# The Twister Hash Function Family

www.twister-hash.com

October 28, 2008

Ewan Fleischmann* Christian Forler† Michael Gorski‡

*Bauhaus-University Weimar , ewan.fleischmann@uni-weimar.de
†Sirrix AG security technologies , c.forler@sirrix.com
‡Bauhaus-University Weimar , michael.gorski@uni-weimar.de

# Executive Overview

The main advantages of the Twister hash function family are:

- portable to many platforms 8, 32, 64 bit
- using well established and studied design concepts (MDS)
- security level can be expressed explicitly
- speed comparable with SHA-256 and SHA-512
- low memory requirements
- very fast for short message hashes
- very fast diffusion due to the MDS concept
- the initial value depends on the output size of the hash value

One of the most difficult tasks for cryptographer is to design a hash function which reaches a claimed security level by being fast as well. It is often only a trade of between security and speed. On the one hand, a hash function with a huge security margin offers a high level of security but might be useless for practical applications. On the other hand a very fast hash function might offer some unexploited weaknesses, while it is used for crucial applications.

We have learned from the AES competition that a well designed block cipher must serve the demand for speed while been easy to analyze without flaws or trapdoors inside.

Twister is a new family of cryptographic hash functions which we regard as an evolutional step from the AES process as well as from the latest research in the young field of cryptographic hash functions. The core of Twister consists of the well studied MDS concept known from the AES winner Rijndael. The biggest advantage of this construction is its simplicity and the well studied security analysis. This makes the Twister hash family very easy to analyze and we can even give some provable security claims.

Twister is defined for different internal state sizes and a variety of output sizes from 32 bits to 512 bits. This allows Twister to be a drop-in replacement for the entire SHA/SHA2 family of hash functions.

# Contents

# 1 Introduction

The National Institute of Standards and Technology (NIST) is conducting a competition which should lead to a new hash function standard. The new hash algorithm should replace the entire SHA-2 [20] family and therefore must provide message digests of size 224, 256, 384 and 512 bits. The winner of the NIST hash function competition will define the new SHA-3 standard for hash functions.

One of the most used primitives in modern cryptography are hash functions. A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is used to compute an $n$-bit fingerprint out of an arbitrarily-sized input. Established security requirements for cryptographic hash functions are collision-, pre-image and 2nd pre-image resistance – but ideally, cryptographers expect a good hash function to *behave like a random function*. Nearly all iterative hash functions are designed using the Merkle-Damgård construction [17, 33]. A Merkle-Damgård hash function is an iterated hash function that uses a fixed length compression function $C : \{0,1\}^{n_c} \times \{0,1\}^m \rightarrow \{0,1\}^{n_c}$ where $n_c$ is the size of the chaining value and $m$ the size of a message block. We have $n = n_c$ for hash functions using the Merkle-Damgård construction. By assuming a padded message $M = (M_1, \ldots, M_l)$, $|M_i| = m$, $1 \leq i \leq l$ and an internal chaining value $h_i \in \{0,1\}^{n_c}$ ($h_0$ is called the initial value) the computation of the hash value for such a message $M$ is as follows:

```
for $i$ from 1 to $l$ do
    $h_i = C(h_{i-1}, M_i)$
return $h_l$
```

The main benefit of the Merkle-Damgård transformation is that it preserves collision resistance: if the compression function $C$ is collision resistant, then so is the hash function. Unfortunately, this result does not extend to 2nd pre-image resistance. Recent results highlight some intrinsic limitations of the Merkle-Damgård approach. This includes being vulnerable to multicollision attacks [24], long second-pre-images attacks [26], and herding [25]. Even though the practical relevance of these attacks is unclear, they highlight some security issues which designers are well advised to avoid or take care of.

## 1.1 Related Work.

Most popular hash functions such as MD5 [40], SHA-0 [35] or SHA-1 [34] possess weaknesses in their design, leading to a huge amount of attacks [6, 7, 12, 19, 39, 42, 43, 44].

But also most new hash functions [23, 27, 28, 2] which try to take care for weaknesses in the MERKLE-DAMGÅRD -construction itself were broken soon after their publications [31, 38, 37, 32, 22].

The concept of sponge functions [5] uses an a big internal state that absorbs a message of infinite length and that later squeezes out an hash value of variable size. RADIOGATÚN [4] with XOR sponges and GRINDAHL [28] with truncate-overwrite sponges are the first hash function that use this framework. GRINDAHL was shown to be vulnerable to several attacks [37, 22].

## 1.2 TWISTER Hash Function Family

TWISTER is a family of hash functions with a large variety of output sizes. The minimum output size is 32 bits. In our implementation the output size growths in 32 bits steps up to 512 bits. We choose the 32 bits steps to cover up all SHA hash funtions.

Its internal state is 512 bit when the output size is smaller then 256 bit, else it is 1024 bit.

For all instances of TWISTER the internal state can be enlarged from output size $n$ to $2n, 3n$ or $4n$. This depends on the security requirements and the memory which is available on the desired platform. We developed a new design for the TWISTER hash function family which is different from the classical MERKLE-DAMGÅRD design in many ways.

# 2 An Informal Description of Twister

In this section, we give a general description of the Twister hash function family and its building blocks. For a complete description of Twister, see the formal specifications in Section 4. Twister follows a very simple and clear design goal. It consists of an iterated compression function and of an output function.

## 2.1 The Compression Function

The compression function of Twister consists of building blocks called `Mini-Rounds` which are grouped into `Maxi-Rounds`. Each `Mini-Round` is a combination of well studied primitives, which are easy to analyze and fast to implement in software and hardware. The instances of the Twister hash function family differ only in their construction of a `Maxi-Round`. We will give an informal description of these principles below.

### 2.1.1 A `Mini-Round`

`Mini-Round` consists of the following primitives:

- MessageInjection inserts a 64-bit message block into the last row of the state matrix,
- SubBytes applies a non-linear S-Box table look up on each byte of the state matrix in parallel,
- AddTwistCounter XOR's a round dependent counter into the second column of the state matrix,
- ShiftRows rotates $i$ by $i$ positions to the left,
- MixColumns applies a linear diffusion on each column of the state matrix in parallel

A visualization of a mini round is show in Figure 2.1

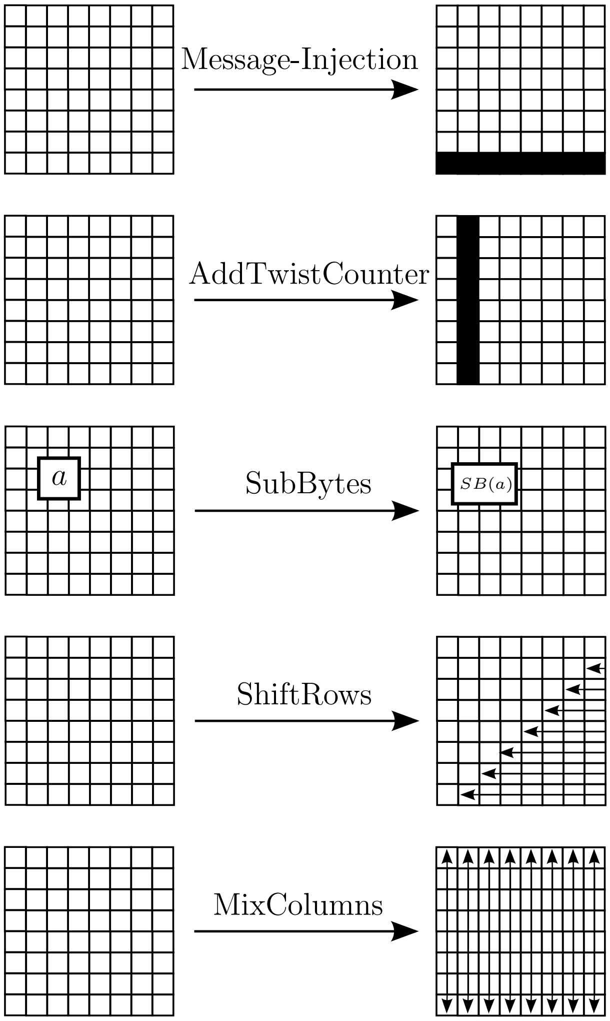


Figure 2.1: A `Mini-Round`

### 2.1.2 A `Maxi-Round`

A `Maxi-Round` contains between three and four `Mini-Rounds` and zero or one `blank rounds`. A `blank round` is a `Mini-Round` with no message input, which is equivalent to the all zero message block. Each `Maxi-Round` uses also a feed forward operation, i.e., the state before a `Maxi-Round` is feed forwarded with the state after a `Maxi-Round`. Figure 4.4 gives a high level description of a `Maxi-Round`.

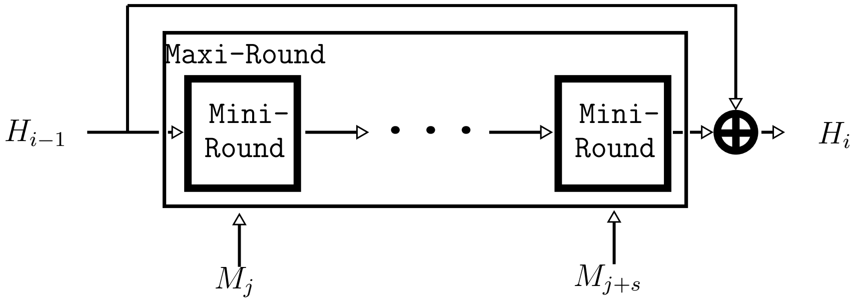


Figure 2.2: A `Maxi-Round`

### 2.1.3 Instances

Now we give an informal description of the compression function of the Twister instances.

#### Twister-224 and Twister-256

The compression function of Twister-224 and Twister-256 consists of three `Maxi-Rounds`, where each of them contains three `Mini-Rounds`. The compression function is displayed in Figure 2.3. A zero indicates a `blank round`.

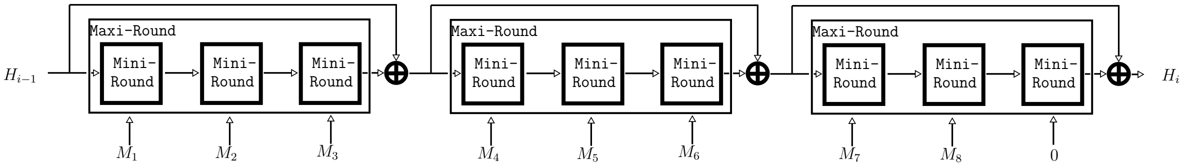


Figure 2.3: The compression function of Twister-224 and Twister-256

#### Twister-384 and Twister-512

The compression function of Twister-384 and Twister-512 consists of three `Maxi-Rounds`, where the first and the second `Maxi-Round` contains three `Mini-Rounds` each. The last `Maxi-Round` contains four `Mini-Rounds`. We show the compression function in Figure 2.4. Furthermore Twister-384 and Twister-512 contains a 512-bit check sum which will be computed as follows. The check sum can be regarded as a $8 \times 8$ matrix of 16 byte entries. It is initialized with zero and updated before a message input takes place. Let $C[i]$ be the $i$-th column of the check sum and $state[i]$ be the $i$-th column of the state. Then the check sum is updated as $C[i] = C[i] \oplus state[0] \boxplus C[i+1]$. Whenever a `blank round` takes place no update of the check sum is performed.

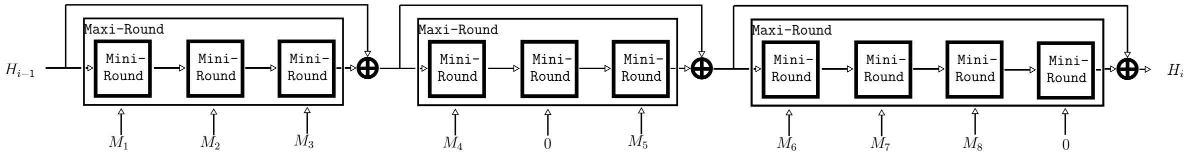


Figure 2.4: The compression function of Twister-384 and Twister-512

## 2.2 The Output Function

The `Output-Round` of Twister contains a global feed forwards as well as some `Mini-Rounds` depending on the size of the hash output. First a `Mini-Round` is applied on the state $H_{i-1}$, then the resulting state is XOR'ed with $H_{i-1}$ another `Mini-Rounds` is applied which gives the state $H_i$. Let $H_f$ be the final state after the last compression function call. A 64-bit output stream $out_i$ is then obtained by XORing the first column of $H_i$ with the firs column of $H_f$. This procedure takes place until the needed amount of output bits are obtained. The last output stream can be varied between 32 bits and 64 bits by taking only the first half of $out_i$. This allows to vary the output size for a huge amount of applications. Figure 2.5 gives a high level description of a `Output-Round`.

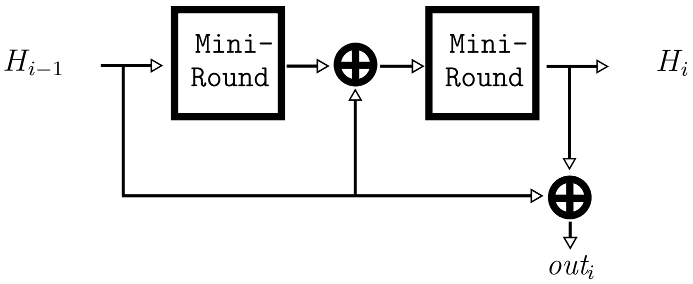


Figure 2.5: `An Output-Round`

# 3 Design Principles of Twister

In this section we exlain our design purpose and describe why Twister was designed as it is.

## 3.1 Security

The security of Twister is based on well studied concepts known from the AES [16]. The concept of MDS allows us to obtain a maximum of diffusion inside each column of the state matrix. Since the message input is orthogonal with the diffusion of the state, we allow a minimum of control on the state for an attacker. Introducing local feedforward as well as `blank rounds` (rounds with no message input) furthermore reduce the influence of an attacker on the state.

## 3.2 Evolutionary

During the last decade many hash function which uses a lot of different concepts. Even in some cases it turns out that hash functions which seem to be weak due to a fast break are even stronger than hash functions which were not broken so fast. But then it turns out that using newly developed techniques lead to stronger attacks which make pretended strong hash functions weak.

The Twister design is in some way evolutionary since we have learned from the AES process in many ways. The well studied and analyzed block ciphers that were in the final round of the competition lead to some well established design principles offering a high level of cryptographic knowledge. Rijndael [14] therefore can be seen as one of the most studied block ciphers during this process and also in the time after. Its concepts of simple byte-wise operations SubBytes, ShiftRows and MixColumns are well analyzed and it turns out that their combination can offer a high level of speed and security. We adopt some of these concepts for Twister and we also learn from resent hash function breakouts.

## 3.3 Simplicity

A strong hash function should not be hard to analyze, since if one cannot find an attack due to the algorithmic complexity does not mean that there is not simple attack which breaks the whole function in an easy way. We therefore only use very simple component which form our `Mini-Round`. These components are well studied and well known but combining these components to obtain a very good hash function is new. Our very simple design and clear structure of Twister makes cryptanalysis easy and serves the purpose that there are no simple attacks which cannot be found due to a complex and unreadable algorithm.

## 3.4 Portability and Scalability

A many design criteria of Twister is its use to a huge range of applications. Due to its byte-wise operations it scales perfect on 8, 16, 32 and 64-bit platforms. Twister can be applied on smart cards with small 8-bit processors very efficiently. We offer an optimized version for 32-bit and 64-bit environments. The portability will be enhanced by its low memory requirements, which makes Twister even for low end platforms valuable. For 32 and 64-bit multi-core we offer a high speed parallel mode of operation which scales to reach the optimal level of processor usage.

## 3.5 Analyzability

Twister uses well known and well analyzed components inside its design. The security level of Twister can be proven for the inner components which is more worth than just a security claim. Using the concept of MDS lead to a very fast diffusing after just two input blocks. This high level of diffusion makes Twister very close to a randomized hash function which offers a hugh level of security by being fast as well.

## 3.6 Speed

Twister has not only a very high security level and is very fast as well. Compared with members of the SHA-2 family Twister is at least as fast on 32 and 64-bit platforms. Our developed modes of operation leads to very fast hashes of hugh messages in a multi-core environment. This makes a parallel implementation of Twister very fast and efficient. But, even short messages can be hash very fast with only few overhead.

# 4 Specification: The Twister Hash Function Family

In this section we present our hash function. We start off with a description of the general design strategy. The design is based on a blockcipher that is iterated using the Davies-Meyer (DM) mode of operation [9]. Twister is a byte-oriented framework that operates on a square state matrix. The building block of the blockcipher is called `Mini-Round`. It takes a sub portion of the message and processes it into the state $\mathcal{S}$ whereas $\mathcal{S}$ is a $N \times N$ byte-matrix, $N \in \mathbb{N}$. After two `Mini-Rounds`, the state is guaranteed to have full diffusion. Also, two subsequent iterations of the `Mini-Round` is can be proved to be collision free. See Section 5 for a detailed discussion on our security issues.

After processing the padded message (i.e. the message is completely absorbed by the state $\mathcal{S}$), the output follows. This technique follows the design ideas of the sponge function [5] by not presenting the complete internal state to the attacker at once but slice by slice.

The following notations are used in the following:

| | |
|---|---|
| $\mathcal{S} = (S_{i,j})_{1 \leq i,j \leq N}$ | internal state matrix |
| $\mathcal{C} = (C_{i,j})_{1 \leq i,j \leq N}$ | internal checksum matrix |
| $N$ | number of rows and columns of the internal state matrix |
| $msg_{size}$ | size of unpadded message (in bits) |
| $R_{total}$ | number of total `Mini-Rounds` per compression function |
| $R_{msg}$ | number of N-byte blocks processed per compression function |
| $m$ | size of the padded message, measured in $N \cdot R_{msg}$ -byte blocks, i.e. the number of compression function calls needed to absorb the message into the state $\mathcal{S}$ |
| $M = (M_1, \ldots, M_m)$ | padded message to be handled by the Twister hash function |
| $M_{unpad}$ | unpadded message |
| $out$ | size of the hash value measured in $N$-byte blocks, i.e. $out = n/(8 \cdot N)$ |
| $n$ | size of the hash value in bits, i.e. $n = out \cdot 8 \cdot N$, where $n \leq (8 \cdot N)^2$ |
| $H = (H_1, \ldots, H_{out})$ | number of $N$-byte blocks of the hash output |
| $\phi$ | TwistCounter |

## 4.1 Twister Components

This section describes in detail the Twister components.

### 4.1.1 State $S$

Twister operats on a square state matrix $\mathcal{S} = (S_{i,j})$, $1 \leq i, j \leq N$, consisting out of $N$ rows and columns, where each cell $S_{i,j}$ represents one byte.

| | | | |
|---|---|---|---|
| $S_{1,1}$ | $S_{1,2}$ | $\cdots$ | $S_{1,N}$ |
| $S_{2,1}$ | $S_{2,2}$ | $\cdots$ | $S_{2,N}$ |
| $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $S_{N,1}$ | $S_{N,2}$ | $\cdots$ | $S_{N,N}$ |

Notation: $S_{(i\,\rightarrow)} := (S_{i,1}, \ldots, S_{i,N})$ denotes the $i$-th row vector and $S_{(j\,\downarrow)} := (S_{1,j}, \ldots, S_{N,j})$ the $j$-th column vector.

### 4.1.2 Checksum $C$

The checksum enlarges the state of Twister-384 and Twister-512 to stick to our wide-pipe design [29] decision.

The checksum is as the srtate $S$ a square checksum matrix $\mathcal{C} = (C_{i,j})$, $1 \leq i, j \leq N$, consisting out of $N$ rows and columns, where each cell $C_{i,j}$ represents one byte.

| | | | |
|---|---|---|---|
| $C_{1,1}$ | $C_{1,2}$ | $\cdots$ | $C_{1,N}$ |
| $C_{2,1}$ | $C_{2,2}$ | $\cdots$ | $C_{2,N}$ |
| $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $C_{N,1}$ | $C_{N,2}$ | $\cdots$ | $C_{N,N}$ |

### 4.1.3 TwistCounter $\phi$

The TwistCounter $\phi$ is a unsigned 64 bit integer that is added and decreased within a `Mini-Round` to prevend slide attacks.

### 4.1.4 Processed message counter `hs_ProcessedMsgCounter`

Processed message length `hs_ProcessedMsgCounter` is a unsigned long long integer. The purpose of the counter is to compute the length of a hashed message.

### 4.1.5 Unprocessed message `hs_Data`

Unprocessed message bytes are stored inside a unsigned char array `hs_Data`. This array are filled up with data when the size of a message chunk is not divisible by 512.

### 4.1.6 Unprocessed message length `hs_UnprocessedMsgCounter`

The number of unprocessed message bits are stored inside a integer `UnprocessedMsgCounter`.

## 4.2 The Compression Function

The Twister-hash function calls the compression function using the DM mode of operation. After the message is absorbed in the internal state the output function is called for every $N$ bytes of output.

The compression function of Twister consists of building blocks called `Mini-Rounds` which are grouped into `Maxi-Rounds`. Each `Mini-Round` is a combination of well studied primitives, which are easy to analyze and fast to implement in software and hardware. The instances of the Twister hash function family differ only in their construction of a `Maxi-Round`.

The compression function takes a 512-bit block and processes them into the internal state matrix $\mathcal{S}$. As a `Maxi-Round` only indicates the position of the local feed forward XOR-operation we will normally only discuss a compression function as a set of `Mini-Rounds`. The local feed-forward operation is an optional security feature and is discussed in Section 5. More general, the compression function works as follows. Let $R$ be the number of `Mini-Rounds` in a compression function. (Note: In Figure 4.1 we have $R = 9$.)

#### Twister-224 and Twister-256

The Twister-224 and Twister-256 compression function consists of three `Maxi-Rounds`. Each `Maxi-Rounds` is followd by a feed-forward XOR-operation. The first and second `Maxi-Round` consist of three `Mini-Rounds`. The last `Maxi-Round` consists of two `Mini-Rounds` and one `blank round`. Figure 4.1 illustrats the compression function.

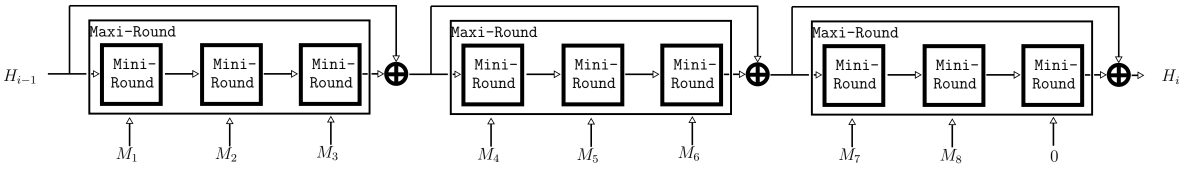


Figure 4.1: The compression function of Twister-224 and Twister-256

The Twister-384 and Twister-512 compression function consists of three Maxi-Rounds, too. The first Maxi-Rounds consists of three Mini-Rounds. The second Maxi-Rounds consists of a Mini-Round followed by a blank round and an other Mini-Round. The last Maxi-Rounds consists of three Mini-Rounds followed by a blank round. Figure 4.2 illustrats the compression function.

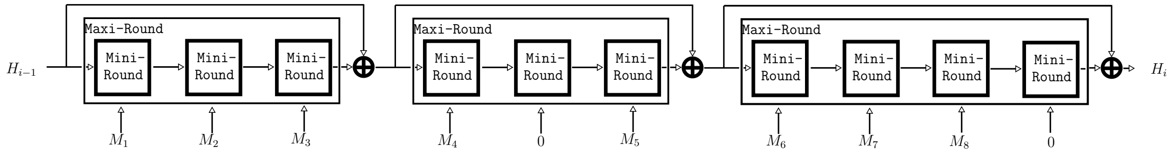


Figure 4.2: The compression function of Twister-384 and Twister-512

## 4.2.1 Mini-Round

The Mini-Round is the underlying building-block of any Twister hash function. It's main purpose is to inject the message (Message-Injection) and to take care for the diffusion of the state matrix $\mathcal{S}$. It is visualized in Figure 4.3. Twister can handle at most $2^{64}$ Mini-Rounds. This limitation causes by the Add TwistCounter operation where a 64 counter is added. Each Mini-Round can process 64 bit of message data. Therefore, with a native usage of a Mini-Round it is possible to process up to $2^{64} \cdot 64$ message bits. If this limitation became in the future a real world issue it is possible to increase the size of the TwistCounter to 128 bit with almost no performance loss.

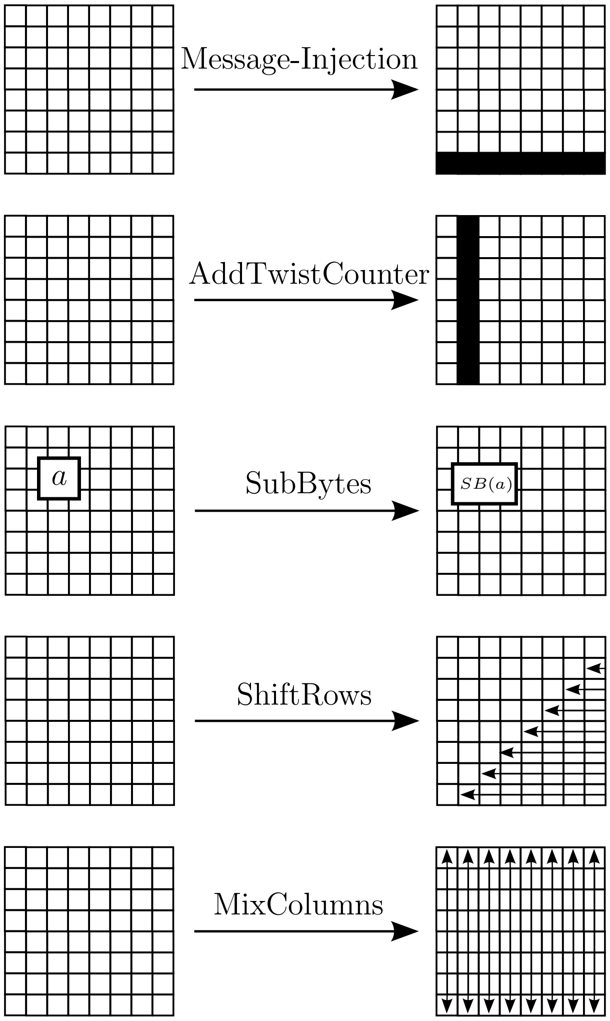


Figure 4.3: A `Mini-Round`

### Message injection

A 64 bit message block $m$ is inserted (via XOR $\oplus$) into the last row. By using the notation $m = (m[1], \ldots, m[N])$ whereas the length of $m[N]$ is one byte, and

$$S_{(\to j)} \oplus m := (S_{1,j} \oplus m[1], \ldots, S_{N,j} \oplus m[N])$$

we define the message injection process by

$$S_{(\to 1)} = S_{(\to N)} \oplus m.$$

### Add TwistCounter

The `TwistCounter` $\phi$ is a unsigned 64 bit integer. The initial state is the maximum value (`0xFFFFFFFFFFFFFFFF`). By using the notation $\phi = (\phi[1], \ldots, \phi[N])$ whereas the length of $\phi[N]$ is one byte, and $\phi[1]$ is the most significant byte of $\phi$. The counter is added byte by byte - via the XOR operation - into the second column of the state $\mathcal{S}$.

$$S_{2,\downarrow} \oplus \phi := S_{2,1} \oplus \phi[1], \dots, S_{2,N} \oplus \phi[N])$$

We define the TwistCounter addition by

$$S = S_{2,\downarrow} \oplus \phi$$

After the additon $\phi$ is decreased by one.

### SubBytes

The function is defined as an bijection

$$SubBytes : \{0,1\}^8 \longrightarrow \{0,1\}^8$$

and is used as an S-box for each byte. It should, among other properties, be highly non-linear. A discussion on how to obtain such cryptographically strong S-Boxes (for 8x8 S-Boxes) can be found in [45]. TWISTER uses the well known and studied AES S-Box. The S-Box can be found in the Appendix B.

We define the `SubBytes` operation by

$$S_{i,j} = SB(S_{i,j}) \quad \forall i, j.$$

### ShiftRows

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

We define the `ShiftRows` operation by

$$S_{(i,j-1)} := S_{(i,j)} \quad \forall i, j.$$

### MixColumns

The MixColumn step is a permutation operation on the state. It applies a $N \times N$-MDS $A$ (a maximum distance separable matrix as defined below) to each column, i.e. performs the operation $A \cdot S_{(j\downarrow)}$.

**Definition** An [n,k,d] code with generator matrix

$$G = [I_{k\times k}\ A_{k\times(n-k)}]$$

is an MDS code if every square submatrix of $A$ is nonsingular. The matrix $A$ is called a MDS-matrix.

Our chosen MDS matrix is cyclic, i.e., its i-th row can be obtained by a cyclic right rotation of (02 01 01 05 07 08 06 01) by $i$ entries. It has a branch number of 9 meaning that if two 8 byte input vectors differ in $1 \leq k \leq 8$ bytes, the output of MixColumns differs in at least $9 - k$ bytes. The $8 \times 8$-MDS matrix used for all proposed instances of Twister is:

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

All of the byte entries are considered to be elements of $\mathbb{F}_{2^8}$. An element $\sum_{i=0}^{7} a_i x^i \in \mathbb{F}_{2^8}$ is represented by $\sum_{i=0}^{7} a_i 2^i$. The reduction polynomial $m(x)$ of $\mathbb{F}_{2^8}$ is defined as

$$m(x) = x^8 + x^6 + x^3 + x^2 + 1. \tag{4.1}$$

Properties of MDS matrices/codes can be found e.g. in [30]. A discussion on how to obtain suitable MDS matrices can be found in Appendix A.

### 4.2.2 Maxi-Round

A `Maxi-Round` contains of several `Mini-Rounds`, `blank round` and a optional checksum updates. We defined a checksum update operation as

$$C_{(i,\downarrow)} = C_{(i,\downarrow)} \oplus C_{(i+1,\downarrow)} \boxplus S_{(i,\downarrow)}$$

`Maxi-Rounds` use also a feed forward operation where the state before a `Maxi-Round` $S^k$ is feed forwarded with the state after a `Maxi-Round` $S^{k+1}$. We defined the feed forward operation as

$$S_{(i,j)} := S^k_{(i,j)} \oplus S^{k+1}_{(i,j)} \quad \forall i, j.$$

Figure 4.4 illustrates a `Maxi-Round`.

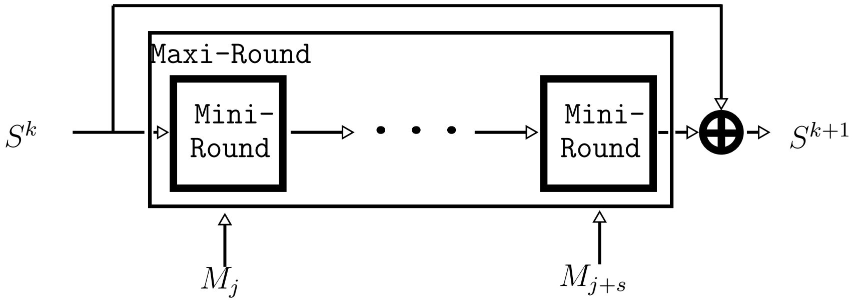


Figure 4.4: A `Maxi-Round`

## 4.3 Postprocessing

This section describes the Twister finalization process. It starts when the message is completely over hand to the

Postprocessing shall take place after the message processed by the compression funtion. This postprocessing consists of three steps:

1. Padding the message $M$
2. Twister state finalization
3. Message digest computation

### 4.3.1 Padding

The message, M, shall be padded before hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 512. Suppose that the length of the message, $M$ is $l$ bits then the message can divide in $n = l/512$ 512 bit message blocks $m_1, \dots, m_n$ and one final message block $m_f$ betweeen 0 and 511 bits. The 512 message blocks $m_1, \dots, m_n$ are processed by the compression function. The final message block $m_f$ is padded by append a '1' to the end followed by $k = 511 - (l \bmod 512)$ zero bits. Afterwards the bitlength of $m_f$ is 512 and it can processed by the compression function like all other message blocks bevor.

### 4.3.2 Twister state finalization

After the padding procedure the message length that is stored in a 64 unsigned integer is injected like a message by XOR it byte by byte to the last row of the state $S$. Afterwards the state is updated via a `Mini-Round`. the following steps depends on the hash value size. The finalization ends with a `blank round` for Twister-224 and Twister-256 or a Twister-256 compression function with the checksum state $C$ as input for Twister-384 and Twister-512. The checksum are transformed in a 64 byte message block $m$ where $m = m[1], \dots, m[n]$ column by column.

$$m[i] = C_{(\lceil i/8 \rceil, i \pmod 9))} \forall i = 1, \dots, 64$$

### 4.3.3 Message digest computation

The `Output-Round` computs the message digest. It contains a global feed forwards as well as some `Mini-Rounds` depending on the size of the hash output. For every 64 message digest bits `Mini-Round` is applied on the state $S$, then the resulting state is XOR'ed with $S^{k-1}$ another `Mini-Rounds` is applied which gives the state $S^k$. Let $S^f$ be the final state after the last compression function call. A 64-bit output stream $out_i$ is then obtained by XORing the first column of $S^k$ with the first column of $S^{k-1}$. This procedure takes place until the needed amount of message digest bits are obtained. The last output stream can be varied between 32 bits and 64 bits by taking only the first half of $out_i$. This allows to vary the output size for a huge amount of applications. Figure 4.5 illustrates the `Output-Round`.

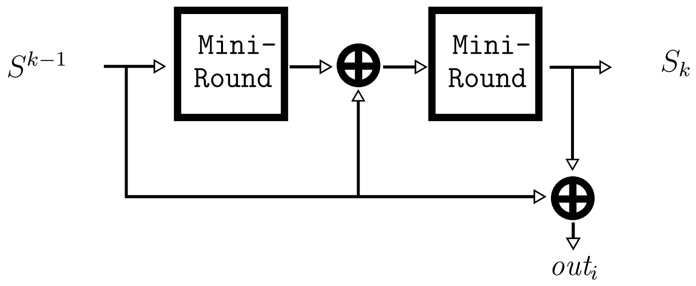


Figure 4.5: `An Output-Round`

## 4.4 The `Init()` – Function

The `Init()` function of TWISTER performs the the following steps:

1. Set the state to zero.
2. Copy the most significant byte of the output length to $S_{7,1}$ and the least significant byte to $S_{7,2}$. The output length is specified in bits and the maximum supported output length is 512 bit. Hence, the output length can be stored in two bytes.
3. The twist counter $\phi$ is initialized with $2^{64} - 1$.
4. Set the checksum for TWISTER-384 and TWISTER-512 to zero.
5. Set the unprocessed message counter `hs_Databitlen` to zero.
6. Set the unprocessed message data `hs_Data` to zero.

## 4.5 The `Update()` – Function

The `Update()` function of TWISTER process data using the TWISTER compression function. It acts as follows:

1. Concatenate the unprocessed message `hs_Data` with the message $M$ to the unprocessed message $M'$.
2. Divide $M'$ in $k$ 512 bit message blocks $m_1, ...m_k$ and a final message block $m_f$ between 0 and 511 bits.
3. Call for every $m_i$ for $i = 0, ..., k$ the TWISTER compression function.

4. Add $512 \cdot k$ to the processed message counter `hs_ProcessedMsgCounter`.
5. Copy the value of $m_f$ to `hs_Data`.
6. Set the unprocessed message counter `hs_Databitlen` to the bitlen of $m_f$.

## 4.6 The `Final()` – Function

The `Finalize()` function of Twister provides for the length padding of the message, processes all unprocessed data, hashes the message length, (process the checksum) and finally outputs the hash value. It acts as follows:

1. Concatenate a '1' bit to the unprocessed message data `hs_Data`.
2. Fill `hs_Data` up with '0' bits until the bit length is 512.
3. Call the compression function for `hs_Data`.
4. Add the unprocessed message counter `hs_Databitlen` to the processed message counter `hs_ProcessedMsgCounter`.
5. XOR `hs_ProcessedMsgCounter` to the last row of the state $S_{(N,\rightarrow)}$.
6. Perform a `Mini-Round`.
7. Perform a `blank round` if the output hash length `hs_Hashbitlen` is less equal 256 bits.
8. Perform a Twister-256 compression function on the checksum state $C$ when `hs_Hashbitlen` is between 256 and 512 bits.
9. Perform four times the `Mini-Round` for Twister-224 and Twister-256, six times for Twister-384 and eight times for Twister-512 to compute the message digest.

## 4.7 Randomized Hashing

Twister supports randomized hashing in the following way. One can chose a so called salt value of size at most 64 bits. If the salt is zero then the usual hash computation will be performed. For salt values not equal to zero and smaller than 64-bit as many zero's will be padded such that a 64-bit block occurs. The salt $s$ will be introduced in the hash computation as a first input block followed by three `blank rounds`. This will prevent Twister to any chosen salt attacks.

One can try to mount the following attack scenario if the salt is inserted as the first input block without some `blank rounds` at the end. It might be more possible to transform a chosen-IV attack into a real attack if an attacker chooses salt values $s$ and $s' \neq s$ for the first two input blocks $M_0$ and $M_0' \neq M_0$ such that

$$\texttt{Mini-Round}(\texttt{Mini-Round}(IV, s), M_0) = \texttt{Mini-Round}(\texttt{Mini-Round}(IV, s'), M_0')$$

Using at least two `Mini-Rounds` will lead to full diffusion 5.2.2 and will reduces an attacker's influence to generate different IV's. Therefore our randomized hashing schema is secure for two reasons. First, a salt value will not be introduced into the compression function, which leads not additional degrees of freedom for an attacker. Second, the degrees of freedom to choose the IV before the first input block are very low which gives no additional power to an attacker and does not lead to further attacks on a chosen salt.

# 5 Security

In this section we analyze the security of TWISTER and show that it is resistant to all known generic attacks.

## 5.1 Generic Attacks

**Length-Extension Attacks.** Given a hash function $H$ based on a MERKLE-DAMGÅRD construction. If one can find a collision for two messages $M, M'$ with $M \neq M'$, such that $H(M)$ and $H(M')$ collide, then one can apply a length extension attack. For any message $N$ one can easily produce a collision for $M||N$ and $M'||N$ as $H(M||N) = H(M'||N)$. Our padding rule avoids such type of attacks since we concatenate the length of the message to the message itself. Another attack can be as follows. For a known hash value $H(M)$ one can compute the hash value $H(M||X||N)$ for any suffix $N$, if the length of an unknown message $M$ is known as well as the padding $X$ of $M$. We prevent TWISTER to this kind of attack by two countermeasures: (i) By knowing only the hash value an attacker can not easily determine the state $\mathcal{S}$ after the last compression function call as he has only access to the result of the `final()` function. This function squeezes out some bits of the state applying output transformation and squeezes out some bits again. The bits of a squeezing process do not leave enough information to recover the internal state. (ii) The multiple feed-forward does also prevent any attacker to successfully gain any knowledge of prior state information. In each squeezing process the one feed forward takes place.

**Multi-Collision Attacks.** Joux [24] found that when iterative hash functions are used, finding a set of $2^k$ messages all colliding on the same hash value (a $2^k$-multi-collision) is as easy as finding $k$ single collision for the hash function. Finding a collision in the compression function, i.e., a single block collision one can find $k$ of such collisions each starting from the chaining value produced by the previous one-block collision. In other words, one have to find two messages blocks $M_i$ and $M'_i \neq M_i$ with $C(h_{i-1}, M_i) = C(h_{i-1}, M'_i)$, where $C(\cdot)$ represents the compression function and $h_i$ the chaining value. Then it is possible to construct $2^k$ messages with the same hash value by choosing for block $i$ either the message block $M_i$ or $M'_i$. Joux also showed that the concatenation of two different hash functions is not more secure against collision attacks than the strongest one. This attack can find $2^k$-way internal multi-collisions with a complexity of $k \cdot 2^{n_c/2}$. An instance of TWISTER fully resists the multi-collision attack if $8 \cdot N^2 \geq 2n_c$, since the complexity is determined by $k \cdot 2^{(8 \cdot N^2)/2}$. All instances of TWISTER have this feature, although the state of TWISTER-384 and TWISTER-512 is not big enough to prevent this attack alone, including the check sum can be viewed as an enlarging of the state, which then guarantees to prevent for this attack.

**Herding Attacks.** The herding attack [25] works as follows. An attacker takes $2^k$ chaining values which are fixed or randomly chosen. Then he chooses $O(2^{n_c/2-k/2})$ message blocks. He computes the output of the compression function for each chaining value and each block. It is expected that for each chaining value there exists another chaining value, such that both collide to the same value. The attacker stores the message block that leads to such a collision in a table and repeats this process again with the newly found chaining values. Once the attacker has only one chaining value, it is used to compute the hash value to be published. To find a message whose chaining value is among the $2^k$ original values, the attacker has to perform $O(2^{n_c-k})$ operations. For such a message the attacker can retrieve from the stored messages the message blocks that would lead to the desired hash value. The time complexity of this attack is about $O(2^{n_c/2+k/2})$ operations for the first step and $O(2^{n_c-k})$ operations for the second step. The whole attack on an $n$-bit hash function requires approximately $2^{(2n_c-5)/3}$ work. For TWISTER we have $n_c = 8 \cdot N^2$ and with $8 \cdot N^2 \geq (3n_c + 5)/2$ the attack has the same complexity as for for a (second) pre-image attack on a random oracle. The complexity of this attack decreases with increased size of the message. If the message is of size about $2^t$, then the complexity of the attack is $2^{(2n_c-5)/3-t}$. One has to choose $N$ such that the hash function is protected against this kind of attack for a given upper bound. All of our proposed instances of TWISTER resist this kind of attack.

**Long 2nd pre-image Attacks.** Dean [18] found that fix-points in the compression function can be used for a second-pre-image attack against long messages in time $O(n_c \cdot 2^{n_c/2})$ and memory $O(n_c \cdot 2^{n_c/2})$, where $n_c = 8 \cdot N^2$ is the size of the internal state (which is equal to the size of the hash output for a plain MERKLE-DAMGÅRD constructions). Kelsey and Schneier [26] extend this result and provide an attack to find a 2nd pre-image on a MERKLE-DAMGÅRD construction with MERKLE-DAMGÅRD strengthening much faster than the expected workload of $2^{n_c}$. The complexity of the attack is determined by the complexity of finding expandable messages. These are messages of varying sizes such that all these messages collide internally for a given IV. Expandable message can either be found using internal collisions or fixed points between a one-block message and an $\alpha$-block message for varying values of $\alpha$. The complexity of the generic attack to find a 2nd pre-image for a $2^k$-message block is about $k \cdot 2^{n_c/2+1} + 2^{n_c-k+1}$ compression function calls.

Long 2nd pre-image attacks in this form cannot be applied to the TWISTER framework for three reasons. First, we include the twist counter which does not allow to find expandable messages. Second, we make use of multiple feed-forwards and, third, the internal chaining value is in general much larger than $n$. This make it harder to find collisions and fix points since we essentially have a constructions similar to the wide-pipe design [29].

Andreeva et al. [1] have shown that a combination of the attacks from [18, 26, 25] can be mounted to dithered hash functions, which gives the attacker more control on the second-pre-image, since he can choose about the first half of the message in an arbitrary way. This attack can be done in time $2^{n_c/2+k/2+2} + 2^{n_c-k}$. Although it is more expensive than the attack of Kelsey and Schneier [26], it works even when an additional input to the compression function (dithering) is given. One have to make the dithering as huge as possible, such that there are no small cycles. Twister includes the twist counter $\phi$ which is very large, i.e., the twist counter is of size of the maximal message length. The larger this counter is the longer cycles we have, which increases the protection against this type of attack.

**Slide Attacks.** Slide attacks are common in block cipher cryptanalysis, but they also applicable to hash functions. Given a hash function $H$ and two messages $M$ and $M'$ where $M$ is a prefix of $M'$, one can find a slid pair of messages $(M, M')$ such that the the last message input block of the longer message $M'$ performs only an additional blank round, e.g. for sponge constructions. These two messages are then slid by one blank round. This attack allows to recover the internal state of a slid pair of messages an even backward computation as shown in [22]. The twist counter $\phi$ avoids the possibility of slide attacks, since XOR-ing a different value in each `Mini-Round` into the state matrix does not allow to find slid pairs of messages. Furthermore, the last inserted message block cannot be the all zero block due to the padding rules. Thus slide attacks are unavailable for Twister.

**Differential Attacks.** The essential idea of differntial attack on hash functions [11], as used to break MD5 and SHA-0/1, is to exploit a high-probability input/output differential over some component of the hash function, e.g. under the form of a "perturb-and-correct" strategy for the latter functions, exploiting high probability linear/non-linear characteristics. In the design of the Twister framework, we applied the following countermeasures against differential attacks:

- *High-Speed-Diffusion.* The strong diffusion capabilities of the `Mini-Round` in combination with the non-linear S-Box make the exploit of linear approximations highly implausible. As it is impossible to find a collision after one `Mini-Round`, any attacker has to trail presumably very long paths to be able to find a collision.
- *Nested feed-forward.* The internal feed-forward operations aim at strengthening the function against differential paths.
- *Optional internal wide-pipe.* this makes internal collision unlikely, and the output-rounds make the differences much harder to predict in the hash value.
- Using different operators (e.g. $\boxplus$ and $\oplus$) highly complicates the computation of good differential paths.

## 5.2 On Collision Resistance

Twister provides maximal diffusion after two rounds due to the chosen MDS matrix for MixColumns. If two input blocks differ in only one byte, then MC generates a difference in all message bytes of the column where the difference is inserted. The SR operation of the next round spreads the differences such that all columns contain at least one byte difference. Twister also prevent that the following inputs cancel these differences out, since they where inserted in different positions in each round.

Due to its very fast diffusion process (after two `Mini-Rounds`), near-collision are practically useless at the end of any compression function. To prevent collision attacks on the internal state in Twister-512 we choose to employ an extra `Mini-Round` after every "message-insertion" as we have chose to not have the same security margin as with Twister-256. As one can easily see, after a single `Mini-Round` one is not able to find a collision, i.e. any collision has to involve more that one `Mini-Round`. We claim a collision resistance of $\mathcal{O}(2^{out \cdot N \cdot 8})$ if $out \leq N$ (as in Twister-256 and Twister-512).

### 5.2.1 Properties of Twister

We have designed Twister such that a single `Mini-Round` is proven to be collision free. This is expressed by the following lemma.

**Lemma 5.2.1** *From any state $h_{i-1}$ one cannot find input message blocks $M$, $M' \neq M$ such that*

$$\texttt{Mini-Round}(h_{i-1}, M) = \texttt{Mini-Round}(h_{i-1}, M')$$

*for all $M$, $M' \neq M$.*

**Proof** Assume that $h_i^M$ is the state after inserting the message block $M$ and $h_i^{M'}$ is the state after inserting $M'$. Then, if $M$ and $M'$ are different in byte $j$ the states $h_i^M$ and $h_i^{M'}$ are different in column $j$ in at least $9 - k$ bytes. This is due to the MDS property of our diffusion layer, which has a brach number of 9.

■

We can also show that Twister offers full diffusion after two input blocks.

**Lemma 5.2.2** *Given an internal state $h_{i-1}$ and two input blocks $M_1$ and $M_1'$ where $M_1 \neq M_1'$. Then, we have full diffusion of the state after two `Mini-Rounds`.*

**Proof** The message distribution process is visualized in Figure 5.1. Two messages $M_1$ and $M_1' \neq M_1$ are different in at least one byte. Due to the diffusion of MixColumns at

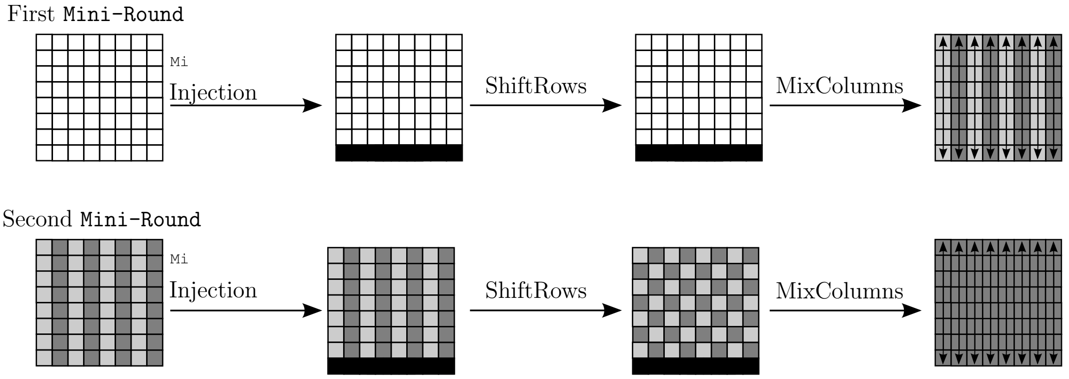


Figure 5.1: Visualization of the diffusion of a message after two `Mini-Rounds`

least one state column differs in 8 bytes. The ShiftRows of the following `Mini-Round` with no message input will distribute the all difference column into a one byte difference in each state column. After that MixColumns generates a difference in all state bytes. Which leaves to full diffusion.

■

### 5.2.2 A Collision Attack Method on Twister

The first step in finding a collision for any instance of the Twister hash family is to analyze a `Mini-Round`. We known from Lemma 5.2.1 that one cannot find a collision after one input block and we know from Lemma 5.2.2 that we have full diffusion of the state after two input blocks. Therefore we claim that an internal collision needs at least three applications of a `Mini-Round`, i.e., three input blocks. This can be verified by regarding the differential properties of MixColumns that are shown in Table 5.1.

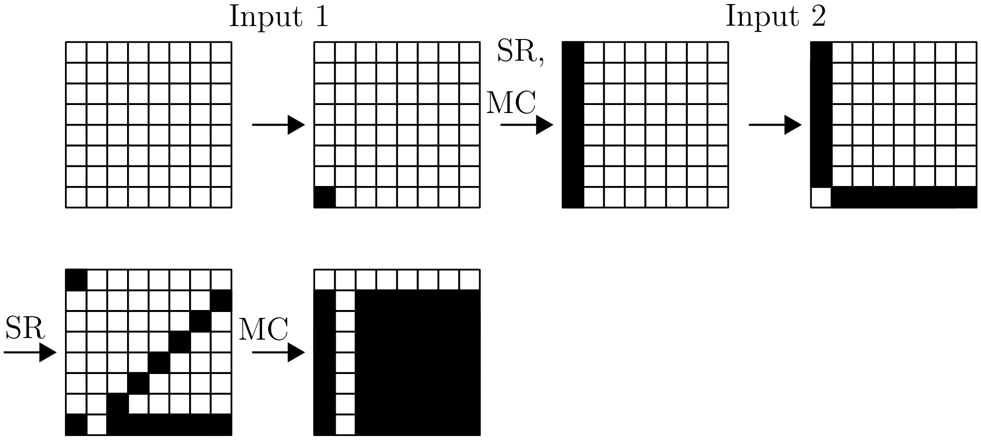
| $D_I$ \ $D_O$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |
| 1 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | 0 |
| 2 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -8 | 0 |
| 3 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -16 | -8 | 0 |
| 4 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -24 | -16 | -8 | 0 |
| 5 | $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ | -32 | -24 | -16 | -8 | 0 |
| 6 | $-\infty$ | $-\infty$ | $-\infty$ | -40 | -32 | -24 | -16 | -8 | 0 |
| 7 | $-\infty$ | $-\infty$ | -48 | -40 | -32 | -24 | -16 | -8 | 0 |
| 8 | $-\infty$ | -56 | -48 | -40 | -32 | -24 | -16 | -8 | 0 |

Table 5.1: Column Properties of the state matrix after multiplication with an MDS matrix. Approximate probability that two 8-byte input words with $D_I$ different bytes on predefined positions maps to two 8-byte output words with $D_O$ different bytes on predefined positions by the *MixColumns* operation. The values are base-2 logarithms.

We find the following lemma:

**Lemma 5.2.3** *Starting from an all zero internal state and inserting a one byte difference in the first input block, then at least* 49 *bytes are different after the second input round.*

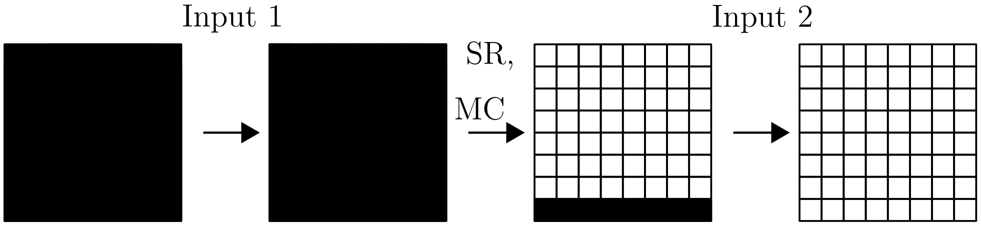
**Proof** A one byte input difference will cause an difference in a whole state column due to the MDS property of MixColumns. The next input difference can be chosen such that the last byte difference in the all difference state column will be canceled out. Thus, the state contains seven columns having 2 non-zero bytes and one all zero byte column after the following ShiftRows. Thus, at least seven bytes are non-zero when applying MixColumns in an column with two non-zero byte differences. Finally, the complete state matrix contains at least 49 non-zero bytes. A graphical representation is shown in Figure 5.2.2.



■

**Lemma 5.2.4** *i) Starting from an all difference internal state one needs at least two input rounds to obtain an all zero internal state. This occurs for random messages with probability* $2^{-512}$. *ii) Any differential starting from an all zero internal state and ending in an all zero internal state needs at least three input rounds. iii) Such a three round differential occurs with probability* $2^{-512}$.

**Proof** i) By starting from an all difference internal state and inserting a any difference which does not cancel any state byte difference out we apply a first `Mini-Round`. Due to Table 5.1 MixColumns generates a non-zero difference in the last byte of a column and a zero difference in the remaining bytes with probability $2^{-56}$. This occurs for eight columns with probability $2^{-448}$. The input block in of the second round will cancel the difference in the first row out. This happens for a randomly chosen input block with



probability $2^{-64}$. Thus, an all zero internal state occurs after at least two input rounds with probability $2^{-512}$.

ii) If we insert a non-zero difference in all bytes of the first input block, the internal state after the first MixColumns is all different. The argument of i) then can be applied. iii) follows from i) and ii).

■

These lemmas show that we can compute lower bounds on the differential probabilities which are needed to find a collision.

### 5.2.3 Pseudo Collisions on a `Mini-Round`

It definitely would be easy to construct a collision of a `Mini-Round` by the following scenario. Choosing two states $IV_1$ and $IV_2 \neq IV_1$ such that for a message block $M_i$

$$\texttt{Mini-Rounds}(IV_1, M_i) = \texttt{Mini-Rounds}(IV_2, M_i),$$

holds. Such an attack would never lead to a collision of a `Maxi-Round` due to the local feedforward that captures a `Maxi-Round`. Thus, although pseudo collisions can easily be found for a `Mini-Round` it thus not lead to a collision of a `Maxi-Round` and thus also not leave to a collision in the compression function of Twister.

## 5.3 On (2nd) Pre-image Resistance

The `Mini-Round`s are easily invertible as they are (after message injection) permutations. This is beneficial for collision-resistance but might lead to problems with (2nd) pre-image resistance. We care for this problem by applying feed-forward XORs with the internal state. So any attacker that tries to mount an pre-image attack has to recover the state from the hash value. For this, the attacker has to guess one bit for every hash output bit (this is due to the `Output`-function). Let $H = (h_1, \ldots, h_{out \cdot N})$ be the hash output for hashing a message $M$, $|h_i| = 1$ byte. Assume that an attacker tries to mount a (2nd) pre-image attack. In order to invert a `Mini-Round`, the attacker has to recover the entire internal state $\mathcal{S}$. As the attacker has only 1/N-th of the internal state $\mathcal{S}_{(1,\downarrow)} \oplus \mathcal{T}_{(N,\downarrow)}$ he has to guess $\mathcal{T}_{(N,\downarrow)}$ in order to recover one colum $\mathcal{S}_{(1,\downarrow)}$ of $\mathcal{S}$. Furthermore, he has to guess all of the other colums to be able to invert. The complexity for guessing essentially the whole matrix is $\mathcal{O}(2^{N \cdot 8})$ so if we choose the output size of $\cdot out \cdot N \leq N \cdot N$ smaller than the size of the internal state no attacker has any signigicant advantage in finding a (2nd)-pre-image for any given hash value. Pre-images as well as (2nd) pre-images can be found if an attacker can easily compute fix points in at least one `Maxi-Round`. Due to the feed forward a fix point in a `Mini-Round` will not lead to a fix point in a `Maxi-Round`.

# 6 Implementational Aspects and Performance

In this chapter we discuss issues related to the implementation of TWISTER on different platforms. In essence, our techniques for implementing TWISTER rely on the following key sources of information:

- Optimization techniques given by Daemen and Rijmen in [16].
- Bernstein and Schwabe presented at Indocrypt 2008 (preliminary version available at [3]) some new techniques on how to reduce the number of instructions for an AES implementation.

Several of the discussed issues are relevant to more than one platform.

## 6.1 Finite Field Multiplication

In the algorithm of TWISTER there are no multiplications of two variables in $GF(2^8)$, but only the multiplications of a variable with a constant. The latter is easier to implement than the former – especially in the context of hardware and high-speed software implementations.

## 6.2 $64$-Bit Platforms

All the different steps for the round transformation can be combined in a single set of look-up tables, allowing for very fast implementations on processors with word lengths of 64 bits (or greater). We will use the following notations (for $1 \leq x, y \leq 8$):

$a_{y,x}$ input state matrix element at position $(x, y)$,
$b_{y,x}$ state matrix element after SUBBYTES() at position $(x, y)$,
$c_{y,x}$ state matrix element after SHIFTROWS() at position $(x, y)$,
$d_{y,x}$ state matrix element after MIXCOLUMNS() at position $(x, y)$.

After the MixColumn-Operation, we have for $1 \leq j \leq 8$:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = \begin{bmatrix} 02 & 01 & 01 & 05 & 07 & 08 & 06 & 01 \\ 01 & 02 & 01 & 01 & 05 & 07 & 08 & 06 \\ 06 & 01 & 02 & 01 & 01 & 05 & 07 & 08 \\ 08 & 06 & 01 & 02 & 01 & 01 & 05 & 07 \\ 07 & 08 & 06 & 01 & 02 & 01 & 01 & 05 \\ 05 & 07 & 08 & 06 & 01 & 02 & 01 & 01 \\ 01 & 05 & 07 & 08 & 06 & 01 & 02 & 01 \\ 01 & 01 & 05 & 07 & 08 & 06 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j+1\,mod\,8}] \\ S[a_{2,j+2\,mod\,8}] \\ S[a_{3,j+3\,mod\,8}] \\ S[a_{4,j+4\,mod\,8}] \\ S[a_{5,j+5\,mod\,8}] \\ S[a_{6,j+6\,mod\,8}] \\ S[a_{7,j+7\,mod\,8}] \end{bmatrix}$$

where $S : \{0,1\}^8 \longrightarrow \{0,1\}^8$ denotes the S-Box operation.

The matrix multiplication can be interpreted as a linear combination of all eight column vectors:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \end{bmatrix} S[a_{0,j}] \oplus \begin{bmatrix} 01 \\ 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \end{bmatrix} S[a_{0,j}] \oplus \ldots \oplus \begin{bmatrix} 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \\ 02 \end{bmatrix} S[a_{0,j+7\,mod\,8}]$$

We define now eight $T$-tables: $T_0, T_1, \ldots, T_7$:

$$T_0[\alpha] = \begin{bmatrix} 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \end{bmatrix} S[\alpha], \quad T_1[\alpha] = \begin{bmatrix} 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \end{bmatrix} S[\alpha] \quad \ldots \quad T_7[\alpha] = \begin{bmatrix} 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \\ 02 \end{bmatrix} S[\alpha].$$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = T_0[a_0, j\,mod\,8] \oplus T_1[a_1, j+1\,mod\,8] \oplus T_1[a_1, j+2\,mod\,8] \oplus \ldots$$

$$\oplus T_0[a_0, j+7\,mod\,8].$$

All operations are 64-bit XOR operations that can be implemented quite efficiently on 64-bit platforms.

## 6.3 32-Bit Platforms

By splitting the 64-bit look-up tables $T_0, \ldots, T_7$ into 32-bit chunks we take presumably about twice the time for a MINI-ROUND as compared to the 64-bit implementation, i.e. the speed linearly scales down with the bandwidth available on a specific platform.

## 6.4 Specific Remarks for 8-Bit Platforms

The performance on 8-bit processors is an important issues, since most smart cards have such a processor and many cryptographic applications run on smart cards. There are several options for implementing TWISTER, depending on whether the requirements demand for minimum space (i.e. low possibility for storing lookup-tables) or maximum speed. If minimum space is requested, the multiplication of two elements in $GF(2^8)$ has to be performed in software and could not be stored as a lookup table. Specific details for such issues can be found in [16, Chapter 4.1.1]. If space limitations is not an issue, the technique for implementing TWISTER via lookup-tables should be chosen as was discussed in section 6.2. As nearly all of the operations scale linearly down (i.e. a 64-bit XOR can be easily implemented via 8 times an 8-bit XOR) we simply have the running time of 8 × running time on 64-bit.

## 6.5 Dedicated Hardware

TWISTER is suited to be implemented in dedicated hardware. There are several trade-offs between chip area and speed possible. Because the implementation in software on general-purpose processors is already very fast, the need for hardware implementation will very probably be limited to very specific cases like:

1. Extreme high-speed chips with no area restrictions: the $T$-tables can be hardwired and the XOR operations can be conducted in parallel.
2. Compact coprocessors on smart cards, there can either be only the S-Box hardwired or, additionally (and if enough memory is available) the $T$-tables generated at runtime.
3. If there is essentially no space to hardwire anything, even the S-Box can generated at runtime.

### 6.5.1 Decomposition of the S-Box

As we use the Rijndael S-Box, we can assemble it using two transformations:

$$S[\alpha] = f(g(\alpha)),$$

where $g(\alpha)$ is the transformation

$$\alpha \rightarrow \alpha^{-1} \text{ in } GF(2^8)$$

and $f(\alpha)$ is an affine transformation.

The problem of designing efficient circuits for inversion in finite field has been studien extensively before; e.g. by C. Paar and M. Rosner in [36] or, for a short summary, in [16].

## 6.6 Multiprocessor Platforms

There is considerable parallelism possible. If look-up tables are used, all table lookups can be done in parallel. The XOR operations can be done mostly in parallel as well. But the limiting factor for Twister implementations is likely to be the number of memory references that can be done per cycle, rendering the *inner* parallelism somewhat non-productive.

## 6.7 Performance Measurements

### 6.7.1 Performance Measurement Results on $64$-Bit Platforms

Twister was especially designed with 64-platforms in mind by making it possible to aggregate 8 times an 8-bit table lookup into one single 64-bit table lookup. The following performance measurements were conducted on:

Processor: Core2Duo $T7300$

Clock Speed: 2000 MHz

Memory: 2048 MB

Operating System: Linux, GNU Debian *Lenny*, Kernel 2.6.26-1 x64

Compiler: GCC 4.3, Optimization settings: -Os

For comparison, performance measurement results for SHA2 on the this platform are given.

**SHA-256**: total 20.1 clock cycles per byte

**SHA-512**: total 13.1 clock cycles per byte

For Twister we have the following performance results:

Twister-**224**
*Setup*: negligible
*Generation of Message Digest*: 15.8 cycles per byte

Twister-**256**
*Setup*: negligible
*Generation of Message Digest*: 15.8 cycles per byte

Twister-**384**
*Setup*: negligible
*Generation of Message Digest*: 17.5 cycles per byte

Twister-**512**
*Setup*: negligible
*Generation of Message Digest*: 17.5 cycles per byte

### 6.7.2 Performance Measurement Results on $32$-Bit Platforms

Processor: Core2Duo $T$7300

Clock Speed: 2000 MHz

Memory: 2048 MB

Operating System: Linux, GNU Debian *Lenny*, Kernel 2.6.26-1 x64

Compiler: GCC 4.1, Optimization settings: -Os

For comparison, performance measurement results for SHA2 on the this platform are given.

**SHA-256**: total 29.3 clock cycles per byte

**SHA-512**: total 55.2 clock cycles per byte

For Twister we have the following performance results:

Twister-**224**
*Setup*: negligible
*Generation of Message Digest*: 35.8 cycles per byte

Twister-**256**
*Setup*: negligible
*Generation of Message Digest*: 35.8 cycles per byte

Twister-**384**

*Setup*: negligible
*Generation of Message Digest*: 39.6 cycles per byte

Twister-**512**
*Setup*: negligible
*Generation of Message Digest*: 39.6 cycles per byte

### 6.7.3 Performance Estimates for 8-Bit Platforms

As Twister nearly linearly scales down on low-end platforms we expect (assuming the same technical background) that Twister has approximately the following performance on 8-bit platforms. We assuem that all the $T$-tables are precomputed and available on the platform.

Twister-**224**
*Setup*: negligible
*Generation of Message Digest*: 200 cycles per byte

Twister-**256**
*Setup*: negligible
*Generation of Message Digest*: 200 cycles per byte

Twister-**384**
*Setup*: negligible
*Generation of Message Digest*: 220 cycles per byte

Twister-**512**
*Setup*: negligible
*Generation of Message Digest*: 220 cycles per byte

This (downward) scalability is largely due to the fact that a 64-bit XOR operation can be implemented using eight 8-bit XOR operations.

# 7 Legal Disclaimer

To the best of our knowledge, the TWISTER hash function is not encumbered by any patents. We have not, and will not, apply for patents on anything in this document, and we are unaware of any other patents or patent filings that cover this work. The example source code – and all other code on the TWISTER website `www.twister-hash.com` – is in the public domain and can be freely used.

We make this submission to NIST's hash function competition solely as individuals. Our respective employers neither endorse nor condemn this submission.

# 8 About the Authors

The Twister team consists of young, dynamic and high motivated PhD students which are working all on topics of symmetric cryptography, especially on hash functions design and cryptanalysis. Our team consists of researcher that are experts in theory and also in practice.

# Bibliography

[1] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second Preimage Attacks on Dithered Hash Functions. In Smart [41], pages 270–288.

[2] Jean-Philippe Aumasson, Willi Meier, and Raphael C.-W. Phan. The Hash Function Family LAKE. In *FSE*, page inproceeding, 2008.

[3] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. Cryptology ePrint Archive, Report 2008/381, 2008. `http://eprint.iacr.org/`.

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara (August 24-25, 2006), 2006. See `http://radiogatun.noekeon.org/`.

[5] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions. Ecrypt Hash Workshop, 2007. See `http://gva.noekeon.org/papers/bdpv07.html`.

[6] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Franklin [21], pages 290–305.

[7] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In Cramer [13], pages 36–57.

[8] Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002.

[10] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[11] Christophe De Cannière and Christian Rechberger. Finding sha-1 characteristics: General results and applications. In Xuejia Lai and Kefei Chen, editors, *ASI-*

*ACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[12] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[13] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[14] J. Daemen and V. Rijmen. AES Proposal: Rijndael. NIST AES Homepage: `http://csrc.nist.gov/encryption/aes/round2/r2algs.htm`, 1999.

[15] Joan Daemen and Vincent Rijmen. The design of rijndael: AES - the advanced encryption standard, 2002.

[16] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[17] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [10], pages 416–427.

[18] Richared D. Deam. Formal Aspects of Mobile Code Security. Ph.D. dissertation, Princeton University, 1999.

[19] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[20] FIPS. *Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD 20899-8900, USA, August 2002.

[21] Matthew K. Franklin, editor. *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*. Springer, 2004.

[22] Michael Gorski, Stefan Lucks, and Thomas Peyrin. Slide Attacks on Hash Functions. In *ASIACRYPT*, pages 143 – 160, 2008.

[23] Deukjo Hong, Donghoon Chang, Jaechul Sung, Sangjin Lee, Seokhie Hong, Jaesang Lee, Dukjae Moon, and Sungtaek Chee. A New Dedicated 256-Bit Hash Function: FORK-256. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 2006.

[24] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin [21], pages 306–316.

[25] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[26] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for

Much Less than $2^n$ Work. In Cramer [13], pages 474–490.

[27] Lars R. Knudsen. SMASH - A Cryptographic Hash Function. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2005.

[28] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Biryukov [8], pages 39–57.

[29] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[30] F. I. MacWilliams and N. J. A. Sloane. The Theory of Error-Correcting Codes, 1977.

[31] Krystian Matusiewicz, Thomas Peyrin, Olivier Billet, Scott Contini, and Josef Pieprzyk. Cryptanalysis of FORK-256. In Biryukov [8], pages 19–38.

[32] Florian Mendel and Martin Schläffer. Collisions for Round-Reduced LAKE. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2008.

[33] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [10], pages 428–446.

[34] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. April 1995. See `http://csrc.nist.gov`.

[35] National Institute of Standards and Technology. FIPS 180: Secure Hash Standard. 1993. See `http://csrc.nist.gov`.

[36] Christof Paar and Martin Rosner. Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware. In *FCCM*, pages 219–225. IEEE Computer Society, 1997.

[37] Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT*, pages 551–567, 2007.

[38] Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Breaking a New Hash Function Design Strategy Called SMASH. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 233–244. Springer, 2005.

[39] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[40] R. Rivest. The MD5 Message-Digest Algorithm, 1992.

[41] Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*. Springer, 2008.

[42] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [13], pages 1–18.

[43] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[44] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [13], pages 19–35.

[45] Xun Yi, Shi Xing Cheng, Xiao Hu You, and Kwok Yan Lam. A Method for Obtaining Cryptographically Strong 8x8 S-boxes. In *IEEE Global Telecommunications Conference, GLOBECOM 97, Volume 2*, pages 689–693, 1997.

# A How to find (good) MDS matrices

The common way for finding MDS matrices is to use a generator matrix $G = (I_k\ A)$ of a Reed-Solomon code and take a sub matrix of $A$. This approach is deterministic and does always give a MDS matrix of the desired size (assuming that the size is not to big for MDS matrices over $\mathbb{F}_{256}$). Another way is to create random matrices and check for the desired properties. This way, we found the MDS-matrix which is given in section 4.2.1. Our MDS-finder (see our website on `www.twister-hash.com` algorithm favors MDS-matrices that have the following properties that lead to less working-memory requirements for look-up tables, higher speed (a multiplication by *one* is and addition) and better hardware performance (cyclicity) [15]:

- 'low' value of the entries especially as much values being '1' as possible,
- low number of different entries,
- cyclicity.

# B The AES/Twister S-BOX

The twister S-box is taken from AES [16] which is as follows.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0x63 | 0x7c | 0x77 | 0x7b | 0xf2 | 0x6b | 0x6f | 0xc5 |
| 0x30 | 0x01 | 0x67 | 0x2b | 0xfe | 0xd7 | 0xab | 0x76 |
| 0xca | 0x82 | 0xc9 | 0x7d | 0xfa | 0x59 | 0x47 | 0xf0 |
| 0xad | 0xd4 | 0xa2 | 0xaf | 0x9c | 0xa4 | 0x72 | 0xc0 |
| 0xb7 | 0xfd | 0x93 | 0x26 | 0x36 | 0x3f | 0xf7 | 0xcc |
| 0x34 | 0xa5 | 0xe5 | 0xf1 | 0x71 | 0xd8 | 0x31 | 0x15 |
| 0x04 | 0xc7 | 0x23 | 0xc3 | 0x18 | 0x96 | 0x05 | 0x9a |
| 0x07 | 0x12 | 0x80 | 0xe2 | 0xeb | 0x27 | 0xb2 | 0x75 |
| 0x09 | 0x83 | 0x2c | 0x1a | 0x1b | 0x6e | 0x5a | 0xa0 |
| 0x52 | 0x3b | 0xd6 | 0xb3 | 0x29 | 0xe3 | 0x2f | 0x84 |
| 0x53 | 0xd1 | 0x00 | 0xed | 0x20 | 0xfc | 0xb1 | 0x5b |
| 0x6a | 0xcb | 0xbe | 0x39 | 0x4a | 0x4c | 0x58 | 0xcf |
| 0xd0 | 0xef | 0xaa | 0xfb | 0x43 | 0x4d | 0x33 | 0x85 |
| 0x45 | 0xf9 | 0x02 | 0x7f | 0x50 | 0x3c | 0x9f | 0xa8 |
| 0x51 | 0xa3 | 0x40 | 0x8f | 0x92 | 0x9d | 0x38 | 0xf5 |
| 0xbc | 0xb6 | 0xda | 0x21 | 0x10 | 0xff | 0xf3 | 0xd2 |
| 0xcd | 0x0c | 0x13 | 0xec | 0x5f | 0x97 | 0x44 | 0x17 |
| 0xc4 | 0xa7 | 0x7e | 0x3d | 0x64 | 0x5d | 0x19 | 0x73 |
| 0x60 | 0x81 | 0x4f | 0xdc | 0x22 | 0x2a | 0x90 | 0x88 |
| 0x46 | 0xee | 0xb8 | 0x14 | 0xde | 0x5e | 0x0b | 0xdb |
| 0xe0 | 0x32 | 0x3a | 0x0a | 0x49 | 0x06 | 0x24 | 0x5c |
| 0xc2 | 0xd3 | 0xac | 0x62 | 0x91 | 0x95 | 0xe4 | 0x79 |
| 0xe7 | 0xc8 | 0x37 | 0x6d | 0x8d | 0xd5 | 0x4e | 0xa9 |
| 0x6c | 0x56 | 0xf4 | 0xea | 0x65 | 0x7a | 0xae | 0x08 |
| 0xba | 0x78 | 0x25 | 0x2e | 0x1c | 0xa6 | 0xb4 | 0xc6 |
| 0xe8 | 0xdd | 0x74 | 0x1f | 0x4b | 0xbd | 0x8b | 0x8a |
| 0x70 | 0x3e | 0xb5 | 0x66 | 0x48 | 0x03 | 0xf6 | 0x0e |
| 0x61 | 0x35 | 0x57 | 0xb9 | 0x86 | 0xc1 | 0x1d | 0x9e |
| 0xe1 | 0xf8 | 0x98 | 0x11 | 0x69 | 0xd9 | 0x8e | 0x94 |
| 0x9b | 0x1e | 0x87 | 0xe9 | 0xce | 0x55 | 0x28 | 0xdf |
| 0x8c | 0xa1 | 0x89 | 0x0d | 0xbf | 0xe6 | 0x42 | 0x68 |
| 0x41 | 0x99 | 0x2d | 0x0f | 0xb0 | 0x54 | 0xbb | 0x16 |

Table B.1: The AES/Twister S-box

# C Examples

This appendix is for informational purposes only.

## C.1 Twister-224 Examples

In the following we show the result of an extremely long messages which is hash using Twister-224: The message

$$abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmno$$

is repeated 16777216 times which give the hash value

$$7D4235BD495A99F75302CB6547966B0BAC9206C174E954AD83DCF080$$

The following table shows some test vectors for short message inputs.

| length | message | hash value |
|---|---|---|
| 0 | 00 | 93CFDE255C161D1D06644413C6D7D7C1442E4654671AFAFC09D4A40E |
| 1 | 00 | 0BD9F4888660AEDB72B943FEF54F69293691E991316C2F5F0CB87FE2 |
| 2 | C0 | 33B9EE086BA1E051AB91281AA06846ACC891DCA5624B5A4F4A99A1D8 |
| 3 | C0 | 8042AE83808FA215822A46B8409260B420FC09636D46C8B1C9EB231B |
| 4 | 80 | 4F530B4AD1EA61B04FFD42C83C836307B3D7AC30A9E60C9710BD1242 |
| 5 | 48 | 4C0A0B005677E59427D43EE38A66946476A399E0B956D1633781F679 |
| 6 | 50 | FCEB85A9CE861A8BA136A23833F9605A9D0626E541E960A4E937CE65 |

Table C.1: Short messages for /Twister-224

## C.2 TWISTER-256 Examples

In the following we show the result of an extremely long messages which is hash using TWISTER-256: The message

$$abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmno$$

is repeated 16777216 times which give the hash value

$$C2EF0A386154AB0A34137799053C37D9A66CA4919FBC$$
$$6DECA873AE7580CAF132$$

The following table shows some test vectors for short message inputs.

| length | message | hash value |
|---|---|---|
| 0 | 00 | CC043565D3016A5F4F0635FF0DE23<br>EEF069CE518266A488748DB9D69F0664484 |
| 1 | 00 | 97BAF8F9060BFAB752CDE28F5ACCC<br>780B1858D0A024A283926FAB696D5AF546B |
| 2 | C0 | EEEAF40C2712BCEF07E746220B7D23<br>C6BA2DD9F5C3ECF590D1E0C17C3482077F |
| 3 | C0 | 72CCCBC49B05C7F7F5E9C383616594<br>7B36C14F334053E2C7B6C8DC906C55066A |
| 4 | 80 | 31AF0D16733F327824F542427B1CB7F<br>6320D85750B33057F97C87043BE1E4C9A |
| 5 | 48 | B291BB6E8F1200D3EEB3BB07E359EA<br>AE8F65199FD2599883EEE3E99BDDCAF7A9 |
| 6 | 50 | D6E468BEBBCA6D1BC4F34CFF0B4409<br>97C61B9F01F3E6C0E23BD81ECCA6780A31 |

Table C.2: Short messages for /TWISTER-256

## C.3 Twister-384 Examples

In the following we show the result of an extremely long messages which is hash using Twister-384: The message

$$abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmno$$

is repeated 16777216 times which give the hash value

$$70C2D4390429DB964F939B92A3C74F8A8D851A4DAA6F82481B2471A$$
$$AD5FA6768D64254B433D509A4DBC64718F9E8026F$$

The following table shows some test vectors for short message inputs.

| length | message | hash value |
|---|---|---|
| 0 | 00 | C19F4457E1B269F4C4502A54F5C1946544A0F1B6D1C99<br>85B5C94137806C4F8D7<br>F188374A2D7E01685B97E40F18DD6808 |
| 1 | 00 | F154FE7598363EE2763930F5F399F0AD7830852641703A6<br>C4B752897EDA9AC64F08A91AECE9E50DCDCB80FFA33E338BD |
| 2 | C0 | 722BF0C945E15863C80588F47533BC4E7FE3207B11407F6<br>D525E365CE43F2F60BF3FBC0B1F32969192086D5D005AA9C2 |
| 3 | C0 | 6748A9704C2452B5C6CF5262C5BE9F0ED29C0FACD821B<br>658CA93857513C40C616B346F6B1794E1E74964DF7564977909 |
| 4 | 80 | F8FF0FA880919F7D302B66EA95C796DF97CE0CB204E91C<br>40626776E0E7A64E55491406FA8164196B33039538A2563DF3 |
| 5 | 48 | BBBD380230A91CB05AD43ACE01B024585528D971FFC87<br>A194CE0BF4564EC1656AA7DC4E6F4E8264F0816905C8EF5606B |
| 6 | 50 | 53024F774BDFF0AC54A4F42D7EDE8490BCEC2F955F60B<br>B9837EBB13F084AAD55B01C007CAF7C7DDDB4A436ADEB59F69B |

Table C.3: Short messages for /Twister-384

## C.4 Twister-512 Examples

In the following we show the result of an extremely long messages which is hash using Twister-512: The message

$$abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmno$$

is repeated 16777216 times which give the hash value

$$371C7D667057E67177901164A3CC627A55EAA2E5$$
$$BBB63D141F3344538A26F0277FFC3BA1B184E091871$$
$$6EA1C9DD2CCAE8C7ACB15FA9946583EDA5B9805FE9F9A$$

The following table shows some test vectors for short message inputs.

| length | message | hash value |
|---|---|---|
| 0 | 00 | 8F5A88005B2833AFCAA6EC3762F8135C00DCE5A5 |
| | | 0B74E4C340AFF10165745EC88B0B542D49345E863AFAA2 |
| | | 1A3F2D735191612DC214A8E4A8CA54147BDA477AD2 |
| 1 | 00 | DB69DD03E42C4D256CFE87FD67E56039E172A985AC3F2 |
| | | 6C1BEE30563C262BD15EF06D9F1296BD92524874BA7A2 |
| | | 9E4B4C6AAB1267919061C77CD2E27AF69DF4D6 |
| 2 | C0 | E799E0EE71DBEC91B34B4E2E91481498E01A6FCE4A12F5BA |
| | | AE085ED8D31047F0E64E80857CCC224EA057F492F |
| | | A8DFBC98155D67197CB9DBEAC252673A423724C |
| 3 | C0 | 50DD967493AA124B145DF4753807B23C7165E4B85AC |
| | | 51EEA2B0F1BE2BB541B17C20F27D1B1D923093E62DDACE |
| | | EEE41A8196839CFCD354F28D5F2A61F94E8E01F |
| 4 | 80 | DCC95B18A2F2E0D6FFA5A007EBA8DD2B1B0B97F96 |
| | | BD04423B9BA3FFF2E2C2E723820451D04F7F83BB122 |
| | | D9F8E27FE34346EDFDC8E9F9FB4D732C0CE937709AF7 |
| 5 | 48 | BDF4009E3C01E4BB96DA092C406C8F87200 |
| | | D415D00252D204B0774F49CA66AFAB1A6B375680CDFB9646587 |
| | | FD35B71C67BD788BA7CFEEA13639EABCCB01FD2259 |
| 6 | 50 | 903470C64B53DFBBCE29CE003F7C98F25DDBE7AF81 |
| | | C888778985C7D962F1979C97E657E2925EE8E3F85D |
| | | E96B245DA11BCC66C015ED107D79E85FE071BE58A5B5 |

Table C.4: Short messages for /Twister-512