An improved preimage attack on MD2

Søren S. Thomsen

Department of Mathematics
Technical University of Denmark
DK-2800 Kgs. Lyngby
Denmark

Abstract. This paper describes an improved preimage attack on the cryptographic hash function MD2. The attack has complexity equivalent to about 2^{73} evaluations of the MD2 compression function. This is to be compared with the previous best known preimage attack, which has complexity about 2^{97}.

1 Introduction

A cryptographic hash function takes an arbitrary length input, the message, and produces a fixed length output. The output is often called the hash or the fingerprint of the message. A cryptographic hash function needs to satisfy certain security criteria in order to be considered secure. Let

$$H : \{0,1\}^* \to \{0,1\}^n$$

denote a hash function, whose output is of length n bits. A cryptographic hash function should be resistant to the following attacks:

- **Collision:** Find x and x' such that $x \neq x'$ and $H(x) = H(x')$.
- **2nd preimage:** Given x and $y = H(x)$ find $x' \neq x$ such that $H(x') = y$.
- **Preimage:** Given $y = H(x)$, find x' such that $H(x') = y$.

A collision for any hash function can be found by a birthday attack with complexity $2^{n/2}$. Preimages and 2nd preimages can be found by a brute force search with complexity 2^n. Typically, one considers a hash function secure only if no attack better than these brute force attacks is known.

The Merkle-Damgård construction [1, 5] is a typical method of constructing hash functions. This method works as follows. Given a so-called compression function $f : \{0,1\}^n \times \{0,1\}^\mu \to \{0,1\}^n$ and an initial n-bit value h_0, the message m is split into a number of μ-bit message blocks $m_1, m_2, \ldots, m_t$. Then, for every i from 1 to t one computes

$$h_i \leftarrow f(h_{i-1}, m_i), \tag{1}$$

and finally $H(m) = h_t$ is returned as output. Examples of hash functions based on the Merkle-Damgård construction are MD4 [8], MD5 [9], SHA-1 [7] and many others.

MD2 [3] is an example of a hash function which does not directly follow the Merkle-Damgård principle. It was developed in 1989 by R. Rivest. MD2 deviates from Merkle-Damgård-based hash functions in that a second state, the so-called checksum, is computed from the message, and this checksum is subsequently appended to the message as an additional message block. Another feature which separates MD2 from immediate successors such as MD4, MD5 and SHA-1 is the use of an S-box.

The first cryptanalytic result on MD2 was a collision attack by Rogier and Chauvaud [10] on the compression function of MD2. The attack cannot be immediately extended to the full MD2 hash function due to the checksum. The first attack against the full MD2 hash function was a preimage attack published by F. Muller [6] in 2004. This attack was improved by Knudsen and Mathiassen in [4].

This paper contains a new preimage attack on MD2 based on Muller's pseudo-preimage attack on the MD2 compression function [6]. The attack requires an amount of work equivalent to about 2^{73} evaluations of the compression function of MD2. In comparison, the previous best known preimage attack (of Knudsen and Mathiassen) requires an amount of work equivalent to about 2^{97} evaluations of the compression function.

2 Description of MD2

MD2 takes messages of any length and returns a 128-bit hash. The message is padded so that its length becomes a multiple of 16 in bytes. Padding is described in Section 2.1. The message is then split into t blocks $m_1, m_2, \ldots, m_t$ of 16 bytes each, and a 16-byte checksum block c is computed from the padded message. c is appended to the message as the $(t+1)$-th message block. The $t+1$ blocks are then processed sequentially: starting from the initial state which is the all zero 16-byte string, every message block updates the state, and the state after m_{t+1} has been processed is the output of the hash function. Hence, once the checksum block has been appended to the message, MD2 can be seen as following the Merkle-Damgård principle (1) on the resulting message. We now give the relevant details of the MD2 hash function. Since the internals of the checksum function are irrelevant to the attacks presented in this paper, a detailed description is postponed to Appendix A.2. We would like to mention here, however, that the checksum function can be seen as taking two inputs, the current checksum and a message block, and producing a new checksum. The checksum function is invertible, i.e., given two of the three values, the third can be easily computed.

2.1 Padding

If the original message consists of r bytes, then d bytes each having the value d are appended to the message, where d is the integer between 1 and 16 such that $r+d$ is a multiple of 16. Hence, all messages are padded, even if r is itself a multiple of 16. This padding rule ensures that there is a one-to-one relationship between the original message and the padded message.

2.2 The compression function

The compression function $f : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ works as follows. Let X be a 19×48 matrix of bytes. Given 16-byte strings h_{in} (called the *chaining input*) and m (the *message block*), fill in the first row (row 0) of X as follows (X_i^j is the byte in row i, column j of X):

$$\begin{aligned} X_0^i &\leftarrow h_{\text{in}}[i] \\ X_0^{16+i} &\leftarrow m[i] \\ X_0^{32+i} &\leftarrow h_{\text{in}}[i] \oplus m[i] \end{aligned}$$

Here, $m[i]$ means byte i of the string m. We shall often think of X as consisting of the submatrices A, B, and C of dimension 19×16, such that $X = [A\ B\ C]$.

The compression function fills in the positions in X. At the end, when all positions have been filled in, the 16 bytes $X_{18}^0, \ldots, X_{18}^{15}$ are returned as the output of the compression function.

The positions of X are filled in as follows:

1. $T \leftarrow 0$
2. For $i = 1, \ldots, 18$ do
   (a) For $j = 0, \ldots, 47$ do
      i. $X_i^j \leftarrow S[T] \oplus X_{i-1}^j$
      ii. $T \leftarrow X_i^j$
   (b) $T \leftarrow T + i - 1$

Here, S is an 8-bit S-box, see Appendix A.1. Since the bytes in the last row of B and C are never used, they do not have to be filled in.

See also Figure 1. This view of the compression function is instructive when studying the attack presented in this paper. Note that evaluating the compression function requires the computation

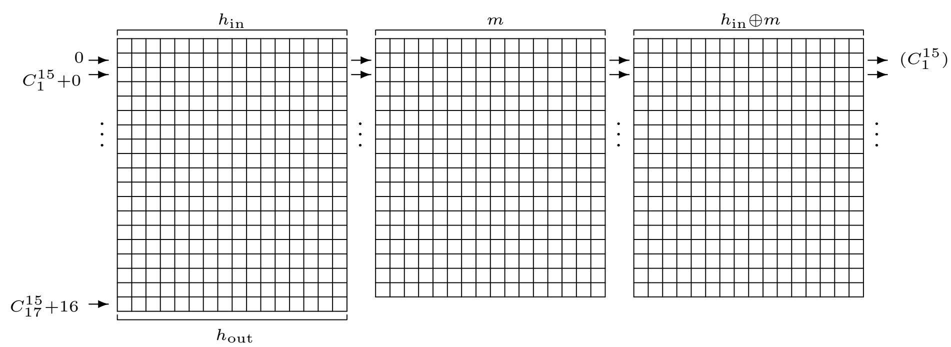


Fig. 1. The MD2 compression function.

of $17 \times 48 + 16 = 832$ bytes in total.

3 A variant of Muller's pseudo-preimage attack

In Section 4.4 of [6], F. Muller describes a pseudo-preimage attack on the MD2 compression function: Given a target h_{out}, he describes how to find h_{in} and m such that $h_{\text{out}} = f(h_{\text{in}}, m)$. The complexity of the attack is about $2^{73.6}$ evaluations of the compression function (our estimate). We now describe a variant of the attack using the notation and terminology introduced in Section 2.

The attack makes use of the observation, that given two out of three bytes in the "triangular" pattern seen in Figure 2, the third byte can be computed. In other words, one does not always have to compute the compression function in the usual forward direction.

This means that when the target, i.e., the last row of submatrix A, is known, a large part of A can be computed immediately, see Figure 3(a). By choosing A_1^{15} and A_2^{15}, one is able to compute a further two diagonals in A, see Figure 3(b). With these values known, all of A can be computed given the chaining input h_{in}. On the other hand, once A_1^{15} and A_2^{15} are fixed, the contents of the matrix B do not depend on h_{in}, only on the message block m.

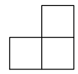
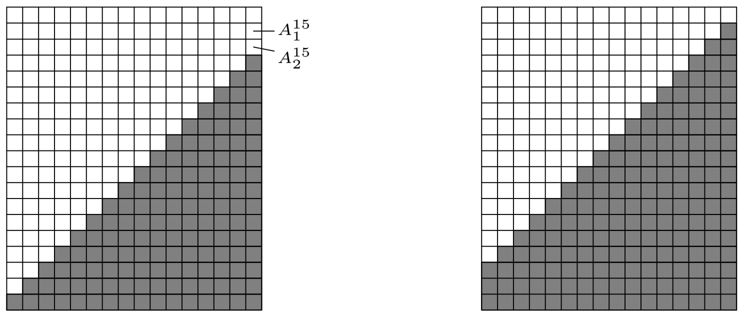


Fig. 2. Given two bytes out of three in this pattern, the third can be computed.

Fig. 3. Values of A that can be computed...



(a) ...from h_{out} (b) ...from h_{out}, A_1^{15} and A_2^{15}

It should be added that once A_1^{15} is fixed, one byte of the chaining input is fixed, since A_1^{15} only depends on the chaining input. Hence, one degree of freedom (in terms of bytes) is lost in the choice of h_{in}.

Apart from being able to compute all of A given h_{in}, also a large part of the matrix C can be computed, since the triangular structure of Figure 2 may in fact be "wrapped around" from matrix A to C. See Figure 4. In the attack a large number of message blocks are chosen, where the last

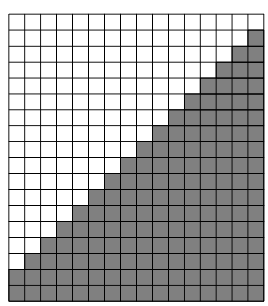


Fig. 4. Values of C that may be computed given h_{in} and h_{out}.

6 bytes are fixed. When this is the case, and h_{in} is known, then, since the first row of C is the xor of h_{in} and m, the last 6 bytes of this row are known, which means another 6 diagonals can be computed, see Figure 5. Also a part of B can be computed. The black values play a certain role in the attack, as we shall see.

The idea of the attack is to compute the values of B that are blackened in Figure 5 in two different ways: given only the chaining input h_{in} (assuming the last 6 bytes of m are fixed); and given only the message block m. We find collisions between these two methods, and for each collision,

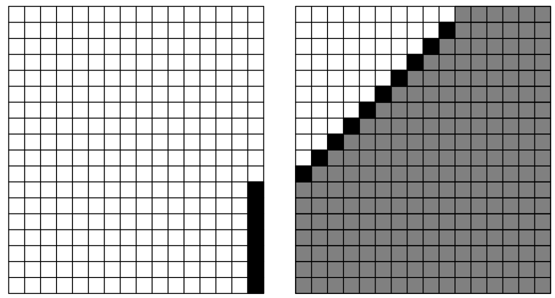


Fig. 5. The blackened and shaded values of B and C can be computed from the chaining input and the last 6 bytes of the message.

we check if there is also a match in the blackened bytes of C. An algorithmic version of the attack is the following:

1. Given target h_{out}, compute as much of A as possible, see Figure 3(a).
2. Choose A_1^{15} and A_2^{15} arbitrarily, and compute a further two diagonals of A, see Figure 3(b).
3. Choose 2^{72} different values of the message block with the last 6 bytes fixed. For each value, compute B and store the last column of B in table T_1.
4. Choose 2^{64} different values of the chaining input in a way such that A_1^{15} gets the right value. Compute all of A and the part of B and C seen in Figure 5. Store the blackened bytes in table T_2.
5. Find collisions between T_1 and T_2 in the bytes of B that are blackened in Figure 5. Since there are 7 bytes, i.e., 2^{56} possible values, and $2^{72} \times 2^{64}$ pairs of values from table T_1 and table T_2, about $2^{72+64-56} = 2^{80}$ collisions are expected.
6. For each collision, check if there is match on the values of C that are blackened in Figure 5. This requires computing the unshaded part of C in Figure 5 for each collision. There are 10 bytes, so one solution is expected. This solution corresponds to a pseudo-preimage.

The most time-consuming part of the attack is finding the (expected) single solution among the 2^{80} collisions. For each collision, on average about 10 bytes of C must be computed, since one may abort once C_1^9 has been computed (from the last column of B, which we stored in T_1), if there is no match on this byte. The amount of work corresponds to about $10/832 \approx 2^{-6.4}$ evaluations of the compression function (since 832 bytes are computed in the compression function, see Section 2). Hence, the complexity of the attack is about $2^{80-6.4} = 2^{73.6}$.

There is a difference between the description of the attack given here, and the description in Section 4.4 of [6]: In Muller's description, the last 6 bytes of the chaining input h_{in} are also fixed. The main observation that leads to the improved preimage attack presented here is that *fixing the last 6 bytes of the chaining input is not necessary.* In fact, any chaining value producing the right value in A_1^{15} can be used in the attack. Hence, to obtain a true preimage attack from the pseudo-preimage attack, a number of chaining values may be computed from a known, fixed initial chaining value, and a number of message blocks. A more detailed description is now given.

4 Converting the pseudo-preimage attack into a preimage attack

The attack described in the previous section is extended in a simple way to obtain a preimage attack. To begin with we shall ignore the checksum.

4.1 An attack without the checksum

We generalise the attack and fix the last k bytes of the message block. This enables us to compute the last $k+1$ bytes of the last column of B. With 2^b different message blocks, and 2^c different chaining inputs, we expect about $2^{b+c-8(k+1)}$ collisions in the last $k+1$ bytes of B. Given a collision we check for a match in $16-k$ bytes of C. Hence, if $2^{b+c-8(k+1)} \geq 2^{8(16-k)}$, then we expect to have found a (pseudo-)preimage. This means we must choose b and c such that $b+c \geq 136$. There is the additional condition on b that it must be possible to produce 2^b different message blocks, given that k bytes are fixed. Hence, we must have $b \leq 8(16-k)$.

We again fix the two bytes A_1^{15} and A_2^{15}. As mentioned, fixing these induces a condition on the chaining input. Apart from this condition, there are no further conditions on the chaining input.

In this extended attack, we make use of two messages blocks. The first block is used to produce a chaining input for the attack of the previous section. The second block is the one found in the attack. The attack can be described as follows.

1. Given target h_{out}, compute as much of A as possible, see Figure 3(a).
2. Choose A_1^{15} and A_2^{15} arbitrarily, and compute a further two diagonals of A, see Figure 3(b).
3. Choose 2^b different values of the message block with the last k bytes fixed. For each value, compute B and store the last column of B in table T_1.
4. Given some chaining value h_0, choose 2^{c+8} message blocks arbitrarily, and evaluate the compression function on h_0 and these. This yields 2^{c+8} new chaining values. Identify the expected 2^c of these which produce the value of A_1^{15} chosen in Step 2. Store the 2^c message blocks and their chaining value in table U.
5. For each of the 2^c chaining values, compute all of A and the parts of B and C seen in Figure 5 (in the figure, $k=6$). Store in table T_2 the bytes of B and C that are blackened in the figure.
6. Find collisions between T_1 and T_2 in the bytes of B that are blackened in Figure 5.
7. For each collision, check if there is match on the values of C that are blackened in Figure 5. The solutions correspond to a preimage (the first message block is found in table U).

As mentioned, with $b+c \geq 136$, we expect at least one preimage. Let us find b, c, k such that this attack has the lowest possible complexity.

Steps 1 and 2 are only done once, and do not contribute to the complexity of the attack. Step 3 requires that we evaluate about a third of the compression function 2^b times. Hence, the complexity is about 2^{b-1}. In Step 4, we evaluate the entire compression function 2^{c+8} times. Step 5 requires 2^c evaluations of at most half the compression function; we must compute about half of A, and most of C. Hence, the complexity may be estimated to 2^{c-1}. In Step 6 we find about $2^{b+c-8(k+1)}$ collisions between T_1 and T_2. The complexity of this step may be estimated to about $\max(2^b, 2^c)/832$, if we assume a comparison is equivalent to the computation of one byte in the compression function. Finding the preimages among the $2^{b+c-8(k+1)}$ collisions (Step 7) requires on average the computation of $16-k$ bytes of C for each collision, hence complexity around $2^{b+c-8(k+1)} \cdot (16-k)/832$.

If we want to find only one preimage, then we would choose b, c, k such that $2^{b+c-8(k+1)} = 2^{8(16-k)} \iff b+c = 136$. Since we need, respectively, 2^{c+8} and 2^{b-1} evaluations of the compression function, we choose b as large as possible, i.e., $b = 8(16-k)$. This means $c = 8(k+1)$. The most time-consuming parts of the attack are expected to be (a) evaluating the compression function 2^{c+8} times, and (b) finding the preimage in the last step. Hence, we may want to make these two tasks equally hard. In other words, we want to find k such that $2^{b+c-8(k+1)} \cdot (16-k)/832 \approx 2^{c+8}$. With the choices of b and c already made, we get that k should be either 6 or 7.

With $k = 7$, we get $b = 72$ and $c = 64$, such that:

- Step 3 takes time about 2^{71}
- Step 4 takes time about 2^{72}
- Step 5 takes time about 2^{63}
- Step 6 takes time about $2^{72}/832 \approx 2^{62.3}$
- Step 7 takes time about $2^{65.5}$.

In total, the complexity is below 2^{73}. Memory requirements are about 2^{73} message blocks.

With $k = 6$ we get $b = 80$ and $c = 56$, so we see already that the complexity must be higher due to Step 3. We could then choose different values for b and c, e.g., $b = 72$ and $c = 64$ as before, but Step 7 is independent of b and c and with $k = 6$ takes time about $2^{73.6}$.

4.2 Taking the checksum into account

In the attack just described, we have completely ignored the checksum. However, the attack works for any chaining input h_0. In other words, given h_0, we are able to find a message consisting of two blocks, such that when this message is processed by MD2, the output is the target h_{out}. Furthermore, the checksum function of MD2 can be inverted in the same time as it takes to evaluate it in the usual forward direction. This means that we may extend the attack as follows (given a target hash value h_{out}).

1. Produce, by the method of A. Joux [2], a 2^{128}-multicollision on MD2 ignoring the checksum. Let the resulting chaining value (common for all messages in the multicollision) be h^*.
2. Apply the attack of Section 4.1 with $h_0 = h^*$ and with target hash value h_{out}. Let the two message blocks resulting from the attack be m_1 and m_2.
3. Let m_2 be the checksum block, and invert the checksum function on m_1 and m_2. This results in a checksum state, call it c^*.
4. Given initial checksum state c_0, compute 2^{64} checksums using the first 64 pairs of blocks of the multicollision. Store the 2^{64} resulting checksum states in table V_1 (also store the corresponding message blocks).
5. Given checksum state c^*, invert the checksum function using the last 64 pairs of message blocks of the multicollision. Store the resulting 2^{64} checksum states in table V_2 (also store the corresponding message blocks).
6. Find (with good probability) a collision between the two tables V_1 and V_2, i.e., a checksum state that appears in both tables. The collision also gives a 128-block message M. A preimage of h_{out} is then $M \| m_1$ (and m_2 is the checksum of this message).

We note that m_1 must have correct padding, but this is easily ensured by always selecting, in the attack of Section 4.1, message blocks with the last byte equal to 1.

Step 1 has complexity equivalent to about $128 \times 2^{64} = 2^{71}$ compression function evaluations. Step 2 has complexity below 2^{73}, as we saw in Section 4.1. Step 3 has negligible complexity. Steps 4 and 5 each have complexity equivalent to about 2^{64} checksum function evaluations. One checksum function evaluation requires the computation of 16 bytes, where each computation is approximately equivalent to the computation of one byte in the compression function (see Appendix A.2). Hence, steps 4 and 5 have total complexity about $2 \times 2^{64} \times 16/832 \approx 2^{59.3}$ compression function evaluations. Step 6 can be done efficiently if V_1 and V_2 are sorted. About 2^{64} comparisons are then needed, equivalent to about $2^{64}/832 \approx 2^{54.3}$ compression function evaluations. The total complexity of the attack is about 2^{73} compression function evaluations. Memory requirements are about 2^{73} message blocks.

5 Conclusion

We have described a preimage attack on MD2 of complexity about 2^{73} in terms of compression function evaluations. The previous best known preimage attack [4] on MD2 has complexity about 2^{97}. Although the task of finding a preimage of MD2 is still an enormous one, it is now close to being within the range of feasibility.

References

1. I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.
2. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.
3. B. S. Kaliski Jr. The MD2 Message-Digest Algorithm, April 1992. RFC 1319.
4. L. R. Knudsen and J. E. Mathiassen. Preimage and Collision Attacks on MD2. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2005.
5. R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.
6. F. Muller. The MD2 Hash Function Is Not One-Way. In P. J. Lee, editor, *Advances in Cryptology – ASIACRYPT 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2004.
7. National Institute of Standards and Technology. Federal Information Processing Standards Publication 180-2. 2002 August 1, Announcing the Secure Hash Standard.
8. R. L. Rivest. The MD4 Message Digest Algorithm. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology – CRYPTO '90, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1991.
9. R. L. Rivest. The MD5 Message-Digest Algorithm, April 1992. RFC 1321.
10. N. Rogier and P. Chauvaud. MD2 Is not Secure without the Checksum Byte. *Designs, Codes and Cryptography*, 12(3):245–251, 1997.

A MD2 details

Some details of the MD2 compression function are omitted in the paper, but given here.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	29	2e	43	c9	a2	d8	7c	01	3d	36	54	a1	ec	f0	06	13
1	62	a7	05	f3	c0	c7	73	8c	98	93	2b	d9	bc	4c	82	ca
2	1e	9b	57	3c	fd	d4	e0	16	67	42	6f	18	8a	17	e5	12
3	be	4e	c4	d6	da	9e	de	49	a0	fb	f5	8e	bb	2f	ee	7a
4	a9	68	79	91	15	b2	07	3f	94	c2	10	89	0b	22	5f	21
5	80	7f	5d	9a	5a	90	32	27	35	3e	cc	e7	bf	f7	97	03
6	ff	19	30	b3	48	a5	b5	d1	d7	5e	92	2a	ac	56	aa	c6
7	4f	b8	38	d2	96	a4	7d	b6	76	fc	6b	e2	9c	74	04	f1
8	45	9d	70	59	64	71	87	20	86	5b	cf	65	e6	2d	a8	02
9	1b	60	25	ad	ae	b0	b9	f6	1c	46	61	69	34	40	7e	0f
a	55	47	a3	23	dd	51	af	3a	c3	5c	f9	ce	ba	c5	ea	26
b	2c	53	0d	6e	85	28	84	09	d3	df	cd	f4	41	81	4d	52
c	6a	dc	37	c8	6c	c1	ab	fa	24	e1	7b	08	0c	bd	b1	4a
d	78	88	95	8b	e3	63	e8	6d	e9	cb	d5	fe	3b	00	1d	39
e	f2	ef	b7	0e	66	58	d0	e4	a6	77	72	f8	eb	75	4b	0a
f	31	44	50	b4	8f	ed	1f	1a	db	99	8d	33	9f	11	83	14

Fig. 6. The MD2 S-box.

A.1 The S-box

The S-box is defined as follows. View the input as a two-digit hexadecimal value. In Figure 6, find the first input digit in the first column, and find the second input digit in the first row. Where the row and the column meet, find the output of S (in hexadecimal). This S-box is derived from the digits of the fractional part of π.

A.2 The checksum function

The checksum function of MD2 operates with a 128-bit (16-byte) state (initially all bytes are zero), which is updated by a message block of the same size. Let D denote the state, and D^i be the ith byte of the state. Let m be the message block with ith byte m^i. The state D is updated by m as follows.

1. $L \leftarrow D^{15}$
2. For increasing i from 0 to 15 do
   (a) $D^i \leftarrow D^i \oplus S[L \oplus m^i]$
   (b) $L \leftarrow D^i$

The reader may note the similarity between the checksum function and the compression function of MD2.