Second Preimage Analysis of Whirlwind

Riham AlTawy and Amr M. Youssef(✉)

Concordia Institute for Information Systems Engineering, Concordia University, Montréal, Québec, Canada
youssef@ciise.concordia.ca

Abstract. Whirlwind is a keyless AES-like hash function that adopts the Sponge model. According to its designers, the function is designed to resist most of the recent cryptanalytic attacks. In this paper, we evaluate the second preimage resistance of the Whirlwind hash function. More precisely, we apply a meet in the middle preimage attack on the compression function which allows us to obtain a 5-round pseudo preimage for a given compression function output with time complexity of 2^{385} and memory complexity of 2^{128}. We also employ a guess and determine approach to extend the attack to 6 rounds with time and memory complexities of 2^{496} and 2^{112}, respectively. Finally, by adopting another meet in the middle attack, we are able to generate n-block message second preimages of the 5 and 6-round reduced hash function with time complexity of 2^{449} and 2^{505} and memory complexity of 2^{128} and 2^{112}, respectively.

Keywords: Cryptanalysis · Hash functions · Meet in the middle · Second preimage attack · Whirlwind

1 Introduction

Building a cryptographic primitive based on an existing component model has a very important advantage other than the possibility of sharing optimized components in a resource constrained environment. Namely, the advantage of adopting a model that has took its fair share of cryptanalysis and is still going strong. Consequently, the new primitive is expected to inherit most of the good qualities and the underlying features. The Advanced Encryption Standard (AES) wide trail strategy [8] has proven solid resistance to standard differential and linear attacks over more than a decade. This fact has made AES-like primitives an attractive alternative to dedicated constructions. Besides the ISO standard Whirlpool [21], we have seen a strong inclination towards proposing AES-like hash functions during the SHA-3 competition [20] (e.g., the SHA-3 finalists Grøstl [9] and JH [28], and LANE [12]). Additionally, Stribog [16] the new Russian hash standard, officially known as GOST R 34.11-2012 [1], is also among the recently proposed AES-like hash functions. This shift in the hash functions design concepts is due to the fact that Wang *et al.* attacks [26,27] are most effective on Add-Rotate-Xor (ARX) based hash functions where one can find differential patterns that propagate with acceptable probabilities. Moreover, these attacks take advantage

D. Lin et al. (Eds.): Inscrypt 2014, LNCS 8957, pp. 311–328, 2015.
DOI: 10.1007/978-3-319-16745-9_17

of the weak message schedules of most of ARX-designs. Hence, using message modification techniques [27], significant reduction in the attack complexity can be achieved.

Whirlwind is a keyless AES-like hash function that adopts the Sponge model [7]. It is proposed by Baretto *et al.* in 2010 as a response to the recent cryptanalytic attacks that have improved significantly during the SHA-3 competition. Sharing the designers of Whirlpool, Whirlwind design is inspired by Whirlpool and takes into account the recent development in hash function cryptanalysis, particularly the rebound attack [17]. In fact, the designers add more security features as a precaution against possible improvements. The most important features are adopting an extended Sponge model where the compression function operates on $2n$-bit state and outputs an n-bit chaining value, and employing 16×16-bit Sboxes. Using large Sboxes aims to decrease the probability of a given differential trail. Unlike Whirlpool, the Whirlwind compression function has no independent mixing of the chaining value. In the latter, the chaining value is processed independently and mixed with the message state at an XOR transformation. The presence of the key schedule has been exploited as an additional degree of freedom by cryptanalysts. Consequently, it has contributed to many improvements of the inbound phase of the rebound attack [14,18]. These improvements have enabled the attack to cover more rounds. Accordingly, for the designers of Whirlwind, eliminating both the key schedule and the interaction between the message and the compression function output via the feedforward, and employing large Sboxes, limit both the effect and scope of the rebound attack to a great extent. However, from our perspective, some of these features made one of our meet in the middle (MitM) pseudo preimage attacks on the compression function easier and with lower complexity than that on Whirlpool [29]. More precisely, with the absence of the key schedule, using large Sboxes and the output truncation has enabled us to find an execution separation such that the matching probability can be balanced with the available forward and backward starting values as will be discussed in Sects. 4 and 5.

Aoki and Sasaki proposed the meet in the middle preimage attack [5] following the work of Laurent on MD4 [15]. Afterwards, the first MitM preimage attack on the AES block cipher in hashing modes was proposed by Sasaki in FSE 2011 [22]. He applied the attack on Whirlpool and a 5-round pseudo preimage attack on the compression function was presented and used for a second preimage attack on the whole hash function in the same work. In the sequel, Wu *et al.* [29] formalized the approach and employed a time-memory trade off to improve the time complexity of the 5-round attack on the Whirlpool compression function. Moreover, they applied the MitM pseudo preimage attack on Grøstl and adapted the attack to produce pseudo preimages of the reduced hash function. Afterwards, a pseudo preimage attack on the 6-round Whirlpool compression function and a memoryless preimage attack on the reduced hash function were proposed in [24]. Finally, AlTawy and Youssef, combined MitM pseudo preimages of the compression function of Stribog with a multicollision attack to generate preimages of the reduced hash function [2].

In this work, we investigate the security of Whirlwind and its compression function, assessing their resistance to the MitM preimage attacks. Employing the partial matching and initial structure concepts [22], we present a pseudo preimage attack on the compression function reduced to 5 out of 12 rounds. More precisely, we present an execution separation for the compression function that balances the forward and backward starting values with the corresponding matching probability [29]. Furthermore, we employ a guess and determine approach [24] to guess parts of the state. This approach helps in maintaining partial state knowledge for one more round. Consequently, we are able to extend the attack by one more round. In spite of the compression function truncated output, the proposed 6-round execution separation maximizes the overall probability of the attack by balancing the chosen number of starting values and the guess size. Finally, we show how to generate n-block messages second preimages of the Whirlwind hash function using the presented pseudo preimage attacks on the compression function.

The rest of the paper is organized as follows. In the next section, the description of the Whirlwind hash function along with the notation used throughout the paper are provided. A brief overview of the MitM preimage attack and the used approaches are given in Sect. 3. Afterwards, in Sects. 4 and 5, we provide detailed description of the attacks and their corresponding complexity. In Sect. 6, we show how second preimages of the hash function are generated using the attacks presented in Sects. 4 and 5. Finally, the paper is concluded in Sect. 7.

2 Whirlwind Description

Whirlwind [6] is a keyless AES-like hash function that adopts a Sponge-like model. The function employs a 12-round compression function which operates on 1024-bit state. The internal state is represented by an 8×8 matrix S of 16-bit (word) elements where each element is indexed by its position in row i and column j.

$$\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} & S_{0,6} & S_{0,7} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} & S_{1,6} & S_{1,7} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} & S_{2,6} & S_{2,7} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} & S_{3,6} & S_{3,7} \\ S_{4,0} & S_{4,1} & S_{4,2} & S_{4,3} & S_{4,4} & S_{4,5} & S_{4,6} & S_{4,7} \\ S_{5,0} & S_{5,1} & S_{5,2} & S_{5,3} & S_{5,4} & S_{5,5} & S_{5,6} & S_{5,7} \\ S_{6,0} & S_{6,1} & S_{6,2} & S_{6,3} & S_{6,4} & S_{6,5} & S_{6,6} & S_{6,7} \\ S_{7,0} & S_{7,1} & S_{7,2} & S_{7,3} & S_{7,4} & S_{7,5} & S_{7,6} & S_{7,7} \end{bmatrix}$$

An element $S_{i,j}$ can be seen as 4×1 matrix of 4-bit nibbles.

$$S_{i,j} = \begin{bmatrix} S_{i,j,0,0} \\ S_{i,j,0,1} \\ S_{i,j,1,0} \\ S_{i,j,1,1} \end{bmatrix}.$$

Accordingly, each row of the state matrix S is in fact 4×8 4-bit nibble matrix. The reason for the switch between the 16-bit elements and the 4-bit nibbles is due to the fact that the adopted round transformations operate on different fields ($GF(2^{16})$ and $GF(2^4)$). More precisely, the round function updates the state by applying the following four transformations:

- γ: A nonlinear bijective mapping over $GF(2^{16})$. This substitution layer works on the 16-bit elements where it replaces each 16-bit element by its multiplicative inverse over $GF(2^{16})$ and zero is replaced by itself.
- θ: A linear transformation that mixes rows. It works by applying the linear transformations λ_0 and λ_1 on the 4-bit nibble elements. Hence, if each state row is 4×8 4-bit nibble matrix, the updated row is:

$$\theta(S_i) = \begin{cases} \lambda_0(S_{i,*,0,0}) = S_{i,*,0,0} \cdot M_0 \\ \lambda_1(S_{i,*,0,1}) = S_{i,*,0,1} \cdot M_1 \\ \lambda_1(S_{i,*,1,0}) = S_{i,*,1,0} \cdot M_1 \\ \lambda_0(S_{i,*,1,1}) = S_{i,*,1,1} \cdot M_0, \end{cases}$$

  where $*$ denotes the column index, $M_0 = \text{dyadic}(0x5, 0x4, 0xA, 0x6, 0x2, 0xD, 0x8, 0x3)$, $M_1 = \text{dyadic}(0x5, 0xE, 0x4, 0x7, 0x1, 0x3, oxF, 0x8)$, and dyadic$(m)$ denotes the MDS dyadic matrix M corresponding to the sequence m over $GF(2^4)$, i.e., $M_{i,j} = m_{i \oplus j}$. Nevertheless, θ inherits the optimal diffusion properties of its underlying transformations. However, these transformations cannot be directly applied on elements of $GF(2^{16})$ through simple matrix multiplication as this requires the use of a linearized polynomial.
- τ: A transposition layer where the 16-bit 8×8 matrix is transposed.
- σ^r: A linear transformation where that 16-bit state is XORed with a round dependant constant state C^r.

As depicted in Fig. 1, the compression function $\phi(h, m)$ operates on 512-bit message block m and 512-bit chaining value h, both represented by 8×4 matrices of 16-bit elements. The internal state S is initialized such that its first four columns are set to h and the last four columns are set to m. The state is then updated by applying the four transformations for twelve rounds. Finally, the last four columns of the last state are truncated and the input chaining value h is XORed with the first four columns of the last state to generate the compression function output.

Whirlwind employs a finalization step. More precisely, after processing all the message blocks, an extra compression function call with a null message block is adopted. If the desired output size is $\log_2(N)$ bits, the output of the finalization step is then reduced modulo N. Whirlwind also uses an adaptable initialization value where the IV used to process the first message block depends on the desired reduction value N. To compute the initial value h_0, the reduction value is converted to an 8×4 matrix, thus $h_0 = \phi(0, N)$. To compute the hash of a given message M, it is first padded by 1 followed by zeros to obtain a bit string whose length is an odd multiple of 256, and finally with the 256-bit right justified

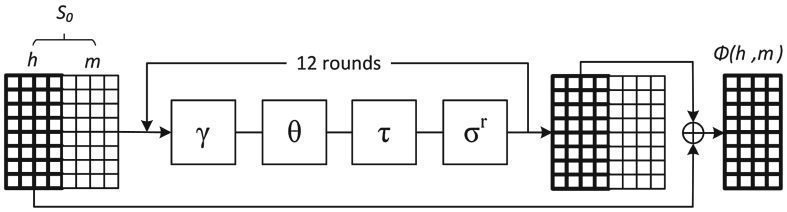

Fig. 1. The compression function ϕ.

binary representation of $|M|$. The padded message is then divided into t 512-bit blocks: $m_0, m_1, ...m_{t-1}$. Finally, the message blocks are processed as follows:

$$\begin{aligned} h_0 &= \phi(0, N), \\ h_i &= \phi(h_{i-1}, m_{i-1}), \text{ for } i = 1, 2, ..., t, \\ h_{t+1} &= \phi(h_t, 0). \end{aligned}$$

The output $H(M)$ is equal to h_{t+1} mod N. For further details, the reader is referred to [6,25].

2.1 Notation

Let S be (8×8) 16-bit state denoting the internal state of the function. The following notation is used in our attacks:

- S_i: The message state at the beginning of round i.
- S_i^U: The message state after the U transformation at round i, where $U \in \{\gamma, \theta, \tau, \sigma^r\}$.
- $S_i[r, c]$: A word at row r and column c of state S_i.
- S_i[row r]: Eight words located at row r of state S_i.
- S_i[col c]: Eight words located at column c of state S_i.

3 MitM Preimage Attacks

Given a compression function CF that processes a chaining value h and a message block m, a preimage attack on CF is defined as follows: given h and x, where x is the compression function output, find m such that $CF(h, m) = x$. However, in a pseudo preimage attack, only x is given and one must find h and m such that $CF(h, m) = x$. The effect of a pseudo preimage attack on the compression function by itself is not important. However, these attacks can be used to build a preimage or second preimage attacks on the whole hash function [19]. As demonstrated in Sect. 6, pseudo preimages of the Whirlwind compression function can be utilized to compose an n-block second preimages of the hash function.

The main concept of the proposed MitM attacks is to divide the attacked execution rounds at the starting point into two independent executions that proceed in opposite directions (forward and backward chunks). The two executions

must remain independent until the point where matching takes place. To maintain the independence constraint, each execution must depend on a different set of inputs, e.g., if only the forward chunk is influenced by a change in a given input, then this input is known as a forward neutral input. Consequently, all of its possible values can be used to produce different outputs of the forward execution at the matching point. Accordingly, all neutral inputs for each execution direction attribute for the number of independent starting values for each execution. Hence, the output of the forward and the backward executions can be independently calculated and stored at the matching point. Similar to all MitM attacks, the matching point is where the outputs of the two separated chunks meet to find a solution from both the forward and backward directions that satisfies both executions. While for block ciphers, having a matching point is achieved by employing both the encryption and decryption oracles, for hash function, this is accomplished by adopting the cut and splice technique [5] which utilizes the employed mode of operation. In other words, given the compression function output, this technique chains the input and output states through the feedforward as we can consider the first and last states as consecutive rounds. Subsequently, the overall attacked rounds behave in a cyclic manner and one can find a common matching point between the forward and backward executions and consequently can also select any starting point.

The MitM preimage attack has been applied to MD4 [5,10], MD5 [5], HAS-160 [11], and all functions of the SHA family [3,4,10]. The attack exploits the fact that all the previously mentioned functions are ARX-based and operate in the Davis-Mayer (DM) mode, where the state is initialized by the chaining value and some of the expanded message blocks are used independently in each round. Thus, one can determine which message blocks affect each execution for the MitM attack. However, several AES-like hash functions operate in the Miyaguchi-Preneel mode, where the input message is fed to the initial state which undergoes a chain of successive transformations. Consequently, the process of separating independent executions becomes relatively more complicated. Cryptanalysts are forced to adopt a pseudo preimage attack when the compression function operates in Davis-Mayer mode. This is due to the fact that the main execution takes place on a state initialized by the chaining value. Subsequently, using the cut and splice technique enforces changes in the first state through the feedforward. Additionally, even if function operates in the Miyaguchi-Preneel mode, attempting a MitM preimage attack usually generates pseudo preimages when the complexity of finding a preimage is higher than the available degrees of freedom in the message. Consequently, the chaining value is utilized as a source of randomization to satisfy the number of multiple restarts required by the attack. As a result, we end up with a pseudo preimage rather than a preimage of the compression function output.

This class of attacks has witnessed significant improvements since its inception. Most of these attacks aim to make the starting and matching points span over more than one round transformation and hence increase the number of the overall attacked rounds. More precisely, the initial structure approach [22,23]

provides the means for the starting point to cover a few successive transformations where words in the states belong to both the forward and backward chunks. Although neutral words of both chunks are shared within the initial structure, independence of both executions is achieved in the rounds at the edges of the initial structure. Additionally, the partial matching technique [5] allows only parts of the state to be matched at the matching point. This method is used to extend the matching point further and makes use of the fact that round transformations may update only parts of the state. Thus the remaining unchanged parts can be used for matching. This approach is highly successful in ARX-based hash functions which are characterized by the slow diffusion of their round update functions and so some state variables remain independent in one direction while execution is in the opposite direction. The unaffected parts of the states at each chunk are used for partial matching at the matching point. However, in AES-like hash functions, full diffusion is achieved after two rounds and this approach can be used to extend the matching point of two states for a limited number of transformations. Once a partial match is found, the inputs of both chunks that resulted in the matched values are selected and used to evaluate the remaining undetermined parts of the state at the matching point to check for a full state match. Figure 2 illustrates the MitM preimage attack approaches for the Whirlwind compression function. The red and blue arrows denote the forward and backward executions on the message state, respectively. S_0 is the first state initialized by h and m and S_t is the last attacked state.

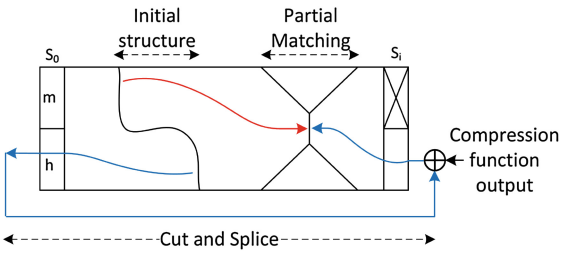

Fig. 2. MitM preimage attack techniques customized for Whirlwind operation (Color figure online).

In the next section, we apply the techniques discussed in this section to generate a 5-round pseudo preimage of the Whirlwind compression function.

4 A Pseudo Preimage of the 5-Round Compression Function

To proceed with the attack, we first need to separate the two execution chunks around the initial structure. More precisely, we divide the five attacked rounds of execution into a 2-round forward chunk and a 2-round backward chunk around a starting point (initial structure). The proposed chunk separation is shown

in Fig. 3. Our choices of forward and backward starting values in the initial structure determine the complexity of the attack. Specifically, we try to balance the number starting values in each direction and the number of known words at the matching point at the end of each chunk. The total number of starting values in both directions should produce candidate pairs at the matching point to satisfy the matching probability. For further clarification, we first explain how the initial structure is constructed. The main idea is to have maximum state knowledge at the start of each execution chunk. This can be achieved by choosing several words as neutral so that the number of corresponding output words of the θ and θ^{-1} transformations at the start of both chunk that are constant or relatively constant is maximized. A relatively constant word is a word at the state directly after the initial structure whose value depends on the value of the neutral words in one execution direction but remains constant from the opposite execution perspective. The initial structure for the 5-round MitM preimage attack on the compression function of Whirlwind is shown in Fig. 4.

Following Fig. 4, our aim is to have five constants in the three lowermost rows in state d and determine the available values of the corresponding blue rows that make them hold. The values of the three lowermost blue rows are the available

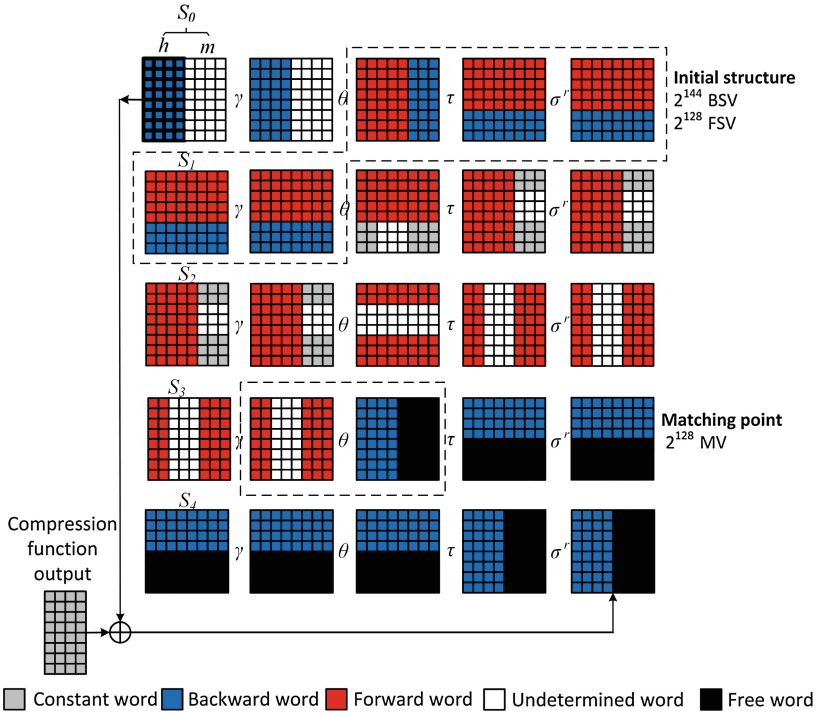

Fig. 3. Chunk separation for a 5-round MitM pseudo preimage attack the compression function. BSV: Backward starting value, FSV: Forward starting value, MV: Matching value (Color figure online).

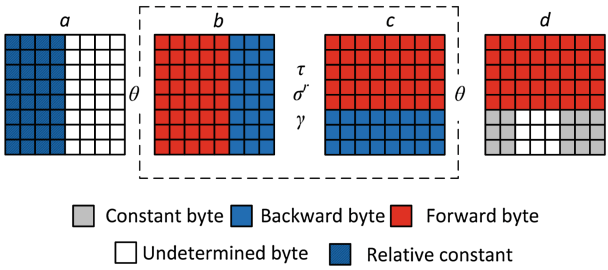

Fig. 4. Initial structure used in the attack on the 5-round compression function (Color figure online).

backward starting values. For each row, we randomly choose the five constant words in d[row 7] and then determine the values of blue words in c[row 7] so that after applying θ on c[row 7], the chosen values of the five constants hold. Since the linear mapping is applied on the 4-bit nibbles, we need twenty constant nibbles in d[row 7]. This can be achieved by maintaining twenty variable nibbles in c[row 7] to solve a system of twenty equations when the other twelve nibbles are fixed. Accordingly, for any of the last three rows in state c, we can randomly choose any three blue words and compute the remaining five so that the output of θ maintains the previously chosen five constant words at d[row 7]. To this end, we have nine free (blue) words, three for each row in state c. Thus the number of backward starting values is 2^{144} which means that we can start the backward execution by 2^{144} different starting values and hence 2^{144} different output values at the matching point S_3^{θ}. Similarly, we choose 32 constant words in state a and for each row in state b we randomly choose four red nibbles and compute the other sixteen red nibbles such that after the θ^{-1} transformation we get the predetermined constants at each row in a. However, the value of the four shaded blue words in each row of state a depends also on the three blue words in the rows of state b. We call these bytes relative constants because their final values cannot be determined until the backward execution starts and these values are different for each backward execution iteration. Specifically, their final values are the predetermined constants acting as an offset XORed with the corresponding blue nibbles multiplied by M_0^{-1} or M_1^{-1} coefficients. In the sequel, we have eight free words (one for each row in b) which means 2^{128} forward starting value and hence 2^{128} different input values to the matching point S_3^{γ}.

As depicted in Fig. 3, the forward chunk starts at S_1^{θ} and ends at S_3^{γ} which is the input state to the matching point. The backward chunk starts at S_0^{γ} and ends after the feedforward at S_3^{θ} which is the output state of the matching point. The red words are the neutral ones for the forward chunk and after choosing them in the initial structure, all the other red words can be independently calculated. White words in the forward chunk are the ones whose values depend on the neutral words of the backward chunk which are the blue words in the initial structure. Accordingly, their values are undetermined, i.e., these words cannot be

evaluated until a partial match is found. Same rationale applies to the backward chunk and the blue words. Grey bytes are constants which can be either the compression function output or the chosen constants in the initial structure.

To find the pseudo preimage of the given compression function output, we have to find a solution that satisfies both executions. This takes place at the matching point where we match the partial state output from the forward execution at S_3^{γ} with the full state (due to truncation) output from the backward execution at S_3^{θ} through the θ transformation. As depicted in Fig. 3, at the matching point, in each row we have knowledge of five words from the forward execution and four words from the backward execution. Since the linear mapping is performed on 4-bit nibbles, we can form sixteen 4-bit linear equations using twelve 4-bit unknowns and match the resulting forward and backward values through the remaining four 4-bit equations. More precisely, we use the following equation to compute the first 4-bit nibble row in the first state row $b_{0,j,0,0}$ through the linear transformation λ_0 given the 4-bit nibble input row $a_{0,j,0,0}$. For ease of notation, we denote the first 4-bit nibble in a word located in the first row and column j as a_j (i.e., $a_{0,j,0,0} = a_j$). We use a similar notation for b.

$$\begin{bmatrix} a_0 & a_1 & \overline{a_2} & \overline{a_3} & \overline{a_4} & a_5 & a_6 & a_7 \end{bmatrix} \begin{bmatrix} 0x5 & 0x4 & 0xA & 0x6 \\ 0x4 & 0x5 & 0x6 & 0xA \\ 0xA & 0x6 & 0x5 & 0x4 \\ 0x6 & 0xA & 0x4 & 0x5 \\ 0x2 & 0xD & 0x8 & 0x3 \\ 0xD & 0x2 & 0x3 & 0x8 \\ 0x8 & 0x3 & 0x2 & 0xD \\ 0x3 & 0x8 & 0xD & 0x2 \end{bmatrix} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}$$

Because we are evaluating the first 4-bit nibble in the output words and the four rightmost column of the output state are truncated, we only use half of the dyadic matrix M_0. In the above equation, we use the overline to denote the unknown first 4-bit nibbles at the first row words. More precisely, there are three unknown nibbles a_2, a_3, and a_4 in the input and all the nibbles in the output are known. Accordingly, given the λ_0 transformation linear matrix M_0, we can form four linear equations to compute b_0, b_1, b_2, and b_3. Then we evaluate the values of the three unknown nibbles a_2, a_3, and a_4 from three out of the four equations and substitute their values in the remaining one. With probability 2^{-4} the right hand side of the remaining equation is equal to the corresponding known backward nibble. Hence, the matching size per 4-bit nibble row is 2^4 and since we have four 4-bit nibble rows per word row, the matching size is 2^{16} for state row, Thus, the matching probability for the whole state is 2^{-128}. The choice of the number of forward and backward starting values directly affects the matching probability as their number determines the number of red and blue words at a given state row at the matching point. If the number of blue and red words are not properly chosen at the initial structure, we can reach no matching value. More precisely, we cannot have a matching value if the total number of red and blue words in a given row at the matching point is less than or equal to eight. In what follows we summarize the attack steps:

1. We randomly choose the constants in states S_1^θ and S_0^γ at forward and backward output of the initial structure.
2. For each forward starting value fw_i in the 2^{128} forward starting values at S_0^γ, we evaluate the forward matching value fm_i at S_3^γ and store (fw_i, fm_i) in a lookup table T.
3. For each backward starting value bw_j in the 2^{144} backward starting values in S_1^γ, we evaluate the backward matching value bm_j at S_3^θ and check if there exists an $fm_i = bm_j$ in T. If found, then these solutions partially match through the linear transformations and the full match should be checked using the matched starting points fw_i and bw_i. If a full match exists, then output the chaining value and the message M_i, else go to step 1.

To minimize the attack complexity, the number of the starting values of both execution and the matching value must be kept as close as possible to each other. In the chunk separation shown in Fig. 3, the number of forward and backward starting values, and the matching values are 2^{128}, 2^{144}, and 2^{128}, respectively. To further explain the complexity of the attack, we consider the attack procedure. After step 2, we have 2^{128} forward matching values at S_3^γ and we need 2^{128} memory to store them. At the end of step 3, we have 2^{144} backward matching values at S_3^θ. Accordingly, we get $2^{128+144} = 2^{272}$ candidate pairs for partial matching. Since the probability of a partial match is 2^{-128}, we expect 2^{144} pairs to partially match. The probability that a partially matching pair results in a full match is the probability that the matching forward and backward starting values generates the three unknown columns in S_3^γ equal to the ones that resulted from the partial match. This probability is equal to $2^{24\times16} = 2^{-384}$. As we have 2^{144} partially matching pairs, we expect $2^{144-384} = 2^{-240}$ pairs to fully match. Thus we need to repeat the attack 2^{240} times to get one fully matching pair. The time complexity for one repetition of the attack is 2^{128} for the forward computation, 2^{144} for the backward computation, and 2^{144} to test if the partially matching pairs fully match. Consequently, the overall complexity of the attack is $2^{240}(2^{128} + 2^{144} + 2^{144}) \approx 2^{385}$ time and 2^{128} memory.

5 Extending the Attack by One More Round

The wide trail strategy adopted by Whirlwind implies that one unknown word leads to a full unknown state after two rounds. Consequently, in the previous 5-round attack, the matching point is chosen exactly two rounds away from the initial structure in each direction. Attempting to go one more round in either directions always fails because at the end of each chunk execution the state has undetermined bytes at each row. Consequently, applying the linear transformation θ to such state results in a full loss of state knowledge and matching cannot be achieved. To maintain partial state knowledge, we adopt a guess and determine approach [24]. Hence, we can probabilistically guess some of the undetermined row words in the state before the linear transformation in either direction. Thus, we maintain knowledge of some state rows after the linear transformation θ which are used for matching. Due to truncation and the large size of Sboxes, we

have to carefully choose both starting values in the initial structure to minimize the number of guessed words as much as possible and to result in an acceptable number of correctly guessed matching pairs. The proposed chunk separation for the 6-round MitM pseudo preimage attack is shown in Fig. 5. In order to be able extend the attack by one extra round in the forward direction, we guess the six unknown words (yellow words) in state S_4^γ. As a result, we can reach state S_5^γ with three determined columns where the matching takes place.

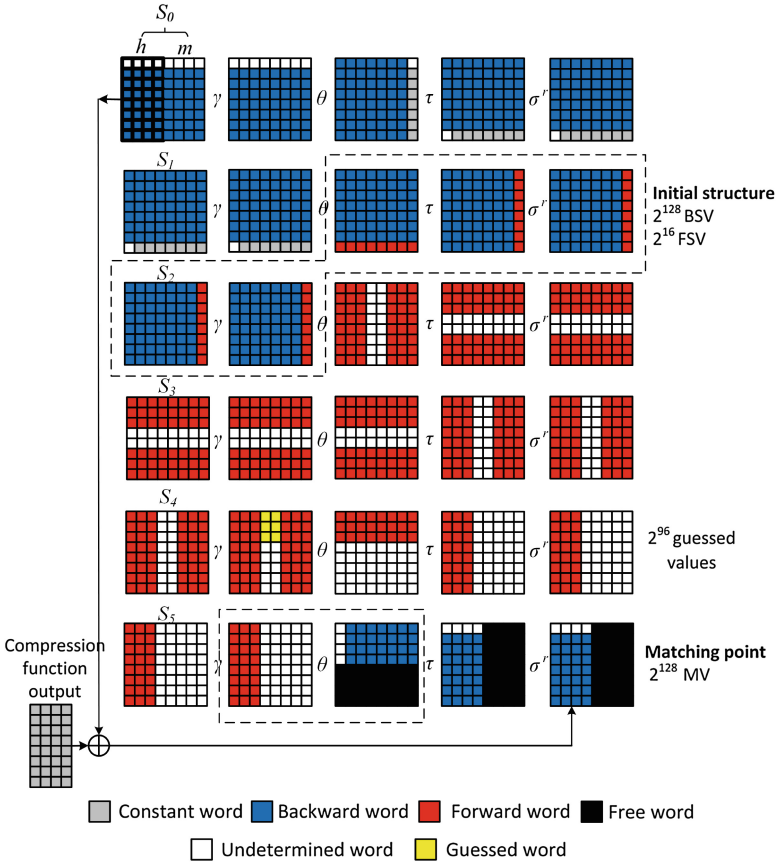

Fig. 5. Chunk separation for a 6-round MitM pseudo preimage attack on the compression function. BSV: Backward starting value, FSV: Forward starting value, MV: Matching value (Color figure online).

The chosen separation and guessed values maximize the attack probability by carefully selecting the forward, backward, and guessed bit values. We aim to increase the number of starting forward values and keep the number of backward and matching values as close as possible and larger than the number of guessed

values. For our attack, the chosen number for the forward and backward starting values, and the guessed values are 2^{16}, 2^{128}, and 2^{96}, respectively. Setting these parameters fixes the number of matching values to 2^{128}. In what follows, we give an overview of the attack procedure and complexity based on the above chosen parameters:

1. We first start by randomly choosing the constants in S_1^γ and S_2^θ at the edges of the initial structure.
2. For each forward starting value fw_i and guessed value g_i in the 2^{16} forward starting values and the 2^{96} guessed values, we compute the forward matching value fm_i at S_5^γ and store (fw_i, g_i, fm_i) in a lookup table T.
3. For each backward starting value bw_j in the 2^{128} backward starting values, we compute the backward matching value bm_j at S_5^θ and check if there exists an $fm_i = bm_j$ in T. If found, then a partial match exists and the full match should be checked using the matched forward, guessed, and backwards starting values fw_i, g_i, and bw_i. If a full match exists, then we output the chaining value h_i and the message m_i, else go to step 1.

The complexity of the attack is evaluated as follows: after step 2, we have $2^{16+96} = 2^{112}$ forward matching values which need 2^{112} memory for the look up table. At the end of step 3, we have 2^{128} backward matching values. Accordingly, we get $2^{112+128} = 2^{240}$ partial matching candidate pairs. Since the probability of a partial match is 2^{-128} and the probability of a correct guess is 2^{-96}, we expect $2^{240-128-96} = 2^{16}$ correctly guessed partially matching pairs. Due to truncation, we are interested only in the uppermost four rows at the matching point. More precisely, we want the partially matching starting value to result in the correct values on the twenty four unknown words in both S_4^γ and S_4^θ that make the blue and red words hold. The probability that the latter condition takes place is $2^{24\times-16} = 2^{-384}$. Consequently, the expected number of fully matching pairs is 2^{-368} and hence we need to repeat the attack 2^{368} times to get a full match. The time complexity for one repetition is 2^{112} for the forward computation, 2^{128} for the backward computation, and 2^{16} to check that partially matching pairs fully match. The overall complexity of the attack is $2^{368}(2^{112}+2^{128}+2^{16}) \approx 2^{496}$ time and 2^{112} memory.

6 Second Preimage of the Hash Function

In this section, we show how the previously presented pseudo preimage attacks on the Whirlwind compression function can be utilized to generate second preimages for the whole hash function. The last two compression function calls in the Whirlwind hash function differ than the previous ones, hence they are considered a final step in the execution of the function. In this step, the first compression function call operates on the padded message, and the state of the second compression function call is initialized by the chaining value and an 8×4 all zero message. Accordingly, attempting to use our pseudo preimage attacks to invert

the final compression function call does not result in the expected all zero message and if extended can rarely satisfy the correct padding. Consequently, using these attacks to generate preimages does not work. However, if we can get the correct chaining values for the last two compression function calls such that when both the correct padding and null message are used we get the target compression function output, then we can use our pseudo preimage attack to get the right messages. This requirements can be fulfilled if we consider a second preimage attack. When one attempts a second preimage attack, one is given a hash function H that operates with an initial value IV and a message block m. Then, one must find m' such that $H_{IV}(m) = H_{IV}(m')$. When we consider a second preimage attack, using the give message m, we can know exactly the input chaining values for the last two compression function calls such that we get the desired hash function output. We only need to find another equal length message m' that is, given the IV, generates the chaining value required by the padding compression function call. Our attack is an n-block second preimage attack ($n \geq 2$) where given an n-block message m, we generate another n-block message m' such that both messages hash to the same value. More precisely, to build m', we copy the finalization step of m and use our pseudo preimage attacks along with another meet in the middle attack to search for m'. For illustration, we are using 2-block messages to describe our attack. As depicted in Fig. 6, the attack is divided into three stages:

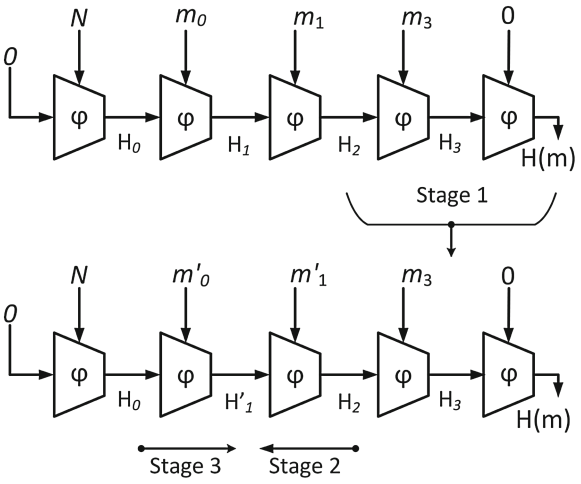

Fig. 6. Second preimage attack on the hash function.

1. Given a 2-block message $m = m_0 \| m_1$ and the truncation value N, we compute the adaptable initialization vector $H_0 = \phi(0, N)$, compose the padding message $m_2 = 1 \| 0^{500} \| 1 \| 0^{10}$, and hash m and get the desired $H(m)$. This process is shown

in the upper hash function execution in Fig. 6. To begin building m', we copy the last compression function calls with there chaining values. Specifically, we consider H_2 to be the output of the compression function call operating on the second block m'_1 of the massage we are searching for.

2. In this stage, given H_2, we produce 2^p pseudo preimages for the second message block compression function call. The output of this step is 2^p pairs of a candidate chaining value H'_1 and a candidate second message block m'_1. We store these resulting candidate pairs (H'_1, m'_1) in a table T.
3. To this end, we try to search for the first message block m'_0 such that using the initial vector H_0, $\phi(H_0, m'_0)$ produces one of the chaining values H'_1 in the table T. In the sequel, we randomly choose m'_0, compute H'_1 and check if it exists in T. As T contains 2^p entries, it is expected to find a match after 2^{512-p} evaluations of the following compression function call with random m'_0 each time:

$$H'_1 = \phi(H_0, m'_0)$$

Once a matching H'_1 value is found in T, the chosen m'_0 is the first message block and the corresponding m'_1 is the second message block such that $m' = m'_0 \| m'_1$ and $H(m) = H(m')$.

The time complexity of the attack is evaluated as follows: we need $2^p \times$ (complexity of pseudo preimage attack) in stage 2, and 2^{512-p} evaluations of one compression function call for the MitM attack at stage 3. The memory complexity for the attack is as follows: 2^p states to store the pseudo preimages for the MitM in stage 2, in addition to the memory complexity of the pseudo preimage attack on the compression function which is 2^{128} or 2^{112} for the 5-round or 6-round compression function. Since the time complexity is highly influenced by p, we have chosen $p = 64$ for the 5-round attack and $p = 8$ for the 6-round attack to obtain the maximum gain. Accordingly, 2-block second preimages for 5-round Whirlwind hash function are produced with a time complexity of $2^{64+385} + 2^{512-64} \approx 2^{449}$ and memory complexity of $2^{128} + 2^{64} \approx 2^{128}$. The time complexity for the 6-round attack is $2^{8+496} + 2^{512-8} \approx 2^{505}$ and the memory complexity is $2^{112} + 2^8 \approx 2^{112}$.

7 Conclusion

In this paper, we have analyzed Whirlwind and its compression function with respect to preimage attacks. We have shown that with a carefully balanced chunk separation, pseudo preimages for the 5-round reduced compression function are generated. Additionally, we have adopted a guess and determine technique and we were able to extend the 5-round attack by one more round. Finally, using another MitM attack, we utilized the compression function pseudo preimage attacks to produce 5 and 6-round hash function n-block second preimages.

Whirlwind is proposed to improve the Whirlpool design. While, the new improvements limit the extent of rebound attacks significantly, they do not consider MitM preimage attacks. It should be noted that the elimination of the

compression function key schedule and using large Sboxes in the same time made our attacks possible. Indeed, with the large state, the chosen constants in the initial structure are enough to satisfy the number of restarts required by the attack complexity. On the other hand, for Whirlpool, the available freedom in the internal state only cannot by itself fulfill the attack complexity. Also, while the adopted truncation and feedforward prohibit interaction between the input message block and the output state thus limiting the ability of difference cancellation, it enhanced the full matching probability, particularly, if we can have full state knowledge at one side of the matching point like our 5-round attack. It is interesting to note that if the adopted model follows the exact Sponge construction where the message is XORed to the internal state and truncation is performed in the finalization step, thus the compression function always maintains a state larger than the hash function output size, our compression function attacks would not work. It should also be noted that the switch between $GF(2^{16})$ and $GF(2^4)$ in different round transformations does not only alleviate potential concerns regarding algebraic attacks but also enhances the resistance of the function to integral attacks [13]. More precisely, the integral properties that are preserved by the substitution layer are shared independently among nibbles by the following linear transformation for the span of one round only. Finally, we know that the presented results do not directly impact the practical security of the Whirlwind hash function. However, they are first steps in the public cryptanalysis of its proposed design concepts with respect to second preimage resistance.

Acknowledgment. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve the quality of the paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Le Fonds de Recherche du Québec - Nature et Technologies (FRQNT).

References

1. The National Hash Standard of the Russian Federation GOST R 34.11-2012. Russian Federal Agency on Technical Regulation and Metrology report (2012). https://www.tc26.ru/en/GOSTR34112012/GOST_R_34_112012_eng.pdf
2. AlTawy, R., Youssef, A.M.: Preimage attacks on reduced-round stribog. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 109–125. Springer, Heidelberg (2014)
3. Aoki, K., Guo, J., Matusiewicz, K., Sasaki, Y., Wang, L.: Preimages for step-reduced SHA-2. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 578–597. Springer, Heidelberg (2009)
4. Aoki, K., Sasaki, Y.: Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 70–89. Springer, Heidelberg (2009)
5. Aoki, K., Sasaki, Y.: Preimage attacks on one-block MD4, 63-step MD5 and more. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 103–119. Springer, Heidelberg (2009)

6. Barreto, P., Nikov, V., Nikova, S., Rijmen, V., Tischhauser, E.: Whirlwind: a new cryptographic hash function. Des. Codes Crypt. **56**(2–3), 141–162 (2010)
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
8. Daemen, J., Rijmen, V.: The Design of Rijndael: AES- The Advanced Encryption Standard. Springer, Berlin (2002)
9. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl a SHA-3 candidate. NIST submission (2008)
10. Guo, J., Ling, S., Rechberger, C., Wang, H.: Advanced meet-in-the-middle preimage attacks: first results on full tiger, and improved results on MD4 and SHA-2. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 56–75. Springer, Heidelberg (2010)
11. Hong, D., Koo, B., Sasaki, Y.: Improved preimage attack for 68-step HAS-160. In: Lee, D., Hong, S. (eds.) ICISC 2009. LNCS, vol. 5984, pp. 332–348. Springer, Heidelberg (2010)
12. Indesteege, S.: The Lane hash function. Submission to NIST (2008). http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf
13. Knudsen, L.R., Wagner, D.: Integral cryptanalysis. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 112–127. Springer, Heidelberg (2002)
14. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound distinguishers: results on the full whirlpool compression function. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 126–143. Springer, Heidelberg (2009)
15. Leurent, G.: MD4 is not one-way. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 412–428. Springer, Heidelberg (2008)
16. Matyukhin, D., Rudskoy, V., Shishkin, V.: A perspective hashing algorithm. In: RusCrypto (2010). (in Russian)
17. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The rebound attack: cryptanalysis of reduced Whirlpool and `Grøstl`. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 260–276. Springer, Heidelberg (2009)
18. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Rebound attacks on the reduced `Grøstl` hash function. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 350–365. Springer, Heidelberg (2010)
19. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (2010)
20. NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register, vol. 72(212) November 2007. http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf
21. Rijmen, V., Barreto, P.S.L.M.: The Whirlpool hashing function. NISSIE submission (2000)
22. Sasaki, Y.: Meet-in-the-middle preimage attacks on AES hashing modes and an application to whirlpool. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 378–396. Springer, Heidelberg (2011)
23. Sasaki, Y., Aoki, K.: Finding preimages in full MD5 faster than exhaustive search. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 134–152. Springer, Heidelberg (2009)
24. Sasaki, Y., Wang, L., Wu, S., Wu, W.: Investigating fundamental security requirements on whirlpool: improved preimage and collision attacks. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 562–579. Springer, Heidelberg (2012)

25. Tischhauser, E.W.: Mathematical aspects of symmetric-key cryptography. Ph.D. thesis, Katholieke Universiteit Leuven, May 2012. http://www.cosic.esat.kuleuven.be/publications/thesis-201.pdf
26. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
27. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
28. Wu, H.: The hash function JH (2011). http://www3.ntu.edu.sg/home/wuhj/research/jh/jh-round3.pdf
29. Wu, S., Feng, D., Wu, W., Guo, J., Dong, L., Zou, J.: (Pseudo) Preimage attack on round-reduced Grøstl hash function and others. In: Canteaut, A. (ed.) FSE 2012. LNCS, pp. 127–145. Springer, Heidelberg (2012)