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, compression function, hash function.

1 Introduction

Hash functions are among the most important basic primitives in cryptography, used in many applications such as digital signatures, message integrity check and message authentication codes (MAC). Informally, a hash function H is a function that takes an arbitrarily long message M as input and outputs a fixed-length hash value of size n bits. Classical security requirements are collision resistance and (second)-preimage resistance. Namely, it should be impossible for an adversary to find a collision (two distinct messages that lead to the same hash value) in less than $2^{n/2}$ hash computations, or a (second)-preimage (a message hashing to a given challenge) in less than 2^n hash computations. More complex security properties can be considered up to the point where the hash function should be indistinguishable from a random oracle, thus presenting no weakness whatsoever. Most standardized hash functions are based upon the Merkle-Damgård paradigm [18, 6] and iterate a compression function h with fixed input size to handle arbitrarily long messages. The compression function itself should ensure equivalent security properties in order for the hash function to inherit from them.

Recent impressive progresses in cryptanalysis [29, 27, 28, 26] led to the fall of most standardized hash 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, but the rapid improvement of the attacks decided the NIST to organize a 4-year SHA-3 competition in order to design a new hash function, eventually leading to the selection of Keccak [1]. This choice was justified partly by the fact that Keccak is built upon a completely different design rationale than the MD-SHA family. Yet, we can not expect the industry to move quickly to SHA-3 unless a real issue is identified in current hash primitives. Therefore, the SHA-3 competition monopolizing most of the cryptanalysis power during the last four years, it is now crucial to continue the study of the unbroken MD-SHA members.

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 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.* [26] 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

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

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	$2^{126.5}$	[25]
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-randomness	52	2^{107}	[22]
RIPEMD-128	**128**	**comp. function**	**non-randomness**	**Full**	$2^{59.57}$	**new**
RIPEMD-128	128	hash function	non-randomness	48	2^{70}	[16]
RIPEMD-128	**128**	**hash. function**	**non-randomness**	**Full**	$2^{105.40}$	**new**

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 [26] on RIPEMD-0 apply to RIPEMD-128 and RIPEMD-160. Then, following the extensive work on preimage attacks for MD-SHA family, [21, 19, 25] 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 collision attack on the full RIPEMD-128 compression function (Section 5), significantly improving the previous free-start collision attack on 48 steps. Any further improvement of our techniques is likely to provide a practical semi-free-start collision attack 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 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 $cv_0 = IV$ (given in Table 2 of Appendix A).

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.

[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.

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 internal state:

$$\begin{aligned} X_{-3} &= h_0 \quad X_{-2} = h_1 \quad X_{-1} = h_2 \quad X_0 = h_3 \\ Y_{-3} &= h_0 \quad Y_{-2} = h_1 \quad Y_{-1} = h_2 \quad Y_0 = h_3 \ . \end{aligned}$$

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 $0 \leq k \leq 15$:

$$W^l_{j\cdot 16+k} = M_{\pi^l_j(k)} \quad \text{and} \quad W^r_{j\cdot 16+k} = M_{\pi^r_j(k)}$$

where permutations π_j^l and π_j^r are given in Table 3 of Appendix A.

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

$$\begin{aligned} X_{i+1} &= (X_{i-3} \boxplus \Phi^l_j(X_i, X_{i-1}, X_{i-2}) \boxplus W_i^l \boxplus K_j^l)^{\lll s_i^l}, \\ Y_{i+1} &= (Y_{i-3} \boxplus \Phi^r_j(Y_i, Y_{i-1}, Y_{i-2}) \boxplus W_i^r \boxplus K_j^r)^{\lll s_i^r}, \end{aligned}$$

where K_j^l, K_j^r are 32-bit constants defined for every round j and every branch, s_i^l, s_i^r are rotation constants defined for every step i and every branch, Φ_j^l, Φ_j^r are 32-bit boolean functions defined for every round j and every branch. All these constants and functions are given in Appendix A.

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:

$$\begin{aligned} h'_0 &= X_{63} \boxplus Y_{62} \boxplus h_1 \quad h'_1 = X_{62} \boxplus Y_{61} \boxplus h_2 \\ h'_2 &= X_{61} \boxplus Y_{64} \boxplus h_3 \quad h'_3 = X_{64} \boxplus Y_{63} \boxplus h_0 \ . \end{aligned}$$

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).

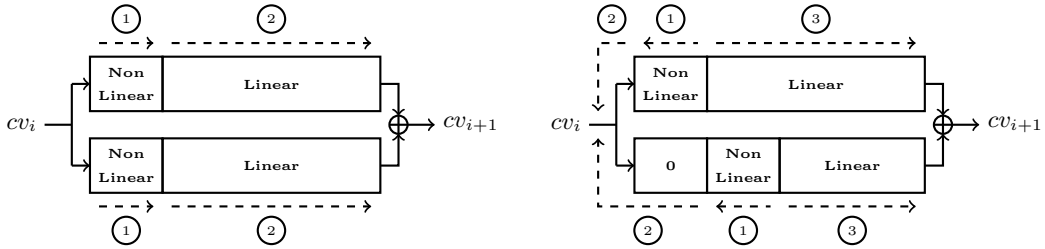


Fig. 1. The previous (left-hand side) and new (right-hand side) approach for collision search on double-branch compression functions.

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 when only a single bit difference is present on its input. 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 free-start collision attack) or very late (for a semi-free-start 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.

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

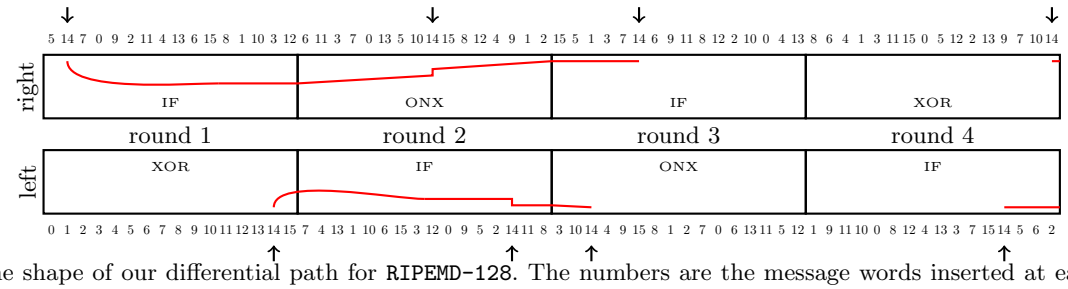


Fig. 2. The shape of our differential path for `RIPEMD-128`. The numbers are the message words inserted at each step and the red curves represent the rough amount differences in the internal state during each steps. The arrows show where the bit differences are injected with M_{14}.

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 (which is yet another argument in favor of M_{14}). 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 semi-free-start collision attack. Following this method and reusing notations from [4], we eventually obtain the differential path depicted in Figure 5 of Appendix F, the "`?`" representing unrestricted bits that will be constrained during the non-linear parts search. 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

Our goal starting from Figure 5 of Appendix F is now to instantiate the unconstrained bits denoted by "`?`" such that only inactive ("`0`", "`1`" or "`-`") or active bits ("`n`", "`u`" or "`x`") remain and such that the path contains no direct inconsistency. This is in general 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 to the trail given in Figure 5 of Appendix F, we obtain the differential path in Figure 6 of Appendix F, 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 $2^{-85.09}$ (i.e. $\prod_{i=0}^{63} \mathrm{P}^l[i] = 2^{-85.09}$) in the left branch and 2^{-145} (i.e. $\prod_{i=0}^{63} \mathrm{P}^r[i] = 2^{-145}$) 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 6 of Appendix F, 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 6 of Appendix F 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 "0000000000000". 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 give in Figure 7 of Appendix F our differential path after having set these constraints (we denote a bit $[X_i]_j$ with the constraint $[X_i]_j = [X_{i-1}]_j$ by "^"). 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 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 3 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 3 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.

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 3, 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 $X_9^{\ggg 11} \boxminus (X_8 \oplus X_7 \oplus X_6) \boxminus M_8 \boxminus K_0^l = (\texttt{0xffffffff} \boxplus M_4)^{\lll 5}$ by replacing M_5 using update formula of step 8 in left branch. Finally, isolating X_6 and replacing it using update formula of step 9 in left branch we obtain:

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

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 3 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}.

[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.

Step	X_i	W^l_i	Π^l_i	$P^l[i]$	Y_i	W^r_i	Π^r_i	$P^r[i]$	$P[i]$
-3:									
-2:									
-1:									
00:	----------0--------	----------	0	0.00	----------0--------	----------	5	-2.00	-287.32
01:	----------	00000101111101110000011001100011 1	1	0.00	----------01--------	x----------011	14	-1.00	-285.32
02:	----------	----------	2	0.00	----------n---------	01000010101100100011001110010110	7	-32.00	-284.32
03:	----------	00101100100000011010000100101111 0	3	0.00	00000000001100101010101011000000	----------	0	-32.00	-252.32
04:	----------	11110000101100100000101111111100 4	4	0.00	00000000001100101010101011000000	----------0---1-1---	9	-31.00	-220.32
05:	----------	----------	5	0.00	10111111101001001001010100111100	----------	2	-32.00	-189.32
06:	----------	00100101011001000111000001010101 6	6	0.00	00nuuuuuu1100011011011001100100	10111001010001001100100111001100	11	0.00	-157.32
07:	----------	01000010101100100011001110010110 7	7	0.00	00011011111101110110010011100000	11110000101100100000101111111100	4	0.00	-157.32
08:	----------	00111100101111111010001110110000 8	8	0.00	10101101110101010010000001001011	01100011101010100010110001110011	13	0.00	-157.32
09:	----------	----------0---1-1--- 9	9	0.00	11100001100101110010101010110n0nnn	00100101011100100011100000101010 1	6	0.00	-157.32
10:	----------	10001010101010011100001100111101 10	10	0.00	1n01000011001001101101010001110	00000110110000010100111010100101 0	15	0.00	-157.32
11:	----------	10111001010001001100100111001100 11	11	-32.00	00111111101110001 0nu1n1000110110	00111100101111111010001110110000 8	8	0.00	-157.32
12:	00111010101011111111101110101000	01101001001010010011101111011100 12	12	-32.00	nuuuuuuu01110111111110110111111001	00000101111011100000110011000111 1	1	0.00	-125.32
13:	01110011001001011011001011011110	01100011101010100010110001110011 13	13	-32.00	0101111110101nn10un11u1001001110	10001010101010011100001100111101 10	10	0.00	-93.32
14:	11111010001111010010110111111100	x----------011	14	-32.00	01011111111100100000u1001100000001	00101100100000011010000100101110 3	3	0.00	-61.32
15:	01101010101111000101111n00110110	00000110110000010100111010100101 0	15	0.00	1010100u1111100000001000111001100	01101001001010010010111011101100 12	12	0.00	-29.32
16:	01010110010unnnn0010011000101111	01000010101100100011001110010110 7	7	0.00	1100101u1111u0011110011000010000	00100101011001000111000001010101 6	6	0.00	-29.32
17:	0100101n01100000000000001111111001	11110000101100100000101111111100 4	4	0.00	11101u101111u0011111001011000010	10111001010001001100100111001100 11	11	0.00	-29.32
18:	10100010110011111111010000010100 0	01100011101010100010110001110011 13	13	0.00	11010u1100000010100110011000111 1	00101100100000011010000100101110 3	3	0.00	-29.32
19:	001u100100000101n0111000101111111	00000101111101110000011001100111 1	1	0.00	01010000011111101010011111100100	01000010101100100011001110010110 7	7	0.00	-29.32
20:	01100001101001101110001100100101	10001010101010011100001100111101 10	10	0.00	u0000000000000000000000000000000	----------	0	-2.00	-29.32
21:	10111101001111111111001101001n110	00100101011001000111000001010101 6	6	0.00	u----------0--	01100011101010100010110001110011 13	13	0.00	-27.32
22:	10111011100010 1unnnn011010000111	00000110110000010100111010100101 0	15	0.00	01111011111101111010000101000u10	----------	5	-1.00	-27.32
23:	10111111101110000000001000100u001	00101100100000011010000100101110 3	3	0.00	----------1--	10001010101010011100001100111101 10	10	-2.00	-26.32
24:	01000010110111n011101001000011110	01101001001010010010111011101100 12	12	0.00	----------10------0--	x----------011	14	-0.25	-24.32
25:	01000010110n1001110100100001110	----------	0	-3.00	----------u---------	00000110110000010100111010100101 0	15	0.00	-24.08
26:	-------u---0-1----------	----------0---1-1--- 9	9	-1.00	----------u---------	00111100101111111010001110110000 8	8	-1.00	-21.08
27:	1------0---1-u----------	----------	5	-3.00	----------0---------	01101001001010010010111011101100 12	12	0.00	-19.08
28:	0------1-----0----------	----------	2	-2.00	----------	11110000101100100000101111111100 4	4	-1.00	-16.08
29:	n----------1----------	x----------011	14	-1.00	-------0----------	----------0---1-1--- 9	9	-1.08	-13.08
30:	u----------	10111001010001001100100111001100 11	11	-1.00	-------u----------	00000101111101110000011001100111 1	1	-1.00	-11.00
31:	u----------	00111100101111111010001110110000 8	8	-1.00	-------1----------	----------	2	-1.00	-9.00
32:	1----------	00101100100000011010000100101110 3	3	0.00	-------1----------	00000110110000010100111010100101 0	15	0.00	-7.00
33:	----------	10001010101010011100001100111101 10	10	0.00	----------	----------	5	-1.00	-7.00
34:	----------	x----------011	14	0.00	u----------	00000101111101110000011001100111 1	1	-2.00	-6.00
35:	----------	11110000101100100000101111111100 4	4	0.00	0----------	00101100100000011010000100101110 3	3	-1.00	-4.00
36:	----------	----------0---1-1--- 9	9	0.00	1----------	01000010101100100011001110010110 7	7	0.00	-3.00
37:	----------	00000110110000010100111010100101 0	15	0.00	----------	x----------011	14	0.00	-3.00
38:	----------	00111100101111111010001110110000 8	8	0.00	----------	00100101011001000111000001010101 6	6	0.00	-3.00
39:	----------	00000101111101110000011001100111 1	1	0.00	----------	----------0---1-1--- 9	9	0.00	-3.00
40:	----------	----------	2	0.00	----------	10111001010001001100100111001100 11	11	0.00	-3.00
41:	----------	01000010101100100011001110010110 7	7	0.00	----------	00111100101111111010001110110000 8	8	0.00	-3.00
42:	----------	----------	0	0.00	----------	01101001001010010010111011101100 12	12	0.00	-3.00
43:	----------	00100101011001000111000001010101 6	6	0.00	----------	----------	2	0.00	-3.00
44:	----------	01100011101010100010110001110011 13	13	0.00	----------	10001010101010011100001100111101 10	10	0.00	-3.00
45:	----------	10111001010001001100100111001100 11	11	0.00	----------	----------	0	0.00	-3.00
46:	----------	----------	5	0.00	----------	11110000101100100000101111111100 4	4	0.00	-3.00
47:	----------	01101001001010010010111011101100 12	12	0.00	----------	01100011101010100010110001110011 13	13	0.00	-3.00
48:	----------	00000101111101110000011001100111 1	1	0.00	----------	00111100101111111010001110110000 8	8	0.00	-3.00
49:	----------	----------0---1-1--- 9	9	0.00	----------	00100101011001000111000001010101 6	6	0.00	-3.00
50:	----------	10111001010001001100100111001100 11	11	0.00	----------	11110000101100100000101111111100 4	4	0.00	-3.00
51:	----------	10001010101010011100001100111101 10	10	0.00	----------	00000101111101110000011001100111 1	1	0.00	-3.00
52:	----------	----------	0	0.00	----------	00101100100000011010000100101110 3	3	0.00	-3.00
53:	----------	00111100101111111010001110110000 8	8	0.00	----------	10111001010001001100100111001100 11	11	0.00	-3.00
54:	----------	01101001001010010010111011101100 12	12	0.00	----------	00000110110000010100111010100101 0	15	0.00	-3.00
55:	----------	11110000101100100000101111111100 4	4	0.00	----------	----------	0	0.00	-3.00
56:	----------	01100011101010100010110001110011 13	13	0.00	----------	----------	5	0.00	-3.00
57:	----------	00101100100000011010000100101110 3	3	0.00	----------	01101001001010010010111011101100 12	12	0.00	-3.00
58:	----------	01000010101100100011001110010110 7	7	-1.00	----------	----------	2	0.00	-3.00
59:	----------0--------	00000110110000010100111010100101 0	15	-1.00	----------	01100011101010100010110001110011 13	13	0.00	-2.00
60:	----------1--------	x----------011	14	0.00	----------	----------0---1-1--- 9	9	0.00	-1.00
61:	----------x--------	----------	5	0.00	----------	01000010101100100011001110010110 7	7	0.00	-1.00
62:	----------	00100101011001000111000001010101 6	6	0.00	----------	10001010101010011100001100111101 10	10	0.00	-1.00
63:	----------	----------	2	-1.00	----------	x----------011	14	0.00	-1.00
64:	----------				----------x--------				

Fig. 3. The differential path for `RIPEMD-128`, after the second phase of the freedom degree utilization. The notations are the same as in [4] and are described in Appendix B. The column $P^l[i]$ (resp. $P^r[i]$) represents the $\log_2()$ differential probability of step i in left (resp. right) branch. The column $P[i]$ represents the cumulated probability (in $\log_2()$) until step i for both branches, i.e. $P[i] = \prod_{j=63}^{j=i}(P^r[j] \cdot P^l[j])$.

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

$$X_2 = X_6^{\ggg 8} \boxminus (X_5 \oplus X_4 \oplus X_3) \boxminus M_5 = C_1 \boxminus M_5$$
$$X_1 = X_5^{\ggg 5} \boxminus (X_4 \oplus X_3 \oplus X_2) \boxminus M_4 = \texttt{0xffffffff} \boxminus (X_4 \oplus X_3 \oplus X_2) = \overline{X_4} \oplus X_3 \oplus X_2 = \overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5)$$

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:

$$\begin{aligned} X_0 &= X_4^{\ggg 12} \boxminus (X_3 \oplus X_2 \oplus X_1) \boxminus M_3 = X_4^{\ggg 12} \boxminus \overline{X_4} \boxminus M_3 \\ = Y_0 &= Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus Y_2 \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r = Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus (C_0 \boxminus M_2) \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r \end{aligned}$$

$$\begin{aligned} X_{-1} &= X_3^{\ggg 15} \boxminus (X_2 \oplus X_1 \oplus X_0) \boxminus M_2 = X_3^{\ggg 15} \boxminus (\overline{X_4} \oplus X_3 \oplus X_0) \boxminus M_2 \\ = Y_{-1} &= Y_3^{\ggg 9} \boxminus (Y_2 \wedge Y_0 \oplus Y_1 \wedge \overline{Y_0}) \boxminus M_7 \boxminus K_0^r = Y_3^{\ggg 9} \boxminus ((C_0 \boxminus M_2) \wedge X_0 \oplus Y_1 \wedge \overline{X_0}) \boxminus M_7 \boxminus K_0^r \end{aligned}$$

$$\begin{aligned} X_{-2} &= X_2^{\ggg 14} \boxminus (X_1 \oplus X_0 \oplus X_{-1}) \boxminus M_1 = (C_1 \boxminus M_5)^{\ggg 14} \boxminus (\overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5) \oplus X_0 \oplus X_{-1}) \boxminus M_1 \\ = Y_{-2} &= Y_2^{\ggg 9} \boxminus (Y_1 \wedge Y_{-1} \oplus Y_0 \wedge \overline{Y_{-1}}) \boxminus M_{14} \boxminus K_0^r = (C_0 \boxminus M_2)^{\ggg 9} \boxminus (Y_1 \wedge X_{-1} \oplus X_0 \wedge \overline{X_{-1}}) \boxminus M_{14} \boxminus K_0^r \end{aligned}$$

$$\begin{aligned} X_{-3} &= X_1^{\ggg 11} \boxminus (X_0 \oplus X_{-1} \oplus X_{-2}) \boxminus M_0 = (\overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5))^{\ggg 11} \boxminus (X_0 \oplus X_{-1} \oplus X_{-2}) \boxminus M_0 \\ = Y_{-3} &= Y_1^{\ggg 8} \boxminus (Y_0 \wedge Y_{-2} \oplus Y_{-1} \wedge \overline{Y_{-2}}) \boxminus M_5 \boxminus K_0^r = Y_1^{\ggg 8} \boxminus (X_0 \wedge X_{-2} \oplus X_{-1} \wedge \overline{X_{-2}}) \boxminus M_5 \boxminus K_0^r \end{aligned}$$

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 give in Appendix C more details on 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 3 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, we were able to do the merging of the two 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.

The attack starts at the end of phase 1, with the path from Figure 7 of Appendix F. From here, he generates $2^{38.32}$ starting points in phase 2, that is $2^{38.32}$ differential paths like the one from Figure 3 (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 semi-free-start 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 computations in each branch).

The merge process has been implemented and we give in Appendix D an example of a message and chaining variable couple that verifies the merge (i.e. they follow the differential path from Figure 6 of Appendix F 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 semi-free-start collision final complexity of $2^{61.88}$ RIPEMD-128 compression function computations. While our practical results confirm our theoretical estimations, we emphasize that the latter are a bit pessimistic, since our attack implementation is not really 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 semi-free-start 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.

5.2 Collision for the RIPEMD-128 compression function

We described in previous sections a semi-free-start collision attack for the full RIPEMD-128 compression function with $2^{61.57}$ computations. It is clear from Figure 3 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 Appendix E 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 complexity on the full primitive.

5.3 Distinguishers

The setting for the distinguisher is very simple. As non-randomness property, the attacker will find one input m, such that $H(m) \oplus H(m \oplus \Delta_I) = \Delta_O$. In other words, he will find an input such that with a fixed and predetermined difference Δ_I applied on it, he observes another fixed and predetermined difference Δ_O on the output. This problem is called the limited-birthday [11] because the fixed differences removes the ability of an attacker to use a birthday-like algorithm when H is a random function. The best known algorithm to find such an input for a random function is to simply pick random inputs m and check if the property is verified. This has a cost of 2^{128} computations for a 128-bit output function.

Of course, considering the differential path we built in previous sections, in our case we will use $\Delta_O = 0$ and Δ_I is defined to contain no difference on the input chaining variable, and only a difference on the most significant bit of M_{14}. If we can find such an input with less than 2^{128} computations for RIPEMD-128, we obtain a distinguisher.

Distinguisher for the RIPEMD-128 compression function. A collision attack on the RIPEMD-128 compression function can already be considered a distinguisher. However, we remark that since the complexity gap between the attack ($2^{61.57}$) and the generic case (2^{128}) is very big, we can relax some of the conditions in the differential path to reduce the distinguisher cost. Indeed, we can straightforwardly relax the collision condition on the compression function finalization, as well as the condition in the last step of the left branch. Overall, the distinguisher complexity is $2^{59.57}$, while the generic cost will be very slightly less than 2^{128} computations because a small set of possible differences Δ_O can now be reached on the output.

Distinguisher for the RIPEMD-128 hash function. The two main distinctions when attacking the hash function compared to the compression function is first that the input chaining variable is now specified to a public IV and second that a part of the message has to contain the padding.

Since the chaining variable is fixed, we can not apply our merging algorithm as in Section 4. Instead, we utilize the available freedom degrees (the message words) to handle only one of the two non-linear parts, namely the one in the right branch because it is the most complex. We use the same method as in phase 2 in Section 4 and we very quickly obtain a differential path such as the one in Figure 8 of Appendix F. One can remark that the six first message inserted in the right branch are free (M_5, M_{14}, M_7, M_0, M_9 and M_2) and we will fix them to merge the right branch to the predefined input chaining variable. The entirety of the left branch will be verified probabilistically (with probability $2^{-84.65}$) as well as the steps located after the non-linear part in the right branch (from step 19 with probability $2^{-19.75}$). The bit condition on the IV can be handled by prepending a random message, and the few conditions in the early steps when computing backwards are directly fulfilled when choosing M_2 and M_9.

Overall, adding the extra condition to obtain a collision after the finalization of the compression function, we end up with a complexity of $2^{105.4}$ computations to get a collision after the first message block. Once this collision found, we add an extra message block without difference to contain the padding and obtain a collision for the whole hash function. In the ideal case, generating a collision for a 128-bit output hash function with a predetermined difference mask on the message input requires 2^{128} computations, and we obtain a distinguisher for the full RIPEMD-128 hash function with $2^{105.4}$ computations.

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 semi-free-start 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 [28], and then MD5-protected certificates [24], we believe that another function than RIPEMD-128 should be used for new security applications.

Aside from reducing the complexity for collision attack on RIPEMD-128 compression function, future works includes application of our methods to RIPEMD-160 and other parallel branches-based functions. It would also be interesting to scrutinize if there is any way that some other freedom degrees techniques (neutral bits, message modifications, etc.) could be used on top of our merging process.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments.

References

1. Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Keccak specifications. Submission to NIST, 2008. `http://keccak.noekeon.org/Keccak-specifications.pdf`.
2. Antoon Bosselaers and Bart Preneel, editors. *Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*, volume 1007 of *LNCS*. Springer, 1995.
3. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, Proceedings*, volume 435 of *LNCS*. Springer, 1990.
4. Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *LNCS*, pages 1–20. Springer, 2006.
5. Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, Proceedings*, volume 3494 of *LNCS*. Springer, 2005.
6. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [3], pages 416–427.
7. Bert den Boer and Antoon Bosselaers. Collisions for the Compressin Function of MD5. In Tor Helleseth, editor, *EUROCRYPT*, volume 765 of *LNCS*, pages 293–304. Springer, 1993.
8. H. Dobbertin. Cryptanalysis of MD5 compress. In Rump Session of Advances in Cryptology EUROCRYPT 1996, 1996. `ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf`.
9. Hans Dobbertin. RIPEMD with Two-Round Compress Function is Not Collision-Free. *J. Cryptology*, 10(1):51–70, 1997.
10. Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.
11. Henri Gilbert and Thomas Peyrin. Super-Sbox Cryptanalysis: Improved Attacks for AES-Like Permutations. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *LNCS*, pages 365–383. Springer, 2010.
12. ISO. *ISO/IEC 10118-3:2004: Information technology — Security techniques — Hash-functions — Part 3: Dedicated hash-functions*. pub-ISO, pub-ISO:adr, feb 2004.
13. Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *LNCS*, pages 244–263. Springer, 2007.
14. Stéphane Manuel and Thomas Peyrin. Collisions on SHA-0 in One Hour. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 16–35. Springer, 2008.
15. Florian Mendel, Tomislav Nad, Stefan Scherz, and Martin Schläffer. Differential Attacks on Reduced RIPEMD-160. In Dieter Gollmann and Felix C. Freiling, editors, *ISC*, volume 7483 of *LNCS*, pages 23–38. Springer, 2012.
16. Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision Attacks on the Reduced Dual-Stream Hash Function RIPEMD-128. In Anne Canteaut, editor, *FSE*, volume 7549 of *LNCS*, pages 226–243. Springer, 2012.
17. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. On the Collision Resistance of RIPEMD-160. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC*, volume 4176 of *LNCS*, pages 101–116. Springer, 2006.
18. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [3], pages 428–446.
19. Chiaki Ohtahara, Yu Sasaki, and Takeshi Shimoyama. Preimage Attacks on Step-Reduced RIPEMD-128 and RIPEMD-160. In Xuejia Lai, Moti Yung, and Dongdai Lin, editors, *Inscrypt*, volume 6584 of *LNCS*, pages 169–186. Springer, 2010.
20. Ronald L. Rivest. The MD4 message-digest algorithm. Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.
21. Yu Sasaki and Kazumaro Aoki. Meet-in-the-Middle Preimage Attacks on Double-Branch Hash Functions: Application to RIPEMD and Others. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 5594 of *LNCS*, pages 214–231. Springer, 2009.
22. Yu Sasaki and Lei Wang. Distinguishers beyond Three Rounds of the RIPEMD-128/-160 Compression Functions. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *ACNS*, volume 7341 of *LNCS*, pages 275–292. Springer, 2012.
23. Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005, Proceedings*, volume 3621 of *LNCS*. Springer, 2005.
24. Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 55–69. Springer, 2009.
25. Lei Wang, Yu Sasaki, Wataru Komatsubara, Kazuo Ohta, and Kazuo Sakiyama. (Second) Preimage Attacks on Step-Reduced RIPEMD/RIPEMD-128 with a New Local-Collision Approach. In Aggelos Kiayias, editor, *CT-RSA*, volume 6558 of *LNCS*, pages 197–212. Springer, 2011.
26. Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [5], pages 1–18.
27. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Shoup [23], pages 17–36.
28. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [5], pages 19–35.
29. Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Shoup [23], pages 1–16.

A Tables and constants for `RIPEMD-128` compression function

Table 2. Initialization values in RIPEMD-128

$X_{-3} = Y_{-3} = \texttt{0x67452301} \quad X_{-2} = Y_{-2} = \texttt{0xefcdab89} \quad X_{-1} = Y_{-1} = \texttt{0x98badcfe} \quad X_0 = Y_0 = \texttt{0x10325476}$

Table 3. Word permutations for the message expansion in RIPEMD-128

round j	$\pi_j^l(k)$																$\pi_j^r(k)$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	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
1	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
2	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
3	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

Table 4. Boolean functions and round constants in RIPEMD-128, with $\mathrm{XOR}(x,y,z) := x \oplus y \oplus z$, $\mathrm{IF}(x,y,z) := x \wedge y \oplus \bar{x} \wedge z$ and $\mathrm{ONX}(x,y,z) := (x \vee \bar{y}) \oplus z$

round j	$\Phi_j^l(x,y,z)$	$\Phi_j^r(x,y,z)$	K_j^l	K_j^r
0	$\texttt{XOR}(x,y,z)$	$\texttt{IF}(z,x,y)$	0x00000000	0x50a28be6
1	$\texttt{IF}(x,y,z)$	$\texttt{ONX}(x,y,z)$	0x5a827999	0x5c4dd124
2	$\texttt{ONX}(x,y,z)$	$\texttt{IF}(x,y,z)$	0x6ed9eba1	0x6d703ef3
3	$\texttt{IF}(z,x,y)$	$\texttt{XOR}(x,y,z)$	0x8f1bbcdc	0x00000000

Table 5. Rotation constants in RIPEMD-128

round j	$s^l_{16\cdot j+k}$																$s^r_{16\cdot j+k}$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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
1	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
2	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
3	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

B Notations for the differential paths

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓
-	✓	-	-	✓
x	-	✓	✓	-
0	✓	-	-	-
u	-	✓	-	-
n	-	-	✓	-
1	-	-	-	✓
#	-	-	-	-

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
3	✓	✓	-	-
5	✓	-	✓	-
7	✓	✓	✓	-
A	-	✓	-	✓
B	✓	✓	-	✓
C	-	-	✓	✓
D	✓	-	✓	✓
E	-	✓	✓	✓

Table 6. Notations used in [4] for a differential path: x represents a bit of the first message and x^* stands for the same bit of the second message.

C Solving the T-function $X_{-1} = Y_{-1}$ during the merging phase

The equation $X_{-1} = Y_{-1}$ can be written:

$$X_3^{\ggg 15} \boxminus (\overline{X_4} \oplus X_3 \oplus X_0) \boxminus M_2 = Y_3^{\ggg 9} \boxminus ((C_0 \boxminus M_2) \wedge X_0 \oplus Y_1 \wedge \overline{X_0}) \boxminus M_7 \boxminus K_0^r$$

which can in turn be transformed into

$$((C_0 \boxminus M_2) \wedge X_0 \oplus Y_1 \wedge \overline{X_0}) = M_2 \boxplus X_3^{\ggg 15} \boxminus (\overline{X_4} \oplus X_3 \oplus X_0) \boxminus Y_3^{\ggg 9} \boxminus M_7 \boxminus K_0^r$$
$$((C_0 \boxminus M_2) \wedge c \oplus a) = M_2 \boxplus b$$

where a, b and c are known random values. Such an equation is a triangular function, or T-function, in the sense that any bit i of the equation depends only on the i first bits of M_2, and it can be solved very efficiently. The algorithm to find a solution M_2 is simply to fix the first bit of M_2 and check if the equation is verified up to its first bit. Then we go to the second bit, etc. and the total cost is 32 operations on average. In practice, a table-based solver is much faster than really going bit per bit. Since the equation is parametrized by 3 random values a, b and c, we can build 24-bit precomputed tables and directly solve byte per byte. On average, finding a solution for this equation only requires a few operations, equivalent to 1 `RIPEMD-128` step computation.

D Solution example for the merging

We provide in hexadecimal notation, a message and chaining variable couple that succeeded in the merge process. The second member of the pair is simply obtained by adding a difference on the most significant bit of M_{14}.

$$\begin{array}{llll} h_3 = \texttt{0x1330db09} & h_2 = \texttt{0xe1c2cd59} & h_1 = \texttt{0xd3160c1d} & h_0 = \texttt{0xd9b11816} \\ M_0 = \texttt{0x4b6adf53} & M_1 = \texttt{0x1e69c794} & M_2 = \texttt{0x0eafe77c} & M_3 = \texttt{0x35a1b389} \\ M_4 = \texttt{0x34a56d47} & M_5 = \texttt{0x0634d566} & M_6 = \texttt{0xb567790c} & M_7 = \texttt{0xa0324005} \\ M_8 = \texttt{0x8162d2b0} & M_9 = \texttt{0x6632792a} & M_{10} = \texttt{0x52c7fb4a} & M_{11} = \texttt{0x16b9ce57} \\ M_{12} = \texttt{0x914dc223} & M_{13} = \texttt{0x3bafc9de} & M_{14} = \texttt{0x5402b983} & M_{15} = \texttt{0xe08f7842} \end{array}$$

E Collision attack for 63-step `RIPEMD-128` compression function

In the case of 63-step `RIPEMD-128` compression function (the first step being removed), the merging process is easier to handle. Indeed, the constraint $X_5^{\ggg 5} \boxminus M_4 = \texttt{0xffffffff}$ is no longer required and the attacker can directly use M_9 for randomization. Therefore, instead of 19 `RIPEMD-128` step computations, one requires only 12 (there are 12 steps to compute backwards after having chosen a value for M_9). Moreover, the message M_9 being now free to use, with two more bits values pre-specified one can remove an extra condition in step 26 of left branch when computing X_{27}. This is depicted in Figure 4. The gain factor is about $(19/12) \cdot 2^1 = 2^{1.66}$ and the collision attack requires $2^{59.91}$ `RIPEMD-128` hash function computations.

The merging phase goal here is to have $X_{-2} = Y_{-2}$, $X_{-1} = Y_{-1}$, $X_0 = Y_0$ and $X_1 = Y_1$ and without the constraint $X_5^{\ggg 5} \boxminus M_4 = \texttt{0xffffffff}$, the value of X_2 must now be written:

$$X_2 = X_6^{\ggg 8} \boxminus (X_5 \oplus X_4 \oplus X_3) \boxminus M_5 = C_2 \boxminus M_5.$$

without further simplification. The equations for the merging are:

$$\begin{aligned} & X_1 = X_5^{\ggg 5} \boxminus (X_4 \oplus X_3 \oplus X_2) \boxminus M_4 = X_5^{\ggg 5} \boxminus (X_4 \oplus X_3 \oplus (C_2 \boxminus M_5)) \boxminus M_4 \\ = {} & Y_1 = Y_5^{\ggg 13} \boxminus (Y_4 \wedge Y_2 \oplus Y_3 \wedge \overline{Y_2}) \boxminus M_9 \boxminus K_0^r = Y_5^{\ggg 13} \boxminus Y_3 \boxminus M_9 \boxminus K_0^r \end{aligned}$$

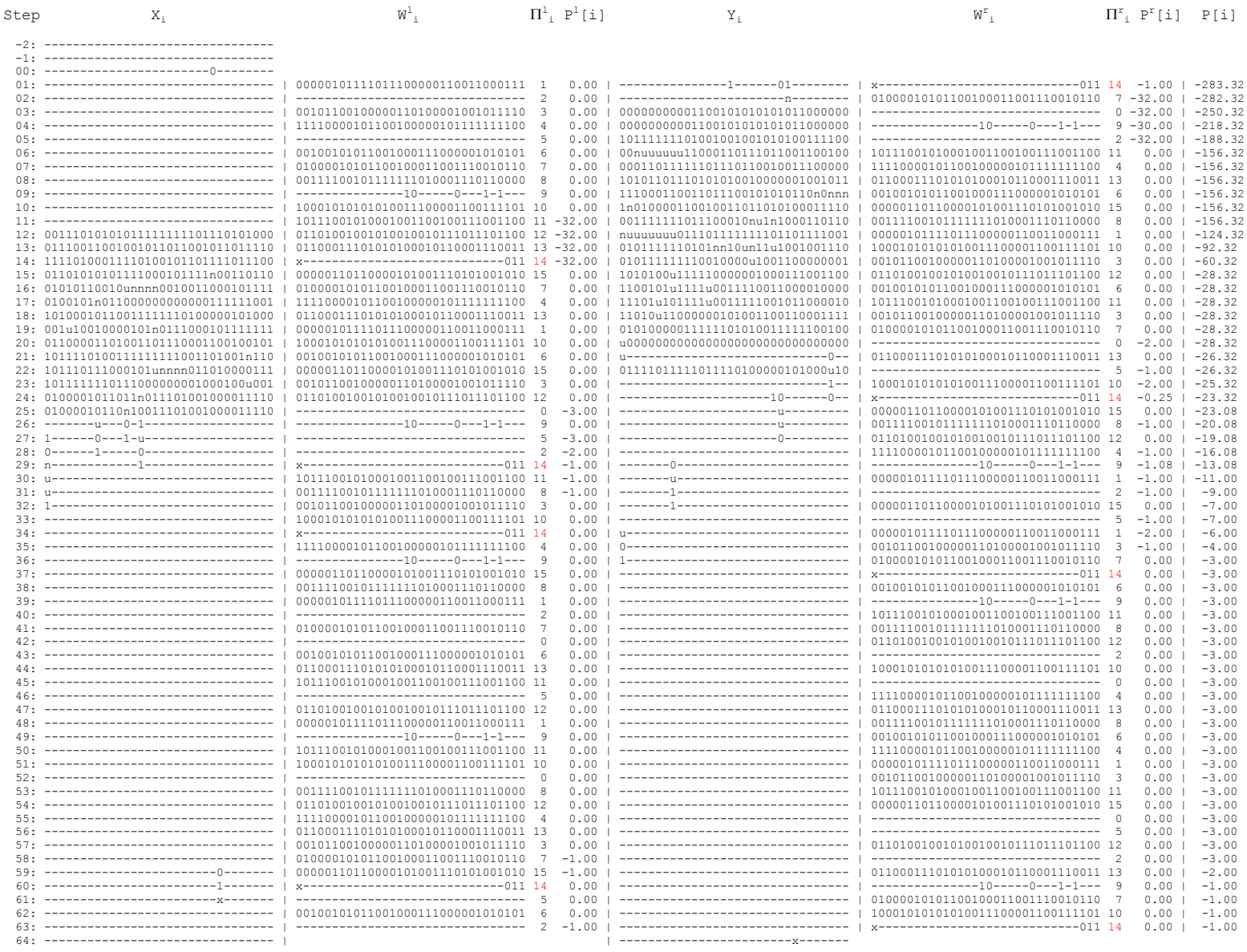
$$\begin{aligned} & X_0 = X_4^{\ggg 12} \boxminus (X_3 \oplus X_2 \oplus X_1) \boxminus M_3 = X_4^{\ggg 12} \boxminus (X_3 \oplus (C_1 \boxminus M_5) \oplus (X_5^{\ggg 5} \boxminus (X_4 \oplus X_3 \oplus (C_2 \boxminus M_5)) \boxminus M_4)) \boxminus M_3 \\ = {} & Y_0 = Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus Y_2 \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r = Y_4^{\ggg 11} \boxminus (Y_3 \wedge Y_1 \oplus (C_0 \boxminus M_2) \wedge \overline{Y_1}) \boxminus M_0 \boxminus K_0^r \end{aligned}$$

$$\begin{aligned} & X_{-1} = X_3^{\ggg 15} \boxminus (X_2 \oplus X_1 \oplus X_0) \boxminus M_2 = X_3^{\ggg 15} \boxminus (\overline{X_4} \oplus X_3 \oplus X_0) \boxminus M_2 \\ = {} & Y_{-1} = Y_3^{\ggg 9} \boxminus (Y_2 \wedge Y_0 \oplus Y_1 \wedge \overline{Y_0}) \boxminus M_7 \boxminus K_0^r = Y_3^{\ggg 9} \boxminus ((C_0 \boxminus M_2) \wedge X_0 \oplus Y_1 \wedge \overline{X_0}) \boxminus M_7 \boxminus K_0^r \end{aligned}$$

$$\begin{aligned} & X_{-2} = X_2^{\ggg 14} \boxminus (X_1 \oplus X_0 \oplus X_{-1}) \boxminus M_1 = (C_1 \boxminus M_5)^{\ggg 14} \boxminus (\overline{X_4} \oplus X_3 \oplus (C_1 \boxminus M_5) \oplus X_0 \oplus X_{-1}) \boxminus M_1 \\ = {} & Y_{-2} = Y_2^{\ggg 9} \boxminus (Y_1 \wedge Y_{-1} \oplus Y_0 \wedge \overline{Y_{-1}}) \boxminus M_{14} \boxminus K_0^r = (C_0 \boxminus M_2)^{\ggg 9} \boxminus (Y_1 \wedge X_{-1} \oplus X_0 \wedge \overline{X_{-1}}) \boxminus M_{14} \boxminus K_0^r \end{aligned}$$

The merging is then very simple: Y_1 is already fully determined so the attacker directly deduces M_5 from the equation $X_1 = Y_1$, which in turns allows him to deduce the value of X_0. Using this information, he solves the T-function to deduce M_2 from the equation $X_{-1} = Y_{-1}$. He finally directly recovers M_0 from equation $X_0 = Y_0$ and the last equation $X_{-2} = Y_{-2}$ is not controlled, thus only verified with probability 2^{-32}.

F The differential paths



Step	X_i	W^l_i	Π^l_i	$P^l[i]$	Y_i	W^r_i	Π^r_i	$P^r[i]$	$P[i]$
-2:	----------								
-1:	----------								
00:	----------0--------								
01:	----------	00000101111101110000011001100011	1	0.00	----------1------01--------	x----------011	14	-1.00	-283.32
02:	----------	----------	2	0.00	----------n--------	01000010101100100011001110010110	7	-32.00	-282.32
03:	----------	00101100100000110100001001011110	3	0.00	00000000001100101010101011000000	----------	0	-32.00	-250.32
04:	----------	11110000101100100000101111111100	4	0.00	00000000001100101010101011000000	----------10-----0---1-1---	9	-30.00	-218.32
05:	----------	----------	5	0.00	10111111101001001001010100111100	----------	2	-32.00	-188.32
06:	----------	00100101011001000111000001010101	6	0.00	00nuuuuuu1100011011101100100100	10111001010001001100100111001100	11	0.00	-156.32
07:	----------	01000010101100100011001110010110	7	0.00	00011011111101110110100100111000000	11110000101100100000101111111100	4	0.00	-156.32
08:	----------	00111100101111111010001110100000	8	0.00	10101101110101010010000001001011	01100011101010100010110001110011	13	0.00	-156.32
09:	----------	----------10-----0---1-1---	9	0.00	11100011001101110010101010n0nnn	00100101011001000111000001010101	6	0.00	-156.32
10:	----------	10001010101010011100001100111101	10	0.00	1n0100001100100110110100100111110	00000110110000101001110100101010	15	0.00	-156.32
11:	----------	10111001010001001100100111001100	11	-32.00	00111111101110000010nu1n1000110110	00111100101111111010001110110000	8	0.00	-156.32
12:	00111010101011111111110111010101000	01101001001010010010111011101100	12	-32.00	nuuuuuuu01110111111110110111001	00000101111101110000011001100111	1	0.00	-124.32
13:	01110011001001011011001011011110	01100011101010100010110001110011	13	-32.00	0101111110101nn10un11u1001001110	10001010101010011100001100111101	10	0.00	-92.32
14:	11110100011110100101101111011100	x----------011	14	-32.00	01011111111001000u1001100000001	00101100100000110100001001011110	3	0.00	-60.32
15:	01101010101111000101111n00110100	00000110110000101001110100101010	15	0.00	1010100u1111100000010001111001100	01101001001010010010111011101100	12	0.00	-28.32
16:	01010110010unnnn0010011000101111	01000010101100100011001110010110	7	0.00	1100101u1111u0011110011000010000	00100101011001000111000001010101	6	0.00	-28.32
17:	0100101n01100000000000001111111001	11110000101100100000101111111100	4	0.00	11101u101111u0011111001011000010	10111001010001001100100111001100	11	0.00	-28.32
18:	10100010110011111111010000101000	01100011101010100010110001110011	13	0.00	11010u11000000101001100110001111	00101100100000110100001001011110	3	0.00	-28.32
19:	001u10010000101n0111000101111111	00000101111101110000011001100111	1	0.00	01010000011111101010011111100100	01000010101100100011001110010110	7	0.00	-28.32
20:	01100001101001011011000100100101	10001010101010011100001100111101	10	0.00	u0000000000000000000000000000000	----------	0	-2.00	-28.32
21:	10111010011111111111001101001n110	00100101011001000111000001010101	6	0.00	u----------0--	01100011101010100010110001110011	13	0.00	-26.32
22:	101110111000101unnnn011010000111	00000110110000101001110101001010	15	0.00	01111011111011110100000101000u10	----------	5	-1.00	-26.32
23:	10111111101110000000001000100u001	00101100100000110100001001011110	3	0.00	----------1--	10001010101010011100001100111101	10	-2.00	-25.32
24:	0100001011011n011101001000011110	01101001001010010010111011101100	12	0.00	----------10------0--	x----------011	14	-0.25	-23.32
25:	01000010110n10011101001000011110	----------	0	-3.00	----------u---------	00000110110000101001110101001010	15	0.00	-23.08
26:	-------u---0-1----------	----------10-----0---1-1---	9	0.00	----------u---------	00111100101111111010001110110000	8	-1.00	-20.08
27:	1------0---1-u----------	----------	5	-3.00	----------0---------	01101001001010010010111011101100	12	0.00	-19.08
28:	0------1-----0----------	----------	2	-2.00	----------	11110000101100100000101111111100	4	-1.00	-16.08
29:	n----------1----------	x----------011	14	-1.00	-------0----------	----------10-----0---1-1---	9	-1.08	-13.08
30:	u----------	10111001010001001100100111001100	11	-1.00	-------u----------	00000101111101110000011001100111	1	-1.00	-11.00
31:	u----------	00111100101111111010001110110000	8	-1.00	-------1----------	----------	2	-1.00	-9.00
32:	1----------	00101100100000110100001001011110	3	0.00	-------1----------	00000110110000101001110101001010	15	0.00	-7.00
33:	----------	10001010101010011100001100111101	10	0.00	----------	----------	5	-1.00	-7.00
34:	----------	x----------011	14	0.00	u----------	00000101111101110000011001100111	1	-2.00	-6.00
35:	----------	11110000101100100000101111111100	4	0.00	0----------	00101100100000110100001001011110	3	-1.00	-4.00
36:	----------	----------10-----0---1-1---	9	0.00	1----------	01000010101100100011001110010110	7	0.00	-3.00
37:	----------	00000110110000101001110101001010	15	0.00	----------	x----------011	14	0.00	-3.00
38:	----------	00111100101111111010001110110000	8	0.00	----------	00100101011001000111000001010101	6	0.00	-3.00
39:	----------	00000101111101110000011001100111	1	0.00	----------	----------10-----0---1-1---	9	0.00	-3.00
40:	----------	----------	2	0.00	----------	10111001010001001100100111001100	11	0.00	-3.00
41:	----------	01000010101100100011001110010110	7	0.00	----------	00111100101111111010001110110000	8	0.00	-3.00
42:	----------	----------	0	0.00	----------	01101001001010010010111011101100	12	0.00	-3.00
43:	----------	00100101011001000111000001010101	6	0.00	----------	----------	2	0.00	-3.00
44:	----------	01100011101010100010110001110011	13	0.00	----------	10001010101010011100001100111101	10	0.00	-3.00
45:	----------	10111001010001001100100111001100	11	0.00	----------	----------	0	0.00	-3.00
46:	----------	----------	5	0.00	----------	11110000101100100000101111111100	4	0.00	-3.00
47:	----------	01101001001010010010111011101100	12	0.00	----------	01100011101010100010110001110011	13	0.00	-3.00
48:	----------	00000101111101110000011001100111	1	0.00	----------	00111100101111111010001110110000	8	0.00	-3.00
49:	----------	----------10-----0---1-1---	9	0.00	----------	00100101011001000111000001010101	6	0.00	-3.00
50:	----------	10111001010001001100100111001100	11	0.00	----------	11110000101100100000101111111100	4	0.00	-3.00
51:	----------	10001010101010011100001100111101	10	0.00	----------	00000101111101110000011001100111	1	0.00	-3.00
52:	----------	----------	0	0.00	----------	00101100100000110100001001011110	3	0.00	-3.00
53:	----------	00111100101111111010001110110000	8	0.00	----------	10111001010001001100100111001100	11	0.00	-3.00
54:	----------	01101001001010010010111011101100	12	0.00	----------	00000110110000101001110101001010	15	0.00	-3.00
55:	----------	11110000101100100000101111111100	4	0.00	----------	----------	0	0.00	-3.00
56:	----------	01100011101010100010110001110011	13	0.00	----------	----------	5	0.00	-3.00
57:	----------	00101100100000110100001001011110	3	0.00	----------	01101001001010010010111011101100	12	0.00	-3.00
58:	----------	01000010101100100011001110010110	7	-1.00	----------	----------	2	0.00	-3.00
59:	----------0--------	00000110110000101001110101001010	15	-1.00	----------	01100011101010100010110001110011	13	0.00	-2.00
60:	----------1--------	x----------011	14	0.00	----------	----------10-----0---1-1---	9	0.00	-1.00
61:	----------x--------	----------	5	0.00	----------	01000010101100100011001110010110	7	0.00	-1.00
62:	----------	00100101011001000111000001010101	6	0.00	----------	10001010101010011100001100111101	10	0.00	-1.00
63:	----------	----------	2	-1.00	----------	x----------011	14	0.00	-1.00
64:	----------				----------x--------				

Fig. 4. The differential path for `RIPEMD-128` reduced to 63 steps (the first step being removed), after the second phase of the freedom degree utilization. The notations are the same as in [4] and are described in Appendix B. The column $\mathrm{P}^l[i]$ (resp. $\mathrm{P}^r[i]$) represents the $\log_2()$ differential probability of step i in left (resp. right) branch. The column $P[i]$ represents the cumulated probability (in $\log_2()$) until step i for both branches, i.e. $\mathrm{P}[i] = \prod_{j=63}^{j=i}(\mathrm{P}^r[j] \cdot \mathrm{P}^l[j])$.

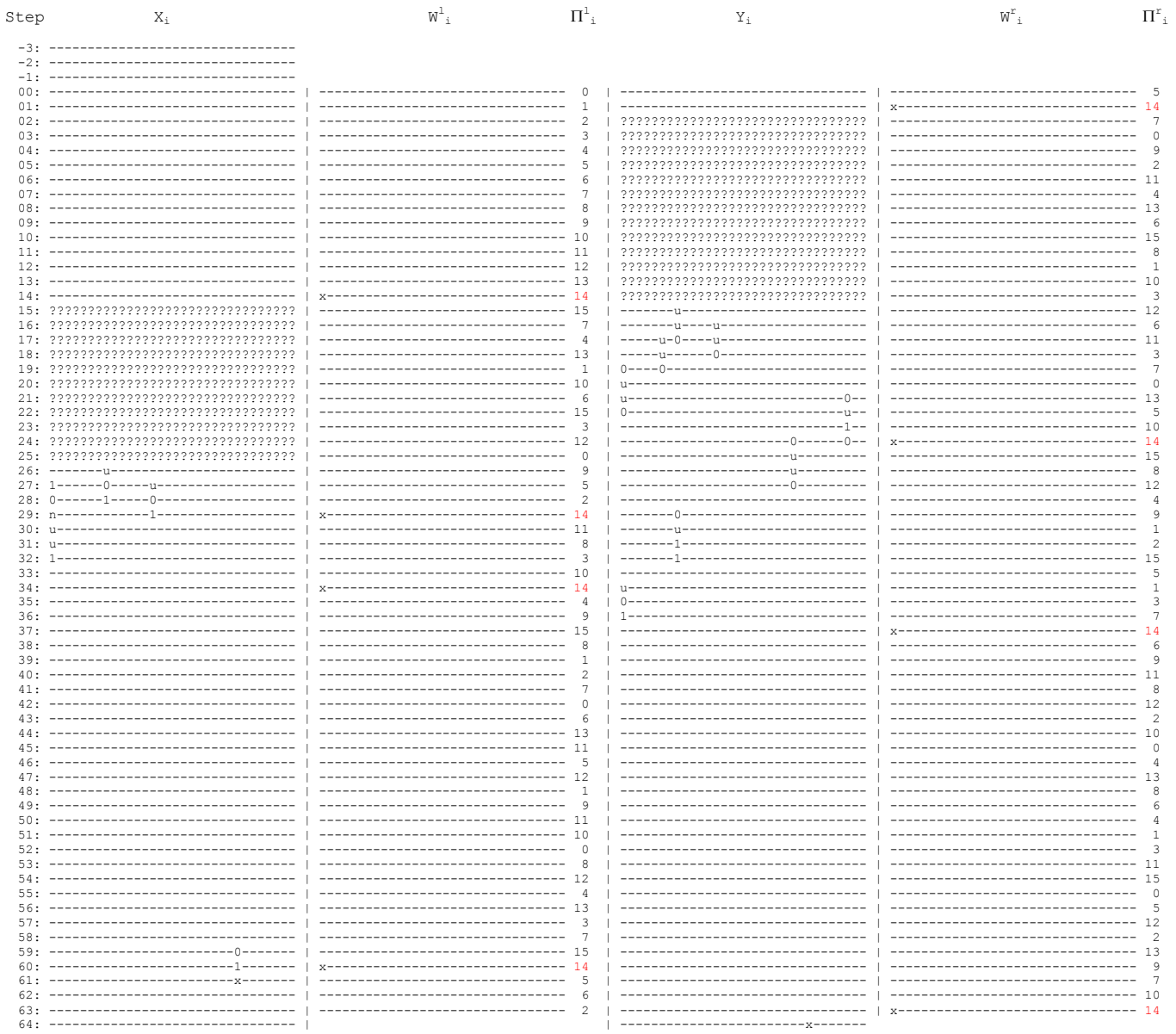


Fig. 5. The differential path for `RIPEMD-128`, before the non-linear parts search. The notations are the same as in [4] and are described in Appendix B. The column $\pi^l_j(k)$ (resp. $\pi^r_j(k)$) represents the message words that are inserted at each step in left branch (resp. right branch).

Fig. 6. The differential path for `RIPEMD-128`, after the non-linear parts search. The notations are the same as in [4] and are described in Appendix B. The column $\mathrm{P}^l[i]$ (resp. $\mathrm{P}^r[i]$) represents the $\log_2()$ differential probability of step i in left (resp. right) branch. The column $P[i]$ represents the cumulated probability (in $\log_2()$) until step i for both branches, i.e. $\mathrm{P}[i] = \prod_{j=63}^{j=i}(\mathrm{P}^r[j] \cdot \mathrm{P}^l[j])$.

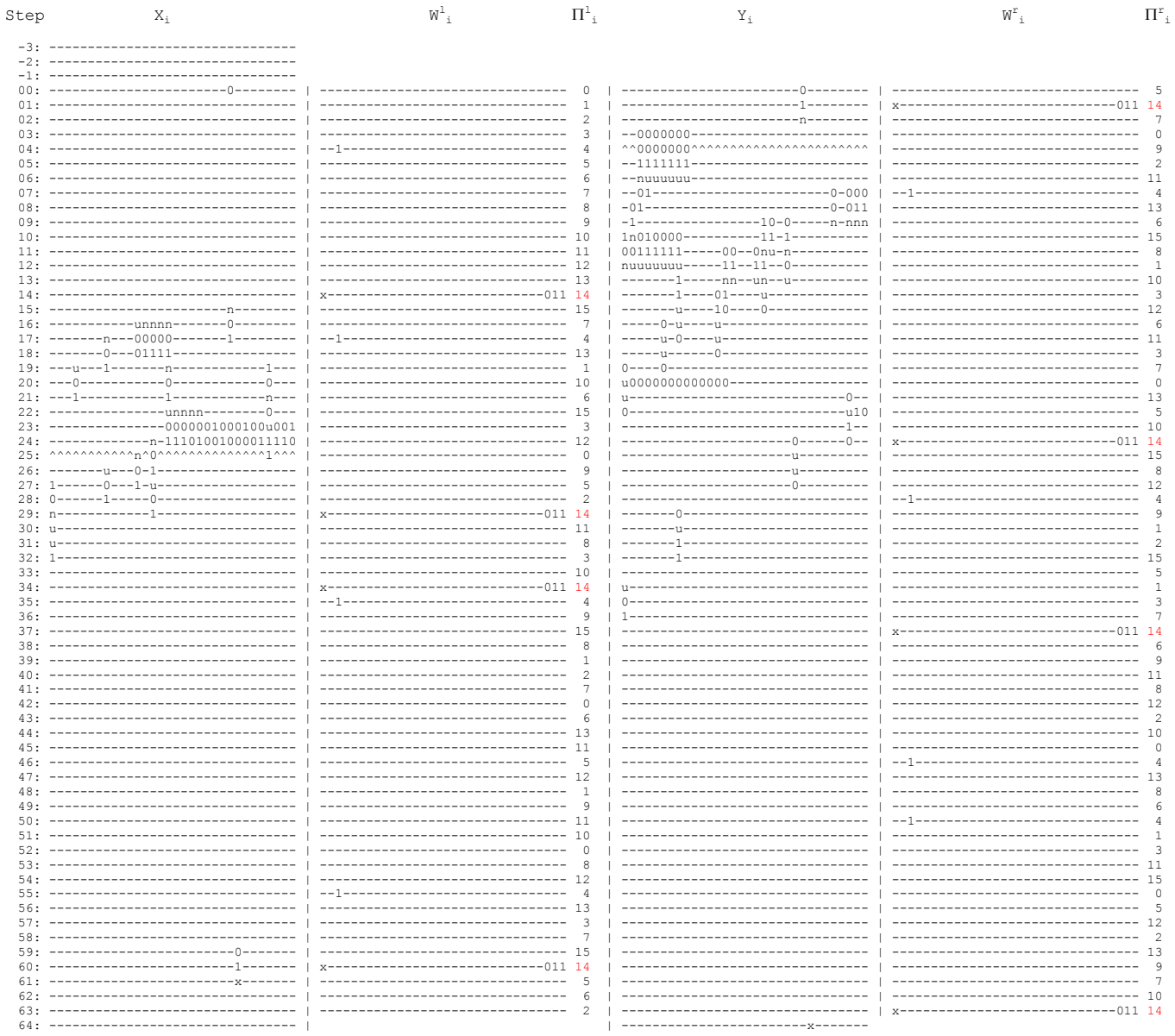


Fig. 7. The differential path for `RIPEMD-128`, after the non-linear parts search. The notations are the same as in [4] and are described in Appendix B. Moreover we denote by "^" the constraint on a bit $[X_i]_j$ that $[X_i]_j = [X_{i-1}]_j$. The column $\pi^l_j(k)$ (resp. $\pi^r_j(k)$) represents the message words that are inserted at each step in left branch (resp. right branch).

Step	X_i	W^l_i	Π^l_i	$P^l[i]$	Y_i	W^r_i	Π^r_i	$P^r[i]$	$P[i]$
-3:	----------								
-2:	----------								
-1:	----------								
00:	----------0--------	----------	0	0.00	----------0--------	----------	5	-1.00	-234.40
01:	----------	00011110011010011100011110010100	1	0.00	----------1--------	x----------	14	-1.00	-233.40
02:	----------	----------	2	0.00	----------n--------	----------	7	-32.00	-232.40
03:	----------	00110101101000011011001110001001	3	0.00	10000000011100101000111111000000	----------	0	-32.00	-200.40
04:	----------	00110100101001010111011010000111	4	0.00	10000000011100101000111111000000	----------	9	-32.00	-168.40
05:	----------	----------	5	0.00	01111111110000011011010011110001	----------	2	-32.00	-136.40
06:	----------	10110101011001110111100100001100	6	0.00	01nuuuuuu11100000011110011110100	00010110101110011100111001010111	11	0.00	-104.40
07:	----------	----------	7	0.00	00010011011110111001100111100000	00110100101001010110111010100111	4	0.00	-104.40
08:	----------	10000001011000101101001010110000	8	0.00	00100101011101111111110100100011	00111011101011111100100111011110	13	0.00	-104.40
09:	----------	----------	9	0.00	11000101101001011010101001nlnnn	10110101011001111011110010001100	6	0.00	-104.40
10:	----------	01010010110001111111110110010010	10	0.00	1n01000000010110001101001100101	11100000010001111011110000010000010	15	0.00	-104.40
11:	----------	00010110101110011100111001010111	11	0.00	00111111011010010nu0n0100000101	10000001011000101101001010110000	8	0.00	-104.40
12:	----------	10010001010011011100001000100011	12	0.00	nuuuuuuu01100111111100000110101	00011110011010011100011110010100	1	0.00	-104.40
13:	----------	00111011101011111100100111011110	13	0.00	00101011110l0nn11unl1u0100001000	01010010110001111111110110100101010	10	0.00	-104.40
14:	----------	x----------	14	-1.00	00101011010101010101u0011110110010	00011010110100001101100111000001	3	0.00	-104.40
15:	----------n--------	11100000100011110111100001000010	15	-7.00	0010001u11011001110011111000110000	10010001010011011100001000100011	12	0.00	-103.40
16:	----------unnnn-------0--------	----------	7	-12.09	0100101u1000u0111000101000100001	10110101011001110111100100001100	6	0.00	-96.40
17:	-------n---00000-------1--------	00110100101001010111011010000111	4	-7.00	00000u0000000u0010001101001100101	00010110101110011100111001010111	11	0.00	-84.31
18:	-------0---01111----------------	00111011101011111100100111011110	13	-4.00	00110u11000000101000011011000000	00110101101000011011001110001001	3	0.00	-77.31
19:	---u---1-------n----------1---	00011110011010011100011110010100	1	-4.00	01100000111000000011010100100011	----------	7	-1.00	-73.31
20:	---0----------0----------0---	01010010110001111111110110010010	10	-2.56	u----------	----------	0	-2.00	-68.31
21:	---1----------1----------n---	10110101011001110111100100001100	6	-6.00	u----------0--	00111011101011111100100111011110	13	-3.75	-63.75
22:	---------unnnn--------0---	11100000100011110111100001000010	15	-10.00	00----------01101u10	----------	5	-1.00	-54.00
23:	---------00000--------u---	00110101101000011011001110001001	3	-7.00	----------1--	01010010110001111111110110100101010	10	-2.00	-43.00
24:	----------n-11101--------1---	10010001010011011100001000100011	12	-4.00	----------0------0--	x----------	14	-1.00	-34.00
25:	----------n-0----------1---	----------	0	-4.00	----------u----------	11100000100011110111100001000010	15	0.00	-29.00
26:	-------u---0-1----------------	----------	9	-5.00	----------u----------	10000001011000101101001010110000	8	-1.00	-25.00
27:	1------0---1-u----------------	----------	5	-3.00	----------0----------	10010001010011011100001000100011	12	0.00	-19.00
28:	0------1-----0----------------	----------	2	-2.00	----------	00110100101001010110110101000111	4	-1.00	-16.00
29:	n----------1----------------	x----------	14	-1.00	-------0----------	----------	9	-1.00	-13.00
30:	u----------	00010110101110011100111001010111	11	-1.00	-------u----------	00011110011010011100011110010100	1	-1.00	-11.00
31:	u----------	10000001011000101101001010110000	8	-1.00	-------1----------	----------	2	-1.00	-9.00
32:	1----------	00110101101000011011001110001001	3	0.00	-------1----------	11100000100011110111100001000010	15	0.00	-7.00
33:	----------	01010010110001111111110110010010	10	0.00	----------	----------	5	-1.00	-7.00
34:	----------	x----------	14	0.00	u----------	00011110011010011100011110010100	1	-2.00	-6.00
35:	----------	00110100101001010110110101000111	4	0.00	0----------	00110101101000011011001110001001	3	-1.00	-4.00
36:	----------	----------	9	0.00	1----------	----------	7	0.00	-3.00
37:	----------	11100000100011110111100001000010	15	0.00	----------	x----------	14	0.00	-3.00
38:	----------	10000001011000101101001010110000	8	0.00	----------	10110101011001110111100100001100	6	0.00	-3.00
39:	----------	00011110011010011100011110010100	1	0.00	----------	----------	9	0.00	-3.00
40:	----------	----------	2	0.00	----------	00010110101110011100111001010111	11	0.00	-3.00
41:	----------	----------	7	0.00	----------	10000001011000101101001010110000	8	0.00	-3.00
42:	----------	----------	0	0.00	----------	10010001010011011100001000100011	12	0.00	-3.00
43:	----------	10110101011001110111100100001100	6	0.00	----------	----------	2	0.00	-3.00
44:	----------	00111011101011111100100111011110	13	0.00	----------	01010010110001111111110110100101010	10	0.00	-3.00
45:	----------	00010110101110011100111001010111	11	0.00	----------	----------	0	0.00	-3.00
46:	----------	----------	5	0.00	----------	00110100101001010110110101000111	4	0.00	-3.00
47:	----------	10010001010011011100001000100011	12	0.00	----------	00111011101011111100100111011110	13	0.00	-3.00
48:	----------	00011110011010011100011110010100	1	0.00	----------	10000001011000101101001010110000	8	0.00	-3.00
49:	----------	----------	9	0.00	----------	10110101011001110111100100001100	6	0.00	-3.00
50:	----------	00010110101110011100111001010111	11	0.00	----------	00110100101001010110110101000111	4	0.00	-3.00
51:	----------	01010010110001111111110110010010	10	0.00	----------	00011110011010011100011110010100	1	0.00	-3.00
52:	----------	----------	0	0.00	----------	00110101101000011011001110001001	3	0.00	-3.00
53:	----------	10000001011000101101001010110000	8	0.00	----------	00010110101110011100111001010111	11	0.00	-3.00
54:	----------	10010001010011011100001000100011	12	0.00	----------	11100000100011110111100001000010	15	0.00	-3.00
55:	----------	00110100101001010110110101000111	4	0.00	----------	----------	0	0.00	-3.00
56:	----------	00111011101011111100100111011110	13	0.00	----------	----------	5	0.00	-3.00
57:	----------	00110101101000011011001110001001	3	0.00	----------	10010001010011011100001000100011	12	0.00	-3.00
58:	----------	----------	7	-1.00	----------	----------	2	0.00	-3.00
59:	----------0--------	11100000100011110111100001000010	15	-1.00	----------	00111011101011111100100111011110	13	0.00	-2.00
60:	----------1--------	x----------	14	0.00	----------	----------	9	0.00	-1.00
61:	----------x--------	----------	5	0.00	----------	----------	7	0.00	-1.00
62:	----------	10110101011001110111100100001100	6	0.00	----------	01010010110001111111110110100101010	10	0.00	-1.00
63:	----------	----------	2	-1.00	----------	x----------	14	0.00	-1.00
64:	----------				----------x--------				

Fig. 8. The differential path for the full `RIPEMD-128` hash function distinguisher. The notations are the same as in [4] and are described in Appendix B. The column $\mathrm{P}^l[i]$ (resp. $\mathrm{P}^r[i]$) represents the $\log_2()$ differential probability of step i in left (resp. right) branch. The column $P[i]$ represents the cumulated probability (in $\log_2()$) until step i for both branches, i.e. $\mathrm{P}[i] = \prod_{j=63}^{j=i}(\mathrm{P}^r[j] \cdot \mathrm{P}^l[j])$.