M_2. Moreover, it is a T-function in M_2 (any bit i of the equation depends only on the i first bits of M_2) and can therefore be solved very efficiently bit per bit. We explain in the full version how to solve this T-function and our average cost in order to find one M_2 solution is one `RIPEMD-128` step computations.

Since X_0 is already fully determined, from the M_2 solution previously obtained we directly deduce the value of M_0 to satisfy the first equation $X_0 = Y_0$. From M_2 we can compute the value of Y_{-2} and we know that $X_{-2} = Y_{-2}$ and we calculate X_{-3} from M_0 and X_{-2}. At this point, the two first equations are fulfilled and we still have the value of M_5 to choose.

The third equation can be rewritten $V^{\ggg 14} = (V \oplus C_2) \boxplus C_3$, where $V = X[2] = (C_1 \boxminus M_5)$ and C_2, C_3 are two constants. Similarly, the fourth equation can be rewritten $V^{\ggg 11} = (V \boxplus C_4) \oplus C_5$, where C_4, C_5 are two constants.

Solving either of these two equations with regards to V can be costly because of the rotations, so we combine them to create simpler one: $((V \oplus C_2) \boxplus C_3)^{\lll 3} = (V \boxplus C_4) \oplus C_5$. This equation is easier to handle because the rotation coefficient is small: we guess the 3 most significant bits of $((V \oplus C_2) \boxplus C_3)$ and we solve simply the equation 3-bit layer per 3-bit layer, starting from the least significant bit. From the value of V deduced, we straightforwardly obtain $M_5 = C_1 \boxminus V$ and the cost of recovering M_5 is equivalent to 8 `RIPEMD-128` step computations (the 3-bit guess implies a factor of 8, but the resolution can be implemented very efficiently with tables).

When all three message words M_0, M_2 and M_5 have been fixed, the first, second and a combination of the third and fourth equalities are necessarily verified. However, we have a probability 2^{-32} that both the third and fourth equations will be fulfilled. Moreover, one can check in Figure 4 that there is one bit condition on $X_0 = Y_0$ and one bit condition on Y_2 and this further adds up a factor 2^{-2}. We evaluate the whole process to cost about 19 `RIPEMD-128` step computations on average: there are 17 steps to compute backwards after having identified a proper couple M_{14}, M_9, and the 8 `RIPEMD-128` step computations to obtain M_5 are only done 1/4 of the time because the two bit conditions on Y_2 and $X_0 = Y_0$ are filtered before.

To summarize the merging: we first compute a couple M_{14}, M_9 that satisfies a special constraint, we find a value of M_2 that verifies $X_{-1} = Y_{-1}$, then we directly deduce M_0 to fulfill $X_0 = Y_0$, and we finally obtain M_5 to satisfy a combination of $X_{-2} = Y_{-2}$ and $X_{-3} = Y_{-3}$. Overall, with only 19 `RIPEMD-128` step computations on average, one can merge the branches with probability 2^{-34}.

5 Results and implementation

5.1 Complexity analysis and implementation

After the quite technical description of the attack in previous section, we would like to rewrap everything to get a clearer view of the attack complexity, the amount of freedom degrees etc. Given a starting point from phase 2, the attacker can perform 2^{26} merge processes (because 3 bits are already fixed in both M_9 and M_{14}, and the extra constraint consumes 32 bits) and since one merge process succeeds only with probability of 2^{-34}, he obtains a solution with probability 2^{-8}. Since he needs $2^{30.32}$ solutions from the merge to have a good chance to verify the probabilistic part of the differential path, a total of $2^{38.32}$ starting points will have to be generated and handled.

From the end of phase 1, he generates $2^{38.32}$ starting points in phase 2, that is $2^{38.32}$ differential paths like the one from Figure 4 (with the same step probabilities). For each of them, in phase 3 he tries 2^{26} times to find a solution for the merge with an average complexity of 19 `RIPEMD-128` step computations for each try. The SFS collision final complexity is $19 \cdot 2^{26+38.32}$ `RIPEMD-128` step computations, which corresponds to $(19/128) \cdot 2^{64.32} = 2^{61.57}$ `RIPEMD-128` compression function computations (there are 64 steps in each branch).

The merge process has been implemented and we give in the full version of the article an example of a message and chaining variable couple that verifies the merge (i.e. they follow the differential path from Figure 3 until step 25 of the left branch and step 20 of the right branch). We measured the efficiency of our implementation in order to confront it to our theoretic complexity estimation. As point of reference, we observed that on the same computer, an optimized implementation of `RIPEMD-160` (OpenSSL v.1.0.1c) performs $2^{21.44}$ compression function computations per second. With 4 rounds instead of 5 and about 3/4 less operations per step, we extrapolated that `RIPEMD-128` would perform at $2^{22.17}$ compression function computations per second. Our implementation performs $2^{24.61}$ merge process (both phase 2 and phase 3) per second on average, which therefore corresponds to a SFS collision final complexity of $2^{61.88}$ `RIPEMD-128` compression computations. While our practical results confirm our theoretical estimations, we emphasize that the latter are a bit pessimistic, since our attack implementation is not optimized. As a side note, we also verified experimentally that the probabilistic part in both left and right branch can be fulfilled.

A last point needs to be checked: the complexity estimation for the generation of the starting points. Indeed, as much as $2^{38.32}$ starting points are required at the end of phase 2 and the algorithm being quite heuristic, it is hard to analyze precisely. The amount of freedom degrees is not an issue since we already saw in Section 4.1 that about $2^{306.91}$ solutions are expected to exist for the differential path at the end of phase 1. A completely new starting point takes about 5 minutes to be outputted on average with our implementation, but from one such path we can directly generate 2^{18} equivalent ones by randomizing M_7. Using the OpenSSL implementation as reference, this amounts to $2^{50.72}$ `RIPEMD-128` computations to generate all the starting points that we need in order to find a SFS collision. This gross estimation is extremely pessimistic since its doesn't even take in account the fact that once a starting point is found, one can also randomize M_4 and M_{11} to find many other valid candidates with a few operations. Finally, one may argue that with this method the starting points generated are not independent enough (in backward direction when merging and/or in forward direction for verifying probabilistically the linear part of the differential path). However, no such correlation was detected during our experiments and previous attacks on similar hash functions [13, 14] showed that only a few rounds were enough to observe independence between bit conditions. In addition, even if some correlations existed, since we are looking for many solutions, the effect would be averaged among good and bad candidates.

Collision for the `RIPEMD-128` compression function. We described in previous sections a SFS collision attack for the full `RIPEMD-128` compression function with $2^{61.57}$ computations. It is clear from Figure 4 that we can remove the 4 last steps of our differential path in order to attack a 60-step reduced variant of the `RIPEMD-128` compression function. No difference will be present in the internal state at the end of the computation and we directly get a collision, saving a factor 2^4 over the full `RIPEMD-128` attack complexity.

We also give in the full version of the article a slightly different freedom degrees utilization when attacking 63 steps of the RIPEMD-128 compression function (the first step being taken out), that saves a factor $2^{1.66}$ over the collision attack.

Distinguishers We provide in the full version of this article a limited-birthday distinguisher [11] on the full RIPEMD-128 compression but also hash function.

Conclusion

In this article, we proposed a new cryptanalysis technique for RIPEMD-128, that led to a collision attack on the full compression function as well as a distinguisher for the full hash function. We believe that our method still presents room for improvements, and we expect a practical collision attack for the full RIPEMD-128 compression function to be found during the incoming years. While our results don't endanger the collision resistance of the RIPEMD-128 hash function as a whole, we emphasize that SFS collision attacks are a strong warning sign which indicates that RIPEMD-128 might not be as secure as the community expected. Considering the history of the attacks on the MD5 compression function [7, 8], MD5 hash function [29], and then MD5-protected certificates [24], we believe that another function than RIPEMD-128 should be used for new security applications. Future works include reducing the attack complexity and applying our methods to RIPEMD-160 and other parallel branches-based functions.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. Submission to NIST, 2008.
2. A. Bosselaers and B. Preneel (Eds.). *Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*, vol. 1007 of *LNCS*. Springer, 1995.
3. G. Brassard (Ed.). *Advances in Cryptology - CRYPTO '89, Proceedings*, vol. 435 of *LNCS*. Springer, 1990.
4. C. De Cannière and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In X. Lai and K. Chen (Eds.), *ASIACRYPT*, vol. 4284 of *LNCS*, pp. 1–20. Springer, 2006.
5. R. Cramer (Ed.). *Advances in Cryptology - EUROCRYPT 2005, Proceedings*, vol. 3494 of *LNCS*. Springer, 2005.
6. I. Damgård. A Design Principle for Hash Functions. In Brassard [3], pp. 416–427.
7. B. den Boer and A. Bosselaers. Collisions for the Compressin Function of MD5. In T. Helleseth (Ed.), *EUROCRYPT*, vol. 765 of *LNCS*, pp. 293–304. Springer, 1993.
8. H. Dobbertin. Cryptanalysis of MD5 compress. In Rump Session of Advances in Cryptology EUROCRYPT 1996, 1996.
9. H. Dobbertin. RIPEMD with Two-Round Compress Function is Not Collision-Free. *J. Cryptology*, 10(1):51–70, 1997.
10. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In D. Gollmann (Ed.), *FSE*, vol. 1039 of *LNCS*, pp. 71–82. Springer, 1996.

11. H. Gilbert and T. Peyrin. Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations. In S. Hong and T. Iwata (Eds.), *FSE*, vol. 6147 of *LNCS*, pp. 365–383. Springer, 2010.
12. ISO. *ISO/IEC 10118-3:2004: Information technology — Security techniques — Hash-functions — Part 3: Dedicated hash-functions*. pub-ISO, feb 2004.
13. A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In A. Menezes (Ed.), *CRYPTO*, vol. 4622 of *LNCS*, pp. 244–263. Springer, 2007.
14. S. Manuel and T. Peyrin. Collisions on SHA-0 in One Hour. In K. Nyberg (Ed.), *FSE*, vol. 5086 of *LNCS*, pp. 16–35. Springer, 2008.
15. F. Mendel, T. Nad, S. Scherz, and M. Schläffer. Differential Attacks on Reduced RIPEMD-160. In D. Gollmann and F. C. Freiling (Eds.), *ISC*, vol. 7483 of *LNCS*, pp. 23–38. Springer, 2012.
16. F. Mendel, T. Nad, and M. Schläffer. Collision Attacks on the Reduced Dual-Stream Hash Function RIPEMD-128. In A. Canteaut (Ed.), *FSE*, vol. 7549 of *LNCS*, pp. 226–243. Springer, 2012.
17. F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. On the Collision Resistance of RIPEMD-160. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *ISC*, vol. 4176 of *LNCS*, pp. 101–116. Springer, 2006.
18. R. C. Merkle. One Way Hash Functions and DES. In Brassard [3], pp. 428–446.
19. C. Ohtahara, Y. Sasaki, and T. Shimoyama. Preimage Attacks on Step-Reduced RIPEMD-128 and RIPEMD-160. In X. Lai, M. Yung, and D. Lin (Eds.), *Inscrypt*, vol. 6584 of *LNCS*, pp. 169–186. Springer, 2010.
20. R. L. Rivest. The MD4 message-digest algorithm. Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.
21. Y. Sasaki and K. Aoki. Meet-in-the-Middle Preimage Attacks on Double-Branch Hash Functions: Application to RIPEMD and Others. In C. Boyd and J.M. González Nieto (Eds.), *ACISP*, vol. 5594 of *LNCS*, pp. 214–231. Springer, 2009.
22. Y. Sasaki and L. Wang. Distinguishers beyond Three Rounds of the RIPEMD-128/-160 Compression Functions. In F. Bao, P. Samarati, and J. Zhou (Eds.), *ACNS*, vol. 7341 of *LNCS*, pp. 275–292. Springer, 2012.
23. Victor Shoup (Ed.). *Advances in Cryptology - CRYPTO 2005, Proceedings*, vol. 3621 of *LNCS*. Springer, 2005.
24. M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. Arne Osvik, and B. de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In S. Halevi (Ed.), *CRYPTO*, vol. 5677 of *LNCS*, pp. 55–69. Springer, 2009.
25. X. Wang, H. Yu, and Y. Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Shoup [23], pp. 1–16.
26. L. Wang, Y. Sasaki, W. Komatsubara, K. Ohta, and K. Sakiyama. (Second) Preimage Attacks on Step-Reduced RIPEMD/RIPEMD-128 with a New Local-Collision Approach. In A. Kiayias (Ed.), *CT-RSA*, vol. 6558 of *LNCS*, pp. 197–212. Springer, 2011.
27. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [5], pp. 1–18.
28. X. Wang, Y. Lisa Yin, and H. Yu. Finding Collisions in the Full SHA-1. In Shoup [23], pp. 17–36.
29. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In Cramer [5], pp. 19–35.