Cryptanalysis of Twister

Florian Mendel and Christian Rechberger and Martin Schläffer

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria.
martin.schlaeffer@iaik.tugraz.at

Abstract. In this paper, we present a semi-free-start collision attack on the compression function for all Twister variants with negligible complexity. We show how this compression function attack can be extended to construct collisions for Twister-512 slightly faster than brute force search. Furthermore, we present a second-preimage and preimage attack for Twister-512 with complexity of about 2^{384} and 2^{456} compression function evaluations, respectively.

Keywords: SHA-3, Twister, hash function, collision-, second-preimage-, preimage attack

1 Introduction

In the NIST SHA-3 competition, many new hash function designs have been submitted. NIST published a list of 51 first round candidates, and Twister [3,4] is one of them. The next step is to reduce the list of 51 candidates to a small set of finalists within the next few years. As an input to this selection process, we describe several cryptanalytic results on Twister in this paper. Similar results on the Whirlpool [1], Grøstl [14] and GOST [13] hash functions, or the Merkle-Damgård [2,10] constrution have been published in [9], [7,8] and [5]. Our results on Twister are summarized in Table 1.

Table 1. Summary of cryptanalytic results on Twister.

type of attack	target	hash size	complexity	memory
semi-free-start collision	compression function	all	2^8	-
collision	hash function	512	2^{252}	2^9
second preimage	hash function	512	2^{384+s}	$2^{10} + 2^{64-s}$
preimage	hash function	512	2^{456}	2^{10}

In the remainder of the paper, we first give a description of Twister in Section 2, outline a practical collision attack on its compression function in Section 3, and give theoretical collision-, second preimage-, and preimage attacks in Sections 4, 5, and 6, respectively. We summarize and conclude in Section 7.

2 Description of Twister

The hash function Twister is an iterated hash function based on the Merkle-Damgård design principle. It processes message blocks of 512 bits and produces a hash value of 224, 256, 384, or 512 bits. If the message length is not a multiple of 512, an unambiguous padding method is applied. For the description of the padding method we refer to [3] and [4]. Let $m = m_1 \| m_2 \| \cdots \| m_t$ be a t-block message (after padding). The hash value $h = H(m)$ is computed as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(H_{i-1}, m_i) \quad \text{for } 0 < i \leq t \\ H_{t+1} &= f(H_t, C) \\ h &= \Omega(H_{t+1}) \ , \end{aligned}$$

where IV is a predefined initial value, C is the value of the checksum and Ω is an output transformation. The checksum C is computed from the intermediate values of the internal state after each Mini-Round. Note that while for Twister-224/256 the checksum is optional it is mandatory for Twister-384/512.

The compression function f of Twister basically consists of 3 Maxi-Rounds. Each Maxi-Rounds consist of 3 or 4 Mini-Rounds (depending on the output size of Twister) and is followed by a feed-forward XOR-operation.

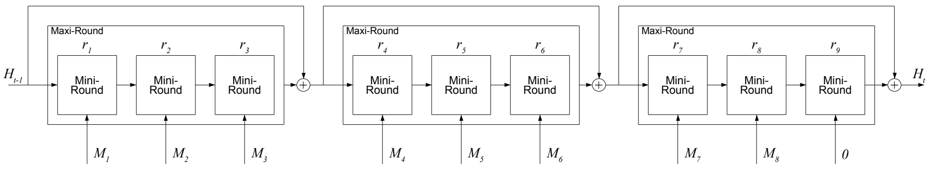

Fig. 1. The compression function of Twister-224/256.

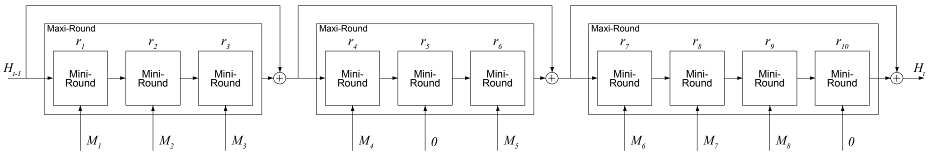

Fig. 2. The compression function of Twister-384/512.

The Mini-Round of Twister is very similar to one round of the Advanced Encryption Standard (AES) [12]. It updates an 8×8 state S of 64 bytes as follows:

MessageInjection A 8-byte message block M is inserted (via XOR) into the last row of the 8×8 state S.

AddTwistCounter A 8-byte block counter is xored to the second column of the state S.

SubBytes is identical to the SubBytes operation of AES. It applies an S-Box to each byte of the state independently

ShiftRows is a cyclic left shift similar to the ShiftRows operation of AES. It rotates row j by $(j-1) \pmod 8$ bytes to the left.

MixColumns is similar to the MixColumns operation of AES. It applies a 8×8-MDS matrix A to each column of the state S. The matrix A and its inverse B are given in Appendix A.

After the last message block and/or the checksum has been processed, the final hash value is generated from the last chaining value H_{t+1} by an output transformation Ω (see Figure 3).

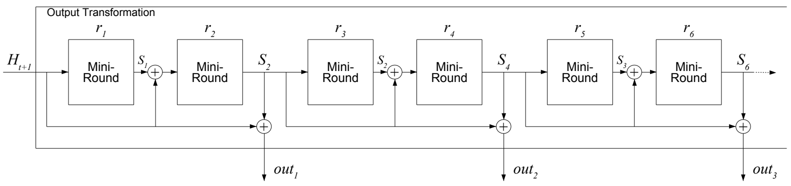

Fig. 3. The output transformation of Twister.

In the output transformation, two Mini-Rounds are applied to subsequently output 64 bits of the hash value. The output stream consists of the XOR of the first column of the state prior and after the two Mini-Rounds. This 64-bit output stream continues until the full hash size has been received. For a more detailed description of Twister we refer to [3] and [4].

3 Semi-Free-Start Collision for the Compression Function

In this section, we present a semi-free-start collision attack on the compression function of Twister for all output sizes. The complexity to find a differential characteristic is about 2^8 compression function evaluations. However, for each differential characteristic, we can construct up to 2^{64} message pairs and the complexity to find one of these conforming message pairs is *one*.

In the attack we use the differential characteristic of Figure 4 for the first Maxi-Round (3 Mini-Rounds) of Twister. The 3 Mini-Rounds are denoted by r_1, r_2 and r_3 and the state after the Mini-Round r_i is denoted by S_i and the state after the corresponding feed-forward S_i^F. The initial state or chaining value is denoted by S_0. In the attack we add a difference in message word M_1 (8 active bytes) to the state S_0, which results in a full active state S_1 after the first Mini-Round r_1. After the MixColumns transformation of the second Mini-Round r_2, the differences result in 8 active bytes of the last row of state S_2, which can be canceled by the message word M_3 in the third Mini-Round r_3.

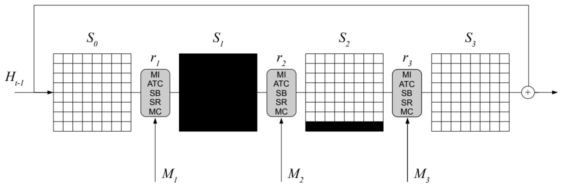

Fig. 4. Characteristic to construct a semi-free-start collision in the first Maxi-Round.

The message differences and values for the state are found using a *rebound* approach as proposed in [9]. Figure 5 shows the characteristic in detail. We start with message word differences in M_1 and M_3 at states S_1' and S_2 (we do not use differences in M_2 to simplify the description of the attack). The differences can be propagated backward and forward through the MixColumns transformation with a probability of one (Step 1). Then, we simply need to find a match for the resulting input and output differences of the SubBytes layer of round r_2 (Step 2) and propagate outwards.

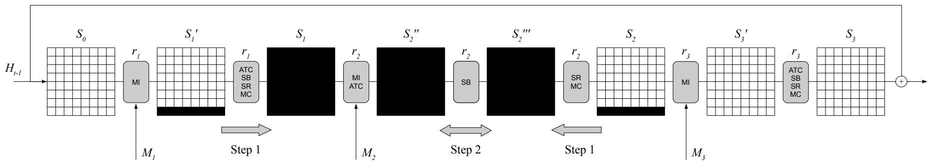

Fig. 5. We start with differences in states S_1' and S_2 injected by message words M_1 and M_3, and propagate backward and forward (Step 1) to find a match for the S-box of round r_2 (Step 2).

Step 1. We start the attack with 8 active bytes in state S_1''' and S_2 (injected by message words M_1 and M_3) and compute backward and forward to two full active states S_2'' and S_2'''. This happens with a probability of one due to the properties of the ShiftRows and MixColumns transformations. Note that we can significantly reduce the complexity of the attack, if we first compute the full active state S_2''' and then compute S_2'' column by column to find a match for 8 S-boxes at once (Step 2).

Step 2. Next, we show how to find a match for the input/output differences of the 64 active S-boxes of round r_2. Note that for a single S-box, the probability that a input/output differential exists is about one half, and for each *valid* input/output differential we can assign at least two possible values to the S-box (for more details we refer to [9]). Note that we can search for valid S-box differentials for each column independently. Hence, we start by choosing a random difference for the first active byte of S_1''' and compute the corresponding row of S_2''. We find a valid differential match for these 8 S-boxes with a probability of $(1/2)^8 = 2^{-8}$. If we find a match, we continue with the remaining active bytes. Alltogether, this step has a complexity of less than 2^8 compression function evaluations.

Once we have found a differential match for the SubBytes layer, we can choose from at least 2^{64} possible states for S_2''. Each of these states can be computed forward and backward and results in a semi-free-start collision for one Maxi-Round. Further, this determines the state S_0 as well as the values and differences of M_1 and M_3 (we can freely choose the values of M_2). Note that the first Maxi-Round is the same for Twister-224/256 and Twister-384/512. Hence, by constructing a semi-free-start collision for the first Maxi-Round we already get a semi-free-start collision for the compression function of Twister-224/256 and Twister-384/512. Since we can find 2^8 semi-free-start collisions with a complexity of 2^8 compression function evaluations, the average complexity to find one semi-free-start collisions is one with negligible memory requirements. An example for a semi-free start collision is given in 2.

Table 2. A colliding message pair (M, M^*) for the semi-free-start collision of the first Maxi-Round (S_3, S_3^*) of Twister. The corresponding semi-free-start collision for Twister-256 (with output transformation) is given by $H(256)$ and $H^*(256)$.

H_0	A63215B04567E389	16D40B5ACFABED9D	C0C4104853084862	C38990B8BEBF7BED
	E936F9AF6406E35B	F5BE6C8455626226	C6C9FA7B806B3BD1	E22C576CDE8ABDB5
H_0^*	A63215B04567E389	16D40B5ACFABED9D	C0C4104853084862	C38990B8BEBF7BED
	E936F9AF6406E35B	F5BE6C8455626226	C6C9FA7B806B3BD1	E22C576CDE8ABDB5
ΔH_0	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
M	0000000000000000	0000000000000000	0000000000000000	
M^*	309F4C5E31CAD0EE	0000000000000000	CF7CA0BD904331CB	
ΔM	309F4C5E31CAD0EE	0000000000000000	CF7CA0BD904331CB	
S_3	8B040660A8F0C7BF	09EE0D5A362F769E	B62FDC8118D186F2	96E6A8E0049B4BA7
	5494AA985B53A83F	B91DE273FA61A073	8082BCD3BB503820	56225FFB45DBA4F8
S_3^*	8B040660A8F0C7BF	09EE0D5A362F769E	B62FDC8118D186F2	96E6A8E0049B4BA7
	5494AA985B53A83F	B91DE273FA61A073	8082BCD3BB503820	56225FFB45DBA4F8
ΔS_3	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
$H(256)$	DE6D957A627CEBBF	88326DBED4135BB0	2039C5411191AD47	A15703E5EA2E66A2
$H^*(256)$	DE6D957A627CEBBF	88326DBED4135BB0	2039C5411191AD47	A15703E5EA2E66A2
$\Delta H(256)$	0000000000000000	0000000000000000	0000000000000000	0000000000000000

4 A Collision Attack on Twister-512

In this section, we show how the semi-free-start collision attack on Twister-512 can be extended to the hash function. We first show how to construct collisions in the compression function of Twister-512 with a complexity of 2^{223} compression function evaluations. This collision attack on the compression function is then extended to a collision attack on the hash function. The extension is possible by combining a multicollision attack and a birthday attack on the checksum. The attack has a complexity of about 2^{252} evaluations of the compression function of Twister-512.

4.1 Collision Attack on the Compression Function

For the collision attack on the compression function of Twister-512 we can use the characteristic of the previous section in the last Maxi-Round (see Figure 6). Remember that in Twister-512 the 3 message words M_6, M_7 and M_8 are injected in the last Maxi-Round. Hence, we can use the first 5 message words $M_1 - M_5$ for a birthday match on 56 state bytes with a complexity of $2^{8 \cdot 56/2} = 2^{224}$. Since the 8 bytes of the last row can always be adapted by using the freedom in the (absolute) values of the message word M_6, we only need to match 56 out of 64 bytes. It can be summarized as follows:

1. Compute 2^{224} semi-free-start collisions for the last Maxi-Round of Twister-512 and save them in a list L. This has a complexity of about $3 \cdot 2^{224}$ Mini-Round computations. Note that we can choose from $2^{3 \cdot 64} = 2^{192}$ differences in M_6, M_7 and M_8 in the attack. Furthermore, by varying the values of M_7, we get additional 2^{64} degrees of freedom. Hence, we can construct up to 2^{256} semi-free-start collisions for the last Maxi-Round.
2. Compute the input of the last Maxi-Round by going forward and check for a match in the list L. After testing about 2^{224} candidates for the input of the last Maxi-Round we expect to find a match in the list L and hence, a collision for the compression function of Twister-512. Finishing this step of the attack has a complexity of about 2^{224} Mini-Round computations.

Hence, we can find a collision for the compression function of Twister-512 for the predefined initial value with a complexity of about 2^{223} compression function evaluations ($10 \cdot 2^{223}$ Mini-Round computations) and memory requirements of 2^{224}. Note that memory requirements of this attack can significantly be reduced by applying a memory-less variant of the meet-in-the-middle attack introduced by Quisquater and Delescaille [15], and applied by Morita, Ohata and Miyaguchi [11].

4.2 Collision Attack on the Hash Function

In this section, we show how the collision attack on the compression function can be extended to the hash function. The attack has a complexity of about

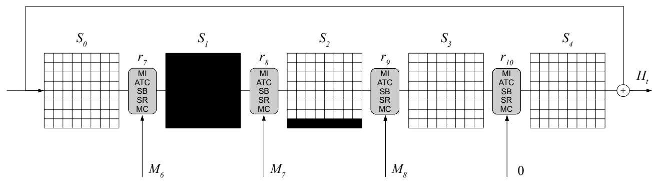

Fig. 6. The characteristic for the last Maxi-Round of Twister-512.

2^{252} evaluations of the compression function of Twister-512. Note that the hash function defines, in addition to the common iterative structure, a checksum computed over the outputs of each Mini-Round which is then part of the final hash computation. Therefore, to construct a collision in the hash function we have to construct a collision in the iterative structure (*i.e.* chaining variables) as well as in the checksum. To do this we use multicollisions.

A multicollision is a set of messages of equal length that all lead to the same hash value. As shown in [6], constructing a 2^t collision, *i.e.* 2^t messages consisting of t message blocks which all lead to the same hash value, can be done with a complexity of about $t \cdot 2^x$ for any iterated hash function, where 2^x is the cost of constructing a collision in the compression function. As shown in the previous section, collisions for the compression function of Twister-512 can be constructed with a complexity of 2^{223}. Hence, we can construct a 2^{256} collision with a complexity of about $256 \cdot 2^{223} \approx 2^{231}$ evaluations of the compression function of Twister-512 and memory requirements of 2^9 (needed to store the 2^{256} collision). With this method we get 2^{256} values for the checksum C that all lead to the same chaining value H_{256}.

To construct a collision in the checksum of Twister-512 we have to find 2 distinct messages consisting of 257 message blocks (256 message blocks for the multicollision and 1 message block for the padding) which produce the same value in the checksum. By applying a birthday attack we can find these 2 messages with a complexity of about 2^{256} checksum computations and memory requirements of 2^{256}. Due to the high memory requirements of the birthday attack, one could see this part as the bottleneck of the attack. However, the memory requirements can be significantly reduced by applying a memory-less variant of the birthday attack [15]. Hence, we can find a collision for Twister-512 with a complexity of about 2^{231} compression function evaluations (10 Mini-Rounds) and about 2^{256} checksum computations (8 xor operations and 8 modular additions of 64-bits). In general the cost for one checksum computation is smaller than one compression function evaluation. Depending on the implementation 1 checksum computation is $1/x$ compression function evaluation. Assume $x = 16$, then we can find a collision for Twister slightly faster than brute force search with a complexity of about 2^{252} compression function evaluations and negligible memory requirements.

4.3 A Remark on the Length Extension Property

Once, we have found a collision, *i.e.* collision in the iterative part (chaining variables) and the checksum, we can construct many more collisions by appending an arbitrary message block. Note that this is not necessarily the case for a straightforward birthday attack. By applying a birthday attack we construct a collision in the final hash value (after the output transformation Ω) and appending a message block is not possible. Hence, we need a collision in the iterative part as well as in the checksum for the extension property. Note that by combining the generic birthday attack and multicollisions, one can construct collisions in both parts with a complexity of about $256 \cdot 2^{256} = 2^{264}$ while our attack has a complexity of 2^{252}.

5 A Second-Preimage Attack on Twister-512

In this section, we present a second-preimage for Twister-512 with complexity of about 2^{384} compression function evaluations and memory requirements of 2^{64}. Assume we want to construct a second preimage for the message $m = m_1 \| \cdots \| m_{513}$. Then, the attack can be summarized as follows.

1. Construct a 2^{512} collision for the first 512 message blocks of Twister-512. This has a complexity of about $512 \cdot 2^{256} \approx 2^{265}$ compression function evaluations (using a birthday attack to construct a collision for each message block) and needs 2^{10} memory to save the multicollision. Hence, we get 2^{512} values for the checksum which all lead to the same chaining value H_{512}.
2. Choose an arbitrary value for the message block m_{513} with correct padding and compute H_{513}.
3. In the last iteration of the compression function, $H_{514} = f(H_{513}, C)$, we first choose arbitrary values for the five checksum words $C_1, \dots, C_5$ with $C = C_1 \| \cdots \| C_8$ and compute the state $S_6^F = H_{513} \oplus S_3 \oplus S_6$. This also determines $S_{10} = H_{514} \oplus S_6^F$. Note that we know H_{514} from the first preimage.
4. For all 2^{64} choices of C_8 compute backward from S_{10} to the injection of C_7 and save the 2^{64} candidates for state $S_7' = \mathsf{MessageInjection}(\mathsf{S_7}, \mathsf{C_7})$ in the list L.
5. For all 2^{64} choices of C_6 compute forward from S_6 to the injection of C_7 and check for match of S_7' in the list L. Since we can still choose C_7, we only need to match 448 (out of 512) bits. In total, we get 2^{128} pairs and this step of the attack will succeed with probability $2^{-448+128} = 2^{-320}$. By repeating steps 3–5 about 2^{320} times we can find a match and fulfill this step of the attack with a complexity of about $2^{320+64} = 2^{384}$ compression function evaluations.
6. Once we have constructed a second-preimage for the iterative part, we still have to ensure that the value of the checksum C is correct. Therefore, we now use the fact that the checksum of Twister is invertible and we have 2^{512} values for the checksum which all lead to the same chaining value H_{512} and hence H_{513} and H_{514}. By using a meet-in-the-middle-attack, we can

construct the needed value in the checksum. This has a complexity of about 2^{257} checksum computations and memory requirements of 2^{256}. Again the memory requirements can be significantly reduced by using a memory-less variant of the meet-in-the-middle attack [15].

Hence, we can construct a second-preimage for Twister-512 with complexity of about 2^{384} and memory requirements of $2^{10} + 2^{64}$. The memory requirements can be significantly reduced at the cost of an higher attack complexity. Several time/memory tradeoffs are possible between 2^{384+s} compression function evaluations and memory requirements of $2^{10} + 2^{64-s}$. Note that our attack requires that the first message consists of at least 513 message blocks. Due to the output transformation of Twister-512, the attack can not be extended to a preimage attack on Twister-512 in a straight-forward way.

6 A Preimage Attack on Twister-512

In order to construct a preimage, we have to invert the output transformation of Twister-512. Once we have inverted the output transformation, we can use the second preimage attack described in the previous section to construct a preimage for Twister-512. Suppose we seek a preimage of $h = out_1 \| \cdots \| out_8$ consisting of 513 message block. Then we have to find the chaining value H_{514} such that $\Omega(H_{514}) = h$.

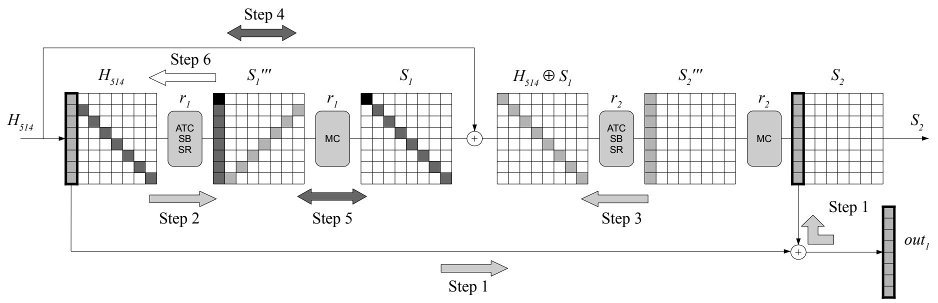

Fig. 7. The inversion of the first part of the output transformation of Twister.

In the following, we show how to find a H_{514} such that out_1 is correct with a complexity of about 2^8 instead of the expected 2^{64}. This reduces the complexity of inverting the whole output transformation Ω to about 2^{456} instead of the expected complexity of 2^{512}. The inversion of the first step of the output transformation can be summarized as follows (see also Figure 7):

1. Choose a random value for the first column of H_{514}. Use out_1 and the first column of H_{514} (8 bytes each) to compute the first column of S_2 (8 bytes).

2. Compute the 8 bytes $S_1'''[i][1 + (9 - i) \mod 8]$ for $(1 \leq i \leq 8)$ of state S_1''' using the first column of H_{514}.
3. Compute backward through the Mini-Round r_2 for the first column of S_2 to get the diagonal 8 bytes of $S_1 \oplus H_{514}$.
4. Choose random values for the 8 diagonal bytes of H_{514}. Note that this determines the first column of S_1'''. Next, compute the 8 diagonal bytes of S_1 from the diagonal bytes of $H_{514} \oplus S_1$ and H_{514} using the feed-forward.
5. Now, we need to connect the states S_1''' and S_1 through the MixColumns operation of Mini-Round r_1. Note that the first column of S_1''' is already fixed (due to step 4). If the first byte of $S_1[1][1]$ does not match, we need to go back to step 1 again. After repeating steps 1-4 about 2^8 times we expect to find a match for $S_1[1][1]$. Once, we have found a match, we have to modify column 2-8 of S_1''' such that the remaining 7 bytes match as well.
   (a) For each column $i = 2 \dots 8$ choose random values for the bytes $S_1'''[i][k]$ with $k \neq 10 - i$. Note that the bytes $S_1'''[i][k]$ with $k = 10 - i$ are already fixed due to step 2 of the attack.
   (b) Next, we compute the MixColumns operation and check if the byte $S_1[i][i]$ matches. If not, we repeat the previous step. This has a complexity of about 2^8.

   Since each column can be modified independently in the attack, finishing this step of the attack has a complexity of about $8 \cdot 2^8 \approx 2^{11}$ Mini-Round computations.
6. After we have found a match for all columns, we can compute backwards from S_1''' to determine H_{514}. Note that values fixed in step 1 and step 4 do not change anymore.

Hence, we can find a H_{514} such that out_1 is correct with a total complexity of about 2^{11} Mini-Round computations, respectively 2^8 compression function evaluations. By repeating the attack about 2^{448} times, we can invert the output transformation of Twister-512 with a complexity of about $2^{438} \cdot 2^8 = 2^{456}$ compression function evaluations and memory requirement of 2^{10}.

Once we have inverted the output transformation, *i.e.* we have found the chaining value H_{514} such that $\Omega(H_{514}) = h$, we can apply the second preimage attack described in the previous section to construct a preimage for Twister-512 consisting of 513 message blocks. The attack has a complexity of about $2^{448} + 2^{456} \approx 2^{456}$ compression function evaluations and negligible memory requirements.

7 Conclusion

In this paper, we have shown two main results: Although Twister is heavily based on a Merkle-Damgård style iteration (as many other hash function like SHA-2), the corresponding reduction proof that reduces the collision resistance of the hash function to the collision resistance of the compression function is not applicable anymore. We show practical (in time and memory) attacks and give an example for a compression function collision. This clearly invalidates the collision resistance assumption of the compression function.

Secondly, we give a theoretical collision, second preimage and preimage attack on the hash function Twister-512. Although the practicality of the proposed attacks might be debatable, it nevertheless exhibits non-random properties that are not present in SHA-512.

Acknowledgements

The authors wish to thank the designers of Twister for useful comments and discussions. The work in this paper has been supported in part by the European Commission under contract ICT-2007-216646 (ECRYPT II) and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

1. Paulo S. L. M. Barreto and Vincent Rijmen. The WHIRLPOOL Hashing Function. Submitted to NESSIE, September 2000. Revised May 2003. Available online at `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html` (2008/07/08).
2. Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 416–427. Springer, 1989.
3. Ewan Fleischmann, Christian Forler, and Michael Gorski. The Twister Hash Function Family. Submission to NIST, 2008.
4. Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. Twister - A Framework for Secure and Fast Hash Functions. In Hui Li and Feng Bao, editors, *ISPEC*. Springer, 2009. To appear.
5. Praveen Gauravaram and John Kelsey. Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 36–51. Springer, 2008.
6. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.
7. Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (Second) Preimage Attack on the GOST Hash Function. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 224–234. Springer, 2008.
8. Florian Mendel, Norbert Pramstaller, Christian Rechberger, Marcin Kontak, and Janusz Szmidt. Cryptanalysis of the GOST Hash Function. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 162–178. Springer, 2008.
9. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *Fast Software Encryption*. Springer, 2009. To appear.
10. Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 428–446. Springer, 1989.
11. Hikaru Morita, Kazuo Ohta, and Shoji Miyaguchi. A Switching Closure Test to Analyze Cryptosystems. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *LNCS*, pages 183–193. Springer, 1991.

12. National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce, November 2001.
13. Government Committee of Russia for Standards. GOST 34.11-94, Gosudarstvennyi Standard of Russian Federation, Information Technology Cryptographic Data Security Hashing Function, 1994. (In Russian).
14. Praveen Gauravaram and Lars R. Knudsen and Krystian Matusiewicz and Florian Mendel and Christian Rechberger and Martin Schläffer and Søren S. Thomsen. Grøstl - a SHA-3 candidate. Available online at `http://www.groestl.info`, 2008.
15. Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 408–413. Springer, 1989.

A MixColumns and Inverse MixColumns

The MDS matrix A of the MixColumns operation of Twister and its inverse B are given as follows:

$$A = \begin{pmatrix} 02 & 01 & 01 & 05 & 07 & 08 & 06 & 01 \\ 01 & 02 & 01 & 01 & 05 & 07 & 08 & 06 \\ 06 & 01 & 02 & 01 & 01 & 05 & 07 & 08 \\ 08 & 06 & 01 & 02 & 01 & 01 & 05 & 07 \\ 07 & 08 & 06 & 01 & 02 & 01 & 01 & 05 \\ 05 & 07 & 08 & 06 & 01 & 02 & 01 & 01 \\ 01 & 05 & 07 & 08 & 06 & 01 & 02 & 01 \\ 01 & 01 & 05 & 07 & 08 & 06 & 01 & 02 \end{pmatrix}$$

$$B = A^{-1} = \begin{pmatrix} 3E & C5 & 7A & E7 & 1B & A9 & 8A & 23 \\ 23 & 3E & C5 & 7A & E7 & 1B & A9 & 8A \\ 8A & 23 & 3E & C5 & 7A & E7 & 1B & A9 \\ A9 & 8A & 23 & 3E & C5 & 7A & E7 & 1B \\ 1B & A9 & 8A & 23 & 3E & C5 & 7A & E7 \\ E7 & 1B & A9 & 8A & 23 & 3E & C5 & 7A \\ 7A & E7 & 1B & A9 & 8A & 23 & 3E & C5 \\ C5 & 7A & E7 & 1B & A9 & 8A & 23 & 3E \end{pmatrix}.$$