Practical Near-Collisions on the Compression Function of BMW

Gaëtan Leurent[1] and Søren S. Thomsen[2⋆]

[1] University of Luxembourg
gaetan.leurent@uni.lu
[2] Technical University of Denmark
s.thomsen@mat.dtu.dk

Abstract. Blue Midnight Wish (BMW) is one of the fastest SHA-3 candidates in the second round of the competition. In this paper we study the compression function of BMW and we obtain practical partial collisions in the case of BMW-256: we show a pair of inputs so that 300 pre-specified bits of the outputs collide (out of 512 bits). Our attack requires about 2^{32} evaluations of the compression function. The attack can also be considered as a near-collision attack: we give an input pair with only 122 active bits in the output, while generic algorithm would require 2^{55} operations for the same result. A similar attack can be developed for BMW-512, which will gives message pairs with around 600 colliding bits for a cost of 2^{64}. This analysis does not affect the security of the iterated hash function, but it shows that the compression function is far from ideal.
We also describe some tools for the analysis of systems of additions and rotations, which are used in our attack, and which can be useful for the analysis of other systems.

1 Introduction

Blue Midnight Wish (BMW) is a candidate in the SHA-3 hash function competition [7] which made it to the second round of the competition, but was not selected as a finalist. It is one of the fastest second round candidates in software, and belongs to the ARX family, using only additions, rotations, and xors.

BMW is built by iterating a compression function, similarly to the ubiquitous Merkle-Damgård paradigm [5, 9]. More precisely, BMW uses a chaining value twice as large as the output of the hash function (this is known as wide-pipe, or Chop-MD), and uses a final transformation similar to the HMAC construction. There are several security proofs for this mode of operation and similar modes [2–4], which essentially show that if the compression function behaves like a random function, then the hash function will behave like a random function (up to some level determined by the width of the chaining variable).

In this paper we explain how to find partial-collisions in the BMW-256 compression function. The same technique could be used to find partial-collisions in

⋆ Supported by a grant from the Villum Kann Rasmussen Foundation.

the BMW-512 compression function, but the complexity would be too high to carry out the attack in a reasonable amount of time, and so we have not implemented this attack. The attacks are not affected by the value of the security parameter of BMW.

1.1 Compression function attacks

A natural step in the analysis of iterated hash functions is to study the compression function. Most attacks on the compression function do not lead to attacks on the iterated hash function, but they can invalidate the assumptions of the security proofs. This does not weaken the hash function in itself, but it can undermine the confidence in the design, because the security of the hash function is no longer a consequence of a simple assumption (namely the security of the compression function).

Recently, new results have shown that some attacks on the compression function can be integrated inside the security proof of the mode of operation [2]. This shows that the security of the hash function does not need a truly perfect compression function: some classes of weaknesses of the compression function cannot be used to attack the iterated hash function. As a general rule, it seems that most attacks that require control over the chaining value can be covered by this kind of proofs. However, those attacks usually reveal some unwanted properties of the function, and might be extended to attacks on the full hash function using more advanced techniques.

To put such attacks into perspective, one might look at the attacks on MD5. The first attack on the compression function was found in 1993 by den Boer and Bosselaers [6], using a very simple differential path. This attack did not threaten the iterated hash function, but the path used in the attack is a core element of the successful attack of Wang et al. in 2005 [12].

1.2 Description of BMW

BMW comes in four variants BMW-n, with $n \in \{224, 256, 384, 512\}$, returning output size n. There are two variants of the BMW compression functions; The BMW-256 compression function is used in both BMW-224 and BMW-256, and the BMW-512 compression function is used in both BMW-384 and BMW-512.

The compression function of BMW-256 takes two inputs, H and M, of 16 32-bit words each. The general structure of the compression function is shown in Figure 1. It consists of three functions named f_0, f_1, f_2. The function f_0 applies an invertible linear transformation P to $H \oplus M$ and adds H wordwise modulo 2^{32}. We denote by '$\boxplus$' modular addition, by '$\boxminus$' modular subtraction, and by '$\oplus$' the exclusive or. The output of f_0 is a 16-word vector Q. P consists of a matrix multiplication over $\mathbb{Z}_{2^{32}}$, followed by linear functions $s_{i \bmod 5}$ (see Appendix A) applied to each word W_i individually and by a wordwise rotation by 1 position ($W_{i+1} \leftarrow W_i$).

The function f_1 is a feedback shift register. To begin with, the vector Q contains 16 elements; in each one of 16 rounds of f_1, one more element is added

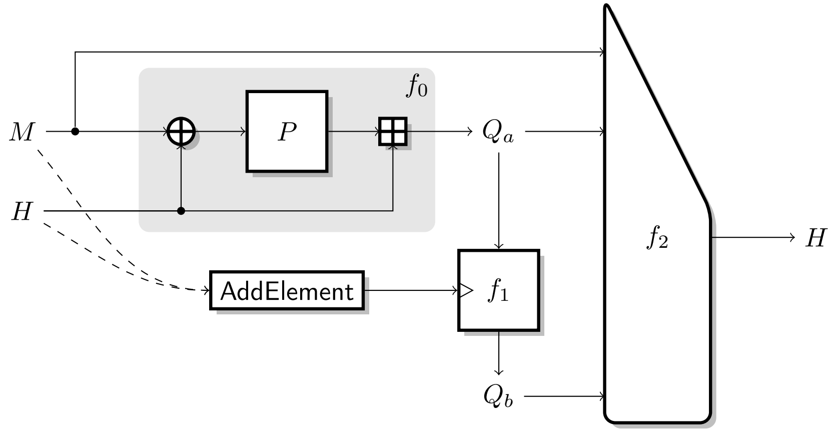

Fig. 1. Compression function of BMW

to Q. This element is computed from the previous 16 elements of Q, and from a value called $\mathsf{AddElement}(i)$ (where i is the round number + 16), which is the following function of three words of M and one word of H:

$$\mathsf{AddElement}(i) = \\ (M_i^{\lll 1+(i \bmod 16)} \boxplus M_{i+3}^{\lll 1+(i+3 \bmod 16)} \boxminus M_{i+10}^{\lll 1+(i+10 \bmod 16)} \boxplus K_i) \oplus H_{i+7}$$

(all indices are to be taken modulo 16, and K_i is a round constant). We note that if there is a collision in the output of f_0 and also in the first, say, j instances of $\mathsf{AddElement}(i)$, then there is a collision in the first $16 + j$ elements of Q. We denote by Q_a the output of f_0, and by Q_b the 16 elements computed in f_1.

The function f_2 performs some final mixing of the elements in Q with M, and produces the 16-word output of the compression function.

Further details on the compression function of BMW can be found in [7].

1.3 Previous results

During the first round of the SHA-3 competition, the best attacks on BMW have been pseudo-attacks due to Thomsen [11]. However, BMW was quite heavily tweaked at the end of the first round, and those attacks do not apply to the current version of BMW. In this paper we only consider the second-round version of BMW.

For the current version of BMW, the best results are differential properties of the compression function, due to Aumasson and Guo and Thomsen [1, 8]. These papers essentially show that for some particular differences in the input of the compression function, a few output bits will be biased.

1.4 Our results

In this paper we describe a partial-collision attack on the compression function of BMW. Our attack is based on differential techniques, and we try to control the propagation of differences inside the compression function. The general idea is to control the differences in M, in Q_a, and in the first instances of AddElement. This means that we also control the difference in the first elements of Q_b, and since the final function f_2 only has limited diffusion, we will control the differences in several output words. We managed to cancel all the differences in $Q_0, \dots Q_{26}$, and to get small differences in Q_{27}, Q_{28}, Q_{29}. This gives a pair of inputs such that 300 pre-specified output bits collide, for a cost similar to 2^{32} evaluations of the compression function, using negligible memory. We note that for a random function, it is expected to take 2^{150} evaluations before finding such a pair of inputs. Moreover, we expect a difference in only half of the uncontrolled bits, and this gives a near-collision attack better than generic algorithms.

Before describing our new attack, we present some useful tools for the analysis of ARX systems in Section 2. In Section 3, we show how to obtain collisions in f_0 without any message difference, which leads to collisions in Q_a, the first half of the Q register. We then show how to find such collisions with some words of H inactive, which leads to a collision in Q_0 to Q_{22}. In Section 4, we extend this result by introducing some differences in the message, and we use the message differences to cancel the chaining value differences in the AddElement function up to Q_{26}. Finally in Section 5 we use near collisions in AddElement instead of full collisions, and we can control the differences up to Q_{29}.

2 Solving a System of Additions and Xor

An important step in our attack requires to solve a system of equations involving only xors and modular additions. In particular, we will often have to solve $x \oplus \Delta = x \boxplus \delta$, where x is a variable, and Δ and δ are given parameters representing respectively the xor-difference and the modular-difference in x. It is well-known that those systems are T-functions, and can be solved from the least significant bit to the most significant bit. However, the naive approach to solve such a system uses backtracking, and can lead to an exponential complexity in the worst case.[3] A more efficient strategy is to use an approach based on automata: any system of such equations can be represented by an automaton, and solving a particular instance take time proportional to the word length. This kind of approach has been used to study differential properties of S-functions in [10]. Here we use this technique to decide whether a system is solvable, and to compute a solution efficiently.

We consider a system of additions and xors, which involves v variables and p parameters. Our goal is twofold: first determine for which values of the parameters the system is compatible, and second, when the system is compatible, determine the set of solutions.

[3] e.g., to solve the system $x \oplus \texttt{0x80000000} = x$, the backtracking algorithm will try all possible values for the 31 lower bits of x before concluding that there is no solution.

The first step in applying this technique is to build an automaton corresponding to the system of equations. The states of this automaton correspond to the possible values of the carry bits: a system with s modular additions gives an automaton with 2^s states. The alphabet is $\{0,1\}^{v+p}$, and each transition reads one bit from each parameter and each variable, starting from the least significant bit. Figure 2 shows an example of such automaton.

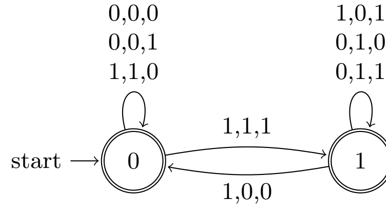

Fig. 2. Carry transitions for $x \oplus \Delta = x \boxplus \delta$. The edges are indexed by Δ, δ, x

Then we remove the variables from the edges, and this gives a non-deterministic automaton which can decide whether a system is solvable or not. We can then build an equivalent deterministic automaton using the powerset construction, as shown in Figure 3.

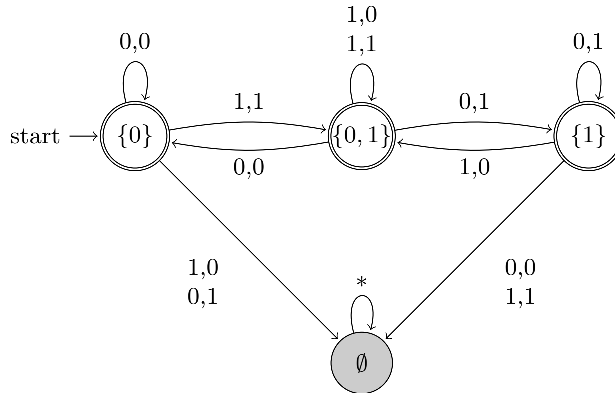

Fig. 3. Decision automaton for $x \oplus \Delta = x \boxplus \delta$. The edges are indexed by Δ, δ

This automaton reveals a lot of information about the system of equation. For instance, one can see that if the state $\{0,1\}$ is reached, then setting $\Delta_i = 1$ assures that the system will have a solution.

In the case of the simple system $x \oplus \Delta = x \boxplus \delta$, we can find an extremely efficient way to check the satisfiability of the system for given parameters, and

to find the actual solutions. By looking at Figure 3, we see that the state $\{0\}$ can only be reached as the initial state or after reading $0, 0$, and that the state $\{1\}$ can only be reached after reading $0, 1$. Moreover, reading $0, 0$ can only lead to state $\{0\}$ or $\emptyset$ and reading $0, 1$ can only lead to state $\{1\}$ or $\emptyset$. This allows a very simple description of the parameters that lead to an inconsistent system, i.e., that reach state $\emptyset$:

– $\Delta_0 \neq \delta_0$, or
– one of the following patterns is seen: $(0, 0), (1, 0)$; $(0, 0), (0, 1)$; $(0, 1), (0, 0)$; $(0, 1), (1, 1)$.

The second condition can be expressed as:

$$\exists i : \Delta_i = 0 \quad \text{and} \quad \delta_i \oplus \Delta_{i+1} \oplus \delta_{i+1} = 1$$

Since those conditions are local they can be tested in parallel using bitwise operations. The following C expression evaluates to one if the system is incompatible, and to zero if it is compatible:

```
((D^d)&1) || ((((D^d)>>1)^d) & (~D)) << 1
```

Note that the rotation to the left is just used to ignore the MSB of the second expression.

Given a compatible pair (Δ, δ), we can use the automaton in Figure 2 to compute a solution x to the equation $x \oplus \Delta = x \boxplus \delta$. First we can remark that if we are in state 0, the next inputs have to satisfy $\delta_i = \Delta_i$, while if we are in state 1, the next inputs have to satisfy $\delta_i \neq \Delta_i$. We can now express the possibles values for x depending on δ and Δ, by looking at the possible transitions in the automata:

$$\begin{cases} \text{if } (\Delta_i, \delta_i) = (0, 0) & \text{then } x_i \text{ is arbitrary:} & x_i \in \{0, 1\} \\ \text{if } (\Delta_i, \delta_i) = (0, 1) & \text{then } x_i \text{ is arbitrary:} & x_i \in \{0, 1\} \\ \text{if } (\Delta_i, \delta_i) = (1, 0) & \text{then } x_i \text{ is given by the next state:} & x_i = \delta_{i+1} \oplus \Delta_{i+1} \\ \text{if } (\Delta_i, \delta_i) = (1, 1) & \text{then } x_i \text{ is given by the next state:} & x_i = \delta_{i+1} \oplus \Delta_{i+1} \end{cases}$$

This can be expressed by the following C expression, where `r` is a random value:

```
(D^d)>>1 ^ (r&(~D|0x8000000))
```

3 Using Collisions in f_0

The first step of our attack is to build collisions in f_0 without any message difference. In the following we denote $x = H \oplus M$ and $y = P(H \oplus M)$. We have $f_0(H, M) = y \boxplus H$ (see Figure 4).

We propose the following algorithm to find such collisions:

1. Pick a random x, x'.

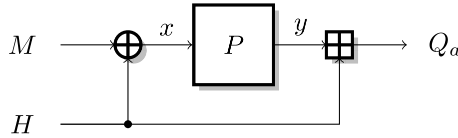

Fig. 4. BMW f_0 function

2. Compute $y = P(x), y' = P(x')$.
3. We have $H \oplus H' = x \oplus x'$ and $H \boxminus H' = y' \boxminus y$. We can solve this and find H using the tools of Section 2.
4. Compute M from x and H: $M = x \oplus H$.

On average, for a random x, x', we expect one solution. However, there is a high probability that there will be no solution for a given x, x', because the xor-difference and the mod-difference for H will not be compatible. (Experiments suggests there is a probability around $2^{-13.9}$ for random differences to be compatible).

To find collisions in practice, we use the degrees of freedom in x to set an xor-difference that has a better probability than a random difference. The best choice is $x' = \neg x$, which works with probability 2^{-1} for each word. However, due to the structure of P, this leads to incompatible systems (the differences in y are constrained by the difference in x). Therefore we use differences of high weight, but we leave some low order bits inactive. This allows to find a compatible system after a few choices of x, x'.

3.1 Collisions in f_0 with some words of H inactive

The next step is to find collisions where some of the words of H are inactive. This will lead to some instances of AddElement being inactive, and some words of Q_b being inactive.

To achieve this, we need an x, x' with some inactive words, but we also require that the same words are inactive in y, y'. Since the inter-word mixing of P is achieved by a linear transformation over $\mathbb{Z}_{2^{32}}$, we can easily find a suitable mod-difference in x. Then we can build the pair x, x' by extending the carries, so that the xor-difference in x is of high Hamming weight.

We use the following algorithm:

1. Pick a random mod-difference in the kernel of the linear transformation P.
2. Build x, x' by extending the carries as much as possible.
3. Compute y, y'.
4. Solve for H.

We can have up to 7 inactive words in H. We use $H_7 \dots H_{13}$ because they are used in the first 7 AddElement rounds. This gives a collision in $Q_0 \dots Q_{22}$. Once we have a solution, we can modify the values of $H_7 \dots H_{13}$ to generate new

solutions by adjusting $M_7 \dots M_{13}$ (we have to keep the value of x). This can be used to get a small difference in Q_{23} as well.

Each choice of x, x' gives a new value for the xor-difference and the mod-difference in H, and we use the tools of Section 2 to check very efficiently whether those values are compatible. The cost of finding a compatible system is negligible before the cost of 2^{32} that we require in order to put a small difference in Q_{23}. This gives some restrictions on the output of the compression function because the f_2 function has 16 outputs and only 9 active inputs, but we do not have any colliding outputs yet.

4 Using Partial-collisions in f_0

In order to improve this result, and to have stronger properties on the output, we have to make more values of Q_b collide. To achieve this, we now put some differences in M, so that differences in M and H can cancel each other in the AddElement function.

More precisely, our best path allows to find collisions in $Q_0 \dots Q_{26}$ using:

- differences in M_{13}, M_{14}, M_{15};
- differences in $H_1 \dots H_6$, H_{10}, H_{11} and H_{12}.

The first step of the attack is to choose a pair x, x' such that $x_{0,7,8,9}$ are inactive, and $y_{0,7,8,9,13,14,15}$ are inactive. Moreover, we fix three more differences: $\delta^{\boxplus} x_{13} = 2^{18}$, $\delta^{\boxplus} y_1 = \texttt{0x04010c43}$, and $\delta^{\boxplus} x_1 = 1$. This is used in order to have $\delta^{\oplus} x_{13} = 2^{18}$, $\delta^{\oplus} y_1 = 1$ (note that $s_0(\texttt{0x04010c43}) = 1$), and $\delta^{\oplus} x_1 = 1$. This gives 14 constraints so we have a solution space of dimension 2.

After fixing the modular difference in x and y, we choose the values of x, x' by extending the carries as much as possible, in order to have a dense xor-difference in M_{14}, M_{15} and $H_2 \dots H_6$, H_{10}, H_{11} and H_{12}. On the other hand, we keep the difference in M_{13} and H_1 sparse so as to have $\delta^{\oplus} x_{13} = 2^{18}$ and $\delta^{\oplus} y_1 = 1$.

When we find a pair x, x' with compatible differences for H, this fixes the values of:

- all active H's: $H_1 \dots H_6, H_{10}, H_{11}, H_{12}$.
- all M's whose corresponding H is active: $M_1 \dots M_6, M_{10}, M_{11}, M_{12}$.

The remaining degrees of freedom are:

- H_0, H_7, H_8, H_9 and M_0, M_7, M_8, M_9, but the values of $H_i \oplus M_i = x_i$ are fixed (4 degrees of freedom).
- H_{13}, H_{14}, H_{15}; M_{13}, M_{14}, M_{15}; and M'_{13}, M'_{14}, M'_{15}, but the values of $H_i \oplus M_i = x_i$ and $M_i \oplus M'_i = x_i \oplus x'_i$ are fixed (3 degrees of freedom).

In order to achieve a collision in $Q_0 \dots Q_{26}$, we need to cancel differences in AddElement 19, 20, 21 and 26.

AddElement(19) $(M_3^{\lll 4} \boxplus M_6^{\lll 7} \boxminus M_{13}^{\lll 14} \boxplus K_{19}) \oplus H_{10}$
We use the freedom of M_{13} to extend carries in $(M_3^{\lll 4} \boxplus M_6^{\lll 7} \boxminus M_{13}^{\lll 14})$.

AddElement(20) $(M_4^{\lll 5} \boxplus M_7^{\lll 8} \boxminus M_{14}^{\lll 15} \boxplus K_{20}) \oplus H_{11}$
We use the freedom of M_{14} and M_7 to extend carries in $(M_4^{\lll 5} \boxplus M_7^{\lll 8} \boxminus M_{14}^{\lll 15})$.

AddElement(21) $(M_5^{\lll 6} \boxplus M_8^{\lll 9} \boxminus M_{15}^{\lll 16} \boxplus K_{21}) \oplus H_{12}$
We use the freedom of M_{15} and M_8 to extend carries in $(M_5^{\lll 6} \boxplus M_8^{\lll 9} \boxminus M_{15}^{\lll 16})$.

AddElement(26) $(M_{10}^{\lll 11} \boxplus M_{13}^{\lll 14} \boxminus M_4^{\lll 5} \boxplus K_{26}) \oplus H_1$
We don't have degrees of freedom available to make this collide, but the differences have been selected so that it happens with high probability: the differences in $M_{13}^{\lll 14}$ and H_1 are only in the least significant bit.

Finally, when we have one solution which collides in $Q_0 \ldots Q_{26}$, we use the freedom in M_0 and M_9 to generate many solutions, until we have a collision in $XH = \bigoplus_{i=16}^{31} Q_i$. This gives a collision in the first three output words for a cost of 2^{32} (see Appendix B for details of the f_2 function). Here is an example of an input pair showing this property is given in Table 1

Table 1. Partial-collision example with 96 controlled bits

Chaining Value							
6ae0a10c	4f14abca	57e66e71	6075a601	6ae0a10c	4f14abcb	a819918f	9f8a59fe
bba141a1	46fb0506	e001fffd	e89b2ebf	445ebe5f	8934faf9	9ffe0002	e89b2ebf
cb1e82d3	ae2d53d6	cb55b67f	e6b080a1	cb1e82d3	ae2d53d6	34aa4980	194f7f5e
8b8c0a70	98d0080b	adaacc99	88f0cf2d	7473f58f	98d0080b	adaacc99	88f0cf2d
Message							
4f5381d3	f96e7f0a	72879df2	e8150fa2	4f5381d3	f96e7f0a	72879df2	e8150fa2
476caf9f	fbacf685	d1c47cb8	73a7bf61	476caf9f	fbacf685	d1c47cb8	73a7bf61
445261cf	a4c0f69f	a2316fdd	12dbc43a	445261cf	a4c0f69f	a2316fdd	12dbc43a
e5197bf4	af952392	c2966021	46cab397	e5197bf4	af992392	3d699fde	39354c6b
Output							
fe57177e	d1e1157d	ccf82758	6aecc4d0	fe57177e	d1e1157d	ccf82758	80b0c87d
cf3d27ab	590788dc	eafe31d9	0e95fe74	0f1b49b9	e0b92229	cf1c1fb4	1fd1f3ab
5b069cc1	b1039e9e	a5049da0	c38e8490	174ab741	7768d4bc	947374c1	74ddf4f9
cb6f569c	96fff629	ee5d89a4	71e405a4	8b4d7466	d075a056	0f8d8b0c	d987e0cb

5 Using near collisions in AddElement

In order to extend the attack with more colliding bits in the output of the compression function, we use near-collisions in the next instances of AddElement. Since we do not have enough remaining degrees of freedom with a given pair x, x', we use freedom in the choice of the x, x' pair in order to go further.

In a practical implementation of the attack, one computes some words of H as a solution to the equation $x \oplus \Delta = x \boxplus \delta$ as described in the previous sections. As an example, in order to find H_5 and H_5' such that $H_5' \oplus H_5 = \delta^{\oplus} x_5$

and $H'_5 \boxminus H_5 = \delta^{\boxplus} y_5$, one may compute H_5 as $(\delta^{\oplus} x_5 \oplus \delta^{\boxplus} y_5)^{\ggg 1}$ (assuming that the pair $(\delta^{\oplus} x_5, \delta^{\boxplus} y_5)$ is compatible). The value y_5 is computed as $s_4(x_1 \boxplus x_2 \boxplus x_9 \boxminus x_{11} \boxminus x_{14})$. Thus, the freedom in the inactive word x_9 can be used to somewhat control H_5 without affecting other conditions. Since M_5 is computed as $H_5 \oplus x_5$, this leads to some freedom in $\mathsf{AddElement}(27) = (M_{11}^{\lll 12} \boxplus M_{14}^{\lll 15} \boxminus M_5^{\lll 6} \boxplus K_{27}) \oplus H_2$, and one may use this freedom to search for a collision in AddElement(27). However, the differences on H_2 and $M_{14}^{\lll 15}$ turn out to be incompatible, so one can only hope for a near-collision in AddElement(27). Still, this will lead to a near-collision in Q_{27}, which will lead to a near-collision in output word 3 of the compression function.

In a similar manner, one can use the freedom in x_0 (through H_{12} and thereby M_{12}) to find a full collision in AddElement(28), which (due to the small difference in Q_{27}) will lead to a near-collision in Q_{28}. Since Q_{28} is the only active word affecting output words 4, 8, and 12 of the compression function, all these three words will contain a near-collision.

Finally, we can use the freedom of M_0 to extend carries in AddElement(29). However, we cannot reach a full collision because the differences in $M_{13}^{\lll 14}$ and H_4 are incompatible.

To summarize, we use the following techniques to extend our attack:

AddElement(27) $(M_{11}^{\lll 12} \boxplus M_{14}^{\lll 15} \boxminus M_5^{\lll 6} \boxplus K_{27}) \oplus H_2$
We use the freedom in x_9 (through H_{15} and thereby M_5) to find a near-collision.

AddElement(28) $(M_{12}^{\lll 13} \boxplus M_{15}^{\lll 16} \boxminus M_6^{\lll 7} \boxplus K_{28}) \oplus H_3$
We use the freedom in x_0 (through H_{12} and thereby M_{12}) to find a full collision.

AddElement(29) $(M_{13}^{\lll 14} \boxplus M_0^{\lll 1} \boxminus M_7^{\lll 8} \boxplus K_{29}) \oplus H_4$
We use the freedom of M_0 to extend carries and find a near-collision.

We stress that these (near-)collisions can be found before searching for a collision in XH, and therefore, since the complexity is still below 2^{32}, the full cost of the attack is still around 2^{32}. Due to carries, however, it cannot be said beforehand how many bits will collide, unless one introduces a few additional bit conditions that will slightly increase the complexity. The search of a collision in XH is done using the freedom in M_9.

Table 2 gives an example of an input pair with an output colliding in 300 pre-specified bits (the search for a (near-)collision in $Q_0, \ldots, Q_{28}$ required the equivalent of about $2^{29.5}$ compression function evaluations). This example can be also be considered as a near-collision with 122 active bits.

6 Conclusion

In this paper we describe a technique to build partial-collisions in the compression function of BMW. We managed to build pairs of input which lead to a collision in 300 pre-specified bits, with complexity 2^{32}. Although it does not weaken the security of the iterated hash function, it is a strong distinguisher

Table 2. Partial-collision example with 300 controlled bits

Chaining Value	
59dfd94b 30b036e3 44ad8a65 47461712	59dfd94b 30b036e2 bb52759b b8b9e8ed
6f56e9b4 425e2d65 40000003 94e62f58	90a9164c bda1d29a bffffffc 94e62f58
12c4bf76 17b18302 4f74ffd3 3ec30f93	12c4bf76 17b18302 b08b002c c13cf06c
8b0f9f9b 7071a4a5 28becf17 6954724f	74f06064 7071a4a5 28becf17 6954724f

Message	
bd050fb4 c6925351 991aa15f 60327d4b	bd050fb4 c6925351 991aa15f 60327d4b
0212e457 9feb065e d6ab8dac 7b52f8ca	0212e457 9feb065e d6ab8dac 7b52f8ca
2f8a9774 1f189302 2043dc85 7b0eac19	2f8a9774 1f189302 2043dc85 7b0eac19
08fe0408 01c2f910 19abe45b 00000000	08fe0408 01c6f910 e6541ba4 ffffffe0

Output	
70588aa3 62e38880 4b32cd23 7da56fd2	70588aa3 62e38880 4b32cd23 7da56fd1
54827a61 d78e6b5f 17cce172 0ae88e5a	54827a62 d78e6b5e f6942bb0 35a96499
232a8830 7f31780e f0865b01 28cb4150	232a8a30 7f31740e 2ad851f7 362f33fb
39ba3bd2 277e9d52 316a7411 c8dbc618	39ba3bd3 27829d53 d239cc6e 29aa1db7

of the compression function. We also note that if the compression function is truncated like in the final transformation of BMW, we can still build pairs of message which collide in more than 110 bits with complexity 2^{32}. This is the first distinguisher on the truncated compression function of BMW.

A similar attack can be mounted on BMW-512 with complexity 2^{64}. It will give pairs of input of the compression function with about 600 colliding bits, including about 220 bits in the second part of the output.

We believe that the techniques developed for this attacks can be useful for further analysis of BMW, and other ARX based SHA-3 candidates.

Acknowledgments

We would like to thank the anonymous reviewers for helpful comments and suggestions.

Bibliography

[1] Aumasson, J.P.: Practical distinguisher for the compression function of Blue Midnight Wish. Available: `http://131002.net/data/papers/Aum10.pdf` (accessed 2011/01/07) (2010)

[2] Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Indifferentiability with Distinguishers: Why Shabal Does Not Require Ideal Ciphers. Cryptology ePrint Archive, Report 2009/199 (2009) `http://eprint.iacr.org/`.

[3] Chang, D., Nandi, M.: Improved Indifferentiability Security Analysis of chopMD Hash Function. In Nyberg, K., ed.: Fast Software Encryption 2008, Proceedings. Volume 5086 of Lecture Notes in Computer Science., Springer (2008) 429–443

[4] Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård Revisited: How to Construct a Hash Function. In Shoup, V., ed.: Advances in Cryptology – CRYPTO 2005, Proceedings. Volume 3621 of Lecture Notes in Computer Science., Springer (2005) 430–448

[5] Damgård, I.: A Design Principle for Hash Functions. In Brassard, G., ed.: Advances in Cryptology – CRYPTO '89, Proceedings. Volume 435 of Lecture Notes in Computer Science., Springer (1990) 416–427

[6] den Boer, B., Bosselaers, A.: Collisions for the Compression Function of MD5. In Helleseth, T., ed.: Advances in Cryptology – EUROCRYPT '93, Proceedings. Volume 765 of Lecture Notes in Computer Science., Springer (1994) 293–304

[7] Gligoroski, D., Klíma, V., Knapskog, S.J., El-Hadedy, M., Amundsen, J., Mjølsnes, S.F.: Cryptographic hash function BLUE MIDNIGHT WISH. Submission to NIST (Round 2). Available: `http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf` (2010/03/22) (September 2009)

[8] Guo, J., Thomsen, S.S.: Deterministic Differential Properties of the Compression Function of BMW. In: Selected Areas in Cryptography 2010, Proceedings. To appear.

[9] Merkle, R.C.: One Way Hash Functions and DES. In Brassard, G., ed.: Advances in Cryptology – CRYPTO '89, Proceedings. Volume 435 of Lecture Notes in Computer Science., Springer (1990) 428–446

[10] Mouha, N., Velichkov, V., Cannière, C.D., Preneel, B.: The Differential Analysis of S-functions. In: Selected Areas in Cryptography 2010, Proceedings. To appear.

[11] Thomsen, S.S.: Pseudo-cryptanalysis of the Original Blue Midnight Wish. In Hong, S., Iwata, T., eds.: Fast Software Encryption 2010, Proceedings. Volume 6147 of Lecture Notes in Computer Science., Springer (2010) 304–317

[12] Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In Cramer, R., ed.: Advances in Cryptology – EUROCRYPT 2005, Proceedings. Volume 3494 of Lecture Notes in Computer Science., Springer (2005) 19–35

A Details on the Permutation P used in f_0

The matrix multiplication taking place in f_0 can be described as follows. Let $x = H \oplus M$. Let C denote the matrix. If $z = C \cdot x$, where x is considered a 16-element vector over $\mathbb{Z}_{2^{32}}$, then

$$\begin{aligned}
z_0 &= x_5 \boxminus x_7 \boxplus x_{10} \boxplus x_{13} \boxplus x_{14} \\
z_1 &= x_6 \boxminus x_8 \boxplus x_{11} \boxplus x_{14} \boxminus x_{15} \\
z_2 &= x_0 \boxplus x_7 \boxplus x_9 \boxminus x_{12} \boxplus x_{15} \\
z_3 &= x_0 \boxminus x_1 \boxplus x_8 \boxminus x_{10} \boxplus x_{13} \\
z_4 &= x_1 \boxplus x_2 \boxplus x_9 \boxminus x_{11} \boxminus x_{14} \\
z_5 &= x_3 \boxminus x_2 \boxplus x_{10} \boxminus x_{12} \boxplus x_{15} \\
z_6 &= x_4 \boxminus x_0 \boxminus x_3 \boxminus x_{11} \boxplus x_{13} \\
z_7 &= x_1 \boxminus x_4 \boxminus x_5 \boxminus x_{12} \boxminus x_{14} \\
z_8 &= x_2 \boxminus x_5 \boxminus x_6 \boxplus x_{13} \boxminus x_{15} \\
z_9 &= x_0 \boxminus x_3 \boxplus x_6 \boxminus x_7 \boxplus x_{14} \\
z_{10} &= x_8 \boxminus x_1 \boxminus x_4 \boxminus x_7 \boxplus x_{15} \\
z_{11} &= x_8 \boxminus x_0 \boxminus x_2 \boxminus x_5 \boxplus x_9 \\
z_{12} &= x_1 \boxplus x_3 \boxminus x_6 \boxminus x_9 \boxplus x_{10} \\
z_{13} &= x_2 \boxplus x_4 \boxplus x_7 \boxplus x_{10} \boxplus x_{11} \\
z_{14} &= x_3 \boxminus x_5 \boxplus x_8 \boxminus x_{11} \boxminus x_{12} \\
z_{15} &= x_{12} \boxminus x_4 \boxminus x_6 \boxminus x_9 \boxplus x_{13}
\end{aligned}$$

The subfunctions s_i, $0 \leq i \leq 4$, used in f_0 are defined as follows.

$$\begin{aligned}
s_0(x) &= x^{\gg 1} \oplus x^{\ll 3} \oplus x^{\lll 4} \oplus x^{\lll 19} \\
s_1(x) &= x^{\gg 1} \oplus x^{\ll 2} \oplus x^{\lll 8} \oplus x^{\lll 23} \\
s_2(x) &= x^{\gg 2} \oplus x^{\ll 1} \oplus x^{\lll 12} \oplus x^{\lll 25} \\
s_3(x) &= x^{\gg 2} \oplus x^{\ll 2} \oplus x^{\lll 15} \oplus x^{\lll 29} \\
s_4(x) &= x^{\gg 1} \oplus x
\end{aligned}$$

B Description of the f_2 function

The f_2 function performs the following computations:

$$XL = \bigoplus_{i=16}^{23} Q_i \qquad\qquad XH = \bigoplus_{i=16}^{31} Q_i$$

$$
\begin{aligned}
HH_0 &= & (XH^{\gg 5} &\oplus Q_{16}^{\gg 5} \oplus M_0) \boxplus (XL &\oplus Q_{24} \oplus Q_0) \\
HH_1 &= & (XH^{\ll 7} &\oplus Q_{17}^{\ll 8} \oplus M_1) \boxplus (XL &\oplus Q_{25} \oplus Q_1) \\
HH_2 &= & (XH^{\gg 5} &\oplus Q_{18}^{\ll 5} \oplus M_2) \boxplus (XL &\oplus Q_{26} \oplus Q_2) \\
HH_3 &= & (XH^{\gg 1} &\oplus Q_{19}^{\ll 5} \oplus M_3) \boxplus (XL &\oplus Q_{27} \oplus Q_3) \\
HH_4 &= & (XH^{\gg 3} &\oplus Q_{20} \oplus M_4) \boxplus (XL &\oplus Q_{28} \oplus Q_4) \\
HH_5 &= & (XH^{\ll 6} &\oplus Q_{21}^{\gg 6} \oplus M_5) \boxplus (XL &\oplus Q_{29} \oplus Q_5) \\
HH_6 &= & (XH^{\gg 4} &\oplus Q_{22}^{\ll 6} \oplus M_6) \boxplus (XL &\oplus Q_{30} \oplus Q_6) \\
HH_7 &= & (XH^{\gg 11} &\oplus Q_{22}^{\ll 2} \oplus M_7) \boxplus (XL &\oplus Q_{31} \oplus Q_7) \\
HH_8 &= HH_4^{\lll 9} \boxplus & (XH &\oplus Q_{24} \oplus M_8) \boxplus (XL^{\ll 8} &\oplus Q_{23} \oplus Q_8) \\
HH_9 &= HH_5^{\lll 10} \boxplus & (XH &\oplus Q_{25} \oplus M_9) \boxplus (XL^{\gg 6} &\oplus Q_{16} \oplus Q_9) \\
HH_{10} &= HH_6^{\lll 11} \boxplus & (XH &\oplus Q_{26} \oplus M_{10}) \boxplus (XL^{\ll 6} &\oplus Q_{17} \oplus Q_{10}) \\
HH_{11} &= HH_7^{\lll 12} \boxplus & (XH &\oplus Q_{27} \oplus M_{11}) \boxplus (XL^{\ll 4} &\oplus Q_{18} \oplus Q_{11}) \\
HH_{12} &= HH_0^{\lll 13} \boxplus & (XH &\oplus Q_{28} \oplus M_{12}) \boxplus (XL^{\gg 3} &\oplus Q_{19} \oplus Q_{12}) \\
HH_{13} &= HH_1^{\lll 14} \boxplus & (XH &\oplus Q_{29} \oplus M_{13}) \boxplus (XL^{\gg 4} &\oplus Q_{20} \oplus Q_{13}) \\
HH_{14} &= HH_2^{\lll 15} \boxplus & (XH &\oplus Q_{30} \oplus M_{14}) \boxplus (XL^{\gg 7} &\oplus Q_{21} \oplus Q_{14}) \\
HH_{15} &= HH_3^{\lll 16} \boxplus & (XH &\oplus Q_{31} \oplus M_{15}) \boxplus (XL^{\gg 2} &\oplus Q_{22} \oplus Q_{15})
\end{aligned}
$$

In the attack of Section 4, we have differences in M_{13}, M_{14}, M_{15}, and $Q_{27}, \dots Q_{31}$, with no difference in XL and XH. In the first part of f_2, this results in differences in $HH_3, \dots HH_7$. In the second part, outputs $HH_8, \dots HH_{15}$ are active.

In the attack of Section 5, we have dense differences in M_{14}, M_{15}, Q_{30}, Q_{31}, and small differences in M_{13}, Q_{27}, Q_{28} and Q_{29}, with no difference in XL and XH. In the first part of f_2, this results in small differences in HH_3, HH_4, HH_5, and dense differences in HH_6 and HH_7. In the second part, there are dense differences in HH_{10}, HH_{11}, HH_{14}, HH_{15}, and small differences in HH_8, HH_9, HH_{12} and HH_{13}.