AURORA:
A Cryptographic Hash Algorithm Family

Submitters:

Sony Corporation[1] and Nagoya University[2]

Algorithm Designers:

Tetsu Iwata[2], Kyoji Shibutani[1], Taizo Shirai[1], Shiho Moriai[1], Toru Akishita[1]

October 24, 2008

Executive Summary

We present a new hash function family AURORA as a candidate for a new cryptographic hash algorithm (SHA-3) family. The hash function family AURORA consists of the algorithms: AURORA-224, AURORA-256, AURORA-384, AURORA-512, AURORA-224M, and AURORA-256M, where AURORA-224M and AURORA-256M are optional instances that are designed to have multi-collision resistance.

AURORA-224 and AURORA-256 are constructed from the secure and efficient compression function using a security-enhanced Merkle-Damgård transform, i.e., the strengthened Merkle-Damgård transform with the finalization function. The compression function is designed based on the well-established design techniques for blockciphers, and uses the Davies-Meyer construction. Since most of existing attacks on hash functions exploited simplicity of message scheduling, we employ a secure message scheduling, which is a different design philosophy from the MDx family including SHA-2.

AURORA-384 and AURORA-512 employ a novel domain extension transform called the Double-Mix Merkle-Damgård (DMMD) transform. The DMMD transform consists of two parallel lines of the compression functions and the mixing functions inserted at intervals. This domain extension transform enables an efficient collision-resistant construction for double length hash functions. Furthermore, the combination of the compression function and the DMMD transform achieves further efficiency by sharing the message scheduling of two compression functions.

The overall structure of AURORA-224M and AURORA-256M is the same as AURORA-384/512 except constants and the final mixing function. The DMMD transform also opens a new efficient way of providing multi-collision resistance. By using the DMMD transform, AURORA-224M and AURORA-256M efficiently achieve multi-collision resistance. As a result, the AURORA family achieves consistency of the design, because all algorithms use similar 256-bit compression functions as building blocks.

Moreover, the AURORA family achieves high efficiency on many platforms. In software implementation on the NIST reference platform (64-bit), AURORA-256 achieves 15.4 cycles/byte and AURORA-512 achieves 27.4 cycles/byte. Also, AURORA shows good performance across a variety of platforms, because it uses platform-independent operations. In hardware implementation, AURORA enables a variety of implementations, from high-speed to area-restricted implementations. Using a $0.13\mu m$ CMOS ASIC library, AURORA-256 can be implemented with only 11.1 Kgates in an area-optimized implementation. In a speed-optimized implementation, AURORA-256 achieves the highest throughput of 10.4 Gbps. For AURORA-512, the smallest size is 14.6 Kgates and the highest throughput is 9.1 Gbps.

Achieving these good performance both in hardware and in software in a single algorithm family which is based on the above design techniques makes a clear distinction between the AURORA family and the SHA-2 family.

Contents

Chapter 1

Introduction

This document describes the algorithm specifications and supporting documentation including design rationale, security, efficient implementation, applications, advantage and limitations of the hash function family AURORA, which we submit as a candidate for a new cryptographic hash algorithm (SHA-3) family.

Since SHA-3 is expected to provide a substitute of the SHA-2 family, AURORA is designed to preserve certain properties of the SHA-2 family including the input parameters, the output sizes, collision resistance, preimage resistance, second-preimage resistance, and the one-pass streaming mode of execution, according to the requirements for SHA-3 candidates [38]. Moreover, AURORA is designed to offer features that exceed the SHA-2 family.

AURORA is designed based on the following design philosophy:

- **Security**: Its security level should be guaranteed by security proofs or security arguments.
  - There is no known structural weakness in the design of the domain extension transform, and the security of the hash function is supported by security proofs.
  - In the design of a compression function, the structure and the components should be chosen to facilitate analysis and to utilize the well-established techniques for blockcipher design and analysis.
  - It should be designed based on different design criteria from the MDx family including SHA-2 so that a possibly successful attack on SHA-2 is unlikely to be applicable to it.
- **Implementation Efficiency and Flexibility**: It should be designed to have better efficiency than the SHA-2 family on many platforms. Also, it should be designed to be less platform-specific.
  - It should be implemented efficiently in a wide range of software platforms (32-bit, 64-bit and 8-bit processors with various compilers and operating systems) without dedicated optimization techniques for specific processors.
  - It should be suitable to flexible hardware implementations with wide variety of area/speed trade-offs.
- **Originality**: It should contain technical breakthroughs to improve security and/or efficiency, not just a combination of existing techniques.
- **Similarity among the Algorithm Family**: According to the NIST requirements [38] (NIST does not intend to select a wholly distinct algorithm for each of the minimally required message digest sizes), all the hash function instances with hash sizes of 224, 256, 384, and 512 bits should be designed under a consistent design philosophy. Concretely, by using the same structure and components e.g., S-boxes and matrices, they should provide security argument and performance evaluation in a unified framework.

Table 1.1: AURORA family.

Name	max. message size (bits)	message block size (bits)	chaining value size (bits)	hash size (bits)
AURORA-224	$512 \times (2^{64} - 1)$	512	256	224
AURORA-256	$512 \times (2^{64} - 1)$	512	256	256
AURORA-384	$512 \times (2^{64} - 1)$	512	512	384
AURORA-512	$512 \times (2^{64} - 1)$	512	512	512
optional instances				
AURORA-224M	$512 \times (2^{64} - 1)$	512	512	224
AURORA-256M	$512 \times (2^{64} - 1)$	512	512	256

The hash function family AURORA. To practice the design philosophy, we designed the hash function family AURORA which consists of the algorithms called AURORA-224, AURORA-256, AURORA-384, AURORA-512, AURORA-224M, and AURORA-256M. AURORA-224, AURORA-256, AURORA-384 and AURORA-512 support hash sizes of 224, 256, 384, and 512 bits, respectively. AURORA-224M and AURORA-256M support hash sizes of 224 and 256 bits, respectively. They are *optional instances* that are designed to have multi-collision resistance by increasing the internal chaining value size ("M" means multi-collision resistance). Every instance of the AURORA family supports a maximum message length of $512 \times (2^{64} - 1)$ bits, which meets the minimum acceptability requirement regarding the maximum message length. Table 1.1 presents the basic properties of the AURORA family.

AURORA-224 and AURORA-256 are constructed from the secure and efficient compression function using a security-enhanced Merkle-Damgård transform, i.e., the strengthened Merkle-Damgård transform with the finalization function. The compression function is designed based on the well-established design techniques for blockciphers, and uses the Davies-Meyer construction. Since most of existing attacks on hash functions exploited simplicity of message scheduling, we employ a secure message scheduling, which is a different design philosophy from the MDx family including SHA-2.

AURORA-384 and AURORA-512 employ a novel domain extension transform called the Double-Mix Merkle-Damgård (DMMD) transform. The DMMD transform consists of two parallel lines of the compression functions and the mixing functions inserted at intervals. This domain extension transform enables an efficient collision-resistant construction for double length hash functions. Furthermore, the combination of the compression function of AURORA and the DMMD transform achieves further efficiency by sharing the message scheduling of two compression functions.

The overall structure of AURORA-224M and AURORA-256M is the same as AURORA-384/512 except constants and the final mixing function. The DMMD transform also opens a new efficient way of providing multi-collision resistance. By using the DMMD transform, AURORA-224M and AURORA-256M efficiently achieve multi-collision resistance. As a result, the AURORA family achieves consistency of the design, because all algorithms use similar 256-bit compression functions as building blocks.

Moreover, the AURORA family achieves high efficiency on many platforms. In software implementation on the NIST reference platform (64-bit), AURORA-256 achieves 15.4 cycles/byte and AURORA-512 achieves 27.4 cycles/byte. Also, AURORA shows good performance across a variety of platforms, because it uses platform-independent operations. In hardware implementation, AURORA enables a variety of implementations, from high-speed to area-restricted implementations. Using a $0.13\mu m$ CMOS ASIC library, AURORA-256 can be implemented with only 11.1 Kgates in an area-optimized implementation. In a speed-optimized implementation, AURORA-256 achieves the highest throughput of 10.4 Gbps. For AURORA-512, the smallest size is 14.6 Kgates and the highest throughput is 9.1 Gbps.

Organization of the document. This document is organized as follows: Chapter 2 describes the specification of the AURORA family. Chapter 3 provides the design rationale. Chapter 4 explains all aspects of security: security argument and algorithm analysis. Chapter 5 shows efficient implementation results of AURORA. Chapter 6 describes the usage of AURORA in important applications. Finally, AURORA's advantages and limitation are described in Chapter 7.

Chapter 2

Specification of AURORA

2.1 Notation

We first describe notation, conventions and symbols used throughout this document.

- We use the prefix `0x` to denote hexadecimal numbers.
- A bit string x with the suffix, $x_{(n)}$, indicates that x is n bits. This suffix is omitted if there is no ambiguity.
- For bit strings x and y, $x \,\|\, y$ or (x, y) is their concatenation.
- For bit strings x and y, $x \leftarrow y$ means that the bit string x is updated by the bit string y. For an nl-bit x, we write $(x_{0\,(n)}, x_{1\,(n)}, \ldots, x_{l-1\,(n)}) \leftarrow x_{(nl)}$ to mean that x is divided into $(x_0, x_1, \ldots, x_l)$, where $(x_{0\,(n)} \,\|\, x_{1\,(n)} \,\|\, \cdots \,\|\, x_{l-1\,(n)}) = x_{(nl)}$.
- For a bit string $x_{(n)}$ and an integer l, $x \lll_n l$ is the l-bit left cyclic shift of x, and $x \ggg_n l$ is the l-bit right cyclic shift of x.
- For bit strings $x_0, x_1, \ldots, x_{n-1}$, $\{x_j\}_{0 \le j < n}$ is a shorthand for $(x_0, x_1, \ldots, x_{n-1})$.
- For an integer l, 0^l is the l times repetition of zero bits and 1^l is the l times repetition of one bits.
- For a bit string x, $\overline{x}$ is the bit-wise complement of x.
- For an element of GF(2^n) represented as a polynomial $x_{n-1}\alpha^{n-1} + x_{n-2}\alpha^{n-2} + \ldots + x_1\alpha + x_0$ where α is a root of an irreducible polynomial, $x_{n-1}||x_{n-2}||\ldots||x_1||x_0$ denotes the bit representation of the polynomial.

Following variables and symbols have specific meanings.

M	The input message.
M_i	The i-th block of the message (after the padding).
m	The length of M in blocks (after the padding).
H_i	The i-th chaining value.
MSM	The Message Scheduling Module.
CPM	The Chaining value Processing Module.
BD	The Byte Diffusion function.
DR	The Data Rotating function.
$PROTL$	The Partial ROTating Left function.
$PROTR$	The Partial ROTating Right function.
Pad	The Padding function.
Len_n	The Length of the input message in blocks encoded into n bits.
TF_n	The Truncation Function that outputs n bits.
F_0, F_1, F_2, and F_3	The F-Functions.
$\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$	The matrices used in the F-functions.
S	The S-box.

Following symbols are used for AURORA-224/256.

CF	The Compression Function for AURORA-224/256.
MS_L and MS_R	The Message Scheduling functions for CF.
CP	The Chaining value Processing function for CF.
FF	The Finalization Function for AURORA-224/256.
MSF_L and MSF_R	The Message Scheduling functions for Finalization for FF.
CPF	The Chaining value Processing function for Finalization for FF.
$CONM_{L,j}$ and $CONM_{R,j}$	The CONstants for MS_L, MS_R, MSF_L, and MSF_R.
$CONC_j$	The CONstant for CP and CPF.

Following symbols are used for AURORA-384/512.

$CF_0, CF_1, \ldots, CF_7$	The Compression Functions for AURORA-384/512.
MF	The Mixing Function for AURORA-384/512.
MFF	The Mixing Function for Finalization for AURORA-384/512.
$MS_{L,s}$ and $MS_{R,s}$	The Message Scheduling functions for CF_s $(0 \leq s \leq 7)$, MF $(s = 8)$, and MFF $(s = 9)$.
$CP_{L,s}$ and $CP_{R,s}$	The Chaining value Processing functions for CF_s $(0 \leq s \leq 7)$, MF $(s = 8)$, and MFF $(s = 9)$.
$CONM_{L,s,j}$ and $CONM_{R,s,j}$	The CONstants used in $MS_{L,s}$ and $MS_{R,s}$, respectively.
$CONC_{L,s,j}$ and $CONC_{R,s,j}$	The CONstants used in $CP_{L,s}$ and $CP_{R,s}$, respectively.

Following symbols are used for AURORA-224M/256M.

$CF_0^M, CF_1^M, \ldots, CF_7^M$	The Compression Functions for AURORA-224M/256M.
MF^M	The Mixing Function for AURORA-224M/256M.
MFF^M	The Mixing Function for Finalization for AURORA-224M/256M.
$ME_{L,s}^M$ and $ME_{R,s}^M$	The Message Expansion functions for CF_s^M $(0 \leq s \leq 7)$, MF^M $(s = 8)$, and MFF^M $(s = 9)$.
$CP_{L,s}^M$ and $CP_{R,s}^M$	The Chaining value Processing functions for CF_s^M $(0 \leq s \leq 7)$, MF^M $(s = 8)$, and MFF^M $(s = 9)$.
$CONM_{L,s,j}^M$ and $CONM_{R,s,j}^M$	The CONstants used in $ME_{L,s}^M$ and $ME_{R,s}^M$, respectively.
$CONC_{L,s,j}^M$ and $CONC_{R,s,j}^M$	The CONstants used in $CP_{L,s}^M$ and $CP_{R,s}^M$, respectively.

2.2 Building Blocks

In this section, specifications of the essential building blocks for constructing AURORA algorithms are described.

2.2.1 Message Scheduling Module: *MSM*

The message scheduling module, *MSM*, takes the following two inputs;

- a bit string $X_{(256)}$, and
- a set of bit strings $\{Y_{j\,(32)}\}_{0\le j<32}$.

The output is a set of bit strings $\{Z_{j\,(32)}\}_{0\le j<72}$.

MSM internally uses a byte diffusion function $BD : (\{0,1\}^{32})^8 \rightarrow (\{0,1\}^{32})^8$, which is a permutation over $(\{0,1\}^{32})^8$ and is defined in Sec. 2.2.3. *MSM* is parameterized by two functions F and F', where

$$\begin{cases} F : \{0,1\}^{32} \rightarrow \{0,1\}^{32}, \\ F' : \{0,1\}^{32} \rightarrow \{0,1\}^{32}. \end{cases} \tag{2.1}$$

We write $MSM[F, F']$ when we emphasize that it is parameterized by functions F and F'. We now describe the specification of $MSM[F, F']$.

Step 1. Let $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) \leftarrow X_{(256)}$.

Step 2. Let $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_0, Y_1, Y_2, Y_3)$.

Step 3. Let $(Z_0, Z_1, \ldots, Z_7) \leftarrow (X_0, X_1, \ldots, X_7)$.

Step 4. (7 round iterations) The following operations are iterated for $i = 1$ to 7.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_{4i}, Y_{4i+1}, Y_{4i+2}, Y_{4i+3}) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ (Z_{8i}, Z_{8i+1}, \ldots, Z_{8i+7}) \leftarrow (X_0, X_1, \ldots, X_7) \end{cases}$$

Step 5. (8-th round) Then the following operations are executed.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ (Z_{64}, Z_{65}, \ldots, Z_{71}) \leftarrow (X_0, X_1, \ldots, X_7) \end{cases}$$

Step 6. Finally, the output is $\{Z_{j\,(32)}\}_{0\le j<72}$.

See Fig. 2.1 for an illustration and Fig. 2.13 for a pseudocode.

2.2.2 Chaining Value Processing Module: *CPM*

The chaining value processing module, *CPM*, takes the following three inputs;

- a bit string $X_{(256)}$,
- a set of bit strings $\{Y_{j\,(32)}\}_{0\le j<144}$, and
- a set of bit strings $\{W_{j\,(32)}\}_{0\le j<68}$.

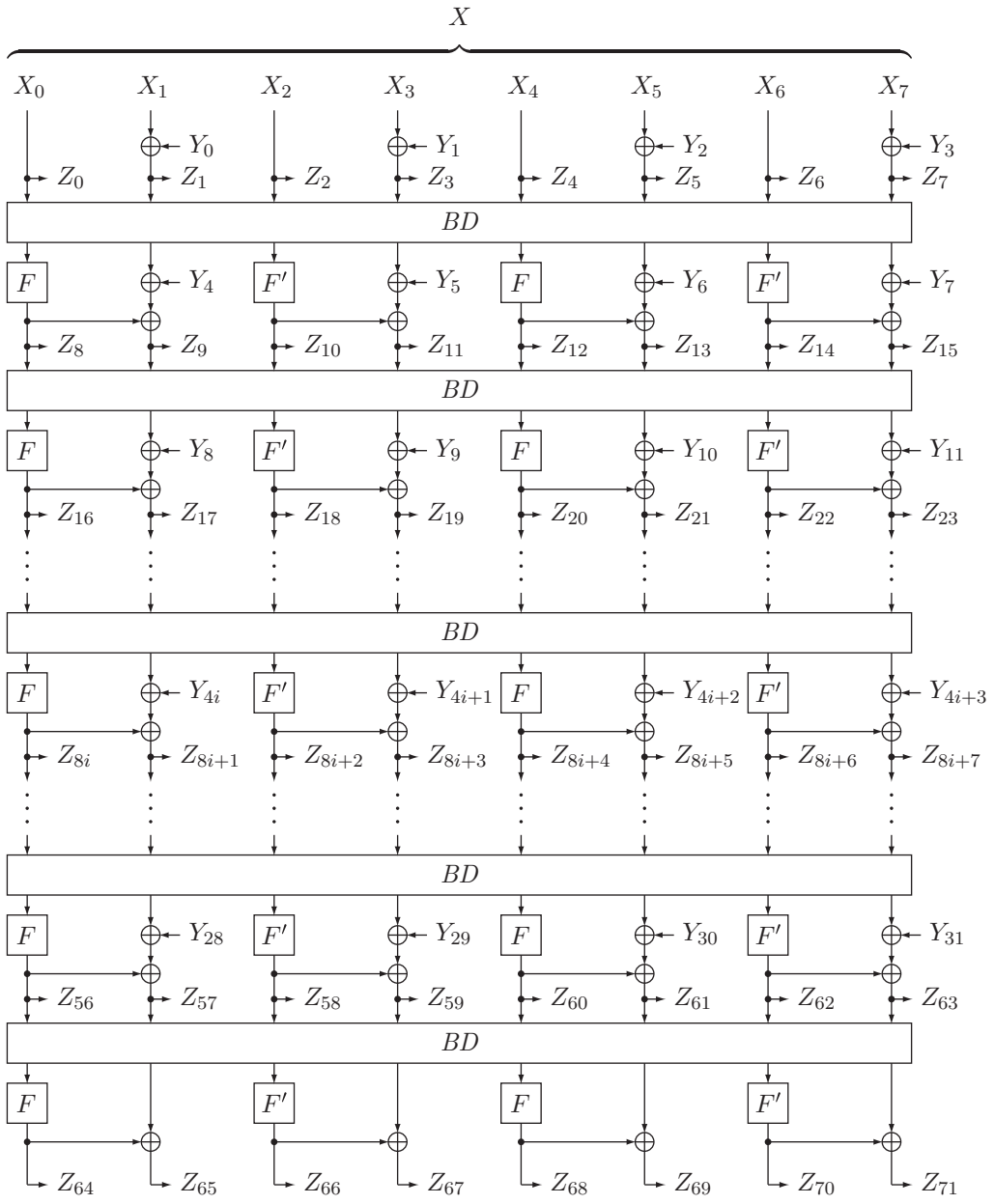

Figure 2.1: $\{Z_{j\,(32)}\}_{0\le j<72} \leftarrow MSM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0\le j<32})$.

The output is a bit string $Z_{(256)}$.

CPM internally uses a byte diffusion function *BD*, which is also used in *MSM*, and is defined in Sec. 2.2.3. As with *MSM*, *CPM* is parameterized by two functions F and F' over $\{0,1\}^{32}$, and we write $CPM[F, F']$ when we use functions F and F'.

We now describe the specification of $CPM[F, F']$.

Step 1. Let $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) \leftarrow X_{(256)}$.

Step 2. Let $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (W_0, W_1, W_2, W_3)$.

Step 3. Let $(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_0, Y_1, \ldots, Y_7)$.

Step 4. (16 round iterations) The following operations are iterated for $i = 1$ to 16.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3}) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ (X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7}) \end{cases}$$

Step 5. (17-th round) Then the following operations are executed.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ (X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{136}, Y_{137}, \ldots, Y_{143}) \end{cases}$$

Step 6. Finally, the output is $Z_{(256)} \leftarrow (X_{0\,(32)} \,\|\, X_{1\,(32)} \,\|\, \cdots \,\|\, X_{7\,(32)})$.

See Fig. 2.2 for an illustration and Fig. 2.14 for a pseudocode.

2.2.3 Byte Diffusion Function: *BD*

The byte diffusion function, *BD*, takes a bit string $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$ as the input, and outputs the updated bit string $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$.

It works as follows.

Step 1. For $i = 0, 1, \ldots, 7$, $X_{i\,(32)}$ is divided into a 4-byte sequence as

$$(x_{4i\,(8)}, x_{4i+1\,(8)}, x_{4i+2\,(8)}, x_{4i+3\,(8)}) \leftarrow X_{i\,(32)},$$

and $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$ is now regarded as a sequence of bytes;

$$(x_{0\,(8)}, x_{1\,(8)}, \ldots, x_{31\,(8)}) = (X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}).$$

Step 2. Next we permute $(x_0, x_1, \ldots, x_{31})$ according to the permutation π defined in Fig. 2.3, where the i-th byte x_i is moved to the $\pi(i)$-th byte. In other words, let $x'_{\pi(i)} = x_i$ for $i = 0, 1, \ldots, 31$. Then $(x'_0, x'_1, \ldots, x'_{31})$ is the result of the permutation. For example, $x'_0 = x_4$, $x'_1 = x_{29}$, and so on.

Step 3. For $i = 0, 1, \ldots, 7$, the 4-byte sequence $(x'_{4i\,(8)}, x'_{4i+1\,(8)}, x'_{4i+2\,(8)}, x'_{4i+3\,(8)})$ is concatenated to form the updated $X_{i\,(32)} = (x'_{4i\,(8)} \,\|\, x'_{4i+1\,(8)} \,\|\, x'_{4i+2\,(8)} \,\|\, x'_{4i+3\,(8)})$, and the output is $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$.

See Fig. 2.4 for an illustration and Fig. 2.15 for a pseudocode.

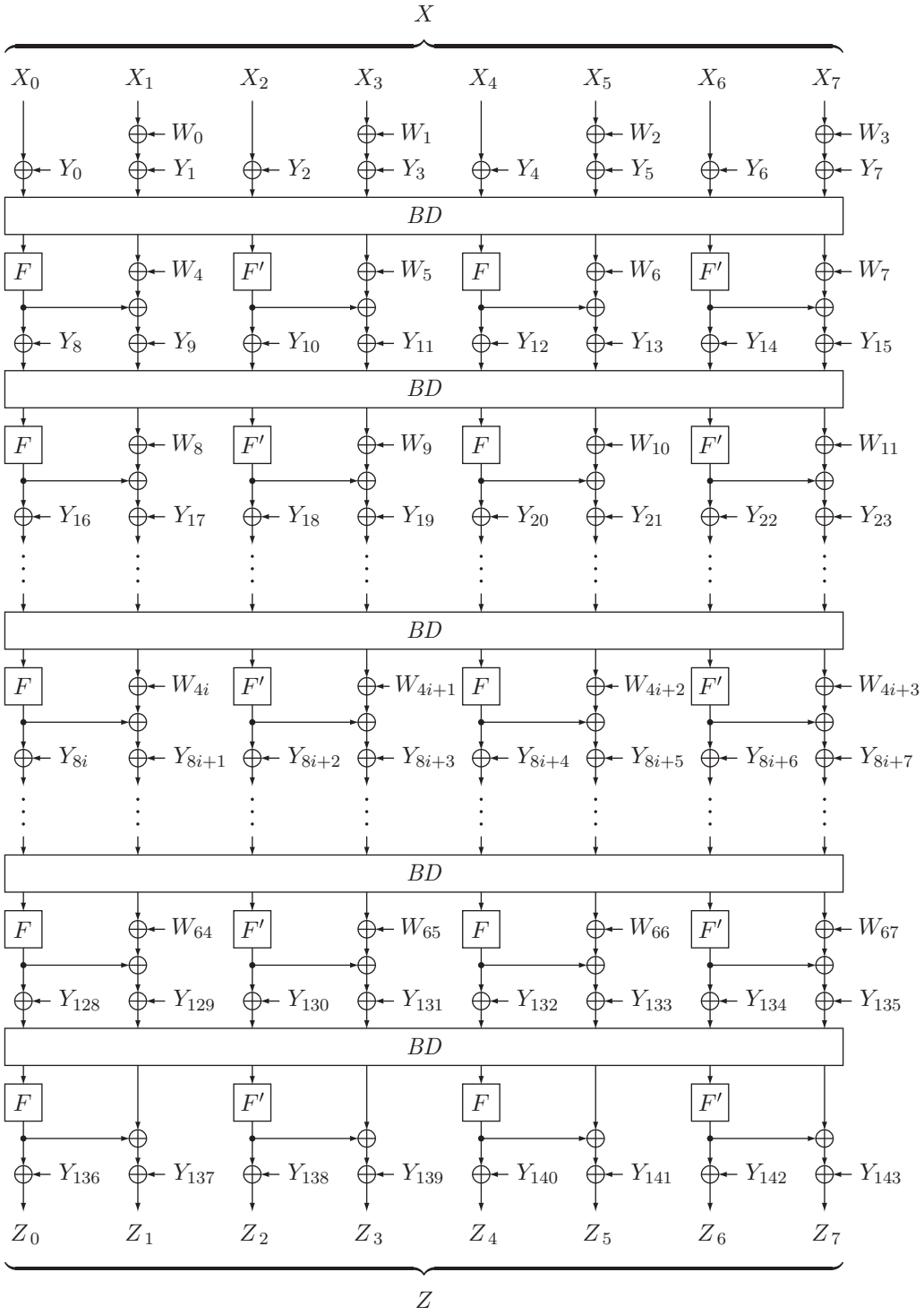

Figure 2.2: $Z_{(256)} \leftarrow CPM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0 \le j < 144}, \{W_{j\,(32)}\}_{0 \le j < 68})$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(i)$	28	29	30	31	0	9	18	27	4	5	6	7	8	17	26	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(i)$	12	13	14	15	16	25	2	11	20	21	22	23	24	1	10	19

Figure 2.3: Definition of the permutation $\pi(\cdot) : \{0, 1, \ldots, 31\} \rightarrow \{0, 1, \ldots, 31\}$.

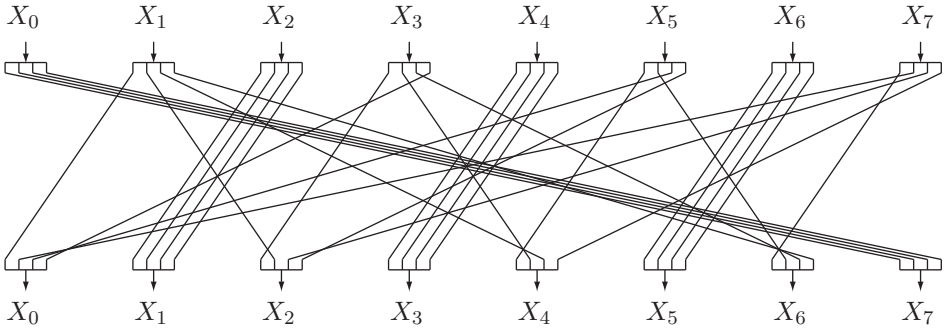

Figure 2.4: $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) \leftarrow BD(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$.

2.2.4 F-Functions: F_0, F_1, F_2, and F_3

We use four F-functions, F_0, F_1, F_2, and F_3, where they take 32-bit input X as input and produce 32-bit output Y. Each function is used as an instantiation of a parameter functions F or F' in *MSM* and *CPM*.

Before defining these F-functions, we first define the S-box $S : \{0,1\}^8 \rightarrow \{0,1\}^8$, and four 4×4 matrices, $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$.

- The S-box $S : x_{(8)} \rightarrow y_{(8)}$ is defined as follows.

$$y = \begin{cases} g(f(x)^{-1}) & \text{if } f(x) \neq 0 \\ g(0) & \text{if } f(x) = 0 \end{cases} .$$

The inverse function is performed in $\mathrm{GF}((2^4)^2)$ defined by an irreducible polynomial $z^2 + z + \{1001\}$ for which the underlying $\mathrm{GF}(2^4)$ is defined by an irreducible polynomial $z'^4 + z' + 1$. Moreover, $f : x_{(8)} \rightarrow y_{(8)}$ and $g : x_{(8)} \rightarrow y_{(8)}$ are affine transformations over $\mathrm{GF}(2)$, which are defined as

$$f : \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} , \qquad (2.2)$$

Table 2.1: S

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

and

$$g: \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{2.3}$$

where $(x_{0(1)}||x_{1(1)}||x_{2(1)}||x_{3(1)}||x_{4(1)}||x_{5(1)}||x_{6(1)}||x_{7(1)}) \leftarrow x_{(8)}$ and $(y_{0(1)}||y_{1(1)}||y_{2(1)}||y_{3(1)}||y_{4(1)}||y_{5(1)}||y_{6(1)}||y_{7(1)}) \leftarrow y_{(8)}$. Table 2.1 shows the output values of S.

- The four matrices are defined as follows.

$$\mathcal{M}_0 = \begin{pmatrix} \texttt{0x01} & \texttt{0x02} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x02} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \end{pmatrix}, \tag{2.4}$$

$$\mathcal{M}_1 = \begin{pmatrix} \texttt{0x01} & \texttt{0x06} & \texttt{0x08} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x06} & \texttt{0x08} \\ \texttt{0x08} & \texttt{0x02} & \texttt{0x01} & \texttt{0x06} \\ \texttt{0x06} & \texttt{0x08} & \texttt{0x02} & \texttt{0x01} \end{pmatrix}, \tag{2.5}$$

$$\mathcal{M}_2 = \begin{pmatrix} \texttt{0x03} & \texttt{0x01} & \texttt{0x02} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x02} & \texttt{0x03} \end{pmatrix}, \tag{2.6}$$

$$\mathcal{M}_3 = \begin{pmatrix} \texttt{0x06} & \texttt{0x08} & \texttt{0x02} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x06} & \texttt{0x08} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x06} & \texttt{0x08} \\ \texttt{0x08} & \texttt{0x02} & \texttt{0x01} & \texttt{0x06} \end{pmatrix}. \tag{2.7}$$

Multiplications are operated over GF(2^8) defined by an irreducible polynomial $z^8+z^4+z^3+z^2+1$.

Now we describe F-functions.

Step 1. Let $(x_{0\,(8)}, x_{1\,(8)}, x_{2\,(8)}, x_{3\,(8)}) \leftarrow X_{(32)}$.

Step 2. Let $(x_0, x_1, x_2, x_3) \leftarrow (S(x_0), S(x_1), S(x_2), S(x_3))$.

Step 3. For $i \in \{0, 1, 2, 3\}$, the output of F_i is $Y_{(32)} = (y_{0\,(8)} \,\|\, y_{1\,(8)} \,\|\, y_{2\,(8)} \,\|\, y_{3\,(8)})$, where

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \mathcal{M}_i \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

2.2.5 Data Rotating Function: *DR*

The data rotating function, *DR*, takes the following two inputs;

- a set of bit strings $\{X_{j\,(32)}\}_{0 \le j < 72}$, and
- a set of bit strings $\{Y_{j\,(32)}\}_{0 \le j < 72}$.

The output is a set of bit strings $\{Z_{j\,(32)}\}_{0 \le j < 144}$.

DR uses the following two functions;

$$\begin{cases} PROTL : (\{0,1\}^{32})^8 \to (\{0,1\}^{32})^8, \\ PROTR : (\{0,1\}^{32})^8 \to (\{0,1\}^{32})^8, \end{cases}$$

which we define as

$$PROTL(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) = (X'_{0\,(32)}, X'_{1\,(32)}, \ldots, X'_{7\,(32)}), \tag{2.8}$$

where $X'_i = X_i$ for $i = 0, 2, 4, 5, 6, 7$, and $(X'_1 \,\|\, X'_3) = (X_1 \,\|\, X_3) \lll_{64} 1$.

Similarly, we define

$$PROTR(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) = (X'_{0\,(32)}, X'_{1\,(32)}, \ldots, X'_{7\,(32)}), \tag{2.9}$$

where $X'_i = X_i$ for $i = 0, 2, 4, 5, 6, 7$, and $(X'_1 \,\|\, X'_3) = (X_1 \,\|\, X_3) \ggg_{64} 1$.

In other words, they rotate the two words by one bit, where these words are concatenated and regarded as one 64 bit string.

Now *DR* works as follows.

Step 1. For inputs $\{X_{j\,(32)}\}_{0 \le j < 72}$ and $\{Y_{j\,(32)}\}_{0 \le j < 72}$, we define $\{Z_{j\,(32)}\}_{0 \le j < 144}$ by iterating the following operations for $i = 0$ to 8.

$$\begin{cases} (Z_{16i}, Z_{16i+1}, \ldots, Z_{16i+7}) \leftarrow PROTL(X_{8i}, X_{8i+1}, \ldots, X_{8i+7}) \\ (Z_{16i+8}, Z_{16i+9}, \ldots, Z_{16i+15}) \leftarrow PROTR(Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7}) \end{cases}$$

Step 2. The output is $\{Z_{j\,(32)}\}_{0 \le j < 144}$ defined in the above operations.

See Fig. 2.5 for an illustration and Fig. 2.16 for a pseudocode.

$$
\begin{array}{cccccc|ccc|ccccc}
X_0 & X_1 & \cdots & X_7 & \rightarrow & (Z_1 \,\|\, Z_3) & \leftarrow & (X_1 \,\|\, X_3) \lll_{64} 1 & \rightarrow & Z_0 & Z_1 & \cdots & Z_7 \\
Y_0 & Y_1 & \cdots & Y_7 & \rightarrow & (Z_9 \,\|\, Z_{11}) & \leftarrow & (Y_1 \,\|\, Y_3) \ggg_{64} 1 & \rightarrow & Z_8 & Z_9 & \cdots & Z_{15} \\
X_8 & X_9 & \cdots & X_{15} & \rightarrow & (Z_{17} \,\|\, Z_{19}) & \leftarrow & (X_9 \,\|\, X_{11}) \lll_{64} 1 & \rightarrow & Z_{16} & Z_{17} & \cdots & Z_{23} \\
Y_8 & Y_9 & \cdots & Y_{15} & \rightarrow & (Z_{25} \,\|\, Z_{27}) & \leftarrow & (Y_9 \,\|\, Y_{11}) \ggg_{64} 1 & \rightarrow & Z_{24} & Z_{25} & \cdots & Z_{31} \\
X_{16} & X_{17} & \cdots & X_{23} & \rightarrow & (Z_{33} \,\|\, Z_{35}) & \leftarrow & (X_{17} \,\|\, X_{19}) \lll_{64} 1 & \rightarrow & Z_{32} & Z_{33} & \cdots & Z_{39} \\
Y_{16} & Y_{17} & \cdots & Y_{23} & \rightarrow & (Z_{41} \,\|\, Z_{43}) & \leftarrow & (Y_{17} \,\|\, Y_{19}) \ggg_{64} 1 & \rightarrow & Z_{40} & Z_{41} & \cdots & Z_{47} \\
\vdots & \vdots & & \vdots & & & \vdots & & & \vdots & \vdots & & \vdots \\
X_{64} & X_{65} & \cdots & X_{71} & \rightarrow & (Z_{129} \,\|\, Z_{131}) & \leftarrow & (X_{65} \,\|\, X_{67}) \lll_{64} 1 & \rightarrow & Z_{128} & Z_{129} & \cdots & Z_{135} \\
Y_{64} & Y_{65} & \cdots & Y_{71} & \rightarrow & (Z_{137} \,\|\, Z_{139}) & \leftarrow & (Y_{65} \,\|\, Y_{67}) \ggg_{64} 1 & \rightarrow & Z_{136} & Z_{137} & \cdots & Z_{143}
\end{array}
$$

Figure 2.5: $\{Z_{j\,(32)}\}_{0 \le j < 144} \leftarrow DR(\{X_{j\,(32)}\}_{0 \le j < 72}, \{Y_{j\,(32)}\}_{0 \le j < 72})$.

2.3 Specification of AURORA-256

2.3.1 Overall Structure

AURORA-256 takes the input message of length at most $512 \times (2^{64} - 1) = 2^{73} - 512$ bits, and outputs the hash value of 256 bits. It internally uses a compression function CF and a finalization function FF, where

$$\begin{cases} CF(\cdot,\cdot) : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}, \\ FF(\cdot,\cdot) : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}. \end{cases}$$

The compression function CF is defined in Sec. 2.3.2 and a finalization function FF is defined in Sec. 2.3.3.

Now AURORA-256 works as follows.

Step 1. The input message M is padded with the following padding function $Pad(\cdot)$;

$$Pad(M) = M \,\|\, 1 \,\|\, 0^b \,\|\, Len_{64}, \tag{2.10}$$

where b is the minimum non-negative integer (possibly zero) such that $|M| + b + 65 = 512m$ for some integer m, and Len_{64} is an encoding of $\lceil |M|/512 \rceil$ in 64-bit string. That is, Len_{64} is the length of M in blocks, where a partial block counts for one block, and b is the minimal integer such that the total length of $Pad(M)$ is a multiple of 512 bits. Then $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., we let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Let $H_{0\,(256)} = 0^{256}$, and compute $H_{1\,(256)}, H_{2\,(256)}, \ldots, H_{m-1\,(256)}$ by iterating

$$H_{i+1} \leftarrow CF(H_i, M_i)$$

for $i = 0$ to $m - 2$.

Note that when $Pad(M)$ has one block (i.e., when $m = 1$ and $Pad(M) = M_0$), then Step 2 is not executed.

Step 3. Finally, let $H_m \leftarrow FF(H_{m-1}, M_{m-1})$, and the output is $H_{m\,(256)}$.

See Fig. 2.6 for an illustration and Fig. 2.17 for a pseudocode.

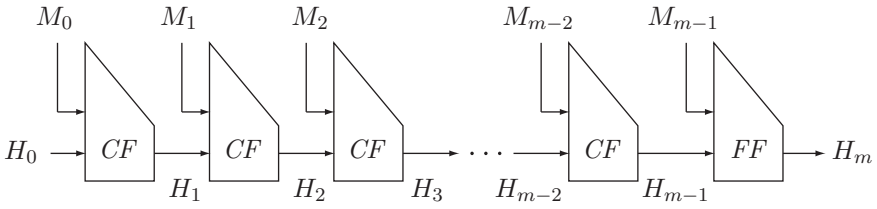

Figure 2.6: AURORA-256.

2.3.2 Compression Function: CF

The compression function, CF, takes the chaining value H_i of 256 bits and the input message block M_i of 512 bits, and outputs the chaining value H_{i+1} of 256 bits.

It internally uses two message scheduling functions MS_L and MS_R, a data rotating function DR, and a chaining value processing function CP, where

$$\begin{cases} MS_L(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MS_R(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ DR(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}, \\ CP(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \to \{0,1\}^{256}. \end{cases}$$

These functions are described below.

Components of CF

- MS_L is an instance of MSM described in Sec. 2.2.1, and for any $X \in \{0,1\}^{256}$, it is defined as

$$MS_L(X) = MSM[F_0, F_1](X, \{CONM_{L,j\,(32)}\}_{0 \le j < 32}), \tag{2.11}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONM_{L,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- Similarly, for any $X \in \{0,1\}^{256}$, MS_R is defined as

$$MS_R(X) = MSM[F_2, F_3](X, \{CONM_{R,j\,(32)}\}_{0 \le j < 32}), \tag{2.12}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM_{R,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.

- CP is an instance of CPM described in Sec. 2.2.2, and for any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, it is defined as

$$CP(X, Y) = CPM[F_1, F_0](X, Y, \{CONC_{j\,(32)}\}_{0 \le j < 68}), \tag{2.13}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC_{j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of CF

Now we present the specification of CF.

Step 1. Let $(M_{L\,(256)}, M_{R\,(256)}) \leftarrow M_{i\,(512)}$, and let $X_{(256)} \leftarrow H_{i\,(256)}$.

Step 2. Let $\{T_{L,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_L(M_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_R(M_{R\,(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72})$.

Step 5. Let $Y_{(256)} \leftarrow CP(X_{(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 6. Finally, the output is $H_{i+1\,(256)} \leftarrow Y_{(256)} \oplus X_{(256)}$.

See Fig. 2.7 for an illustration and Fig. 2.18 for a pseudocode.

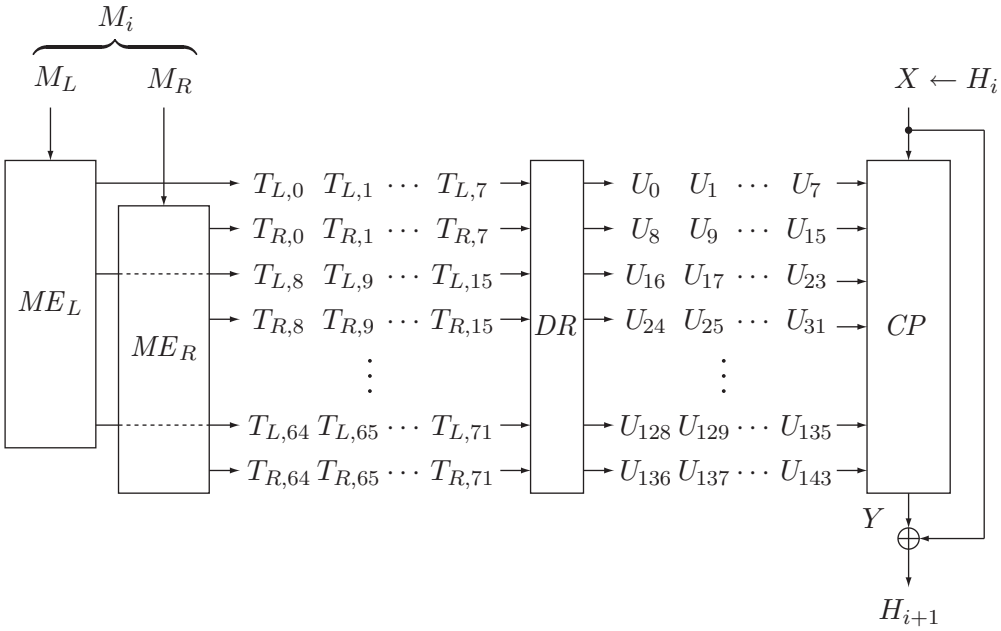

Figure 2.7: $H_{i+1\,(256)} \leftarrow CF(H_{i\,(256)}, M_{i\,(512)})$.

2.3.3 Finalization Function: *FF*

The finalization function, *FF*, is used at the last step of the hash value computation. It takes the chaining value H_{m-1} of 256 bits and the last input message block M_{m-1} of 512 bits, and outputs the final hash value H_m of 256 bits.

FF is structurally equivalent to *CF*, and the only difference is the constants used in the components.

FF internally uses message scheduling functions for finalization, MSF_L and MSF_R, a data rotating function *DR*, and a chaining value processing function for finalization, *CPF*. They have the following syntax.

$$\begin{cases} MSF_L(\cdot) : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\ MSF_R(\cdot) : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\ DR(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \rightarrow (\{0,1\}^{32})^{144}, \\ CPF(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \rightarrow \{0,1\}^{256}. \end{cases} \tag{2.14}$$

These functions are described below.

Components of *FF*

- For any $X \in \{0,1\}^{256}$, MSF_L is defined as

$$MSF_L(X) = MSM[F_0, F_1](X, \{CONM_{L,j\,(32)}\}_{32 \le j < 64}), \tag{2.15}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONM_{L,j\,(32)}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0,1\}^{256}$, MSF_R is defined as

$$MSF_R(X) = MSM[F_2, F_3](X, \{CONM_{R,j\,(32)}\}_{32 \le j < 64}), \tag{2.16}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM_{R,j\,(32)}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.
- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, CPF is defined as

$$CPF(X,Y) = CPM[F_1, F_0](X, Y, \{CONC_{j\,(32)}\}_{68 \le j < 136}), \tag{2.17}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC_{j\,(32)}\}_{68 \le j < 136}$ is the set of constants defined in Sec. 2.9.

Specification of FF

Now the finalization function FF works as follows.

Step 1. Let $(M_{L\,(256)}, M_{R\,(256)}) \leftarrow M_{m-1\,(512)}$, and let $X_{(256)} \leftarrow H_{m-1\,(256)}$.

Step 2. Let $\{T_{L,j\,(32)}\}_{0 \le j < 72} \leftarrow MSF_L(M_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MSF_R(M_{R\,(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72})$.

Step 5. Let $Y_{(256)} \leftarrow CPF(X_{(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 6. Finally, the output is $H_{m\,(256)} \leftarrow Y_{(256)} \oplus X_{(256)}$.

See Fig. 2.19 for a pseudocode.

2.3.4 Alternate Method for Computing CF and FF

The compression function CF and the finalization function FF, components of AURORA-256 hash computation method, are described in an alternative way which requires less memory space in implementation. Firstly, three component functions $RoundC, RoundM_L$ and $RoundM_R$ are defined here for an alternate computation method.

Components $RoundC, RoundM_L$ and $RoundM_R$

$RoundC^{(i)}(\cdot) : (\{0,1\}^{32})^8 \rightarrow (\{0,1\}^{32})^8$ is a round function of the structure for CP. Now we present the computation steps of $RoundC^{(i)}(\cdot)$.

$$\begin{cases} RoundC^{(i)}(X_0, X_1, \ldots, X_7) : \\ (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F_1(X_0), F_0(X_2), F_1(X_4), F_0(X_6)) \\ \text{If } i \neq 17, \text{ do the following line} \\ \quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONC_{4i}, CONC_{4i+1}, CONC_{4i+2}, CONC_{4i+3}) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ Output \ \ (X_0, X_1, \ldots, X_7) \end{cases}$$

Similarly, round functions $RoundM_L$ and $RoundM_R$ for MS_L and MS_R are defined by replacing F-functions and constants as follows.

$$\begin{cases} RoundM_L^{(i)}(X_0, X_1, \ldots, X_7) : \\ (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F_0(X_0), F_1(X_2), F_0(X_4), F_1(X_6)) \\ \text{If } i \neq 8, \text{ do the following line} \\ \quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM_{L,4i}, CONM_{L,4i+1}, CONM_{L,4i+2}, CONM_{L,4i+3}) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ Output \ \ (X_0, X_1, \ldots, X_7) \end{cases}$$

$$\begin{cases} RoundM_R^{(i)}(X_0, X_1, \ldots, X_7) : \\ (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F_2(X_0), F_3(X_2), F_2(X_4), F_3(X_6)) \\ \text{If } i \neq 8, \text{ do the following line} \\ \quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM_{R,4i}, CONM_{R,4i+1}, CONM_{R,4i+2}, CONM_{R,4i+3}) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ Output \ \ (X_0, X_1, \ldots, X_7) \end{cases}$$

Alternative Specification of CF

Now we present an alternative computation method of CF.

Step 1. Initialize input values.

$$\begin{cases} (X_{0(32)}, X_{1(32)}, \ldots, X_{7(32)}, Y_{0(32)}, Y_{1(32)}, \ldots, Y_{7(32)}) \leftarrow M_{i(512)} \\ (Z_{0(32)}, Z_{1(32)}, \ldots, Z_{7(32)}) \leftarrow H_{i\,(256)} \end{cases}$$

Step 2. Add constant values to the initial values.

$$\begin{cases} (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM_{L,0}, CONM_{L,1}, CONM_{L,2}, CONM_{L,3}) \\ (Y_1, Y_3, Y_5, Y_7) \leftarrow (Y_1, Y_3, Y_5, Y_7) \oplus (CONM_{R,0}, CONM_{R,1}, CONM_{R,2}, CONM_{R,3}) \\ (Z_1, Z_3, Z_5, Z_7) \leftarrow (Z_1, Z_3, Z_5, Z_7) \oplus (CONC_0, CONC_1, CONC_2, CONC_3) \end{cases}$$

Step 3. Do the first round function.

$$\begin{cases} (Z_0, Z_1, \ldots, Z_7) \leftarrow (Z_0, Z_1, \ldots, Z_7) \oplus (X_0, X'_1, X_2, X'_3, X_4, X_5, X_6, X_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow \mathit{RoundC}^{(1)}(Z_0, Z_1, \ldots, Z_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow (Z_0, Z_1, \ldots, Z_7) \oplus (Y_0, Y'_1, Y_2, Y'_3, Y_4, Y_5, Y_6, Y_7) \end{cases}$$

Step 4. The following operations are iterated for $j = 1$ to 8.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow \mathit{RoundM}_L^{(j)}(X_0, X_1, \ldots, X_7) \\ (Y_0, Y_1, \ldots, Y_7) \leftarrow \mathit{RoundM}_R^{(j)}(Y_0, Y_1, \ldots, Y_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow \mathit{RoundC}^{(2j)}(Z_0, Z_1, \ldots, Z_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow (Z_0, Z_1, \ldots, Z_7) \oplus (X_0, X'_1, X_2, X'_3, X_4, X_5, X_6, X_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow \mathit{RoundC}^{(2j+1)}(Z_0, Z_1, \ldots, Z_7) \\ (Z_0, Z_1, \ldots, Z_7) \leftarrow (Z_0, Z_1, \ldots, Z_7) \oplus (Y_0, Y'_1, Y_2, Y'_3, Y_4, Y_5, Y_6, Y_7) \end{cases}$$

Step 5. Finally, the output is $H_{i+1\,(256)} \leftarrow (Z_0, Z_1, \ldots, Z_7) \oplus H_i$.

In the above specification, X'_1, X'_3, Y'_1 and Y'_3 are defined as $(X'_1 \,\|\, X'_3) = (X_1 \,\|\, X_3) \lll_{64} 1$ and $(Y'_1 \,\|\, Y'_3) = (Y_1 \,\|\, Y_3) \ggg_{64} 1$.

Alternative Specification of *FF*

An alternative specification of *FF* is obtained by replacing constants in the specification of *CF* as $\mathit{CONC}_j \leftarrow \mathit{CONC}_{j+32}$, $\mathit{CONM}_{L,j} \leftarrow \mathit{CONM}_{L,j+32}$ and $\mathit{CONM}_{R,j} \leftarrow \mathit{CONM}_{R,j+32}$.

2.4 Specification of AURORA-224

AURORA-224 takes the input message of length at most $512 \times (2^{64} - 1) = 2^{73} - 512$ bits, and outputs the hash value of 224 bits. It uses the same padding function Pad, the compression function CF, and the finalization function FF as AURORA-256 defined in Sec. 2.3.

The difference is that AURORA-224 uses $H_0 = 1^{256}$ as the initial value, and the output of FF is truncated to 224 bits by the truncation function TF_{224}.

The truncation function, $TF_{224}(\cdot) : \{0,1\}^{256} \to \{0,1\}^{224}$, first parses the input $H_{m\,(256)}$ into a sequence of bytes $H_{m\,(256)} = (m_{0\,(8)}, m_{1\,(8)}, \ldots, m_{31\,(8)})$ and drops m_7, m_{15}, m_{23}, and m_{31} to produce the 224-bit hash value $H'_{m\,(224)} = (m'_{0\,(8)}, m'_{1\,(8)}, \ldots, m'_{27\,(8)})$. That is, for the 256-bit input $H_{m\,(256)} = (m_{0\,(8)}, m_{1\,(8)}, \ldots, m_{31\,(8)})$, the 224-bit output is $H'_{m\,(224)} = (m'_{0\,(8)}, m'_{1\,(8)}, \ldots, m'_{27\,(8)})$, where

$$\begin{cases} m'_i = m_i & \text{for } 0 \le i \le 6 \\ m'_i = m_{i+1} & \text{for } 7 \le i \le 13 \\ m'_i = m_{i+2} & \text{for } 14 \le i \le 20 \\ m'_i = m_{i+3} & \text{for } 21 \le i \le 27 \end{cases}$$

Now we describe the specification of AURORA-224.

Step 1. The input message M is first padded with $Pad(\cdot)$ in (2.10), and the result of $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Let $H_{0\,(256)} = 1^{256}$, and compute $H_{1\,(256)}, H_{2\,(256)}, \ldots, H_{m-1\,(256)}$ by iterating

$$H_{i+1} \leftarrow CF(H_i, M_i)$$

for $i = 0$ to $m - 2$.

Note that when $Pad(M)$ has one block (i.e., when $m = 1$ and $Pad(M) = M_0$), then Step 2 is not executed.

Step 3. Let $H_m \leftarrow FF(H_{m-1}, M_{m-1})$, and the output is $H'_{m\,(224)} \leftarrow TF_{224}(H_{m\,(256)})$.

See Fig. 2.20 for a pseudocode.

2.5 Specification of AURORA-512

2.5.1 Overall Structure

AURORA-512 takes the input message of length at most $512\times(2^{64}-1) = 2^{73}-512$ bits, and outputs the hash value of 512 bits. It internally uses eight compression functions $CF_0, CF_1, \ldots, CF_7$, a mixing function MF, and a mixing function for finalization MFF, where

$$\begin{cases} CF_s(\cdot,\cdot) : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512} \text{ for } s \in \{0,1,\ldots,7\}, \\ MF(\cdot) : \{0,1\}^{512} \to \{0,1\}^{512}, \\ MFF(\cdot) : \{0,1\}^{512} \to \{0,1\}^{512}. \end{cases}$$

The compression functions $CF_0, CF_1, \ldots, CF_7$ are defined in Sec. 2.5.2, the mixing function MF is defined in Sec. 2.5.3, and the mixing function for finalization MFF is defined in Sec. 2.5.4.

Now we describe the specification of AURORA-512.

Step 1. The input message M is padded with the padding function $Pad(\cdot)$ in (2.10), and $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Now let $H_{0\,(512)} \leftarrow 0^{512}$. Then compute $H_{1\,(512)}, H_{2\,(512)}, \ldots, H_{m\,(512)}$ by iterating the following operations for $i = 0$ to $m-1$.

$$\begin{cases} H_{i+1} \leftarrow CF_{i \bmod 8}(H_i, M_i) \\ \text{if } (0 < i < m-1) \wedge (i \bmod 8 = 7) \text{ then } H_{i+1} \leftarrow MF(H_{i+1}) \end{cases}$$

Step 3. Finally, the output is $H_{m\,(512)} \leftarrow MFF(H_{m\,(512)})$.

See Fig. 2.8 for an illustration and Fig. 2.21 for a pseudocode.

2.5.2 Compression Functions: $CF_0, CF_1, \ldots, CF_7$

The compression function, CF_s, where $s \in \{0,1,\ldots,7\}$, takes the chaining value H_i of 512 bits and the input message block M_i of 512 bits, and outputs the chaining value H_{i+1} of 512 bits.

For each $s \in \{0,1,\ldots,7\}$, CF_s internally uses two message scheduling functions $MS_{L,s}$ and $MS_{R,s}$, a data rotating function DR, and two chaining value processing functions $CP_{L,s}$ and $CP_{R,s}$, where

$$\begin{cases} MS_{L,s}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MS_{R,s}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ DR(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}, \\ CP_{L,s}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \to \{0,1\}^{256}, \\ CP_{R,s}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \to \{0,1\}^{256}. \end{cases}$$

These functions are defined below.

Components of $CF_0, CF_1, \ldots, CF_7$

- For any $X \in \{0,1\}^{256}$, $MS_{L,s}$ is defined as

$$MS_{L,s}(X) = MSM[F_0, F_1](X, \{CONM_{L,s,j\,(32)}\}_{0\le j<32}), \tag{2.18}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONM_{L,s,j\,(32)}\}_{0\le j<32}$ is the set of constants defined in Sec. 2.9.

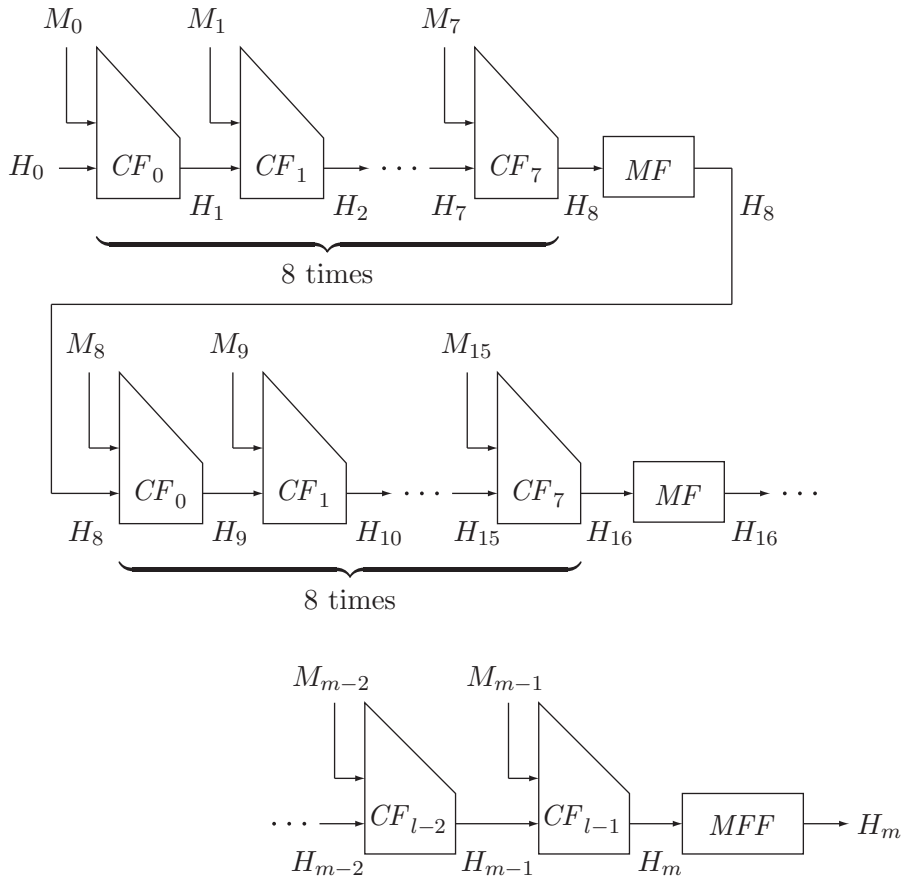

Figure 2.8: AURORA-512, where $l = m \bmod 8$.

- For any $X \in \{0,1\}^{256}$, $MS_{R,s}$ is defined as

$$MS_{R,s}(X) = MSM[F_2, F_3](X, \{CONM_{R,s,j\,(32)}\}_{0 \le j < 32}), \tag{2.19}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM_{R,s,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{L,s}$ is defined as

$$CP_{L,s}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC_{L,s,j\,(32)}\}_{0 \le j < 68}), \tag{2.20}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC_{L,s,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{R,s}$ is defined as

$$CP_{R,s}(X, Y) = CPM[F_3, F_2](X, Y, \{CONC_{R,s,j\,(32)}\}_{0 \le j < 68}), \tag{2.21}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONC_{R,s,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of $CF_0, CF_1, \ldots, CF_7$

Now the compression function CF_s works as follows.

Step 1. Let $(M_{L\,(256)}, M_{R\,(256)}) \leftarrow M_{i\,(512)}$, and $(X_{L\,(256)}, X_{R\,(256)}) \leftarrow H_{i\,(512)}$.

Step 2. Let $\{T_{L,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_{L,s}(M_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_{R,s}(M_{R\,(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72})$.

Step 5. Let $Y_{L\,(256)} \leftarrow CP_{L,s}(X_{L\,(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 6. Let $Y_{R\,(256)} \leftarrow CP_{R,s}(X_{R\,(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 7. Finally, the output is $H_{i+1\,(512)} \leftarrow (Y_{L\,(256)} \oplus X_{L\,(256)} \,\|\, Y_{R\,(256)} \oplus X_{R\,(256)})$.

See Fig. 2.9 for an illustration and Fig. 2.22 for a pseudocode.

2.5.3 Mixing Function: *MF*

The mixing function MF is used to mix the chaining values every after eight calls of CF_s. It takes the chaining value H_i of 512 bits and outputs the updated chaining value H_i of 512 bits. It internally uses two message scheduling functions $MS_{L,8}$ and $MS_{R,8}$, a data rotating function DR, and two chaining value processing functions $CP_{L,8}$ and $CP_{R,8}$, where

$$\begin{cases} MS_{L,8}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MS_{R,8}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ DR(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}, \\ CP_{L,8}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \to \{0,1\}^{256}, \\ CP_{R,8}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \to \{0,1\}^{256}. \end{cases}$$

These functions are defined below.

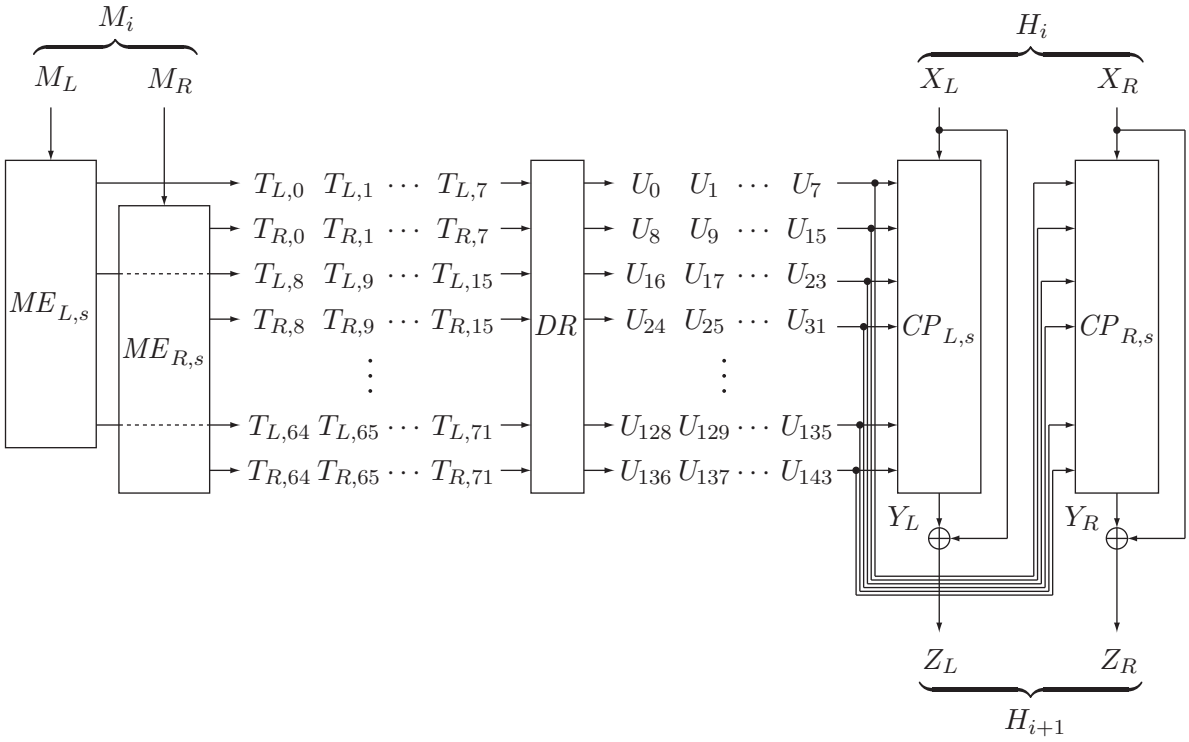

Figure 2.9: $H_{i+1\,(512)} \leftarrow CF_s(H_{i\,(512)}, M_{i\,(512)})$.

Components of MF

- For any $X \in \{0,1\}^{256}$, $MS_{L,8}$ is defined as

$$MS_{L,8}(X) = MSM[F_0, F_1](X, \{CONM_{L,8,j\,(32)}\}_{0 \le j < 32}), \tag{2.22}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONM_{L,8,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0,1\}^{256}$, $MS_{R,8}$ is defined as

$$MS_{R,8}(X) = MSM[F_2, F_3](X, \{CONM_{R,8,j\,(32)}\}_{0 \le j < 32}), \tag{2.23}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM_{R,8,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{L,8}$ is defined as

$$CP_{L,8}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC_{L,8,j\,(32)}\}_{0 \le j < 68}), \tag{2.24}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC_{L,8,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{R,8}$ is defined as

$$CP_{R,8}(X, Y) = CPM[F_3, F_2](X, Y, \{CONC_{R,8,j\,(32)}\}_{0 \le j < 68}), \tag{2.25}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONC_{R,8,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of *MF*

Now we describe the specification of *MF*.

Step 1. Let $(X_{L\,(256)}, X_{R\,(256)}) \leftarrow H_{i\,(512)}$.

Step 2. Let $\{T_{L,j\,(32)}\}_{0\le j<72} \leftarrow MS_{L,8}(X_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0\le j<72} \leftarrow MS_{R,8}(X_{R\,(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0\le j<72}, \{T_{R,j\,(32)}\}_{0\le j<72})$.

Step 5. Let $Y_{L\,(256)} \leftarrow CP_{L,8}(X_{L\,(256)}, \{U_{j\,(32)}\}_{0\le j<144})$.

Step 6. Let $Y_{R\,(256)} \leftarrow CP_{R,8}(X_{R\,(256)}, \{U_{j\,(32)}\}_{0\le j<144})$.

Step 7. Finally, the output is $H_{i\,(512)} \leftarrow (Y_{L\,(256)} \oplus X_{L\,(256)} \,\|\, Y_{R\,(256)} \oplus X_{R\,(256)})$.

See Fig. 2.10 for an illustration and Fig. 2.23 for a pseudocode.

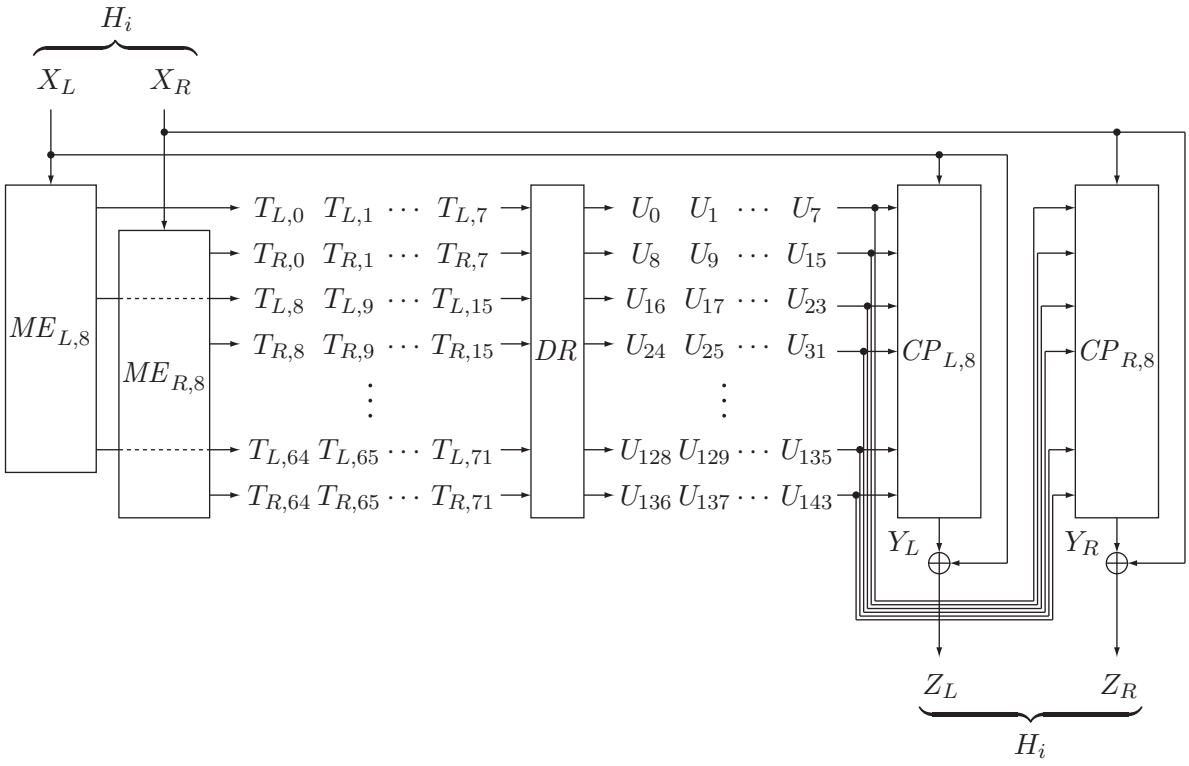

Figure 2.10: $H_{i\,(512)} \leftarrow MF(H_{i\,(512)})$.

2.5.4 Mixing Function for Finalization: *MFF*

The mixing function for finalization *MFF* is used at the last computation of the final hash value. *MFF* is structurally equivalent to *MF*, and the only difference is the constants used in the components. It takes the last chaining value H_m of 512 bits and outputs the updated value H_m of 512 bits, which is the final hash value. It internally uses two message scheduling functions $MS_{L,9}$

and $MS_{R,9}$, a data rotating function DR, and two chaining value processing functions $CP_{L,9}$ and $CP_{R,9}$, where

$$\begin{cases} MS_{L,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{32} \to (\{0,1\}^{32})^{72}, \\ MS_{R,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{32} \to (\{0,1\}^{32})^{72}, \\ DR : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}, \\ CP_{L,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \times (\{0,1\}^{32})^{68} \to \{0,1\}^{256}, \\ CP_{R,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \times (\{0,1\}^{32})^{68} \to \{0,1\}^{256}. \end{cases} \tag{2.26}$$

These functions are defined below.

Components of *MFF*

- For any $X \in \{0,1\}^{256}$, $MS_{L,9}$ is defined as

$$MS_{L,9}(X) = MSM[F_0, F_1](X, \{CONM_{L,9,j\,(32)}\}_{0 \le j < 32}), \tag{2.27}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONM_{L,9,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- Similarly, for any $X \in \{0,1\}^{256}$, $MS_{R,9}$ is defined as

$$MS_{R,9}(X) = MSM[F_2, F_3](X, \{CONM_{R,9,j\,(32)}\}_{0 \le j < 32}), \tag{2.28}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM_{R,9,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{L,9}$ is defined as

$$CP_{L,9}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC_{L,9,j\,(32)}\}_{0 \le j < 68}), \tag{2.29}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC_{L,9,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP_{R,9}$ is defined as

$$CP_{R,9}(X, Y) = CPM[F_3, F_2](X, Y, \{CONC_{R,9,j\,(32)}\}_{0 \le j < 68}), \tag{2.30}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONC_{R,9,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of *MFF*

Now we describe the specification of *MFF*.

Step 1. Let $(X_{L\,(256)}, X_{R\,(256)}) \leftarrow H_{m\,(512)}$.

Step 2. Let $\{T_{L,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_{L,9}(X_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_{R,9}(X_{R\,(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72})$.

Step 5. Let $Y_{L\,(256)} \leftarrow CP_{L,9}(X_{L\,(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 6. Let $Y_{R\,(256)} \leftarrow CP_{R,9}(X_{R\,(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 7. Finally, the output is $H_{m\,(512)} \leftarrow (Y_{L\,(256)} \oplus X_{L\,(256)} \,\|\, Y_{R\,(256)} \oplus X_{R\,(256)})$.

See Fig. 2.24 for a pseudocode.

2.6 Specification of AURORA-384

AURORA-384 takes the input message of length at most $512 \times (2^{64} - 1) = 2^{73} - 512$ bits, and outputs the hash value of 384 bits. It uses the same padding function Pad, the compression functions $CF_0, CF_1, \ldots, CF_7$, the mixing function MF, and the mixing function for finalization MFF as AURORA-512 defined in Sec. 2.5.

The difference is that AURORA-384 uses $H_0 = 1^{512}$ as the initial value, and the output of MFF is truncated to 384 bits by the truncation function TF_{384}.

The truncation function, $TF_{384}(\cdot) : \{0,1\}^{512} \to \{0,1\}^{384}$, first parses the input $H_{m\,(512)}$ into a sequence of bytes $H_{m\,(512)} = (m_{0\,(8)}, m_{1\,(8)}, \ldots, m_{63\,(8)})$ and drops the following bytes;

$$m_6, m_7, m_{14}, m_{15}, m_{22}, m_{23}, m_{30}, m_{31}, m_{38}, m_{39}, m_{46}, m_{47}, m_{54}, m_{55}, m_{62}, m_{63},$$

to produce the 384-bit hash value $H'_{m\,(384)} = (m'_{0\,(8)}, m'_{1\,(8)}, \ldots, m'_{47\,(8)})$.

That is, for the 512-bit input $H_{m\,(512)} = (m_{0\,(8)}, m_{1\,(8)}, \ldots, m_{63\,(8)})$, the 384-bit output is $H'_{m\,(384)} = (m'_{0\,(8)}, m'_{1\,(8)}, \ldots, m'_{47\,(8)})$, where

$$\begin{cases} m'_i = m_i & \text{for } 0 \le i \le 5 \\ m'_i = m_{i+2} & \text{for } 6 \le i \le 11 \\ m'_i = m_{i+4} & \text{for } 12 \le i \le 17 \\ m'_i = m_{i+6} & \text{for } 18 \le i \le 23 \\ m'_i = m_{i+8} & \text{for } 24 \le i \le 29 \\ m'_i = m_{i+10} & \text{for } 30 \le i \le 35 \\ m'_i = m_{i+12} & \text{for } 36 \le i \le 41 \\ m'_i = m_{i+14} & \text{for } 42 \le i \le 47 \end{cases}$$

Now we describe the specification of AURORA-384.

Step 1. The input message M is first padded with $Pad(\cdot)$ in (2.10), and the result of $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Let $H_{0\,(512)} = 1^{512}$, and compute $H_{1\,(512)}, H_{2\,(512)}, \ldots, H_{m\,(512)}$ by iterating

$$\begin{cases} H_{i+1} \leftarrow CF_{i \bmod 8}(H_i, M_i) \\ \text{if } (0 < i < m-1) \wedge (i \bmod 8 = 7) \text{ then } H_{i+1} \leftarrow MF(H_{i+1}) \end{cases}$$

for $i = 0$ to $m - 1$.

Step 3. Let $H_{m\,(512)} \leftarrow MFF(H_{m\,(512)})$, and output $H'_{m\,(384)} \leftarrow TF_{384}(H_{m\,(512)})$.

See Fig. 2.25 for a pseudocode.

2.7 Specification of AURORA-256M (optional)

2.7.1 Overall Structure

AURORA-256M takes the input message of length at most $512 \times (2^{64} - 1) = 2^{73} - 512$ bits, and outputs the hash value of 256 bits. It internally uses eight compression functions $CF_0^M, CF_1^M, \ldots, CF_7^M$, a mixing function MF^M, and a mixing function for finalization MFF^M, where

$$\begin{cases} CF_s^M(\cdot,\cdot) : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512} \text{ for } s \in \{0,1,\ldots,7\}, \\ MF^M(\cdot) : \{0,1\}^{512} \to \{0,1\}^{512}, \\ MFF^M(\cdot) : \{0,1\}^{512} \to \{0,1\}^{512}. \end{cases}$$

Basically, AURORA-256M is structurally very similar to AURORA-512. CF_s^M and MF^M are the same as CF_s and MF, except for constants used in their components, while the output of MFF^M is 256 bits instead of 512 bits for MFF.

The compression functions $CF_0^M, CF_1^M, \ldots, CF_7^M$ are defined in Sec. 2.7.2, the mixing function MF^M is defined in Sec. 2.7.3, and the mixing function for finalization MFF^M is defined in Sec. 2.7.4.

Now we describe the specification of AURORA-256M.

Step 1. The input message M is padded with the padding function $Pad(\cdot)$ in (2.10). Then $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Now let $H_{0\,(512)} \leftarrow 0^{512}$. Then compute $H_{1\,(512)}, H_{2\,(512)}, \ldots, H_{m\,(512)}$ by iterating the following operations for $i = 0$ to $m - 1$.

$$\begin{cases} H_{i+1} \leftarrow CF_{i \bmod 8}^M(H_i, M_i) \\ \text{if } (0 < i < m-1) \wedge (i \bmod 8 = 7) \text{ then } H_{i+1} \leftarrow MF^M(H_{i+1}) \end{cases}$$

Step 3. Finally, the output is $H'_{m\,(256)} \leftarrow MFF^M(H_{m\,(512)})$.

See Fig. 2.11 for an illustration and Fig. 2.26 for a pseudocode.

2.7.2 Compression Functions: $CF_0^M, CF_1^M, \ldots, CF_7^M$

The compression function, CF_s^M, where $s \in \{0,1,\ldots,7\}$, takes the chaining value H_i of 512 bits and the input message block M_i of 512 bits, and outputs the chaining value H_{i+1} of 512 bits.

For each $s \in \{0,1,\ldots,7\}$, CF_s^M internally uses two message scheduling functions $MS_{L,s}^M$ and $MS_{R,s}^M$, a data rotating function DR, and two chaining value processing functions $CP_{L,s}^M$ and $CP_{R,s}^M$. These functions are equivalent to the corresponding functions in Sec. 2.5.2 for AURORA-512, where we use

- $CONM_{L,s,j\,(32)}^M$ for $MS_{L,s}^M$ instead of $CONM_{L,s,j\,(32)}$ for $MS_{L,s}$,
- $CONM_{R,s,j\,(32)}^M$ for $MS_{R,s}^M$ instead of $CONM_{R,s,j\,(32)}$ for $MS_{R,s}$,
- $CONC_{L,s,j\,(32)}^M$ for $CP_{L,s}^M$ instead of $CONC_{L,s,j\,(32)}$ for $CP_{L,s}$, and
- $CONC_{R,s,j\,(32)}^M$ for $CP_{R,s}^M$ instead of $CONC_{R,s,j\,(32)}$ for $CP_{R,s}$.

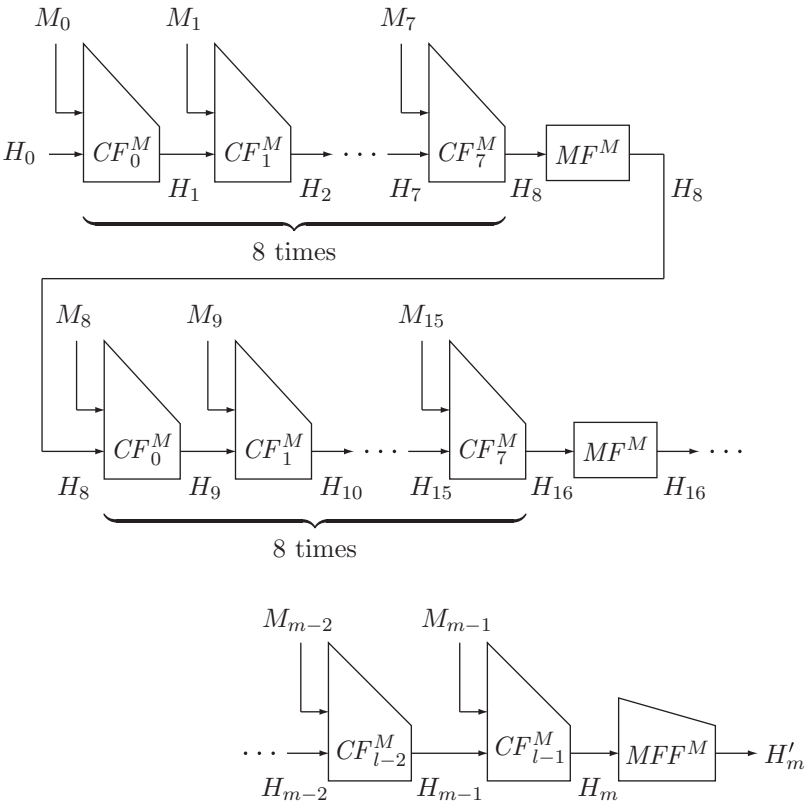

Figure 2.11: AURORA-256M, where $l = m \bmod 8$.

The constants, $CONM^M_{L,s,j\,(32)}$, $CONM^M_{R,s,j\,(32)}$, $CONC^M_{L,s,j\,(32)}$, and $CONC^M_{R,s,j\,(32)}$ are all defined in Sec. 2.9. Below, we present the specification, and show the pseudocode in Fig. 2.27 for completeness.

$$MS^M_{L,s}(X) = MSM[F_0, F_1](X, \{CONM^M_{L,s,j\,(32)}\}_{0\le j<32}), \tag{2.31}$$
$$MS^M_{R,s}(X) = MSM[F_2, F_3](X, \{CONM^M_{R,s,j\,(32)}\}_{0\le j<32}), \tag{2.32}$$
$$CP^M_{L,s}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC^M_{L,s,j\,(32)}\}_{0\le j<68}), \tag{2.33}$$
$$CP^M_{R,s}(X, Y) = CPM[F_3, F_2](X, Y, \{CONC^M_{R,s,j\,(32)}\}_{0\le j<68}). \tag{2.34}$$

In the above specification, F_0, F_1, F_2 and F_3 are F-functions defined in Sec. 2.2.4.

2.7.3 Mixing Function: MF^M

The mixing function MF^M is used to mix the chaining values every after eight calls of CF^M_s. It takes the chaining value H_i of 512 bits and outputs the updated chaining value H_i of 512 bits. It internally uses two message scheduling functions $MS^M_{L,8}$ and $MS^M_{R,8}$, a data rotating function DR, and two chaining value processing functions $CP^M_{L,8}$ and $CP^M_{R,8}$. These functions are equivalent to the corresponding functions in Sec. 2.5.3 for AURORA-512, where we use

- $CONM^M_{L,8,j\,(32)}$ for $MS^M_{L,8}$ instead of $CONM_{L,8,j\,(32)}$ for $MS_{L,8}$,
- $CONM^M_{R,8,j\,(32)}$ for $MS^M_{R,8}$ instead of $CONM_{R,8,j\,(32)}$ for $MS_{R,8}$,
- $CONC^M_{L,8,j\,(32)}$ for $CP^M_{L,8}$ instead of $CONC_{L,s,j\,(32)}$ for $CP_{L,s}$, and
- $CONC^M_{R,8,j\,(32)}$ for $CP^M_{R,8}$ instead of $CONC_{R,8,j\,(32)}$ for $CP_{R,8}$.

The constants, $CONM^M_{L,8,j\,(32)}$, $CONM^M_{R,8,j\,(32)}$, $CONC^M_{L,8,j\,(32)}$, and $CONC^M_{R,8,j\,(32)}$ are all defined in Sec. 2.9. Below, we present the specification, and show the pseudocode in Fig. 2.28 for completeness.

$$MS^M_{L,8}(X) = MSM[F_0, F_1](X, \{CONM^M_{L,8,j\,(32)}\}_{0\le j<32}), \tag{2.35}$$
$$MS^M_{R,8}(X) = MSM[F_2, F_3](X, \{CONM^M_{R,8,j\,(32)}\}_{0\le j<32}), \tag{2.36}$$
$$CP^M_{L,8}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC^M_{L,8,j\,(32)}\}_{0\le j<68}), \tag{2.37}$$
$$CP^M_{R,8}(X, Y) = CPM[F_3, F_2](X, Y, \{CONC^M_{R,8,j\,(32)}\}_{0\le j<68}). \tag{2.38}$$

2.7.4 Mixing Function for Finalization: MFF^M

The mixing function for finalization MFF^M is used at the last computation of the final hash value. It takes the last chaining value H_m of 512 bits and outputs the final hash value H'_m of 256 bits. It internally uses a message scheduling function $MS^M_{R,9}$, a data rotating function DR, and a chaining value processing function $CP^M_{L,9}$, where

$$\begin{cases} MS^M_{R,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{32} \to (\{0,1\}^{32})^{72}, \\ DR : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}, \\ CP^M_{L,9} : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \times (\{0,1\}^{32})^{68} \to \{0,1\}^{256}. \end{cases} \tag{2.39}$$

These functions are defined below.

Components of MFF^M

- For any $X \in \{0,1\}^{256}$, $MS^M_{L,9}$ is defined as

$$MS^M_{R,9}(X) = MSM[F_2, F_3](X, \{CONM^M_{R,9,j\,(32)}\}_{0 \le j < 32}), \tag{2.40}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.4, and $\{CONM^M_{R,9,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.9.

- DR is the data rotating function defined in Sec. 2.2.5.

- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, $CP^M_{L,9}$ is defined as

$$CP^M_{L,9}(X, Y) = CPM[F_1, F_0](X, Y, \{CONC^M_{L,9,j\,(32)}\}_{0 \le j < 68}), \tag{2.41}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.4, and $\{CONC^M_{L,9,j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of MFF^M

Now we describe the specification of MFF^M.

Step 1. Let $(X_{(256)}, Y_{(256)}) \leftarrow H_{m\,(512)}$.

Step 2. Let $T_{L,j\,(32)} \leftarrow 0^{32}$ for $0 \le j < 72$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MS^M_{R,9}(Y_{(256)})$.

Step 4. Let $\{U_{j\,(32)}\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72})$.

Step 5. Let $Z_{(256)} \leftarrow CP^M_{L,9}(X_{(256)}, \{U_{j\,(32)}\}_{0 \le j < 144})$.

Step 6. Finally, the output is $H'_{m\,(256)} \leftarrow Z_{(256)} \oplus X_{(256)}$.

See Fig. 2.12 for an illustration and Fig. 2.29 for a pseudocode.

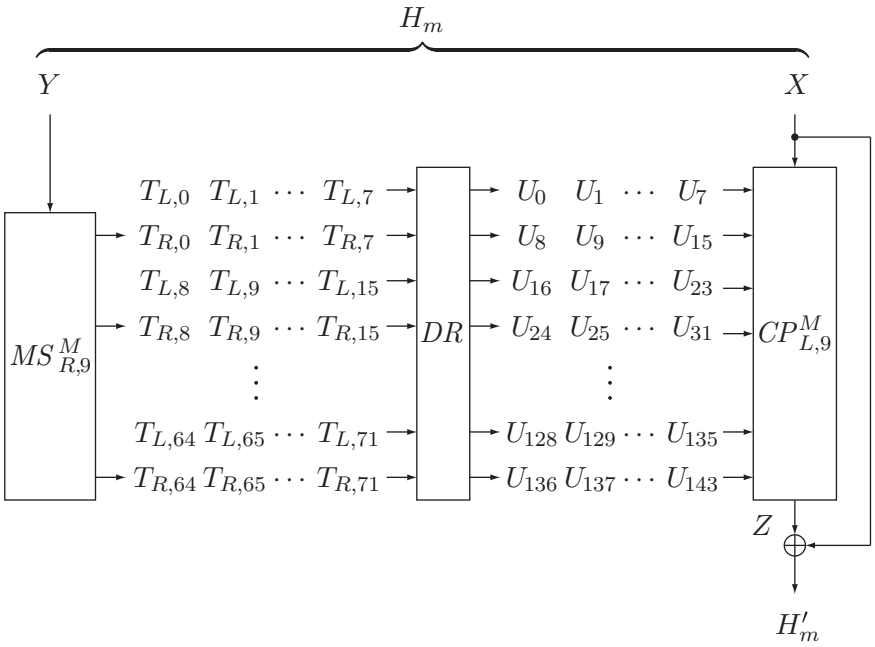

Figure 2.12: $H'_{m\,(256)} \leftarrow MFF^M(H_{m\,(512)})$. Note that $T_{L,j} = 0^{32}$ for $0 \leq j < 72$.

2.8 Specification of AURORA-224M (optional)

AURORA-224M takes the input message of length at most $512 \times (2^{64} - 1) = 2^{73} - 512$ bits, and outputs the hash value of 224 bits. It uses the same padding function Pad, the compression function CF_s^M, the mixing function MF^M, and the mixing function for finalization MFF^M as AURORA-256M defined in Sec. 2.7.

The difference is that AURORA-224M uses $H_0 = 1^{512}$ as the initial value, and the output of MFF^M is truncated to 224 bits by the truncation function TF_{224} in Sec. 2.4.

Now we describe the specification of AURORA-224M.

Step 1. The input message M is padded with the padding function $Pad(\cdot)$ in (2.10). Then $Pad(M)$ is divided into blocks $M_0, M_1, \ldots, M_{m-1}$ each of length 512 bits, i.e., let

$$(M_{0\,(512)}, M_{1\,(512)}, \ldots, M_{m-1\,(512)}) \leftarrow Pad(M).$$

Step 2. Now let $H_{0\,(512)} \leftarrow 1^{512}$. Then compute $H_{1\,(512)}, H_{2\,(512)}, \ldots, H_{m\,(512)}$ by iterating the following operations for $i = 0$ to $m - 1$.

$$\begin{cases} H_{i+1} \leftarrow CF^M_{i \bmod 8}(H_i, M_i) \\ \text{if } (0 < i < m-1) \wedge (i \bmod 8 = 7) \text{ then } H_{i+1} \leftarrow MF^M(H_{i+1}) \end{cases}$$

Step 3. Let $H'_{m\,(256)} \leftarrow MFF^M(H_{m\,(512)})$.

Step 4. Finally, the output is $H''_{m\,(224)} \leftarrow TF_{224}(H'_{m\,(256)})$.

See Fig. 2.30 for a pseudocode.

2.9 Constant Values

This section describes the generation procedures and the lists of constant values.

2.9.1 Constant Values for AURORA-224/256

Following constants are used in AURORA-224/256;

- $\{CONM_{L,j}\}_{0\le j<32}$, $\{CONM_{R,j}\}_{0\le j<32}$, $\{CONC_j\}_{0\le j<68}$ for CF, and
- $\{CONM_{L,j}\}_{32\le j<64}$, $\{CONM_{R,j}\}_{32\le j<64}$, $\{CONC_j\}_{68\le j<136}$ for FF.

Below, we describe the generation process of the constants. The multiplication and the inversion are done in GF(2^{16}) with the primitive polynomial $x^{16}+x^{15}+x^{13}+x^{11}+x^5+x^4+1$, which is `0x1a831`.

Step 1. Let IV_0, IV_1, $mask_0$, $mask_1$, $mask_2$ and $mask_3$ be the following values.

$$\begin{cases} IV_0 \leftarrow (2^{1/2}-1)2^{16} = \texttt{0x6a09} \\ IV_1 \leftarrow (3^{1/2}-1)2^{16} = \texttt{0xbb67} \\ mask_0 \leftarrow (2^{1/3}-1)2^{16} = \texttt{0x428a} \\ mask_1 \leftarrow (3^{1/3}-1)2^{16} = \texttt{0x7137} \\ mask_2 \leftarrow (2^{1/5}-1)2^{16} = \texttt{0x2611} \\ mask_3 \leftarrow (3^{1/5}-1)2^{16} = \texttt{0x3ee8} \end{cases}$$

Step 2. The following operations are iterated for $i = 0$ to 16.

$$\begin{cases} T_{0,i} \leftarrow IV_0 \cdot \texttt{0x0002}^i \\ T_{1,i} \leftarrow IV_1 \cdot \texttt{0x0002}^{-i} \\ CONC_{4i} \leftarrow (T_{0,i} \oplus mask_0 \,\|\, \overline{T_{0,i}} \lll_{16} 8) \\ CONC_{4i+1} \leftarrow (T_{1,i} \oplus mask_1 \,\|\, \overline{T_{1,i}} \lll_{16} 8) \\ CONC_{4i+2} \leftarrow (T_{0,i} \lll_{16} 8 \,\|\, T_{0,i} \oplus mask_2) \\ CONC_{4i+3} \leftarrow (T_{1,i} \lll_{16} 9 \,\|\, T_{1,i} \oplus mask_3) \end{cases}$$

Step 3. The following operations are iterated for $i = 0$ to 7.

$$\begin{cases} CONM_{L,4i} \leftarrow CONC_{8i} \lll_{32} 1 \\ CONM_{L,4i+1} \leftarrow CONC_{8i+1} \lll_{32} 1 \\ CONM_{L,4i+2} \leftarrow CONC_{8i+2} \lll_{32} 1 \\ CONM_{L,4i+3} \leftarrow CONC_{8i+3} \lll_{32} 1 \\ CONM_{R,4i} \leftarrow CONC_{8i+4} \ggg_{32} 1 \\ CONM_{R,4i+1} \leftarrow CONC_{8i+5} \ggg_{32} 1 \\ CONM_{R,4i+2} \leftarrow CONC_{8i+6} \ggg_{32} 1 \\ CONM_{R,4i+3} \leftarrow CONC_{8i+7} \ggg_{32} 1 \end{cases}$$

Step 4. The following operations are iterated for $i = 0$ to 16.

$$\begin{cases} CONC_{4i+68} \leftarrow CONC_{4i} \\ CONC_{4i+69} \leftarrow CONC_{4i+1} \\ CONC_{4i+70} \leftarrow CONC_{4i+2} \\ CONC_{4i+71} \leftarrow CONC_{4i+3} \oplus \texttt{0x01010101} \end{cases}$$

Step 5. The following operations are iterated for $i = 0$ to 7.

$$\begin{cases} CONM_{L,4i+32} \leftarrow CONM_{L,4i} \\ CONM_{L,4i+33} \leftarrow CONM_{L,4i+1} \\ CONM_{L,4i+34} \leftarrow CONM_{L,4i+2} \\ CONM_{L,4i+35} \leftarrow CONM_{L,4i+3} \\ CONM_{R,4i+32} \leftarrow CONM_{R,4i} \\ CONM_{R,4i+33} \leftarrow CONM_{R,4i+1} \\ CONM_{R,4i+34} \leftarrow CONM_{R,4i+2} \\ CONM_{R,4i+35} \leftarrow CONM_{R,4i+3} \end{cases}$$

2.9.2 Constant Values for AURORA-384/512

Following constants are used in AURORA-384/512;

- $\{CONM_{L,s,j}\}_{0\le j<32}$, $\{CONM_{R,s,j}\}_{0\le j<32}$, $\{CONC_{L,s,j}\}_{0\le j<68}$, $\{CONC_{R,s,j}\}_{0\le j<68}$ for CF_s, where $s = 0, 1, \ldots, 7$,
- $\{CONM_{L,8,j}\}_{0\le j<32}$, $\{CONM_{R,8,j}\}_{0\le j<32}$, $\{CONC_{L,8,j}\}_{0\le j<68}$, $\{CONC_{R,8,j}\}_{0\le j<68}$ for MF, and
- $\{CONM_{L,9,j}\}_{0\le j<32}$, $\{CONM_{R,9,j}\}_{0\le j<32}$, $\{CONC_{L,9,j}\}_{0\le j<68}$, $\{CONC_{R,9,j}\}_{0\le j<68}$ for MFF.

These constants are generated with the procedure described below.

Step 1. Let IV_0^{512}, IV_1^{512}, $mask_0^{512}$, $mask_1^{512}$, $mask_2^{512}$ and $mask_3^{512}$ be the following values.

$$\begin{cases} IV_0^{512} \leftarrow (11^{1/2} - 3)2^{16} = \texttt{0x510e} \\ IV_1^{512} \leftarrow (13^{1/2} - 3)2^{16} = \texttt{0x9b05} \\ mask_0^{512} \leftarrow (11^{1/3} - 2)2^{16} = \texttt{0x3956} \\ mask_1^{512} \leftarrow (13^{1/3} - 2)2^{16} = \texttt{0x59f1} \\ mask_2^{512} \leftarrow (11^{1/5} - 1)2^{16} = \texttt{0x9d8a} \\ mask_3^{512} \leftarrow (13^{1/5} - 1)2^{16} = \texttt{0xab97} \end{cases}$$

Step 2. The following operations are iterated for $i = 0$ to 16.

$$\begin{cases} T_{0,i}^{512} \leftarrow IV_0^{512} \cdot \texttt{0x0002}^i \\ T_{1,i}^{512} \leftarrow IV_1^{512} \cdot \texttt{0x0002}^{-i} \\ CONC_{L,0,4i} \leftarrow (T_{0,i}^{512} \oplus mask_0^{512} \,\|\, \overline{T_{0,i}^{512}} \lll_{16} 8) \\ CONC_{L,0,4i+1} \leftarrow (T_{1,i}^{512} \oplus mask_1^{512} \,\|\, \overline{T_{1,i}^{512}} \lll_{16} 8) \\ CONC_{L,0,4i+2} \leftarrow (T_{0,i}^{512} \lll_{16} 8 \,\|\, T_{0,i}^{512} \oplus mask_2^{512}) \\ CONC_{L,0,4i+3} \leftarrow (T_{1,i}^{512} \lll_{16} 9 \,\|\, T_{1,i}^{512} \oplus mask_3^{512}) \end{cases}$$

Step 3. The following operation is iterated for $i = 0$ to 67.

$$CONC_{R,0,i} \leftarrow CONC_{L,0,i} \lll_{32} 3$$

Step 4. The following operations are iterated for $i = 0$ to 7.

$$\begin{cases} CONM_{L,0,4i} \leftarrow CONC_{L,0,8i} \lll_{32} 1 \\ CONM_{L,0,4i+1} \leftarrow CONC_{L,0,8i+1} \lll_{32} 1 \\ CONM_{L,0,4i+2} \leftarrow CONC_{L,0,8i+2} \lll_{32} 1 \\ CONM_{L,0,4i+3} \leftarrow CONC_{L,0,8i+3} \lll_{32} 1 \\ CONM_{R,0,4i} \leftarrow CONC_{L,0,8i+4} \ggg_{32} 1 \\ CONM_{R,0,4i+1} \leftarrow CONC_{L,0,8i+5} \ggg_{32} 1 \\ CONM_{R,0,4i+2} \leftarrow CONC_{L,0,8i+6} \ggg_{32} 1 \\ CONM_{R,0,4i+3} \leftarrow CONC_{L,0,8i+7} \ggg_{32} 1 \end{cases}$$

Step 5. The following operations are iterated for $i = 0$ to 16 and for $s = 1$ to 9.

$$\begin{cases} CONC_{L,s,4i} \leftarrow CONC_{L,0,4i} \\ CONC_{L,s,4i+1} \leftarrow CONC_{L,0,4i+1} \\ CONC_{L,s,4i+2} \leftarrow CONC_{L,0,4i+2} \\ CONC_{L,s,4i+3} \leftarrow CONC_{L,0,4i+3} \oplus CONS_s \\ CONC_{R,s,4i} \leftarrow CONC_{R,0,4i} \\ CONC_{R,s,4i+1} \leftarrow CONC_{R,0,4i+1} \\ CONC_{R,s,4i+2} \leftarrow CONC_{R,0,4i+2} \\ CONC_{R,s,4i+3} \leftarrow CONC_{R,0,4i+3} \oplus CONS_s \end{cases}$$

Each $CONS_s$ is defined as $CONS_1 = \texttt{0x01010101}$, $CONS_2 = \texttt{0x02020202}$, $CONS_3 = \texttt{0x03030303}$, $CONS_4 = \texttt{0x04040404}$, $CONS_5 = \texttt{0x05050505}$, $CONS_6 = \texttt{0x06060606}$, $CONS_7 = \texttt{0x07070707}$, $CONS_8 = \texttt{0x08080808}$, and $CONS_9 = \texttt{0x09090909}$.

Step 6. The following operations are iterated for $i = 0$ to 7 and for $s = 1$ to 9.

$$\begin{cases} CONM_{L,s,4i} \leftarrow CONM_{L,0,4i} \\ CONM_{L,s,4i+1} \leftarrow CONM_{L,0,4i+1} \\ CONM_{L,s,4i+2} \leftarrow CONM_{L,0,4i+2} \\ CONM_{L,s,4i+3} \leftarrow CONM_{L,0,4i+3} \\ CONM_{R,s,4i} \leftarrow CONM_{R,0,4i} \\ CONM_{R,s,4i+1} \leftarrow CONM_{R,0,4i+1} \\ CONM_{R,s,4i+2} \leftarrow CONM_{R,0,4i+2} \\ CONM_{R,s,4i+3} \leftarrow CONM_{R,0,4i+3} \end{cases}$$

2.9.3 Constant Values for AURORA-224M/256M

Following constants are used in AURORA-224M/256M;

- $\{CONM^M_{L,s,j}\}_{0\le j<32}$, $\{CONM^M_{R,s,j}\}_{0\le j<32}$, $\{CONC^M_{L,s,j}\}_{0\le j<68}$, $\{CONC^M_{R,s,j}\}_{0\le j<68}$ for CF^M_s, where $s = 0, 1, \ldots, 7$,
- $\{CONM^M_{L,8,j}\}_{0\le j<32}$, $\{CONM^M_{R,8,j}\}_{0\le j<32}$, $\{CONC^M_{L,8,j}\}_{0\le j<68}$, $\{CONC^M_{R,8,j}\}_{0\le j<68}$ for MF^M, and
- $\{CONM^M_{R,9,j}\}_{0\le j<32}$, $\{CONC^M_{L,9,j}\}_{0\le j<68}$ for MFF^M.

These constants are generated with almost the same procedure as AURORA-384/512.

Step 1. Let IV^M_0, IV^M_1, $mask^M_0$, $mask^M_1$, $mask^M_2$ and $mask^M_3$ be the following values.

$$\begin{cases} IV^M_0 \leftarrow (5^{1/2} - 2)2^{16} = \texttt{0x3c6e} \\ IV^M_1 \leftarrow (7^{1/2} - 2)2^{16} = \texttt{0xa54f} \\ mask^M_0 \leftarrow (5^{1/3} - 1)2^{16} = \texttt{0xb5c0} \\ mask^M_1 \leftarrow (7^{1/3} - 1)2^{16} = \texttt{0xe9b5} \\ mask^M_2 \leftarrow (5^{1/5} - 1)2^{16} = \texttt{0x6135} \\ mask^M_3 \leftarrow (7^{1/5} - 1)2^{16} = \texttt{0x79cc} \end{cases}$$

Step 2. The following operations are iterated for $i = 0$ to 16.

$$\begin{cases} T^M_{0,i} \leftarrow IV^M_0 \cdot \texttt{0x0002}^i \\ T^M_{1,i} \leftarrow IV^M_1 \cdot \texttt{0x0002}^{-i} \\ CONC^M_{L,0,4i} \leftarrow (T^M_{0,i} \oplus mask^M_0 \,\|\, \overline{T^M_{0,i}} \lll_{16} 8) \\ CONC^M_{L,0,4i+1} \leftarrow (T^M_{1,i} \oplus mask^M_1 \,\|\, \overline{T^M_{1,i}} \lll_{16} 8) \\ CONC^M_{L,0,4i+2} \leftarrow (T^M_{0,i} \lll_{16} 8 \,\|\, T^M_{0,i} \oplus mask^M_2) \\ CONC^M_{L,0,4i+3} \leftarrow (T^M_{1,i} \lll_{16} 9 \,\|\, T^M_{1,i} \oplus mask^M_3) \end{cases}$$

Step 3. The following operation is iterated for $i = 0$ to 67.

$$CONC^M_{R,0,i} \leftarrow CONC^M_{L,0,i} \lll_{32} 3$$

Step 4. The following operations are iterated for $i = 0$ to 7.

$$\begin{cases} CONM^M_{L,0,4i} \leftarrow CONC^M_{L,0,8i} \lll_{32} 1 \\ CONM^M_{L,0,4i+1} \leftarrow CONC^M_{L,0,8i+1} \lll_{32} 1 \\ CONM^M_{L,0,4i+2} \leftarrow CONC^M_{L,0,8i+2} \lll_{32} 1 \\ CONM^M_{L,0,4i+3} \leftarrow CONC^M_{L,0,8i+3} \lll_{32} 1 \\ CONM^M_{R,0,4i} \leftarrow CONC^M_{L,0,8i+4} \ggg_{32} 1 \\ CONM^M_{R,0,4i+1} \leftarrow CONC^M_{L,0,8i+5} \ggg_{32} 1 \\ CONM^M_{R,0,4i+2} \leftarrow CONC^M_{L,0,8i+6} \ggg_{32} 1 \\ CONM^M_{R,0,4i+3} \leftarrow CONC^M_{L,0,8i+7} \ggg_{32} 1 \end{cases}$$

Step 5. The following operations are iterated for $i = 0$ to 16 and for $s = 1$ to 9.

$$\begin{cases} CONC^M_{L,s,4i} \leftarrow CONC^M_{L,0,4i} \\ CONC^M_{L,s,4i+1} \leftarrow CONC^M_{L,0,4i+1} \\ CONC^M_{L,s,4i+2} \leftarrow CONC^M_{L,0,4i+2} \\ CONC^M_{L,s,4i+3} \leftarrow CONC^M_{L,0,4i+3} \oplus CONS_s \\ CONC^M_{R,s,4i} \leftarrow CONC^M_{R,0,4i} \\ CONC^M_{R,s,4i+1} \leftarrow CONC^M_{R,0,4i+1} \\ CONC^M_{R,s,4i+2} \leftarrow CONC^M_{R,0,4i+2} \\ CONC^M_{R,s,4i+3} \leftarrow CONC^M_{R,0,4i+3} \oplus CONS_s \end{cases}$$

Each $CONS_s$ is the same as in AURORA-384/512.

Step 6. The following operations are iterated for $i = 0$ to 7 and for $s = 1$ to 9.

$$\begin{cases} CONM^M_{L,s,4i} \leftarrow CONM^M_{L,0,4i} \\ CONM^M_{L,s,4i+1} \leftarrow CONM^M_{L,0,4i+1} \\ CONM^M_{L,s,4i+2} \leftarrow CONM^M_{L,0,4i+2} \\ CONM^M_{L,s,4i+3} \leftarrow CONM^M_{L,0,4i+3} \\ CONM^M_{R,s,4i} \leftarrow CONM^M_{R,0,4i} \\ CONM^M_{R,s,4i+1} \leftarrow CONM^M_{R,0,4i+1} \\ CONM^M_{R,s,4i+2} \leftarrow CONM^M_{R,0,4i+2} \\ CONM^M_{R,s,4i+3} \leftarrow CONM^M_{R,0,4i+3} \end{cases}$$

2.9.4 List of Constant Values

The following tables offer the list of the constant values for reference. These described values are all required constant values for AURORA-224/256 CF, AURORA-384/512 CF_0 and AURORA-224M/256M CF_0^M. In the following tables, the constant values are arranged from the left to the right.

Constant Values for AURORA-224/256 CF $\{CONC_j\}_{0 \le j < 68}$							
2883f695	ca509844	096a4c18	cf76858f	9698ed2b	f89c5476	12d4f203	5713b743
429feaff	e1fa326f	15002604	9b21ae25	42a0d5ff	ed498163	2a00263b	fd38a296
42deabff	3f08c0b1	54002645	7e9c70d7	422257ff	8230f80c	a80026b9	0fe6cdef
43daaffe	dcac6452	50012741	375b9373	402a5ffd	f3e22a7d	a00224b1	ab05bc3d
47cabffa	e4458d6a	40052351	e52aab9a	480a7ff5	3b8e46b5	800a2c91	72957451
578affea	8073bb0e	00153311	89e2cfac	688affd5	09955d87	002a0c11	44f1464a
168affab	4d66aec3	00547211	a27802b9	ea8aff57	bb07cf35	00a88e11	6194f4d8
babbce07	142fe79a	31f8de20	30ca5bf0	1ad9aca7	43bb73cd	53587e42	18650c64
f22c594f	6871b9e6	a6b096b7	8c3227ae				

Constant Values for AURORA-224/256 CF $\{CONM_{L,j}\}_{0\leq j<32}$							
5107ed2a	94a13089	12d49830	9eed0b1f	853fd5fe	c3f464df	2a004c08	36435c4b
85bd57fe	7e118162	a8004c8a	fd38e1ae	87b55ffc	b958c8a5	a0024e82	6eb726e6
8f957ff4	c88b1ad5	800a46a2	ca555735	af15ffd4	00e7761d	002a6622	13c59f59
2d15ff56	9acd5d86	00a8e422	44f00573	75779c0f	285fcf34	63f1bc40	6194b7e0

Constant Values for AURORA-224/256 CF $\{CONM_{R,j}\}_{0\leq j<32}$							
cb4c7695	7c4e2a3b	896a7901	ab89dba1	a1506aff	f6a4c0b1	9500131d	7e9c514b
a1112bff	41187c06	d400135c	87f366f7	a0152ffe	f9f1153e	d0011258	d582de1e
a4053ffa	9dc7235a	c0051648	b94aba28	b4457fea	84caaec3	80150608	2278a325
f5457fab	dd83e79a	80544708	30ca7a6c	8d6cd653	a1ddb9e6	29ac3f21	0c328632

Constant Values for AURORA-384/512 CF_0 $\{CONC_{L,0,j}\}_{0\leq j<68}$							
6858f1ae	c2f4fa64	0e51cc84	0b363092	9b4ae35d	c06b6566	1ca23f96	3533320d
d55ff613	153c32b3	09ec7183	9a99e75a	4975dc8f	ab8f810d	2370eda9	fde459e9
d910b91f	20cec086	46e07dcc	7ef2d2a8	51eb4297	b1767817	bd68f537	0fd14310
e82c852e	f9aaa45f	7ad14cf0	b7400bcc	33933af5	ddc4ca7b	c50a974f	6b082fa2
2cdc75ea	cff3fd69	8a158800	052c3d95	1242ebd4	12f0feb4	142bb69e	0296e096
6f7ed7a9	a869670e	2856cba2	31e35a0f	9506af53	213d3387	50ac31da	98f1d35b
c9c76e0f	659799c3	91f06d1b	cc7897f1	7045ecb6	47c2cce1	1349d499	663cb5a4
ab70d96d	82f0fe24	26920fac	03b67096	b52b8273	e0696746	7d8c11f7	3173120f
899d344f	053d33a3	cbb02d41	98b9f75b				

Constant Values for AURORA-384/512 CF_0 $\{CONC_{R,0,j}\}_{0\leq j<68}$							
42c78d73	17a7d326	728e6420	59b18490	da571aec	035b2b36	e511fcb0	a9999069
aaffb09e	a9e19598	4f638c18	d4cf3ad4	4baee47a	5c7c086d	1b876d49	ef22cf4f
c885c8fe	06760431	3703ee62	f7969543	8f5a14ba	8bb3c0bd	eb47a9bd	7e8a1880
41642977	cd5522ff	d68a6783	ba005e65	9c99d7a9	ee2653de	2854ba7e	58417d13
66e3af51	7f9feb4e	50ac4004	2961eca8	92175ea0	9787f5a0	a15db4f0	14b704b0
7bf6bd4b	434b3875	42b65d11	8f1ad079	a8357a9c	09e99c39	85618ed2	c78e9adc
4e3b707e	2cbcce1b	8f8368dc	63c4bf8e	822f65b3	3e16670a	9a4ea4c8	31e5ad23
5b86cb6d	1787f124	34907d61	1db384b0	a95c139d	034b3a37	ec608fbb	8b989079
4ce9a27c	29e99d18	5d816a0e	c5cfbadc				

Constant Values for AURORA-384/512 CF_0 $\{CONM_{L,0,j}\}_{0\leq j<32}$							
d0b1e35c	85e9f4c9	1ca39908	166c6124	aabfec27	2a786566	13d8e306	3533ceb5
b221723f	419d810c	8dc0fb98	fde5a550	d0590a5d	f35548bf	f5a299e0	6e801799
59b8ebd4	9fe7fad3	142b1001	0a587b2a	defdaf52	50d2ce1d	50ad9744	63c6b41e
938edc1f	cb2f3386	23e0da37	98f12fe3	56e1b2db	05e1fc49	4d241f58	076ce12c

Constant Values for AURORA-384/512 CF_0 $\{CONM_{R,0,j}\}_{0\leq j<32}$							
cda571ae	6035b2b3	0e511fcb	9a999906	a4baee47	d5c7c086	91b876d4	fef22cf4
a8f5a14b	d8bb3c0b	deb47a9b	07e8a188	99c99d7a	eee2653d	e2854ba7	358417d1
092175ea	09787f5a	0a15db4f	014b704b	ca8357a9	909e99c3	285618ed	cc78e9ad
3822f65b	a3e16670	89a4ea4c	331e5ad2	da95c139	7034b3a3	bec608fb	98b98907

Constant Values for AURORA-224M/256M CF_0^M $\{CONC_{L,0,j}^M\}_{0\leq j<68}$							
89ae91c3	4cfab05a	6e3c5d5b	9f4adc83	cd1c2387	6f0a4079	dc7819e9	7f0dff73
4478470e	7ef2b868	b8f1908d	8f2eee8b	fe81beb4	760e4460	414b2a74	773fe677
23427d69	72703a64	8296f7b7	8b37e209	30f5ca7a	704f0566	3585e400	f533e036
179ba45d	a54802b3	5ba2c36e	fa993531	59477813	1bd3990d	87ec8db2	cde48baa
c4ffc08e	9086cc86	3f71100a	66f200ff	57be811d	01347e17	7ee2834b	03d1914d
d90d3293	49eda75f	cd6c0df8	b140d994	6c5a6526	b999d3af	9ad9b8af	58a029e0
aec5fae4	c1a3e9d7	051b7a30	2c5051da	83caf5c9	fdbef4eb	0a36573f	16286dc7
d9d4eb93	37a8e221	146c0d21	3bbca7d1	6de8d727	52a3e944	28d8b91d	2d76c2da
ada19ee7	b43e74a2	61187954	16bb2447				

Constant Values for AURORA-224M/256M CF_0^M $\{CONC_{R,0,j}^M\}_{0\leq j<68}$							
4d748e1c	67d582d2	71e2eadb	fa56e41c	68e11c3e	785203cb	e3c0cf4e	f86ffb9b
23c23872	f795c343	c78c846d	7977745c	f40df5a7	b0722303	0a5953a2	b9ff33bb
1a13eb49	9381d323	14b7bdbc	59bf104c	87ae53d1	82782b33	ac2f2001	a99f01b7
bcdd22e8	2a40159d	dd161b72	d4c9a98f	ca3bc09a	de9cc868	3f646d94	6f245d56
27fe0476	84366434	fb888051	379007fb	bdf408ea	09a3f0b8	f7141a5b	1e8c8a68
c869949e	4f6d3afa	6b606fc6	8a06cca5	62d32933	ccce9d7d	d6cdc57c	c5014f02
762fd725	0d1f4ebe	28dbd180	62828ed1	1e57ae4c	edf7a75f	51b2b9f8	b1436e38
cea75c9e	bd471109	a3606908	dde53e89	6f46b93b	951f4a22	46c5c8e9	6bb616d1
6d0cf73d	a1f3a515	08c3caa3	b5d92238				

Constant Values for AURORA-224M/256M CF_0^M $\{CONM_{L,0,j}^M\}_{0\leq j<32}$							
135d2387	99f560b4	dc78bab6	3e95b907	88f08e1c	fde570d0	71e3211b	1e5ddd17
4684fad2	e4e074c8	052def6f	166fc413	2f3748ba	4a900567	b74586dc	f5326a63
89ff811d	210d990d	7ee22014	cde401fe	b21a6527	93db4ebe	9ad81bf1	6281b329
5d8bf5c9	8347d3af	0a36f460	58a0a3b4	b3a9d727	6f51c442	28d81a42	77794fa2

Constant Values for AURORA-224M/256M CF_0^M $\{CONM_{R,0,j}^M\}_{0\leq j<32}$							
e68e11c3	b785203c	ee3c0cf4	bf86ffb9	7f40df5a	3b072230	20a5953a	bb9ff33b
187ae53d	382782b3	1ac2f200	7a99f01b	aca3bc09	8de9cc86	43f646d9	66f245d5
abdf408e	809a3f0b	bf7141a5	81e8c8a6	362d3293	dccce9d7	cd6cdc57	2c5014f0
c1e57ae4	fedf7a75	851b2b9f	8b1436e3	b6f46b93	2951f4a2	946c5c8e	16bb616d

2.10 Pseudocodes

The pseudocodes of the specifications of the AURORA family are described in this section.

$MSM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0 \le j < 32})$
000 $(X_0, X_1, \ldots, X_7) \leftarrow X$
010 $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_0, Y_1, Y_2, Y_3)$
020 $(Z_0, Z_1, \ldots, Z_7) \leftarrow (X_0, X_1, \ldots, X_7)$
030 for $i \leftarrow 1$ to 7 do
040 $\quad (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$
050 $\quad (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$
060 $\quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_{4i}, Y_{4i+1}, Y_{4i+2}, Y_{4i+3})$
070 $\quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$
080 $\quad (Z_{8i}, Z_{8i+1}, \ldots, Z_{8i+7}) \leftarrow (X_0, X_1, \ldots, X_7)$
090 $(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$
100 $(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$
110 $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$
120 $(Z_{64}, Z_{65}, \ldots, Z_{71}) \leftarrow (X_0, X_1, \ldots, X_7)$
130 return $\{Z_{j\,(32)}\}_{0 \le j < 72}$

Figure 2.13: A pseudocode of $MSM : \{0,1\}^{256} \times (\{0,1\}^{32})^{32} \rightarrow (\{0,1\}^{32})^{72}$. *BD* is defined in Sec. 2.2.3. F and F' are functions over $\{0,1\}^{32}$.

$CPM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0 \le j < 144}, \{W_{j\,(32)}\}_{0 \le j < 68})$
000 $(X_0, X_1, \ldots, X_7) \leftarrow X$
010 $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (W_0, W_1, W_2, W_3)$
020 $(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_0, Y_1, \ldots, Y_7)$
030 for $i \leftarrow 1$ to 16 do
040 $\quad (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$
050 $\quad (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$
060 $\quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3})$
070 $\quad (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$
080 $\quad (X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7})$
090 $(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$
100 $(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$
110 $(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$
120 $(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{136}, Y_{137}, \ldots, Y_{143})$
130 $Z \leftarrow (X_0 \,\|\, X_1 \,\|\, \cdots \,\|\, X_7)$
140 return $Z_{(256)}$

Figure 2.14: A pseudocode of $CPM : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \times (\{0,1\}^{32})^{68} \rightarrow \{0,1\}^{256}$. *BD* is defined in Sec. 2.2.3. F and F' are functions over $\{0,1\}^{32}$.

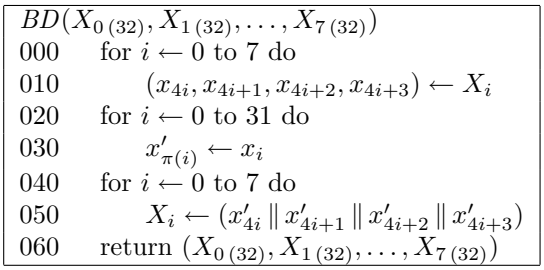

Figure 2.15: A pseudocode of $BD : (\{0,1\}^{32})^8 \to (\{0,1\}^{32})^8$. π is defined in Fig. 2.3.

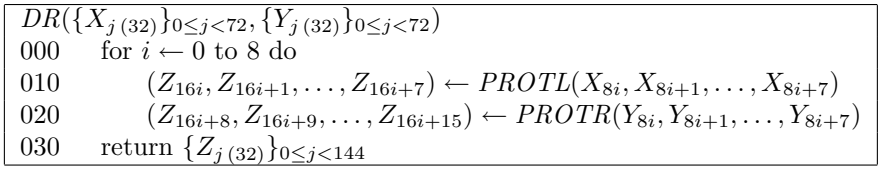

Figure 2.16: A pseudocode of $DR : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{144}$. The functions $PROTL$ and $PROTR$ are defined in (2.8) and (2.9), respectively.

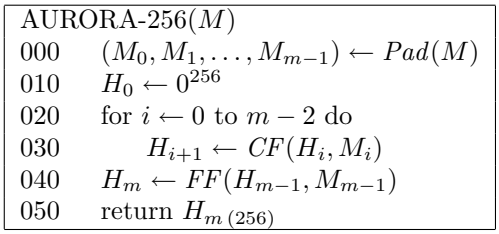

Figure 2.17: A pseudocode of AURORA-256. The padding function, $Pad(\cdot)$, is defined in (2.10), CF is defined in Sec. 2.3.2, and FF is defined in Sec. 2.3.3.

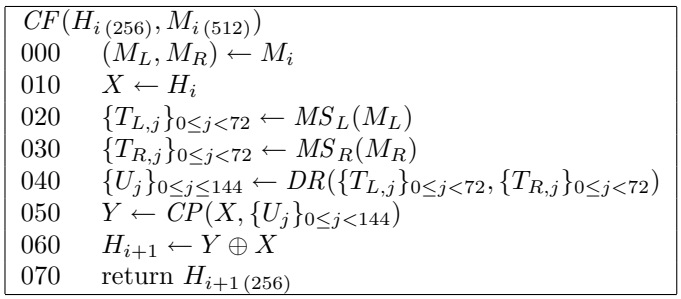

Figure 2.18: A pseudocode of $CF : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$. MS_L, MS_R, DR, and CP are defined in (2.11), (2.12), Sec. 2.2.5, and in (2.13), respectively.

```
FF(H_{m-1 (256)}, M_{m-1 (256)})
000   (M_L, M_R) ← M_{m-1}
010   X ← H_{m-1}
020   {T_{L,j}}_{0≤j<72} ← MSF_L(M_L)
030   {T_{R,j}}_{0≤j<72} ← MSF_R(M_R)
040   {U_j}_{0≤j≤144} ← DR({T_{L,j}}_{0≤j<72}, {T_{R,j}}_{0≤j<72})
050   Y ← CPF(X, {U_j}_{0≤j<144})
060   H_m ← Y ⊕ X
070   return H_{m (256)}
```

Figure 2.19: A pseudocode of $FF : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$. MSF_L, MSF_R, DR, and CPF are defined in (2.15), (2.16), Sec. 2.2.5, and in (2.17), respectively.

```
AURORA-224(M)
000   (M_0, M_1, ..., M_{m-1}) ← Pad(M)
010   H_0 ← 1^256
020   for i ← 0 to m − 2 do
030       H_{i+1} ← CF(H_i, M_i)
040   H_m ← FF(H_{m-1}, M_{m-1})
050   H'_m ← TF_224(H_m)
060   return H'_{m (224)}
```

Figure 2.20: A pseudocode of AURORA-224. Pad, CF, and FF are the same as AURORA-256 and defined in Sec. 2.3.

```
AURORA-512(M)
000   (M_0, M_1, ..., M_{m-1}) ← Pad(M)
010   H_0 ← 0^512
020   for i ← 0 to m − 1 do
030       H_{i+1} ← CF_{i mod 8}(H_i, M_i)
040       if (0 < i < m − 1) ∧ (i mod 8 = 7) then
050           H_{i+1} ← MF(H_{i+1})
060   H_m ← MFF(H_m)
070   return H_{m (512)}
```

Figure 2.21: A pseudocode of AURORA-512. The padding function, $Pad(\cdot)$, is defined in (2.10), CF_s is defined in Sec. 2.5.2, MF is defined in Sec. 2.5.3, and MFF is defined in Sec. 2.5.4.

$CF_s(H_{i\,(512)}, M_{i\,(512)})$
000 $(M_L, M_R) \leftarrow M_i$
010 $(X_L, X_R) \leftarrow H_i$
020 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS_{L,s}(M_L)$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS_{R,s}(M_R)$
040 $\{U_j\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72})$
050 $Y_L \leftarrow CP_{L,s}(X_L, \{U_j\}_{0\le j<144})$
060 $Y_R \leftarrow CP_{R,s}(X_R, \{U_j\}_{0\le j<144})$
070 $Z_L \leftarrow Y_L \oplus X_L$
080 $Z_R \leftarrow Y_R \oplus X_R$
090 $H_{i+1} \leftarrow (Z_L, Z_R)$
100 return $H_{i+1\,(512)}$

Figure 2.22: A pseudocode of $CF_s : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$. $MS_{L,s}$, $MS_{R,s}$, DR, $CP_{L,s}$, and $CP_{R,s}$ are defined in (2.18), (2.19), Sec. 2.2.5, (2.20), and in (2.21), respectively.

$MF(H_{i\,(512)})$
000 $(X_L, X_R) \leftarrow H_i$
010 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS_{L,8}(X_L)$
020 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS_{R,8}(X_R)$
030 $\{U_j\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72})$
040 $Y_L \leftarrow CP_{L,8}(X_L, \{U_j\}_{0\le j<144})$
050 $Y_R \leftarrow CP_{R,8}(X_R, \{U_j\}_{0\le j<144})$
060 $Z_L \leftarrow Y_L \oplus X_L$
070 $Z_R \leftarrow Y_R \oplus X_R$
080 $H_i \leftarrow (Z_L, Z_R)$
090 return $H_{i\,(512)}$

Figure 2.23: A pseudocode of $MF : \{0,1\}^{512} \to \{0,1\}^{512}$. $MS_{L,8}$, $MS_{R,8}$, DR, $CP_{L,8}$, and $CP_{R,8}$ are defined in (2.22), (2.23), Sec. 2.2.5, (2.24), and in (2.25), respectively.

$MFF(H_{m\,(512)})$
000 $(X_L, X_R) \leftarrow H_m$
010 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS_{L,9}(X_L)$
020 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS_{R,9}(X_R)$
030 $\{U_j\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72})$
040 $Y_L \leftarrow CP_{L,9}(X_L, \{U_j\}_{0\le j<144})$
050 $Y_R \leftarrow CP_{R,9}(X_R, \{U_j\}_{0\le j<144})$
060 $Z_L \leftarrow Y_L \oplus X_L$
070 $Z_R \leftarrow Y_R \oplus X_R$
080 $H_m \leftarrow (Z_L, Z_R)$
090 return $H_{m\,(512)}$

Figure 2.24: A pseudocode of $MFF : \{0,1\}^{512} \to \{0,1\}^{512}$. $MS_{L,9}$, $MS_{R,9}$, DR, $CP_{L,9}$, and $CP_{R,9}$ are defined in (2.27), (2.28), Sec. 2.2.5, (2.29), and in (2.30), respectively.

AURORA-384(M)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow Pad(M)$
010 $H_0 \leftarrow 1^{512}$
020 for $i \leftarrow 0$ to $m-1$ do
030 $\quad H_{i+1} \leftarrow CF_{i \bmod 8}(H_i, M_i)$
040 $\quad$ if $(0 < i < m-1) \wedge (i \bmod 8 = 7)$ then
050 $\quad\quad H_{i+1} \leftarrow MF(H_{i+1})$
060 $H_m \leftarrow MFF(H_m)$
070 $H'_m \leftarrow TF_{384}(H_m)$
080 return $H'_{m\,(384)}$

Figure 2.25: A pseudocode of AURORA-384. Pad, CF_s, MF, and MFF are the same as AURORA-512 and defined in Sec. 2.5.

AURORA-256M(M)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow Pad(M)$
010 $H_0 \leftarrow 0^{512}$
020 for $i \leftarrow 0$ to $m-1$ do
030 $\quad H_{i+1} \leftarrow CF^M_{i \bmod 8}(H_i, M_i)$
040 $\quad$ if $(0 < i < m-1) \wedge (i \bmod 8 = 7)$ then
050 $\quad\quad H_{i+1} \leftarrow MF^M(H_{i+1})$
060 $H'_m \leftarrow MFF^M(H_m)$
070 return $H'_{m\,(256)}$

Figure 2.26: A pseudocode of AURORA-256M. The padding function, $Pad(\cdot)$, is defined in (2.10), CF^M_s is defined in Sec. 2.7.2, MF^M is defined in Sec. 2.7.3, and MFF^M is defined in Sec. 2.7.4.

$CF^M_s(H_{i\,(512)}, M_{i\,(512)})$
000 $(M_L, M_R) \leftarrow M_i$
010 $(X_L, X_R) \leftarrow H_i$
020 $\{T_{L,j}\}_{0 \le j < 72} \leftarrow MS^M_{L,s}(M_L)$
030 $\{T_{R,j}\}_{0 \le j < 72} \leftarrow MS^M_{R,s}(M_R)$
040 $\{U_j\}_{0 \le j \le 144} \leftarrow DR(\{T_{L,j}\}_{0 \le j < 72}, \{T_{R,j}\}_{0 \le j < 72})$
050 $Y_L \leftarrow CP^M_{L,s}(X_L, \{U_j\}_{0 \le j < 144})$
060 $Y_R \leftarrow CP^M_{R,s}(X_R, \{U_j\}_{0 \le j < 144})$
070 $Z_L \leftarrow Y_L \oplus X_L$
080 $Z_R \leftarrow Y_R \oplus X_R$
090 $H_{i+1} \leftarrow (Z_L, Z_R)$
100 return $H_{i+1\,(512)}$

Figure 2.27: A pseudocode of $CF^M_s : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$. $MS^M_{L,s}$, $MS^M_{R,s}$, DR, $CP^M_{L,s}$, and $CP^M_{R,s}$ are defined in (2.31), (2.32), Sec. 2.2.5, (2.33), and in (2.34), respectively.

$MF^M(H_{i\,(512)})$
000 $(X_L, X_R) \leftarrow H_i$
010 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS^M_{L,8}(X_L)$
020 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS^M_{R,8}(X_R)$
030 $\{U_j\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72})$
040 $Y_L \leftarrow CP^M_{L,8}(X_L, \{U_j\}_{0\le j<144})$
050 $Y_R \leftarrow CP^M_{R,8}(X_R, \{U_j\}_{0\le j<144})$
060 $Z_L \leftarrow Y_L \oplus X_L$
070 $Z_R \leftarrow Y_R \oplus X_R$
080 $H_i \leftarrow (Z_L, Z_R)$
090 return $H_{i\,(512)}$

Figure 2.28: A pseudocode of $MF^M : \{0,1\}^{512} \to \{0,1\}^{512}$. $MS^M_{L,8}$, $MS^M_{R,8}$, DR, $CP^M_{L,8}$, and $CP^M_{R,8}$ are defined in (2.35), (2.36), Sec. 2.2.5, (2.37), and in (2.38), respectively.

$MFF^M(H_{m\,(512)})$
000 $(X, Y) \leftarrow H_m$
010 for $j \leftarrow 0$ to 71 do
020 $\quad T_{L,j} \leftarrow 0^{32}$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS^M_{R,9}(Y)$
040 $\{U_j\}_{0\le j\le 144} \leftarrow DR(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72})$
050 $Z \leftarrow CP^M_{L,9}(X, \{U_j\}_{0\le j<144})$
060 $H'_m \leftarrow Z \oplus X$
070 return $H'_{m\,(256)}$

Figure 2.29: A pseudocode of $MFF^M : \{0,1\}^{512} \to \{0,1\}^{512}$. $MS^M_{R,9}$, DR, and $CP^M_{L,9}$ are defined in (2.40), Sec. 2.2.5, and in (2.41), respectively.

AURORA-224M(M)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow Pad(M)$
010 $H_0 \leftarrow 1^{512}$
020 for $i \leftarrow 0$ to $m-1$ do
030 $\quad H_{i+1} \leftarrow CF^M_{i \bmod 8}(H_i, M_i)$
040 $\quad$ if $(0 < i < m-1) \wedge (i \bmod 8 = 7)$ then
050 $\quad\quad H_{i+1} \leftarrow MF^M(H_{i+1})$
060 $H'_m \leftarrow MFF^M(H_m)$
070 $H''_m \leftarrow TF_{224}(H'_m)$
080 return $H''_{m\,(256)}$

Figure 2.30: A pseudocode of AURORA-224M. The padding function, $Pad(\cdot)$, is defined in (2.10), CF^M_s is defined in Sec. 2.7.2, MF^M is defined in Sec. 2.7.3, and MFF^M is defined in Sec. 2.7.4.

2.11 AURORA Examples

This section describes example vectors of the AURORA hash algorithm family. Table 2.2 gives three examples for the messages M_1, M_2, and M_3 below for each hash function.

Let the message M_1 be the 24-bit ASCII string "abc", which is equivalent to the following binary string:

`01100001  01100010  01100011`.

Let the message M_2 be the 448-bit ASCII string

"**abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq**".

Let the message M_3 be the binary-coded form of the ASCII string which consists of 1,000,000 repetitions of the character "**a**".

Table 2.2: AURORA Examples.

AURORA-256	
Message	Hash Value
M_1	3e0c31c1 8ef5c404 33844fac 2d4acdf4 9e390962 797821a4 9e3553f3 8189917e
M_2	21621069 e64ec45a eccf140a d881c684 44c30081 32a3b2d0 e9a1d961 d2dc034f
M_3	ec8cede6 3fd1bd3b c6de6702 b6ed25e8 d80f5efa b5433912 446aaefc db026b5f

AURORA-224	
Message	Hash Value
M_1	50fddc1c 77601c2c c01cc258 eccc6a10 37646235 860da74b 6e0280af
M_2	05874948 064d42ca e0ffa686 45034160 8d571731 f9581ca8 b8ea1890
M_3	7977bc32 b66d7b05 6b215153 1545668d 5f3d1c6c 42a48334 5ab31f70

AURORA-512	
Message	Hash Value
M_1	6a4cf6d1 18619abd e8c920d5 9806e483 cc90616f 8d1b4db6 b98abab7 00c4ec47 85eaa639 45bb65e1 52df4901 a1c36f78 9c587f09 49c8e76a a0a8d7de 20f8aa0e
M_2	cbf432c3 01103535 f0cf0027 efe2b0c6 2046414e 6128ec83 bbd0bccf 7425f908 a5061438 6da57647 8f91cd42 1f4a0015 7b2fa527 d81328e7 76be3262 7352ef0c
M_3	577e573e d9bfbc31 a80bcea8 2d1e4441 89d31fe0 7cda57d3 a2c8ad00 9800feae 431e456b 85184399 5c12c5e6 6a7f7272 55880d11 375f08a1 4841fb96 86d390e4

AURORA-384	
Message	Hash Value
M_1	cb7a330f 33ab55ec 98698f49 4ace5996 3dcec8e2 bdfa12f1 f8db22fc 18b5591e a02f267e bdaf1639 49133bf3 b59e94c2
M_2	f16bb878 ddee85ef 51994078 61aeee1c b23c63fb 6498f38d fbecf41e cf24805f 8b28f018 656610f1 26ad1400 0a3f3ab6
M_3	c18722f8 d9e0fe10 de818d07 e8b66734 c23532ee 7d1d9968 18f60ab0 3950b416 cb89c086 8263eb84 3b4264d1 44c2180d

AURORA-256M	
Message	Hash Value
M_1	46c5dba6 cfdc333b 7cfb4242 8fe59345 a0882acb c10c5694 9c248501 b156c457
M_2	3c3353d9 67d30005 de02cae6 e3b1a205 11e3b3a8 3d9048ee 5694df40 2bdc9588
M_3	cd97a51f 79cb722a c2c33a46 62502b10 a13565b4 1f662699 11b9b438 f9fe81fb

AURORA-224M	
Message	Hash Value
M_1	d64eaa68 02030670 3e7d6301 74bd2f9b 607a1e95 b6620ba2 5d2a3248
M_2	587879d0 6eebb1da 87b6de94 06e0dbdf 24e5fbad d98bc0dd 1257ad26
M_3	c78f12a4 308821ab 3d312fdb 9dff6408 5496a44e a1aeebd5 a734166c

Chapter 3

Design Rationale of AURORA

This chapter describes the design rationale of the AURORA hash function family. The design of AURORA is divided in two parts: one is a part of fixed-input-length compression functions and the other is a domain extension transform which utilizes the compression function as a building block to implement a variable-input-length hash function. In this chapter, we describe the design rationale in a top-down approach, from the domain extension to the compression function for AURORA-256, AURORA-512, and AURORA-256M, then explain the components in the common building blocks.

We describe the design rationale for AURORA-256, AURORA-512, and AURORA-256M as representatives of the AURORA family. However, the design rationale of AURORA-256 is applicable to AURORA-224, because AURORA-224 is the same as AURORA-256 except for the initial value and truncation of final hash value. Similarly, the design rationale of AURORA-512 is applicable to AURORA-384, and also the design rationale of AURORA-256M is applicable to AURORA-224M.

3.1 AURORA-256

3.1.1 Domain Extension

AURORA-256 adopts the strengthened Merkle-Damgård (sMD) transform with a *finalization function* which is different from the compression function in the transform. The domain extension of AURORA-256 is shown in the above of Fig. 3.1.

Most of widely-used hash functions employ the strengthened Merkle-Damgård transform because it has been proven to be collision-resistance preserving [35, 15]: if the compression function is collision-resistant (CR), then so is the hash function. However, current usages of hash functions make it obvious that CR no longer suffices for the security goal for hash functions, because hash functions are also often used to instantiate random oracles. Coron et al. [12] introduced a formal definition of "behaving like a random oracle" for hash functions using the indifferentiability framework, which was originally proposed by Maurer et al. [32]. They showed that the sMD transform is not indifferentiable from a random oracle.

We chose the sMD transform with the finalization function, because it preserves CR and indifferentiability (PRO) of the underlying compression function. The collision resistance preservation (CR-Pr) is ensured by the MD strengthening [35]: the input message is padded by the padding function $Pad(\cdot)$ in AURORA. CR-Pr can be proven similarly to the proof in [35]. The pseudorandom oracle preservation (PRO-Pr) is due to the finalization function. The finalization function works to envelope the internal MD iteration as the enveloping mechanism used in NMAC/HMAC constructions [5] and the EMD transform [6]. PRO-Pr can be easily proven from Lemma 5.1 in [6], which is core to the proof that EMD is PRO-Pr.

The structure of the finalization function FF, is the same as the structure of the compression function CF, except for the constants. By using a different set of constants between them, it

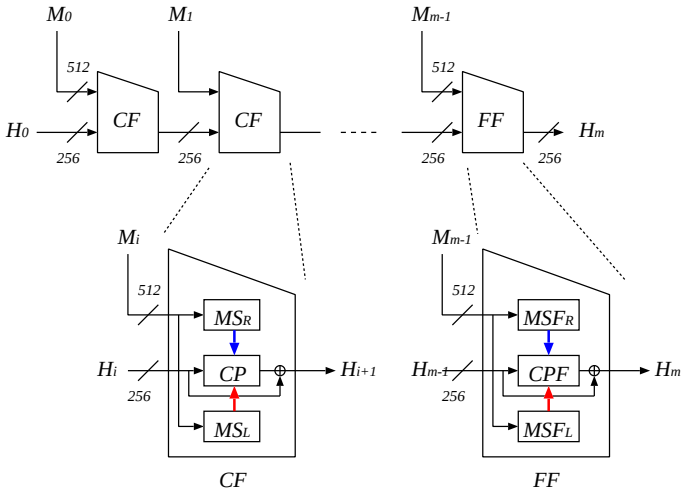

Figure 3.1: AURORA-256: Domain extension and compression function.

is expected that FF behaves as a different function from CF. On the other hand, FF can be efficiently implemented by using the same module as CF.

3.1.2 Compression Function

The AURORA-256 compression function CF uses two message scheduling functions MS_L and MS_R, and the chaining value processing function CP, as shown in the below of Fig. 3.1. It is regarded as the Davies-Meyer construction [34, p.340]. We chose this construction because it is possible to input longer message than the chaining value to achieve higher throughput, while in the Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions [34, p.340] the message and chaining value must be the same size. Although the Davies-Meyer construction has a negative property of easily found fixed points [36, 45, 17], we attached more importance to achieving higher throughput.

Considering recent attacks on hash functions exploiting simple message scheduling [55, 56, 57], we chose to design more secure (and more heavy) message schedule like Whirlpool [3] and DASH [8]. The message scheduling function (MS_L, MS_R) was designed based on a 256-bit permutation using blockcipher design techniques. To achieve both of security and speed, the message scheduling function is composed of two 256-bit functions, not one 512-bit function, because generally constructing a 512-bit ideal primitive requires more than double cost of constructing a 256-bit ideal primitive.

The finalization function FF uses two message scheduling functions MSF_L and MSF_R and the chaining value processing function CPF. The structure of the finalization function FF is the same as the structure of the compression function CF except for the constants.

3.2 AURORA-512

3.2.1 Domain Extension – Double-Mix Merkle-Damgård transform

In order to achieve an efficient 512-bit hash function, a novel domain extension transform, called the Double-Mix Merkle-Damgård (DMMD) transform is introduced. The DMMD transform consists of double lines of the compression functions and the mixing functions inserted every 8 blocks as Fig. 3.2 shows. The DMMD transform enables an efficient collision-resistant construction for

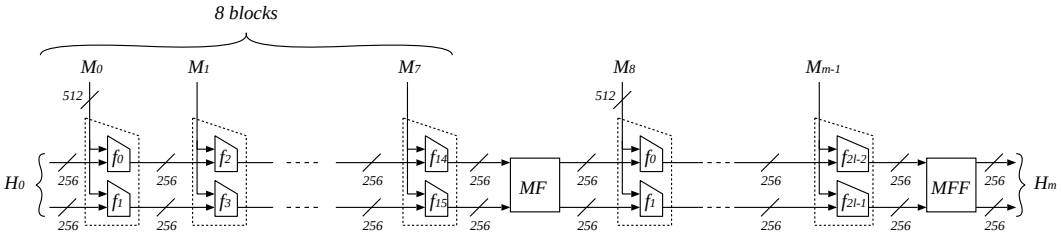

Figure 3.2: Double-Mix Merkle-Damgård (DMMD) transform.

double length hash functions[1], which outputs $2n$-bit hash values using component functions with n-bit output. We adopted this approach because (1) the same compression function can be used in all the AURORA family, and because (2) the message scheduling functions can be shared between two compression functions by making the best use of the structure of the AURORA compression function.

The previous designs for secure (i.e., collision-resistant) double length hash functions include Lucks' double-pipe hash [31] and Hirose's construction [25]. The double-pipe hash uses two compression functions $f : \{0,1\}^{2n} \times \{0,1\}^m \to \{0,1\}^n$ in parallel, i.e., f has a $\boldsymbol{2n}$-bit chaining value and an $\boldsymbol{m}$-bit message as inputs and an n-bit chaining value as output. Similarly, Hirose's construction uses a secure blockcipher $E : \{0,1\}^{n+m} \times \{0,1\}^n \to \{0,1\}^n$ twice, i.e., E is an n-bit blockcipher with $(n+m)$-bit key length. The DMMD transform consists of *smaller* compression functions $f_i : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ with an $\boldsymbol{n}$-bit chaining value and an $\boldsymbol{m}$-bit message as inputs and an n-bit chaining value as output. Generally, it is possible to construct a secure component with small input size at lower cost than a component with large input size. Although the DMMD transform additionally requires the mixing function which is called only once every eight blocks, this approach can achieve an efficient double length hash function.

Security of the DMMD transform. The collision resistance (CR), preimage resistance (Pre), and second preimage resistance (Sec) of the DMMD transform can be achieved with appropriate assumptions on the underlying components. (See Sec. 4.2.2 for the proofs).

The pseudorandom oracle preservation (PRO-Pr) can be proven similarly to the EMD [6, Lemma 5.1]. The PRO-Pr of the DMMD transform is due to the mixing function for finalization *MFF*, which works to envelope the internal iterated compression functions.

Shared message scheduling between two 256-bit compression functions. In AURORA-512, the compression function CF_i consists of two compression functions with 256-bit output (denoted as "256-bit compression functions") f_i and f_{i+1}, as shown in Fig. 3.2. Each of the 256-bit compression function consists of two message scheduling functions and the chaining value processing function. Since the message scheduling can be shared between two 256-bit compression functions, the cost of the 512-bit compression function CF_i is reduced to less than double cost of the 256-bit compression function. In the case of AURORA-512, the cost for the message scheduling functions/the chaining value processing function ratio is about 1:1, the 512-bit compression function CF_i can be implemented with about 1.5 times cost of the 256-bit compression function.

Mixing functions. In the DMMD transform, the mixing function is inserted at intervals. The purpose of the mixing function is to mix the two n-bit chaining values. The number of blocks the mixing function is inserted effects the security bound. In AURORA-512, the mixing function *MF* is inserted every 8 blocks.

[1]They are also called as double-block-length (DBL) hash functions, but we use the term "double length hash functions" following [37].

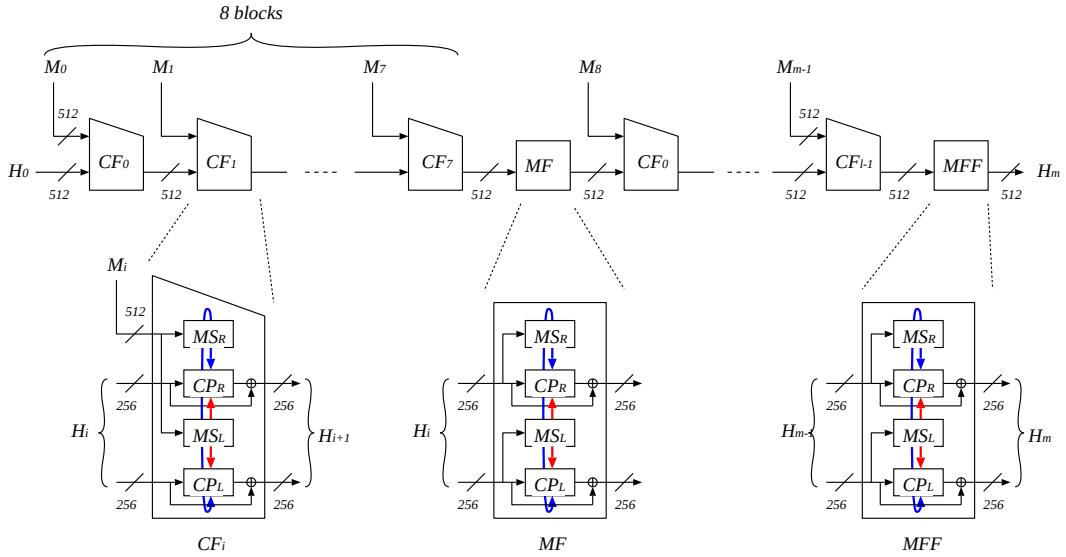

Figure 3.3: AURORA-512: Domain extension and compression function.

Independent instances of compression functions. In order to prove that the DMMD transform has collision resistance, each of the (eight) compression functions between the mixing functions is expected to behave as an independent function. To specify independent compression function instances with limited implementation cost, we use the same components for all compression functions with different sets of constants. For the security proofs of the DMMD transform, see Sec. 4.2.2.

3.2.2 Compression Function

As described in Sec. 3.2.1, the compression function for AURORA-512 is designed based on two 256-bit compression functions, which are the same as the AURORA-256 compression function. One 256-bit compression function consists of two message scheduling functions MS_L and MS_R and a chaining value processing function CP_L; the other 256-bit compression function consists of two message scheduling functions MS_L and MS_R and a chaining value processing function CP_R. By sharing the message scheduling functions, the AURORA-512 compression function CF consists of MS_L, MS_R, CP_L and CP_R (See the below left in Fig. 3.3).

The mixing function MF and the mixing function for finalization MFF have a different interface from the compression function CF_i. In other words, there is no message input to MF and MFF, and the chaining value is input to both of MS and CP (See the below middle and right in Fig. 3.3). However, MF and MFF are composed of the same components as the compression function CF_i, except for constants. This enables us to use the same module in software and hardware implementations.

3.3 AURORA-256M

3.3.1 Domain Extension

AURORA-256M, which outputs 256-bit hash values, is an optional instance with multi-collision resistance ("M" means multi-collision resistance). AURORA-256M has the same *structure* as AURORA-512 except the final mixing function. Therefore, it has the almost same performance as AURORA-512, which is only about 50% additional cost to AURORA-256, i.e. less than double cost of AURORA-256. Thus AURORA-256M achieves multi-collision resistance very efficiently.

It is known that many iterated hash functions including the Merkle-Damgård construction and its variants allow Joux's multi-collision attack [26], Kelsey-Schneier's second preimage attack [28], and Kelsey-Kohno's Herding attack [27]. In particular, Kelsey-Schneier's second preimage attack on n-bit iterated hash functions finds a second preimage for a message of 2^k message blocks with about 2^{n-k+1} work. (Note that the security requirement for SHA-3 regarding the second preimage resistance is approximately $n-k$ bits for any message shorter than 2^k bits, so we understand that multi-collision resistance is not a mandatory requirement.)

We include AURORA-224M/256M in the AURORA family for the use in the applications where multi-collision resistance and/or second-preimage resistance for extremely long messages is considered important. However, we submit AURORA-224/256 as the formal SHA-3 candidates and submit AURORA-224M/256M as optional instances, because (1) AURORA-224/256 are more efficient than AURORA-224M/256M and (2) NIST encourages submitters to submit only one algorithm for each hash size.

3.3.2 Compression Function

The compression function and mixing function for AURORA-256M are the same as those for AURORA-512 except for the constants. Thus AURORA-256M can be implemented with the same module as AURORA-512.

3.4 Components and Constants

The compression functions of all the AURORA family are composed of the *common* building blocks: the message scheduling module (*MSM*) and the chaining value processing module (*CPM*). This section shows the design rationale of the components and constants used in *MSM* and *CPM*.

3.4.1 AURORA Structure

As is known in blockcipher design and analysis, security evaluation tends to be difficult or infeasible as the block/input size increases, because the required computational complexity increases. To facilitate ease of analysis, design choice of the structure and its components is important. We chose a 256-bit permutation based on byte-oriented operations to construct the structure of both of the message scheduling module (*MSM*) and the chaining value processing module (*CPM*). We call it the AURORA structure, which is shown in Fig. 3.4. It can be regarded as a 256-bit generalized modified-Feistel structure with byte-wise diffusion layers.

The AURORA structure itself is novel, but follows the traditional blockcipher design strategy. There are four 32-bit-to-32-bit F-functions in parallel in one round. The F-function consists of a substitution layer and a permutation layer, where four S-boxes and a 4×4 matrix multiplication in GF(2^8) are operated. Details are written in Sec. 3.4.2. In order that the hash function family AURORA has desirable security properties including the collision resistance and indifferentiability, it should be guaranteed that the underlying compression function has no differential paths with high probability that are exploitable in collision-finding attacks or distinguishing attacks. The compression function consists of an underlying 256-bit blockcipher with two message scheduling. Since it is computationally infeasible to estimate maximum differential probability of the overall compression function $CF : \{0,1\}^{256} \times \{0,1\}^{512} \rightarrow \{0,1\}^{256}$, we designed so that each of the 256-bit permutation from the 256-bit input X to the 256-bit output Z in *MSM* and *CPM* (For X and Z, see Sec. 2.2.1 and 2.2.2) has no differential paths with high probability under the assumption that "subkeys" (i.e., constants in *MSM* and expanded messages in *CPM*) are independent and uniformly distributed.

In choosing the structure, we estimated maximum differential characteristic probability obtained by the number of active-S-boxes and compared estimated performance given by the number of required F-functions among several candidates including the generalized Feistel structure and its variants. As a result of consideration discussed in Sec. 4.2.3, we chose 8-round AURORA

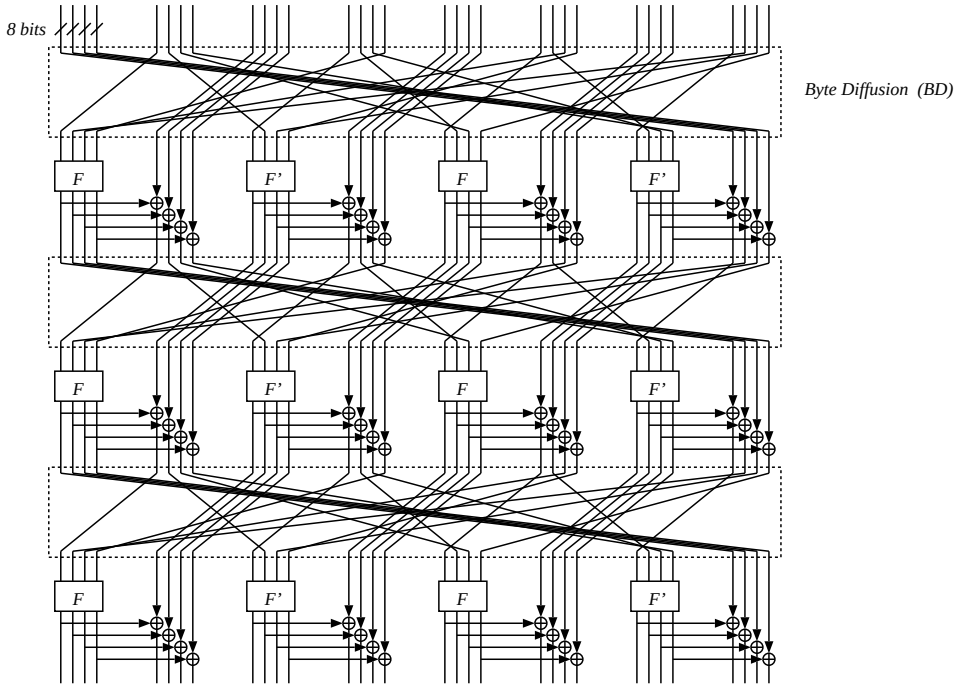

Figure 3.4: AURORA structure.

structure for the message scheduling module and 17-round AURORA structure for the chaining value processing module.

Since (1) AURORA's message scheduling module is designed to be secure by itself by using blockcipher design techniques and (2) AURORA is designed based on byte-oriented operations including the S-box and the matrices in GF(2^8) while SHA-2 makes use of logical operations on 32-bit or 64-bit words, the design strategy is significantly different from SHA-2. Therefore, it is expected that a possibly successful attack on SHA-2 is unlikely to be applicable to AURORA. Furthermore, byte-oriented operations including the S-box and the matrices in GF(2^8) are suitable for a wide range of platforms including 8-bit processors and constrained hardware implementations.

Byte Diffusion function *BD*. The byte diffusion function *BD* is adopted to enhance diffusion and to avoid preserving wordwise structure. For example, there exist 16-round trivial impossible differential paths in the AURORA structure if *BD* is replaced with the traditional *wordwise* permutation. (C.f. There exist 17-round trivial impossible differential paths in the 8-line generalized Feistel structure.) On the other hand, full bytewise diffusion has the downsides including a decrease in efficiency and a reduction of effect by the DSM techniques (For details, see the design rationale of diffusion matrices described later in this section). We examined the effect of several diffusion layer variants on differential characteristic probability in determining the diffusion layer. As a result, we chose the diffusion function where half of the data (i.e. the 2nd, 4th, 6th and 8th words) are input to the bytewise diffusion which is the same as the ShiftRow transformation in the AES [22], and the other half (i.e. the 1st, 3rd, 5th, and 7th words) are input to the 32-bit wordwise permutation similar to that in the generalized Feistel structure.

3.4.2 F-function

The F-function consists of the substitution layer and the permutation layer, where four nonlinear byte substitutions and a 4×4 maximum distance separable (MDS) matrix multiplication over GF(2^8) are operated. The byte substitutions (S-boxes) provide confusion and the matrix

multiplication provides local diffusion in the F-function. The structure and the components of the F-function are chosen to facilitate analysis and to utilize the well-established techniques for blockcipher design and analysis.

AURORA uses four F-Functions F_0, F_1, F_2, and F_3 with different diffusion matrices. Each of the building blocks CP_L, CP_R, ME_L, and ME_R uses two different F-Functions chosen out of four (See Table 3.2). We chose four diffusion matrices so that the Diffusion Switching Mechanism (DSM) [52] works to improve the security against differential and linear attacks.

The details of selection of the S-box and the diffusion matrices are described below.

S-box

We explain the design criteria and procedure for choosing the S-box of AURORA to show that there exist no "trap-doors" in it. The design criteria of the S-box are:

- Immunity against known attacks, and
- Suitability for efficient hardware/software implementations.

To meet the design criteria above, we chose a byte substitution based on an inversion in the finite field GF(2^8), because it provides optimal security in terms of maximum differential/linear probability etc. and optimization techniques for hardware/software implementations are well studied. The AES [22] also employs an S-box based on an inversion in the finite field GF(2^8), however, there is room for both of area/throughput optimizations in hardware implementations. Thus we decided to choose a different S-box from the AES.

The S-box of AURORA is based on the inversion in the finite field GF($(2^4)^2$) defined by an irreducible polynomial $z^2 + z + \{1001\}$ for which the underlying GF(2^4) is defined by an irreducible polynomial $z'^4+z'+1$. These irreducible polynomials were chosen to optimize hardware implementations. The S-box is constructed by the following three steps:

Step 1. Apply the affine transformation over GF(2): f,

Step 2. Take the inverse in GF($(2^4)^2$), then

Step 3. Apply the affine transformation over GF(2): g.

The affine transformations f and g are applied to hide the algebraic structure (such as algebraically simple relations) in the finite field GF($(2^4)^2$). Considering implementation cost, the affine transformations f and g were chosen so that the following conditions are satisfied.

Let $f(x) = M_f \cdot x + c_f$ and $g(x) = M_g \cdot x + c_g$, where M_f and M_g are non-singular 8×8 matrices in GF(2), and c_f and c_g are constant vectors in GF(2) (See (2.2) and (2.3) in Sec. 2.2.4).

Conditions on M_f and M_g

1. The Hamming weight of each row/column vector of M_f and M_g is 2 or less.
2. The Hamming distance between the 1st and the 5th row vectors in M_f and M_g is 1. Similarly, the Hamming distance between the 2nd and the 6th row vectors, the 3rd and the 7th row vectors, and the 4th and the 8th row vectors in M_f and M_g is 1, respectively.
3. The Hamming weights of the 5th, 6th, 7th, and 8th row vectors are 1.

The numbers of candidates of M_f and M_g satisfying the conditions above are 40320, respectively.

Table 3.1: Security properties of the S-box.

maximum differential probability	2^{-6}
maximum linear probability	2^{-6}
minimum degree of Boolean polynomial	7
minimum number of terms in polynomial over GF(2^8)	252
length of cycle	255

Conditions on c_f and c_g

1. The Hamming weight of c_f and c_g is 4.
2. The Hamming weight of the upper 4-bit of c_f and c_g is 3, and the Hamming weight of the lower 4-bit of c_f and c_g is 1, respectively.

The number of candidates of c_f and c_g satisfying the conditions above is 17, respectively.

From all the possible $40320 \times 40320 \times 17 \times 17$ combinations of (M_f, M_g, c_f, c_g) satisfying the conditions above, we chose the first candidate that satisfied the security properties[2] shown in Table 3.1 according to the pseudocode below:

```
Select S-box (i.e. M_f, M_g, c_f, c_g)
000   for M_f index ← 0 to 40319 do
010       for M_g index ← 40319 down to 0 do
020           for c_f index ← 0 to 16 do
030               for c_g index ← 0 to 16 do
040                   if satisfy the conditions in Table 3.1
                      return (M_f index, M_g index, c_f index, c_g index).
```

Note that c_f and c_g are indexed by the values which can be represented as the concatenation of its individual bit values of the 8-bit vector in the order, respectively. M_f and M_g are indexed by the values which are generated by concatenating 8 8-bit row vectors from the most significant byte, respectively.

As a result, the candidate with $M_f$index= 0, $M_g$index= 40319, $c_f$index= 2, $c_g$index= 5 was chosen.

Diffusion Matrices

AURORA employs four different diffusion matrices $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ to improve the immunity against differential (and linear) attacks by using the Diffusion Switching Mechanism (DSM). The concept of DSM was first proposed by Shirai and Shibutani in 2004, followed by extended works [51, 52, 53, 50] and used in the blockcipher CLEFIA [54]. This technique is applicable to the AURORA structure. By using plural different matrices, we can prevent difference cancellations which can happen at the XOR operations in the structure. As a result the guaranteed number of active S-boxes is increased.

Let $\mathcal{B}_8(M)$ be the branch number of matrix M, which is defined as follows:

Definition 1 *Let $x \in \{0,1\}^{pn}$ represented as $x = [x_0 x_1 \dots x_{p-1}]$ where $x_i \in \{0,1\}^n$, then the bundle weight $w_n(x)$ is defined as $w_n(x) = \sharp\{x_i | x_i \neq 0\}$. Let $P : \{0,1\}^{pn} \to \{0,1\}^{qn}$. The branch number of P is defined as*

$$\mathcal{B}_n(P) = \min_{a \neq 0} \{w_n(a) + w_n(P(a))\} \ .$$

[2]The condition for the minimum number of terms in polynomial over GF(2^8) was not included in the selection conditions in the pseudocode, but the selected candidate satisfied this property.

Table 3.2: Diffusion matrices used in each building block of AURORA family.

AURORA-224/256	building block	MS_L	MS_R	CP	
	F-function matrices	F_0,F_1 $\mathcal{M}_0$,$\mathcal{M}_1$	F_2,F_3 $\mathcal{M}_2$,$\mathcal{M}_3$	F_1,F_0 $\mathcal{M}_1$,$\mathcal{M}_0$	
AURORA-384/512	building block	MS_L	MS_R	CP_L	CP_R
	F-function matrices	F_0,F_1 $\mathcal{M}_0$,$\mathcal{M}_1$	F_2,F_3 $\mathcal{M}_2$,$\mathcal{M}_3$	F_1,F_0 $\mathcal{M}_1$,$\mathcal{M}_0$	F_3,F_2 $\mathcal{M}_3$,$\mathcal{M}_2$
AURORA-224M/256M	building block	MS_L^M	MS_R^M	CP_L^M	CP_R^M
	F-function matrices	F_0,F_1 $\mathcal{M}_0$,$\mathcal{M}_1$	F_2,F_3 $\mathcal{M}_2$,$\mathcal{M}_3$	F_1,F_0 $\mathcal{M}_1$,$\mathcal{M}_0$	F_3,F_2 $\mathcal{M}_3$,$\mathcal{M}_2$

To utilize the DSM technique, AURORA uses two pairs of diffusion matrices ($\mathcal{M}_0$, $\mathcal{M}_1$), and ($\mathcal{M}_2$, $\mathcal{M}_3$) which satisfy the conditions I and II. Note that the elements of the matrices are in GF(2^8).

Condition I (MDS)

$$\mathcal{B}_8(\mathcal{M}_0) = \mathcal{B}_8(\mathcal{M}_1) = 5 \tag{3.1}$$
$$\mathcal{B}_8(\mathcal{M}_2) = \mathcal{B}_8(\mathcal{M}_3) = 5 \tag{3.2}$$

This is an optimal branch number for 4×4 matrices in GF(2^8), and the matrices satisfying this condition are called the MDS matrices.

Besides the condition I, the branch numbers of the concatenated matrices $\mathcal{M}_0|\mathcal{M}_1$, ${}^t\mathcal{M}_0^{-1}|\,{}^t\mathcal{M}_1^{-1}$, $\mathcal{M}_2|\mathcal{M}_3$, and ${}^t\mathcal{M}_2^{-1}|\,{}^t\mathcal{M}_3^{-1}$ should be optimal.

Condition II (DSM)

$$\mathcal{B}_8(\mathcal{M}_0|\mathcal{M}_1) = \mathcal{B}_8({}^t\mathcal{M}_0^{-1}|\,{}^t\mathcal{M}_1^{-1}) = 5 \tag{3.3}$$
$$\mathcal{B}_8(\mathcal{M}_2|\mathcal{M}_3) = \mathcal{B}_8({}^t\mathcal{M}_2^{-1}|\,{}^t\mathcal{M}_3^{-1}) = 5 \tag{3.4}$$

We call the pair of the matrices satisfying these conditions the "DSM pair". ($\mathcal{M}_0$, $\mathcal{M}_1$) is a DSM pair.

Actually, ($\mathcal{M}_0$, $\mathcal{M}_1$) is chosen according to (3.1) and (3.3). $\mathcal{M}_2$ and $\mathcal{M}_3$ are obtained by cyclically shifting each column of $\mathcal{M}_0$ and $\mathcal{M}_1$, respectively. It is easily proven that ($\mathcal{M}_2$, $\mathcal{M}_3$) is a DSM pair, i.e. (3.2) and (3.4) hold for $\mathcal{M}_2$ and $\mathcal{M}_3$ obtained in this way. Moreover, it is also shown that ($\mathcal{M}_0$, $\mathcal{M}_3$) and ($\mathcal{M}_1$, $\mathcal{M}_2$) are DSM pairs. Therefore, the DSM technique works not only in the single building block but also across the building blocks such as CP, MS_L, and MS_R. Table 3.2 shows diffusion matrices used in each building block of the AURORA family.

Since there are huge number of matrices satisfying the conditions I and II, we chose ($\mathcal{M}_0$, $\mathcal{M}_1$) considering implementation cost. Among circulant matrices with a low Hamming weight, we chose the pair of matrices which can be implemented efficiently in hardware, i.e., to minimize the XOR gate counts and the maximum delay. We chose $x^8 + x^4 + x^3 + x^2 + 1$ as the primitive polynomial in representing for the field GF(2^8). $\mathcal{M}_2$ and $\mathcal{M}_3$ are obtained by cyclically shifting each column of $\mathcal{M}_0$ and $\mathcal{M}_1$, respectively.

3.4.3 Data Rotating Function

The outputs from the message scheduling functions are XORed to the data in the chaining value processing function via the data rotating function DR. The function DR is adopted to incorporate bitwise operations with minimum additional cost and to prevent generic attacks exploiting byte/word-wise structure of the chaining value processing function and the message scheduling functions.

Table 3.3: Initial values and parameters in constant generation procedure.

AURORA-256	$IV_0 = (2^{1/2} - 1)2^{16}$	$mask_0 = (2^{1/3} - 1)2^{16}$	$mask_2 = (2^{1/5} - 1)2^{16}$
	$IV_1 = (3^{1/2} - 1)2^{16}$	$mask_1 = (3^{1/3} - 1)2^{16}$	$mask_3 = (3^{1/5} - 1)2^{16}$
AURORA-256M	$IV_0 = (5^{1/2} - 2)2^{16}$	$mask_0 = (5^{1/3} - 1)2^{16}$	$mask_2 = (5^{1/5} - 1)2^{16}$
	$IV_1 = (7^{1/2} - 2)2^{16}$	$mask_1 = (7^{1/3} - 1)2^{16}$	$mask_3 = (7^{1/5} - 1)2^{16}$
AURORA-512	$IV_0 = (11^{1/2} - 3)2^{16}$	$mask_0 = (11^{1/3} - 2)2^{16}$	$mask_2 = (11^{1/5} - 1)2^{16}$
	$IV_1 = (13^{1/2} - 3)2^{16}$	$mask_1 = (13^{1/3} - 2)2^{16}$	$mask_3 = (13^{1/5} - 1)2^{16}$

3.4.4 Truncation Functions

In AURORA-224, the 224-bit hash value is obtained by truncating the 256-bit final hash value by the truncation function TF_{224}. Similarly, in AURORA-384, the 384-bit hash value is obtained by truncating the 512-bit final hash value by the truncation function TF_{384}. These truncation functions do not just drop right-most bytes like the SHA-2 family, but drop bytes equally from every 64-bit block to make effective use of all the outputs from the F-functions in the last round of the compression function. See also Sec. 4.2.3.

3.4.5 Constant Generation

Role of Constants in the AURORA family

AURORA-224/256, AURORA-384/512, and AURORA-224M/256M, use 3, 4, and 4 sets of constants, respectively, as listed in Sec. 2.9.4.

The constants play an important role in security. They are used to make each module of *CPM* and *MSM* an independent function. In AURORA-256, it is expected that the finalization function *FF* behaves as an different function from the compression function *CF* by using a different set of constants. In AURORA-512 and AURORA-256M, it is expected that each of 8 compression functions, the mixing function, and the mixing function for finalization behaves an independent function from each other by using a different set of constants.

Design of Constant Generation Procedure

In AURORA, all the constants can be generated by the constant generation procedure. This strategy is more advantageous than storing all the independent random constants, especially in constrained environments where available memory is limited.

The constant generation procedure is designed to generate pseudorandom sequences by using simple operations such as XOR, bit-rotations, and so on. The design strategy is similar to the constant generator of the blockcipher CLEFIA [54]. The four 32-bit constant values used in each module of *CPM* and *MSM* in one round are generated from 16-bit values $T_{0,i}$ and $T_{1,i}$. $T_{0,i}$ and $T_{1,i}$ are updated every round by multiplication by x or x^{-1} in GF(2^{16}), respectively, where the primitive polynomial is $x^{16} + x^{15} + x^{13} + x^{11} + x^5 + x^4 + 1$ (=`0x1a831`). This primitive polynomial is also used in CLEFIA, and the choosing strategy is as follows. The lower 16-bit value is defined as `0xa831`= $(= \sqrt[3]{101} - 4) \cdot 2^{16})$. "101" is the smallest prime number satisfying the primitive polynomial condition in this form.

We set IV_0 and IV_1 (the initial values of $T_{0,i}$ and $T_{1,i}$) and the masking values $mask_0$, $mask_1$, $mask_2$, $mask_3$ as the first 16 bits of the fractional parts of the square/cube/fifth roots of prime numbers 2, 3, 5, 7, 11, and 13 as Table 3.3 shows. This is an evidence that there is no trapdoor in these values.

We selected the amounts of rotation $(r_0, r_1, r_2, r_3) = (8, 8, 8, 9)$ in Step 2 in the generation procedure of the constants, which is described in Sec. 2.9, by checking whether the generated sequences pass the statistical test suites: the mono bit test, the poker test, and the runs test [18].

In details, we checked the pseudorandomness of the first 20,000 bits of the following sequences for all the combinations of the amounts of rotation (r_0, r_1, r_2, r_3):

- Sequences of constants for AURORA-224/256
  - a sequence generated based on $T_{0,i}$: $\{CONC_{4i}, CONC_{4i+2}, CONC_{4i+4}, \dots\}$
  - a sequence generated based on $T_{1,i}$: $\{CONC_{4i+1}, CONC_{4i+3}, CONC_{4i+5}, \dots\}$
  - a sequence of constants used in CP: $\{CONC_{4i}, CONC_{4i+1}, CONC_{4i+2}, \dots\}$
  - a sequence of constants used in MS_L: $\{CONM_{L,4i}, CONM_{L,4i+1}, CONM_{L,4i+2}, \dots\}$
  - a sequence of constants used in MS_R: $\{CONM_{R,4i}, CONM_{R,4i+1}, CONM_{R,4i+2}, \dots\}$
  - a sequence of constants used in CPF: $\{CONC_{4i+68}, CONC_{4i+69}, CONC_{4i+70}, \dots\}$
  - a sequence of constants used in MSF_L: $\{CONM_{L,4i+32}, CONM_{L,4i+33}, CONM_{L,4i+34}, \dots\}$
  - a sequence of constants used in MSF_R: $\{CONM_{R,4i+32}, CONM_{R,4i+33}, CONM_{R,4i+34}, \dots\}$
- Sequences of constants for AURORA-384/512 in a similar manner to above
- Sequences of constants for AURORA-224M/256M in a similar manner to above

From the combinations of (r_0, r_1, r_2, r_3) which passed all the statistical tests above, we selected considering software implementation cost: i.e. we selected (r_0, r_1, r_2, r_3) with the smallest sum of distance from either 0, 8, or 16. As a result, we selected $(r_0, r_1, r_2, r_3) = (8, 8, 8, 9)$.

3.4.6 Initial Value

We believe that the security provided by the structure of the AURORA family does not depend on the value of the initial value, so any value can be used as the initial value. We chose the constants such as all-0 or all-1, because we don't need additional area to memorize the specific constants.

All of AURORA-256, AURORA-512, and AURORA-256M use the same all-0 constants. We don't identify any security problem, because each module used in AURORA-256, AURORA-512, and AURORA-256M are different due to different matrices and constants. Similarly, all of AURORA-224, AURORA-384, and AURORA-224M use the same all-1 constants, but we don't identify any security problem.

Chapter 4

Security of AURORA

4.1 Expected Strength

For AURORA-n, $n \in \{224, 256, 384, 512\}$ and AURORA-nM, $n \in \{224, 256\}$, each hash function is expected to satisfy preimage resistance of approximately n bits, second preimage resistance of approximately $n - k$ bits for any message shorter than 2^k bits, and the collision resistance of approximately $n/2$ bits. Several attempts to attack the AURORA family by the above attack scenarios are described in Sec. 4.3.1-4.3.3.

Moreover, all members in AURORA family provide resistance to length-extension attacks (see Sec. 4.3.4).

Resistance against multicollision attack is achieved in AURORA-224M/256M. See Sec. 4.3.5.

Also, any m-bit hash function specified by taking a fixed subset of the function's output bits is expected to meet the above requirements with m replacing n.

If one of AURORA-n and AURORA-nM is used with HMAC to construct a PRF [23], the PRF resists any distinguishing attack that requires much fewer than $2^{n/2}$ queries and significantly less computation than a preimage attack (see Sec. 4.2.1).

If AURORA-n or AURORA-nM is used in the randomized hashing scheme [39], it provides n bits of security against the following attack. 1) An attacker gets a randomized hash of M_1 and randomization value r_1 that has been randomly chosen without the attacker's control, 2) Find M_2 and r_2 that yield the same randomized hash value. Since AURORA hash functions are secure hash functions, it can be expected that the randomized hashing using AURORA is a secure scheme.

4.2 Security Argument

4.2.1 Security of HMAC using AURORA

HMAC-AURORA-224/256 specified in Sect. 6.2 employs *CF* and *FF* as their compression functions and its domain extension is the same as the MD transform. Fig. 4.1 shows the structure of HMAC-AURORA-224/256. According to the discussion in Sect. 4.2.3, *CF* and *FF* are expected to be pseudorandom functions (PRFs) when keyed via the IV. They are also expected to be PRFs when keyed via its data input. HMAC using the MD transform was proved to be a PRF when keyed via the IV assuming that the underlying compression function is a PRF when keyed via the IV and when keyed via its data input [4]. Therefore HMAC-AURORA-224/256 is expected to be a good PRF [4].

Fig. 4.2 shows a structure of HMAC-AURORA-384/512 specified in Sect. 6.2. It can be regarded that the iterated compression function of HMAC-AURORA-384/512 consists of the following 17 compression functions.

$$
\begin{aligned}
CFMFF_1^1(X, CV) &= MFF(CF_1(X, CV)), \\
&\vdots \\
CFMFF_7^1(X_0||...||X_6) &= MFF(CF_7(X_6, CF_6(X_5, ..., CF_1(X_0, CV)...))), \\
CFMF^1(X_0||...||X_6) &= MF(CF_7(X_6, CF_6(X_5, ..., CF_1(X_0, CV)...))), \\
CFMFF_0^0(X, CV) &= MFF(CF_0(X, CV)), \\
&\vdots \\
CFMFF_7^0(X_0||...||X_7, CV) &= MFF(CF_7(X_7, CF_6(X_6, ..., CF_0(X_0, CV)...))), \\
CFMF^0(X_0||...||X_7, CV) &= MF(CF_7(X_7, CF_6(X_6, ..., CF_0(X_0, CV)...)))
\end{aligned}
$$

NMAC using the MD transform was proved to be a PRF when keyed via the IV assuming that the underlying compression function is a PRF when keyed via the IV. HMAC-AURORA-384/512 can be a PRF when keyed via the IV if it satisfies that (1) the 17 compression functions used in the iterated compression function of HMAC-AURORA-384/512 is PRFs when keyed via the IV, (2) *MFF* is a PRF when keyed via the IV, and (3) keys K'_{IN} and K'_{OUT} are chosen at random. First, since all 17 compression functions employ *MFF* or *MF* as the final function, they can be regarded as PRFs when keyed via the IV. Second, the *MFF* can also be a PRF when keyed via the IV. Finally, if the inputs of two $CF_0(\cdot, H_0)$s, K_{IN} and K_{OUT} are chosen at random, the outputs K'_{IN} and K'_{OUT} will be almost random when H_0 is fixed. Thus HMAC-AURORA-384/512 is expected to be a good PRF when keyed via the IV. Also by the similar manner, HMAC-AURORA-224M/256M is expected to be a good PRF.

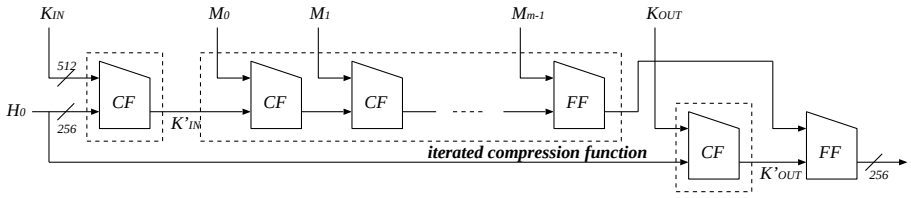

Figure 4.1: HMAC-AURORA-224/256.

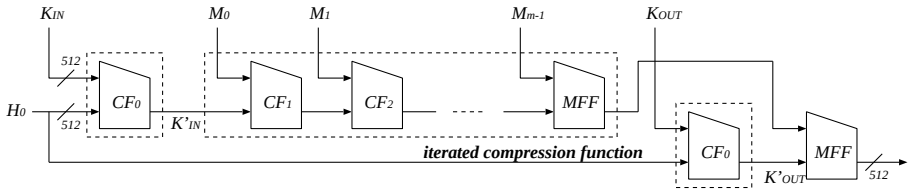

Figure 4.2: HMAC-AURORA-384/512.

4.2.2 Security Proofs of DMMD Transform

In this section, we present the security theorems and their proofs on important security properties of the Double-Mix Merkle Damgård (DMMD) transform. We show that:

- the DMMD transform is collision resistant, as a hash function, in the random oracle model, and
- the DMMD transform is preimage resistant if *MFF* is preimage resistant.

The first result requires some assumption on the adversary, but it still covers a class of known attacks.

Collision resistance of the DMMD transform. We first restate the transform to fix the notation. Let $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1} : \{0,1\}^{m+n} \to \{0,1\}^n$, $F_0 : \{0,1\}^{2n} \to \{0,1\}^{2n}$, and $F_1 : \{0,1\}^{2n} \to \{0,1\}^c$ be functions. The DMMD transform internally uses $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, F_0, and F_1. Two initial values $H_0, \hat{H}_0 \in \{0,1\}^n$ are fixed constants. Without loss of generality, we assume it takes an (already padded) message $M = (M_0, \ldots, M_{\mu-1}) \in (\{0,1\}^m)^*$ as input. The block length, μ, may vary across the messages. As we assume that the padding is properly done, the last block, $M_{\mu-1}$, contains the length of the original message, and therefore $\mu \geq 1$. The output is $H_{\mu+1} \in \{0,1\}^c$. It works as in Figure 4.3.

Algorithm $\mathrm{DMMD}^{f_0,\hat{f}_0\ldots,f_{l-1},\hat{f}_{l-1},F_0,F_1}(M)$**:**
for $i \leftarrow 0$ to $\mu - 1$ do
  $(H_{i+1}, \hat{H}_{i+1}) \leftarrow (f_{i \bmod l}(M_i, H_i), \hat{f}_{i \bmod l}(M_i, \hat{H}_i))$
  if $(0 < i < \mu - 1) \wedge (i \bmod l = l - 1)$ then
    $(H_{i+1}, \hat{H}_{i+1}) \leftarrow F_0(H_{i+1}, \hat{H}_{i+1})$
$H_{\mu+1} \leftarrow F_1(H_\mu, \hat{H}_\mu)$
return $H_{\mu+1}$

Figure 4.3: Algorithm of $\mathrm{DMMD}^{f_0,\hat{f}_0\ldots,f_{l-1},\hat{f}_{l-1},F_0,F_1}(M)$.

Now we describe our collision finding adversary A_1. A_1 has oracle access to $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, F_0, and F_1, and outputs $M, M' \in \{0,1\}^*$ such that $M \neq M'$. A_1 makes q queries to each of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}, F_0$, and F_1. We say A_1 wins if $\mathrm{DMMD}(M) = \mathrm{DMMD}(M')$. A_1 may access the oracles in an arbitrarily order.

Now A_1's advantage is defined as

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}}(A_1) = \Pr(A_1^{f_0,\hat{f}_0,\ldots,f_{l-1},\hat{f}_{l-1},F_0,F_1} \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}, F_0, F_1$ and A_1's coin (if any). A_1 is assumed to know $\mathrm{DMMD}(M) = \mathrm{DMMD}(M')$ holds when A_1 outputs M and M'.

Function $\mathcal{G}$. We next define a function $\mathcal{G}$, which corresponds to "one loop" of the DMMD transform. It internally uses $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$ and F_0. It takes two initial values $H_0, \hat{H}_0 \in \{0,1\}^n$, and a message $M = (M_0, \ldots, M_{\mu-1})$ of at most l blocks (i.e., $\mu \leq l$) as inputs, and produces the output $(H_\mu, \hat{H}_\mu) \in (\{0,1\}^n)^2$. It works as in Figure. 4.4.

We next describe our collision finding adversary A_2 against $\mathcal{G}$. A_2 has access to $l+1$ oracles, $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$, and F_0. $(f_i(\cdot,\cdot), \hat{f}_i(\cdot,\cdot))$ takes $(M, H, \hat{H})$ as input, and the output is $(h, \hat{h}) = (f_i(M, H), \hat{f}_i(M, \hat{H}))$. The $2l+1$ functions, $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, and F_0 are random oracles. A_2 may access the oracles in an arbitrarily order, and outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where M and M' are at most l blocks.

Algorithm $\mathcal{G}^{f_0,\hat{f}_0\cdots,f_{l-1},\hat{f}_{l-1},F_0}((H_0,\hat{H}_0),M)$**:**
$(H_0,\hat{H}_0) \leftarrow F_0(H_0,\hat{H}_0)$
for $i \leftarrow 0$ to $\mu-1$ do
    $(H_{i+1},\hat{H}_{i+1}) \leftarrow (f_i(M_i,H_i),\hat{f}_i(M_i,\hat{H}_i))$
return $(H_\mu,\hat{H}_\mu)$

Figure 4.4: Algorithm of $\mathcal{G}^{f_0,\hat{f}_0\cdots,f_{l-1},\hat{f}_{l-1},F_0}((H_0,\hat{H}_0),M)$, where $M=(M_1,\ldots,M_{\mu-1})$ and $\mu \le l$.

A_2 makes $2q$ queries to each of $(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})$, and q queries to F_0. We say A_2 wins if $\mathcal{G}((H,\hat{H}),M)=\mathcal{G}((H',\hat{H}'),M')$. A_2's advantage is defined as

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G}}(A_2)=\Pr(A_2^{(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1}),F_0} \text{ wins}),$$

where the probability is taken over the choices of $f_0,\hat{f}_0,\ldots,f_{l-1},\hat{f}_{l-1}$, F_0 and A_2's coin (if any). A_2 is assumed to know $\mathcal{G}((H,\hat{H}),M)=\mathcal{G}((H',\hat{H}'),M')$ holds when A_2 outputs $((H,\hat{H}),M)$ and $((H',\hat{H}'),M')$.

Function $\mathcal{F}$. We next describe the function $\mathcal{F}$. It internally uses $f_0,\hat{f}_0,\ldots,f_{l-1},\hat{f}_{l-1}$. It takes two initial values $H_0,\hat{H}_0 \in \{0,1\}^n$, a message $M=(M_0,\ldots,M_{\mu-1})$ of at most l blocks (thus $\mu \le l$) as inputs, and produces the output $(H_\mu,\hat{H}_\mu) \in (\{0,1\}^n)^2$. It works as in Figure. 4.5.

Algorithm $\mathcal{F}^{f_0,\hat{f}_0\cdots,f_{l-1},\hat{f}_{l-1}}((H_0,\hat{H}_0),M)$**:**
for $i \leftarrow 0$ to $\mu-1$ do
    $(H_{i+1},\hat{H}_{i+1}) \leftarrow (f_i(M_i,H_i),\hat{f}_i(M_i,\hat{H}_i))$
return $(H_\mu,\hat{H}_\mu)$

Figure 4.5: Algorithm of $\mathcal{F}^{f_0,\hat{f}_0\cdots,f_{l-1},\hat{f}_{l-1}}((H_0,\hat{H}_0),M)$, where $M=(M_0,\ldots,M_{\mu-1})$, and $\mu \le l$.

$\mathcal{F}$ is the same as $\mathcal{G}$ without initial computation of F_0. We note that $\mathcal{F}$ is not collision resistant as a $2n$-bit compression function. However, by making a restriction on the chaining values, and making an assumption on the adversary on the order of oracle access, it is possible to show its collision resistance.

Let $S=\{S_1,\ldots,S_s\}$, $S_i \in \{0,1\}^n$, be a multi-set of strings. For any integer $K \ge 1$, we say that S is K-coll if there are K indices $1 \le i_1 < \cdots < i_K \le s$ such that $S_{i_1}=\cdots=S_{i_K}$ holds. The strings $(S_{i_1},\ldots,S_{i_K})$ is said to be a K collision. We say S is K-COLL if S is K-coll but it is not $(K+j)$-coll for all $j \ge 1$. If S is not K-COLL, then we say S is K-COLL-free, which means that S may have a $K-1$ collision but does not have K (or more) collision.

Let $K \ge 1$ and $s \ge 1$ be fixed integers and let $\mathcal{H}_0=\{(H_0,\hat{H}_0),\ldots,(H_{s-1},\hat{H}_{s-1})\}$, $(H_i,\hat{H}_i) \in (\{0,1\}^n)^2$, be a fixed set of strings such that

$$\begin{cases} \text{the multi-set } \mathcal{H}_0^H=\{H_0,\ldots,H_{s-1}\} \text{ is } K\text{-COLL-free, and} \\ \text{the multi-set } \mathcal{H}_0^{\hat{H}}=\{\hat{H}_0,\ldots,\hat{H}_{s-1}\} \text{ is } K\text{-COLL-free.} \end{cases} \tag{4.1}$$

Note we assume that $(H_0,\hat{H}_0)$, the fixed initial value for the DMMD transform, is included in $\mathcal{H}_0$. Now we describe our collision finding adversary A_3 against $\mathcal{F}$. A_3 has access to l oracles, $(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})$. The $2l$ functions, $f_0,\hat{f}_0,\ldots,f_{l-1},\hat{f}_{l-1}$, are random oracles. A_3 may access the oracles in an arbitrarily order, and outputs $((H,\hat{H}),M)$ and $((H',\hat{H}'),M')$ such that $((H,\hat{H}),M) \neq ((H',\hat{H}'),M')$, where M and M' are at most l blocks. A_3 makes $2q$ queries to each of $(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})$. We say A_3 wins if $\mathcal{F}((H,\hat{H}),M)=\mathcal{F}((H',\hat{H}'),M')$, where $(H,\hat{H}),(H',\hat{H}') \in \mathcal{H}_0$ must hold.

Now A_3's advantage is defined as

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) = \Pr(A_3^{(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})} \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$ and A_3's coin (if any). A_3 is assumed to know $\mathcal{F}((H,\hat{H}),M) = \mathcal{F}((H',\hat{H}'),M')$ holds when A_3 outputs $((H,\hat{H}),M)$ and $((H',\hat{H}'),M')$.

We next describe another adversary A_4. A_4 tries to make a collision between $\mathcal{G}$ and $\mathcal{F}$. Now A_4 has access to $l+1$ oracles, $(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})$, and F_0, as A_2. A_4 may access the oracles in an arbitrarily order, and outputs $((H,\hat{H}),M)$ and M', where M and M' are at most l blocks in lengths. A_4 makes $2q$ queries to each of $(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})$, and q queries to F_0. We say A_4 wins if $\mathcal{G}((H,\hat{H}),M) = \mathcal{F}((H_0,\hat{H}_0),M')$, where $(H_0,\hat{H}_0)$ is the fixed initial value of the DMMD transform.

Now A_4's advantage is defined as

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G},\mathcal{F}}(A_4) = \Pr(A_4^{(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1}),F_0} \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, F_0 and A_4's coin (if any). A_4 is assumed to know $\mathcal{G}((H,\hat{H}),M) = \mathcal{F}((H_0,\hat{H}_0),M')$ holds when A_4 outputs $((H,\hat{H}),M)$ and M'.

Now we have the following result.

Theorem 1 (Collision resistance of the DMMD transform) *Let A_1, A_2, A_3, and A_4 be adversaries, described as above. Then we have*

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}}(A_1) \le \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G}}(A_2) + \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) + \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G},\mathcal{F}}(A_4) + \frac{q^2}{2^{c+1}}.$$

Proof. Let A_1' be an adversary, exactly the same as A_1, but outputs M and M' such that $\mathrm{DMMD}'(M) = \mathrm{DMMD}'(M')$ and $M \neq M'$, where DMMD' is the same as DMMD but without the final F_1 function. First, we claim that

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}}(A_1) \le \mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}'}(A_1') + \frac{q^2}{2^{c+1}}, \tag{4.2}$$

since without finding a collision against DMMD', A_1 is forced to find a collision against F_1, i.e., a random oracle of c-bit output.

Let $cut(\cdot) : (\{0,1\}^m)^* \to (\{0,1\}^m)^*$ be a function that takes a message $M = (M_0,\ldots,M_{\mu-1})$ as its input. The output is defined as follows.

- if $\mu \bmod l = 0$, then return the last l blocks $(M_{\mu-l},\ldots,M_{\mu-1})$.
- else return the last $\mu \bmod l$ blocks $(M_{\mu-(\mu \bmod l)},\ldots,M_{\mu-1})$.

Now there are three cases depending on the length of messages that A_1' outputs. Let $M = (M_0,\ldots,M_{\mu-1}), M' = (M_0',\ldots,M_{\mu'-1}')$ be the messages;

- Case 1: $(\mu \bmod l \neq 0) \wedge (\mu > l) \wedge (\mu' \bmod l \neq 0) \wedge (\mu' > l)$, or $(\mu \bmod l = 0) \wedge (\mu' \bmod l = 0)$, or $(\mu \bmod l = 0) \wedge (\mu' \bmod l \neq 0) \wedge (\mu' > l)$.
- Case 2: $(\mu \bmod l \neq 0) \wedge (\mu < l) \wedge (\mu' \bmod l \neq 0) \wedge (\mu' < l)$.
- Case 3: $(\mu \bmod l = 0) \wedge (\mu' \bmod l \neq 0) \wedge (\mu' < l)$, or $(\mu \bmod l \neq 0) \wedge (\mu > l) \wedge (\mu' \bmod l \neq 0) \wedge (\mu' < l)$.

In case 1, A_2 can simulate A_1''s oracles and by computing DMMD(M) and DMMD(M'), A_2 obtains the desired $(H, \hat{H}), (H', \hat{H}')$, and $cut(M)$ and $cut(M')$ correspond to the messages that A_2 outputs. Similarly, in case 2, A_3 can output $((H_0, \hat{H}_0), M)$ and $((H_0, \hat{H}_0), M')$, where $(H_0, \hat{H}_0)$ is the fixed initial value of the DMMD transform. In case 3, A_4 can compute $(H, \hat{H})$ by computing DMMD(M), and $cut(M)$ and M' itself are the messages that A_4 outputs.

We note that A_1' makes q queries to f_i and q queries to $\hat{f}_i$, while A_2, A_3, and A_4 make $2q$ queries to $(f_i, \hat{f}_i)$ oracle. Therefore A_2, A_3, and A_4 can simulate A_1''s oracles. □

We next show that A_2 and A_4 are essentially equivalent to A_3. We show the following result.

Lemma 1 (Relation between A_2, A_3, and A_4) *Let A_2, A_3, A_4 be the adversaries, described as above. Then we have*

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G}}(A_2) \leq \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) + \frac{2q^K}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}$$

and

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G},\mathcal{F}}(A_4) \leq \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) + \frac{2q^K}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}.$$

Proof. A_2 has a random oracle F_0, where the $\mathcal{F}$ function is followed. Now since the output of F_0 is a $2n$-bit truly random string, we may give all answers to A_2, before A_2 has oracle access to $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$. That is, we let A_2 know the response before making queries, and let A_2 choose the corresponding input value. Clearly, this does not decrease the success probability of A_2. Now we give q random strings to A_2. Let $\{(H_1, \hat{H}_1), \ldots, (H_q, \hat{H}_q)\}$, $(H_i, \hat{H}_i) \in (\{0,1\}^n)^2$, be the q random strings. Now since we have

$$\begin{cases} \Pr(\{H_1, \ldots, H_q\} \text{ contains } K \text{ collision}) \leq q^K/2^{n(K-1)}, \\ \Pr(\{\hat{H}_1, \ldots, \hat{H}_q\} \text{ contains } K \text{ collision}) \leq q^K/2^{n(K-1)}, \\ \Pr(\{(H_1, \hat{H}_1), \ldots, (\hat{H}_q, \hat{H}_q)\} \text{ contains 2 collision}) \leq q^2/2^{2n+1}, \end{cases}$$

we have

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{G}}(A_2) \leq \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) + \frac{2q^K}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}.$$

By the same argument, we have a bound for A_4. □

Therefore, the bound in Theorem 1 can be re-written as

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}}(A_1) \leq 3\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) + \frac{4q^K}{2^{n(K-1)}} + \frac{q^2}{2^{2n}} + \frac{q^2}{2^{c+1}}.$$

To show that the DMMD transform is secure against collision attacks, it is enough to show that finding a collision among the chaining values for $\mathcal{F}$ is a difficult task. To further relax the assumption, we present another adversary A_6.

Adversary A_6. We describe our collision finding adversary A_6 against $\mathcal{F}$. A_6 is exactly the same as A_3, but the output of A_6 is $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where M and M' are at most l blocks, and $|M| = |M'|$.

Notice that the restriction on the output of A_3 is that M and M' are at most l blocks in lengths, i.e., $|M| \neq |M'|$ is allowed.

We have the following result.

Lemma 2 (Relation between A_3 and A_6) *Let A_3 and A_6 be the adversaries, described as above. Then we have*

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_3) \leq \mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_6) + \frac{4l^2q^2}{2^{2n+1}}.$$

Proof. There are two cases that A_3 wins, case $|M| = |M'|$ and case $|M| \neq |M'|$. Consider the case where A_3 wins with M and M' such that $|M| = \mu m$ and $|M'| = \mu' m$, but $\mu \neq \mu'$. Then A_3 must have found the collision between the outputs of the $(f_{\mu-1}, \hat{f}_{\mu-1})$ oracle and the $(f_{\mu'-1}, \hat{f}_{\mu'-1})$ oracle, i.e., a collision between $2n$ bit independent random strings. Since there are l intermediate values, and since A_3 makes $2q$ oracle calls, we have the bound. □

Next, we further relax the assumption.

Adversary A_7. We describe our collision finding adversary A_7 against $\mathcal{F}$. A_7 is exactly the same as A_6, but it takes $\mu \leq l$ as the input, and the output of A_7 is $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where M and M' are exactly μ blocks. Notice that the output message of A_6 may be adaptively chosen, A_7 has to output μ blocks of messages.

We have the following result.

Lemma 3 (Relation between A_6 and A_7) *Let A_6 and A_7 be the adversaries, described as above. Then we have*

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_6) \leq l\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_7).$$

A proof is based on the fact that, if A_6 succeeds, then A_7 with some input $\mu = 0, \ldots, l-1$ should also succeed.

Overall, the bound on A_1 is thus

$$\mathbf{Adv}^{\mathrm{coll}}_{\mathrm{DMMD}}(A_1) \leq 3l\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_7) + \frac{12l^2q^2}{2^{2n+1}} + \frac{4q^K}{2^{n(K-1)}} + \frac{q^2}{2^{2n}} + \frac{q^2}{2^{c+1}}.$$

To show that the DMMD transform is secure against collision attacks, it is enough to show that $\mathbf{Adv}^{\mathrm{coll}}_{\mathcal{F}}(A_7)$ is small enough. However, it appears that the proof is not simple. Instead, we consider another adversary A'_7 that works exactly the same as A_7, but is restricted in the order of oracle access. A'_7 has to access to oracles $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$ in this order.

Before showing that A'_7 has a small success probability, we present the analysis on the compression function which will be used in the analysis of A'_7.

Collision resistance of the compression function. The compression function of the DMMD transform itself is not collision resistant as a $2n$-bit compression function. However, if we make the assumption on the chaining values that the adversary can use, then it is possible to show its collision resistance.

Let $K' \geq 1$ and $s \geq 1$ be fixed integers and let $\mathcal{H} = \{(S_1, T_1), \ldots, (S_s, T_s)\}$, $(S_i, T_i) \in (\{0,1\}^n)^2$, be a multi-set of strings such that

- the multi-set $\mathcal{H}^S = \{S_1, \ldots, S_s\}$ is K'-COLL-free,
- the multi-set $\mathcal{H}^T = \{T_1, \ldots, T_s\}$ is K'-COLL-free, and
- $\mathcal{H}$ is 2-COLL-free (thus, $\mathcal{H}$ is actually a set).

Now we consider the following adversary A_8 that has access to an oracle $(f_0(\cdot,\cdot), f_1(\cdot,\cdot))$ that, on input (M, U, V), returns $(X, Y) = (f_0(M, U), f_1(M, V))$. Both f_0 and f_1 are random oracles. We consider A_8 with the following constraint: For the i-th query (M_i, U_i, V_i) that A_8 makes, (U_i, V_i) has to be chosen from the multi-set $\mathcal{H}$. Let q' be the number of queries that A_8 makes.

Let us define a multi-set $\mathcal{L}_i$, $0 \leq i \leq q'$, as follows. $\mathcal{L}_i$ is the multi-set of (M, U, V, X, Y) such that A_8 knows $(X, Y) = (f_0(M, U), f_1(M, V))$ for some (M, U, V) where $(U, V) \in \mathcal{H}$, after the i-th query (thus $\mathcal{L}_0 = \emptyset$).

We also define multi-sets $\mathcal{U}_i$ and $\mathcal{V}_i$, $0 \leq i \leq q'$, as follows. $\mathcal{U}_i$ is the multi-set of (M, U, X) such that A_8 knows $X = f_0(M, U)$ for some (M, U) where $U \in \mathcal{H}^S$, after the i-th query. Similarly, $\mathcal{V}_i$ is the multi-set of (M, V, Y) such that A_8 knows $Y = f_1(M, V)$ for some (M, V) where $V \in \mathcal{H}^T$, after the i-th query.

We next define the following associate multi-sets of $\mathcal{L}_i$, $\mathcal{U}_i$, and $\mathcal{V}_i$:

- $\mathcal{L}_i^{M,U,V}$ consists of (M,U,V) such that $(M,U,V,X,Y) \in \mathcal{L}_i$ for some (X,Y).
- $\mathcal{U}_i^{M,U}$ consists of (M,U) such that $(M,U,X) \in \mathcal{U}_i$ for some X.
- $\mathcal{V}_i^{M,V}$ consists of (M,V) such that $(M,V,Y) \in \mathcal{V}_i$ for some Y.

We also use $\mathcal{L}_i^{X,Y}$, $\mathcal{L}_i^X$, $\mathcal{L}_i^Y$, $\mathcal{U}_i^X$, and $\mathcal{V}_i^Y$, which are defined in an obvious way.

On making a query, A_8 may use the same (S_j, T_j) several times, but we assume it does not make pointless queries. That is, A_8 never makes a query $(M_{i+1}, U_{i+1}, V_{i+1})$ if $(M_{i+1}, U_{i+1}) \in \mathcal{U}_i^{M,U}$ and $(M_{i+1}, V_{i+1}) \in \mathcal{V}_i^{M,V}$.

Let $K \geq 2$ be a fixed integer. We say A_8 wins if, after making q' queries,

- $\mathcal{L}_{q'}^X$ is $K'K$-COLL,
- $\mathcal{L}_{q'}^Y$ is $K'K$-COLL, or
- $\mathcal{L}_{q'}^{X,Y}$ is 2-COLL.

We show that A_8 has a low probability in winning the game.

Lemma 4 (Collision resistance of the compression function) *Let A_8 be the adversary, described as above. Assume A_8 makes at most q' queries. Then we have*

$$\Pr(A_8^{f_0,f_1} \text{ wins}) \leq \frac{{q'}^2 K'}{2^{2n}} + \frac{2q'(K')^2 K}{2^n} + \frac{6q'K'}{2^n} + \frac{2{q'}^K}{2^{n(K-1)}}.$$

We present four lemmata to prove Lemma 4.

Let win_i, $0 \leq i \leq q'$, be the event that A_8 wins at the i-th query, and $\overline{\text{win}}_i$ be its complement event. Then we have

$$\Pr(A_8^{f_0,f_1} \text{ wins}) \leq \sum_{0 \leq i \leq q'-1} \Pr(\text{win}_{i+1} \mid \overline{\text{win}}_1 \wedge \cdots \wedge \overline{\text{win}}_i).$$

For notational simplicity, let $\overline{\text{WIN}}_i$ be the event $\overline{\text{win}}_1 \wedge \cdots \wedge \overline{\text{win}}_i$. We then have

$$\begin{aligned}
\Pr(A_8^{f_0,f_1} \text{ wins}) \leq \Pr(\text{win}_{i+1} \mid \overline{\text{WIN}}_i) \quad \leq \quad & \sum_{0 \leq i \leq q'-1} \Pr(\mathcal{L}_{i+1}^X \text{ is } K'K\text{-COLL} \mid \overline{\text{WIN}}_i) && (4.3)\\
& + \sum_{0 \leq i \leq q'-1} \Pr(\mathcal{L}_{i+1}^Y \text{ is } K'K\text{-COLL} \mid \overline{\text{WIN}}_i) && (4.4)\\
& + \sum_{0 \leq i \leq q'-1} \Pr(\mathcal{L}_{i+1}^{X,Y} \text{ is 2-COLL} \mid \overline{\text{WIN}}_i). && (4.5)
\end{aligned}$$

We first have the following lemma that shows the upper bound on (4.3).

Lemma 5

$$\sum_{0 \leq i \leq q'-1} \Pr(\mathcal{L}_{i+1}^X \text{ is } K'K\text{-COLL} \mid \overline{\text{WIN}}_i) \leq \frac{{q'}^K}{2^{n(K-1)}}.$$

Proof. From the assumption on $\mathcal{H}$, for the $(i+1)$-st query $(M_{i+1}, U_{i+1}, V_{i+1})$ that A_8 makes, we see that at most K' colliding elements are added to $\mathcal{L}_i^X$. Suppose that A_8 makes a query $(M_{i+1}, U_{i+1}, V_{i+1})$ such that $(U_{i+1}, V_{i+1}) = (S_j, T_j)$, where

- $S_j = S_{j_1} = \cdots = S_{j_{K'-2}}$,
- $(M_{i+1}, S_j) = (M_{i+1}, S_{j_1}) = \cdots = (M_{i+1}, S_{j_{K'-2}}) \notin \mathcal{U}_i^{M,U}$, and
- $(M_{i+1}, T_j), (M_{i+1}, T_{j_1}), \ldots, (M_{i+1}, T_{j_{K'-2}}) \in \mathcal{V}_i^{M,V}$.

Then, $K'-1 < K'$ colliding elements will be added to $\mathcal{L}_i^X$, but not more. Note that the added value, X_{i+1}, is itself a random n-bit string (even under the condition that $\overline{\mathrm{WIN}}_i$). Therefore, in order to produce a $K'K$ collision, A_8 has to produce a K collision among the random strings returned by the oracle. Since at most q' random values are returned by the oracle, we have

$$\sum_{0 \le i \le q'-1} \Pr(\mathcal{L}_{i+1}^X \text{ is } K'K\text{-COLL} \mid \overline{\mathrm{WIN}}_i) \le \frac{{q'}^K}{2^{n(K-1)}},$$

and we have the claimed bound. □

By exactly the same argument, we have the following lemma for (4.4).

Lemma 6

$$\sum_{0 \le i \le q'-1} \Pr(\mathcal{L}_{i+1}^Y \text{ is } K'K\text{-COLL} \mid \overline{\mathrm{WIN}}_i) \le \frac{{q'}^K}{2^{n(K-1)}}.$$

Before analyzing $\Pr(\mathcal{L}_{i+1}^{X,Y}$ is 2-COLL $\mid \overline{\mathrm{WIN}}_i)$, we define the events bad_i^U and bad_i^V, $0 \le i \le q'$, as follows.

- We say bad_i^U occurs if, the i-th query (M_i, U_i, V_i) satisfies $(M_i, U_i) \notin \mathcal{U}_{i-1}^{M,U}$ where $(U_i, V_i) = (S_\ell, T_\ell)$, and there exists (M_j, U_j, X_j) such that
  - $(M_j, U_j, X_j) \in \mathcal{U}_{i-1}$,
  - $U_j = S_{\ell'}$,
  - $M_i = M_j$, $S_\ell \neq S_{\ell'}$, $T_\ell = T_{\ell'}$, and $X_i = X_j$.
- Similarly, we say bad_i^V occurs if, the i-th query (M_i, U_i, V_i) satisfies $(M_i, V_i) \notin \mathcal{V}_{i-1}^{M,V}$ where $(U_i, V_i) = (S_\ell, T_\ell)$, and there exists (M_j, V_j, Y_j) such that
  - $(M_j, V_j, Y_j) \in \mathcal{V}_{i-1}$,
  - $V_j = T_{\ell'}$,
  - $M_i = M_j$, $S_\ell = S_{\ell'}$, $T_\ell \neq T_{\ell'}$, and $Y_i = Y_j$.

We show that these bad events rarely occur.

Lemma 7

$$\Pr(\mathrm{bad}_i^U) \le \frac{K'}{2^n} \textit{ and } \Pr(\mathrm{bad}_i^V) \le \frac{K'}{2^n}.$$

Proof. We first consider $\Pr(\mathrm{bad}_i^U)$. We claim that there are at most K' choices for (M_j, U_j, X_j). To see this, our $\mathcal{H}^T$ contains only $K'-1$ collisions, and let $(S_1, T_1), \ldots, (S_{K'-1}, T_{K'-1})$ be the elements of $\mathcal{H}$ such that $T_1 = \cdots = T_{K'-1}$. Now we see that the probability of bad_i^U is maximized when A_8 has already obtained $f_0(M_i, S_1), \ldots, f_0(M_i, S_{K'-2})$, and let $U_i = S_{K'-1}$. In this case, A_8 has $K'-2 < K'$ target values for a collision. Now since the returned value X_i is a random n-bit string, we have the claimed bound.

By a similar argument, we have $\Pr(\mathrm{bad}_i^V) \le K'/2^n$. □

Note that if bad_i^U (or bad_i^V) occurs, then A_8 has succeeded in making $\mathcal{L}_i^{X,Y}$ 2-COLL at the i-th query (if $(M_j, U_j, V_j, X_j, Y_j) \in \mathcal{L}_i$), or can obviously succeed at the $(i+1)$-st query by making (M_j, U_j, V_j) (if $(M_j, U_j, V_j, X_j, Y_j) \notin \mathcal{L}_i$). Without loss of generality, we assume that A_8 makes the 2-COLL occur at the $(i+1)$-st query if bad_i^U (or bad_i^V) occurs and $(M_j, U_j, V_j, X_j, Y_j) \notin \mathcal{L}_i$.

We have the following lemma on $\Pr(\mathcal{L}_{i+1}^{X,Y}$ is 2-COLL $\mid \overline{\mathrm{WIN}}_i)$.

Lemma 8

$$\Pr(\mathcal{L}_{i+1}^{X,Y} \text{ is 2-COLL} \mid \overline{\mathrm{WIN}}_i) \le \frac{2K'i}{2^{2n}} + \frac{2(K')^2K}{2^n} + \frac{6K'}{2^n}.$$

Proof. From the assumption on $\mathcal{H}$, for the $(i+1)$-st query $(M_{i+1}, U_{i+1}, V_{i+1})$ that A_8 makes, at most $2K'$ elements are added to $\mathcal{L}_i^{X,Y}$. Suppose that A_8 makes a query $(M_{i+1}, U_{i+1}, V_{i+1})$ such that $(U_{i+1}, V_{i+1}) = (S_j, T_j)$, where

- $S_j = S_{j_1} = \cdots = S_{j_{K'-2}}$,
- $T_j = T_{\ell_1} = \cdots = T_{\ell_{K'-2}}$,
- $\{j_1, \ldots, j_{K'-2}\} \cap \{\ell_1, \ldots, \ell_{K'-2}\} = \emptyset$,
- $(M_{i+1}, S_j) = (M_{i+1}, S_{j_1}) = \cdots = (M_{i+1}, S_{j_{K'-2}}) \notin \mathcal{U}_i^{M,U}$,
- $(M_{i+1}, S_{\ell_1}), \ldots, (M_{i+1}, S_{\ell_{K'-2}}) \in \mathcal{U}_i^{M,U}$,
- $(M_{i+1}, T_{j_1}), \ldots, (M_{i+1}, T_{j_{K'-2}}) \in \mathcal{V}_i^{M,V}$, and
- $(M_{i+1}, T_j) = (M_{i+1}, T_{\ell_1}) = \cdots = (M_{i+1}, T_{\ell_{K'-2}}) \notin \mathcal{V}_i^{M,V}$.

Then, $(2K'-3) < 2K'$ elements will be added, but not more. Therefore, the size of $\mathcal{L}_i^{X,Y}$ is at most $2K'i$.

We divide the elements that are added to $\mathcal{L}_i$ into the following three multi-sets, Type_1, Type_2, and Type_3:

- Type_1 consists of (M, U, V, X, Y) such that X and Y are both randomly chosen at this query.
- Type_2 consists of (M, U, V, X, Y) such that A_8 already knows X, and only Y is randomly chosen at this query. Note that the added elements share the same random Y.
- Type_3 consists of (M, U, V, X, Y) such that A_8 already knows Y, and only X is randomly chosen at this query. The added elements have the same random X.

If the $(i+1)$-st query $(M_{i+1}, U_{i+1}, V_{i+1})$ satisfies $(M_{i+1}, U_{i+1}) \notin \mathcal{U}_i^{M,U}$ and $(M_{i+1}, V_{i+1}) \notin \mathcal{V}_i^{M,V}$, then the element in Type_1 is added, and elements in Type_2 and Type_3 may also be added to $\mathcal{L}_i$. Similarly, if the $(i+1)$-st query satisfies $(M_{i+1}, U_{i+1}) \in \mathcal{U}_i^{M,U}$ and $(M_{i+1}, V_{i+1}) \notin \mathcal{V}_i^{M,V}$, then only elements in Type_2 are added to $\mathcal{L}_i$, and if the $(i+1)$-st query satisfies $(M_{i+1}, U_{i+1}) \notin \mathcal{U}_i^{M,U}$ and $(M_{i+1}, V_{i+1}) \in \mathcal{V}_i^{M,V}$, then only elements in Type_3 are added. A_8 does not make a query $(M_{i+1}, U_{i+1}, V_{i+1})$ if $(M_{i+1}, U_{i+1}) \in \mathcal{U}_i^{M,U}$ and $(M_{i+1}, V_{i+1}) \in \mathcal{V}_i^{M,V}$. Therefore, when (M, U, V, X, Y) is added to $\mathcal{L}_i$, X or Y (or both) are randomly chosen. Also, observe that Type_1 has at most one element, and Type_2 and Type_3 have at most K' elements, respectively.

Let $\mathrm{Type}_i^{X,Y}$ be a short hand for the multi-set of (X, Y) such that $(M, U, V, X, Y) \in \mathrm{Type}_i^{X,Y}$ for some (M, U, V).

Now 2-COLL can occur in the following cases:

Case 1: $\mathrm{Type}_1^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset$. In this case, there are at most $2K'i$ elements in $\mathcal{L}_i^{X,Y}$ for the collision, and each elements will collide with probability $1/2^{2n}$. We thus have

$$\Pr(\mathrm{Type}_1^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset \mid \overline{\mathrm{WIN}}_i) \leq \frac{2K'i}{2^{2n}}.$$

Case 2: $\mathrm{Type}_2^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset$, or $\mathrm{Type}_3^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset$.

Consider $\mathrm{Type}_2^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset$. We know that at most K' elements are added to $\mathcal{L}_i$. Let

$$\begin{gathered}(M_{\ell_1}, U_{\ell_1}, V_{\ell_1}, X_{\ell_1}, Y_{\ell_1}) \\ \vdots \\ (M_{\ell_{K'}}, U_{\ell_{K'}}, V_{\ell_{K'}}, X_{\ell_{K'}}, Y_{\ell_{K'}})\end{gathered}$$

be corresponding elements, where $Y_{\ell_1} = \cdots = Y_{\ell_{K'}}$ is a random n-bit string.

Now for each $(M_{\ell_j}, U_{\ell_j}, V_{\ell_j}, X_{\ell_j}, Y_{\ell_j})$, there are at most $K'K$ elements in $\mathcal{L}_i^{X,Y}$ for the collision, as $\mathcal{L}_i^X$ has most one $K'K$ collision that share the same X_{ℓ_j} with $(M_{\ell_j}, U_{\ell_j}, V_{\ell_j}, X_{\ell_j}, Y_{\ell_j})$. Therefore, at most K' elements are added, each added element has at most $K'K$ elements in $\mathcal{L}_i$ for a collision, and each elements collide with probability $1/2^n$, and we thus have

$$\Pr(\mathrm{Type}_2^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset \mid \overline{\mathrm{WIN}}_i) \leq \frac{(K')^2 K}{2^n}.$$

Similarly, we have the same bound for $\mathrm{Type}_3^{X,Y} \cap \mathcal{L}_i^{X,Y} \neq \emptyset$.

Case 3: $\mathrm{Type}_1^{X,Y} \cap \mathrm{Type}_2^{X,Y} \neq \emptyset$, or $\mathrm{Type}_1^{X,Y} \cap \mathrm{Type}_3^{X,Y} \neq \emptyset$.

Consider $\mathrm{Type}_1^{X,Y} \cap \mathrm{Type}_2^{X,Y} \neq \emptyset$. The elements in $\mathrm{Type}_1^{X,Y}$ and $\mathrm{Type}_2^{X,Y}$ share the same Y. Since X in $\mathrm{Type}_1^{X,Y}$ is randomly chosen, and $\mathrm{Type}_2^{X,Y}$ has at most K' elements, we have

$$\Pr(\mathrm{Type}_1^{X,Y} \cap \mathrm{Type}_2^{X,Y} \neq \emptyset \mid \overline{\mathrm{WIN}}_i) \leq \frac{K'}{2^n}.$$

Similarly, we have the same bound for $\mathrm{Type}_1^{X,Y} \cap \mathrm{Type}_3^{X,Y} \neq \emptyset$.

Case 4: $\mathrm{Type}_2^{X,Y} \cap \mathrm{Type}_3^{X,Y} \neq \emptyset$. We may have 2-COLL only if the randomly chosen Y for Type_2 collides with the Y in $\mathrm{Type}_3^{X,Y}$, or the randomly chosen X for Type_3 collides with the X in $\mathrm{Type}_2^{X,Y}$. Since Type_2 and Type_3 have at most K' elements,

$$\Pr(\mathrm{Type}_2^{X,Y} \cap \mathrm{Type}_3^{X,Y} \neq \emptyset \mid \overline{\mathrm{WIN}}_i) \leq \frac{2K'}{2^n}.$$

Case 5: Two elements in $\mathrm{Type}_2^{X,Y}$ collide, or two elements in $\mathrm{Type}_3^{X,Y}$ collide. Now in order for two elements in $\mathrm{Type}_2^{X,Y}$ to collide, bad_i^U has to occur. Therefore, from Lemma 7,

$$\Pr(\text{Two elements in } \mathrm{Type}_2^{X,Y} \text{ collide} \mid \overline{\mathrm{WIN}}_i) \leq \frac{K'}{2^n}.$$

We have the same bound for $\mathrm{Type}_3^{X,Y}$.

Overall, we have

$$\Pr(\mathcal{L}_{i+1}^{X,Y} \text{ is 2-COLL} \mid \overline{\mathrm{WIN}}_i) \leq \frac{2K'i}{2^{2n}} + \frac{2(K')^2 K}{2^n} + \frac{6K'}{2^n},$$

and this completes the proof. □

Now we present the proof of Lemma 4.

Proof (of Lemma 4). Lemma 5 gives the bound on (4.3), and Lemma 6 gives the bound on (4.4). Lemma 8 shows that (4.5) is at most

$$\begin{aligned}\sum_{0 \leq i \leq q'-1} \Pr(\mathcal{L}_{i+1}^{X,Y} \text{ is 2-COLL} \mid \overline{\mathrm{WIN}}_i) &\leq \sum_{0 \leq i \leq q'-1} \frac{2K'i}{2^{2n}} + \frac{2(K')^2 K}{2^n} + \frac{6K'}{2^n} \\ &\leq \frac{{q'}^2 K'}{2^{2n}} + \frac{2q'(K')^2 K}{2^n} + \frac{6q'K'}{2^n},\end{aligned}$$

and therefore, we have the claimed bound. □

Collision resistance of $\mathcal{F}$ against A_7'. Now we return to A_7'. We first recall its definition. Let $K \geq 1$ and $s \geq 1$ be fixed integers and let $\mathcal{H}_0 = \{(H_0, \hat{H}_0), \ldots, (H_{s-1}, \hat{H}_{s-1})\}$, $(H_i, \hat{H}_i) \in (\{0,1\}^n)^2$, be a fixed set of strings such that

- the multi-set $\mathcal{H}_0^H = \{H_0, \ldots, H_{s-1}\}$ is K-COLL-free, and
- the multi-set $\mathcal{H}_0^{\hat{H}} = \{\hat{H}_0, \ldots, \hat{H}_{s-1}\}$ is K-COLL-free.

The adversary A_7' takes $\mu \leq l$ as the input. It has access to l oracles, $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$, in this order, that is, A_7' first makes queries to $(f_0, \hat{f}_0)$ and then $(f_1, \hat{f}_1)$, until $(f_{l-1}, \hat{f}_{l-1})$ (but the last $l - \mu$ oracles are irrelevant). $(f_i(\cdot,\cdot), \hat{f}_i(\cdot,\cdot))$ takes $(M, H, \hat{H})$ as the input, and the output is $(h, \hat{h}) = (f_i(M, H), \hat{f}_i(M, \hat{H}))$. The $2l$ functions, $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, are random oracles. A_7' outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where M and M' are both μ blocks. We say A_7' wins if $\mathcal{F}((H, \hat{H}), M) = \mathcal{F}((H', \hat{H}'), M')$, where $(H, \hat{H}), (H', \hat{H}') \in \mathcal{H}_0$ must hold.

Now we restate the advantage of A_7', which is defined as

$$\mathbf{Adv}_{\mathcal{F}}^{\mathrm{coll}}(A_7') = \Pr(A_7^{(f_0,\hat{f}_0),\ldots,(f_{l-1},\hat{f}_{l-1})} \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$ and A_7''s coin (if any). A_7' is assumed to know $\mathcal{F}((H, \hat{H}), M) = \mathcal{F}((H', \hat{H}'), M')$ holds when A_7' outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$.

We have the following result.

Theorem 2 (Collision resistance of $\mathcal{F}$) *Let A_7' be a collision finding adversary, described as above, that makes $2q$ queries to each of $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$. Then for any integer $K \geq 2$,*

$$\mathbf{Adv}_{\mathcal{F}}^{\mathrm{coll}}(A_7') \leq \frac{4q^2K^{l+1}}{2^{2n}} + \frac{4qK^{2l+2}}{2^n} + \frac{12qK^{l+1}}{2^n} + \frac{l2^{K+1}q^K}{2^{n(K-1)}}.$$

Proof. Let $\mathcal{H}_i$, $1 \leq i \leq l$, be a multi-set consists of all the strings of $(M, H, \hat{H}, h, \hat{h})$ such that A_7' knows $(h, \hat{h}) = (f_{i-1}(M, H), \hat{f}_{i-1}(M, \hat{H}))$. $\mathcal{H}_i^{h,\hat{h}}$, $\mathcal{H}_i^h$, and $\mathcal{H}_i^{\hat{h}}$ are also defined in the natural way.

For notational simplicity, let E_i be the event that

$$(\mathcal{H}_i^h \text{ is } K^{i+1}\text{-COLL}) \vee (\mathcal{H}_i^{\hat{h}} \text{ is } K^{i+1}\text{-COLL}) \vee (\mathcal{H}_i^{h,\hat{h}} \text{ is 2-COLL}),$$

and $\overline{E}_i$ be its complement event.

We have

$$\mathbf{Adv}_{\mathcal{F}}^{\mathrm{coll}}(A_7') \leq \Pr(E_1 \mid \overline{E}_0) \leq \frac{4q^2K}{2^{2n}} + \frac{4qK^3}{2^n} + \frac{12qK}{2^n} + \frac{2^{K+1}q^K}{2^{n(K-1)}}$$

for $\mu = 1$ from Lemma 4, by letting $q' \leftarrow 2q$ and $K' \leftarrow K$. In general, for $1 \leq i \leq l$, we have

$$\Pr(E_i \mid \overline{E}_0 \wedge \cdots \wedge \overline{E}_{i-1}) \leq \frac{4q^2K^i}{2^{2n}} + \frac{4qK^{2i+1}}{2^n} + \frac{12qK^i}{2^n} + \frac{2^{K+1}q^K}{2^{n(K-1)}}.$$

Therefore, we have

$$\begin{aligned}
\mathbf{Adv}_{\mathcal{F}}^{\mathrm{coll}}(A_7') \leq \Pr(E_l) &\leq \sum_{1 \leq i \leq l} \Pr(E_i \mid \overline{E}_0 \wedge \cdots \wedge \overline{E}_{i-1}) \\
&\leq \sum_{1 \leq i \leq l} \frac{4q^2K^i}{2^{2n}} + \frac{4qK^{2i+1}}{2^n} + \frac{12qK^i}{2^n} + \frac{2^{K+1}q^K}{2^{n(K-1)}} \\
&\leq \frac{4q^2K^{l+1}}{2^{2n}} + \frac{4qK^{2l+2}}{2^n} + \frac{12qK^{l+1}}{2^n} + \frac{l2^{K+1}q^K}{2^{n(K-1)}}.
\end{aligned}$$

	Bound	q
$K=3$	$\frac{q^2}{2^{495}}+\frac{q}{2^{225}}+\frac{q}{2^{238}}+\frac{q^3}{2^{505}}$	2^{168}
$K=4$	$\frac{q^2}{2^{492}}+\frac{q}{2^{218}}+\frac{q}{2^{234}}+\frac{q^4}{2^{760}}$	2^{190}
$K=5$	$\frac{q^2}{2^{489}}+\frac{q}{2^{212}}+\frac{q}{2^{231}}+\frac{q^5}{2^{1015}}$	2^{203}
$K=6$	$\frac{q^2}{2^{486}}+\frac{q}{2^{207}}+\frac{q}{2^{229}}+\frac{q^6}{2^{1270}}$	2^{207}

Table 4.1: Numerical examples of Theorem 2 for $n = 256$ and $l = 8$, and q denotes the minimum number of q that the bound reaches 1. Since the bound holds for any K, A'_7 needs at least 2^{207} queries.

This completes the proof. □

We note that although the restriction on A'_7, several known attacks on concatenating hash functions are covered by A'_7. If A'_7 is close to the best attack, then the above theorem ensures that A_1 cannot succeed in finding a collision against the DMMD transform. See Table 4.1. Although the bound does not reach the level of collision resistance for a $2n$ bit hash function, for a reasonable size of K, say 4 or 5, the bound is much better than the standard birthday bound, and finding a collision for short concatenating hash functions is recognized as a hard problem, it is unlikely that the attack will be found on the DMMD transform.

Preimage resistance of the DMMD transform. It is simple to show that the DMMD transform is preimage resistant if F_1 (which corresponds to *MFF*) is preimage resistant. We follow the notation in Figure. 4.3. A preimage finding adversary against DMMD, is an adversary that is given a hash value $h \in \{0,1\}^c$, outputs a (padded) message $M \in (\{0,1\}^m)^*$ such that $\mathrm{DMMD}(M) = h$. Similarly, a preimage finding adversary against F_1 is given a hash value $h \in \{0,1\}^c$, and outputs $(H, \hat{H}) \in (\{0,1\}^m)^2$ such that $F_1(H, \hat{H}) = h$.

We have the following result.

Theorem 3 (Preimage resistance of the DMMD transform) *If there exists a preimage finding adversary against DMMD, then there exists a preimage finding adversary against F_1.*

Proof. Suppose that $h \in \{0,1\}^c$ is given to the preimage finding adversary, A, against F_1. Now the preimage finding adversary, B, against DMMD is run with $h \in \{0,1\}^c$ as its input. From the assumption, B outputs $M \in (\{0,1\}^m)^*$ such that $\mathrm{DMMD}(M) = h$. Now A computes $\mathrm{DMMD}(M)$ by itself. Let $(H, \hat{H})$ be the input value of F_1 that is obtained during the computation, and A outputs $(H, \hat{H})$. □

4.2.3 Security Properties of AURORA structure

Guaranteed Active S-boxes in AURORA structure

By the recent evolution of research on attacks on hash functions [55, 56, 57], it becomes very important to know the immunity against differential type attacks to design a new hash function. Moreover, in the traditional blockcipher based design strategy of hash functions, the compression function assumes that the underlying blockcipher behaves like an ideal blockcipher. Thus designers should design a strong blockcipher which holds enough strength against differential cryptanalysis as a matter of course. In this section, a permutation used in AURORA called "AURORA structure" is investigated, and security aspect with regard to differential cryptanalysis is discussed.

Table 4.2: Guaranteed Numbers of Active S-boxes in AURORA structure.

Round	♯ of Active S-boxes	Round	♯ of Active S-boxes
1	0	11	36
2	1	12	40
3	5	13	42
4	6	14	46
5	9	15	50
6	15	16	52
7	22	17	56
8	26	18	60
9	30	19	62
10	32	20	66

From the specification of AURORA, it can be seen that both of *MSM* and *CPM* employ 8-round and 17-round AURORA structure, respectively. The AURORA structure is based on an 8-bit S-box, matrices and a byte diffusion *BD* design, and all components are byte-oriented. It is natural for evaluating the immunity against differential cryptanalysis by counting the minimum number of active S-boxes of AURORA structure using a blockcipher evaluation method [50, 51, 53, 54].

We used a simulation program to count the guaranteed numbers of active S-boxes in the structure. The counting method treats a byte data as either 0 or 1 in truncated form, and then tries to find a truncated differential path which holds the minimum number of active S-boxes for a target round in an exhaustive way [50]. During the search, the DSM conditions are used to judge whether given truncated paths are valid or not, and behavior of *BD* is also taken into consideration.

Table 4.2 shows the obtained guaranteed numbers of active S-boxes for each round of AURORA structure. From the fact that AURORA employs an S-box whose maximum differential probability is 2^{-6}, we can conclude that 8-round AURORA structure does not hold characteristics whose differential probability is higher than $2^{-156} < 2^{-128}$. Similarly, 17-round AURORA structure does not hold characteristics with probability higher than $2^{-336} < 2^{-256}$. We explain the immunity of AURORA against differential cryptanalysis by using the above observations.

There are several steps in recently developed differential type attacks for hash functions 1) finding a local collision and a differential path, 2) finding sufficient conditions applied to a message MM, and 3) choosing a message M such that all sufficient conditions hold. Since there is no established way to prevent a hash function from the above attack, we choose one approach to make the Step 1) be difficult for an attacker by introducing non-linearity in the message scheduling part. Consider the situation such that an attacker controls messages to find a collision of AURORA. The attacker will succeed if he finds a collision with less than 128-bit security. But if the attacker insert a difference into *MSM*, the probability of the differential that follows a specific characteristic which is useful for finding collision is very low enough being less than 2^{-128}.

Moreover, we see that a compression function of AURORA-256 uses a 256-bit blockcipher. The obtained numbers of active S-boxes show in Table 4.2 imply that the blockcipher is secure enough against distinguishing attacks in differential cryptanalytic scenarios, which we believe is more important requirement than key recovery attacks on hash functions. As a result, we conclude that the underlying blockcipher behaves randomly with regard to the differential attack and does not hold bad properties which are exploited in differential attack scenarios by attackers.

Output Truncation

As stated in Sec. 4.1, any m bits by taking a fixed subset of the AURORA function's output bits expected to meet the desirable security requirements. On the other hand, if we see the AURORA structure carefully, it is noticed that dropping consecutive 32 bits at once from the output of the

structure sometimes waste the calculation effort of a F-function at the last round. Therefore, we introduced the truncation function *TF* to avoid such the lose to maximize the effect of F-functions.

The output truncation function *TF* is applied for AURORA-256/512/256M with different IVs to generate the output values for AURORA-224/384/224M, respectively. Due to the internal connection of AURORA structure, we adopted a design policy of truncation functions which drop non-successive bytes of output of the compression function to avoid invalidating the calculation effort of a F-function. Let $X_{(256)}$ be a output of AURORA structure, and set $(X_{0(64)}, X_{1(64)}, X_{2(64)}, X_{3(64)}) \leftarrow X$. In this case, a truncation function should not drop any of $X_{i(64)}$ at once, because output of a F-function at the last round in *CPM* only affects one of $X_{i(64)}$, which means that the F-function is invalidated for the calculation of the output values. Therefore, the truncation functions in AURORA family is designed to drop byte data at discontinuous positions.

Impossible Differentials in AURORA Structure

Impossible differential is the differential path that never exists (i.e. its differential probability is 0). The attack using impossible differentials was originally proposed for recovering a blockcipher key [7].

In the hash function case, there is no secret key to recover, and in most cases the adversary is allowed to know the message to be hashed. Therefore, it does not seem that impossible differential attacks work on hash functions. However, existence of impossible differential can allow us to distinguish a hash function from a random function. Indeed, with such a property, one can show a non-random behavior of the hash function. For example, Sasaki et al. recovered the secret data (password) included in the input of the hash function using an impossible differential path in MD4, which is used in a challenge-response password authentication protocol [48].

We searched for impossible differential paths considering that the matrices satisfy the DSM conditions (i.e. Conditions I and II described in Sec. 3.4.2). The longest impossible differential paths that we found in the AURORA structure (with the byte diffusion function *BD*) have 7 rounds. It can be shown that the byte diffusion plays an important role in avoiding long impossible differential paths, because there exist trivial 16-round impossible differential paths in the modified-AURORA structure where byte diffusion function *BD* is replaced with "usual" word-wise diffusion.

Furthermore, the AURORA structure has stronger resistance against impossible differential attacks than the generalized Feistel structure: there exist trivial 17-round impossible differential paths in the 8-branch generalized Feistel structure, and 8-round impossible differential paths in the 8-branch generalized Feistel structure employing the byte diffusion *BD*.

Since the chaining value processing module employs the 17-round AURORA structure, and the message scheduling modules employ the 8-round AURORA structure, it is expected that there is no impossible differential in the AURORA compression function which can allow us to distinguish the AURORA hash function from a random function.

4.3 Algorithm Analysis

This section describes a preliminary analysis of AURORA hash functions regarding collision attacks, preimage attacks, second preimage attacks, length-extension attacks, and multicollision attacks. In this section, "r-round AURORA-256" is used to refer to a variant of AURORA-256 algorithm reduced to r rounds, i.e. the chaining value processing function with r rounds and the corresponding message scheduling functions. The round function begins from the byte diffusion function *BD* and ends by XORing with message words (See Fig. 4.6).

4.3.1 Collision Attack

There are several known approaches for finding collisions of hash functions in the literature. We consider possible approaches and show their results or how the design of AURORA works to prevent the attacks. Beside the analyses below, Sec. 4.2.3 describes differential cryptanalysis of

the AURORA structure, and shows that there is no differential characteristic in *MSM* and *CPM* with high probability.

Approach I : Application of the collision attacks on MDx-SHAx family. A well-known approach for finding collision of hash functions is to (1) find a local collision by analyzing the chaining value processing module, (2) stack local collisions together to form a global collision by analyzing message scheduling module and construct a differential path, and (3) boost success probability of the attack by message modification techniques. This approach has been successful in finding collisions on many hash functions including MD4, MD5, SHA-0, SHA-1 [11, 56, 57, 55].

The local collision is defined as a collision for a fixed number of steps of the compression function under the assumption that the message words from the message scheduling modules can be chosen independently by the attacker. There exists a 2-round local collision in AURORA, which is shown in Table 4.3. In the cases of hash functions with simple message schedule such as MD4 and MD5, this local collision would be useful, because the assumption that message words are independent almost holds. However, in the case of AURORA, this assumption does not hold due to the complicated message scheduling modules. Therefore, the existence of a 2-round local collision does not lead to a certain vulnerability.

In Table 4.3, notice that δ_i can be zero, and that at most only 8 differences are introduced in message words. It is possible to construct longer local collisions, but more message word differences should be involved. It tends to be harder to control.

Table 4.3: A 2-round local collision for AURORA family.

	chaining value difference								message word difference							
round	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	ΔU_{8i}	ΔU_{8i+1}	ΔU_{8i+2}	ΔU_{8i+3}	ΔU_{8i+4}	ΔU_{8i+5}	ΔU_{8i+6}	ΔU_{8i+7}
i	0	0	0	0	0	0	0	0	δ_1	0	δ_2	0	δ_3	0	δ_4	0
$i+1$	0	δ_2	0	δ_3	0	δ_4	0	δ_1	0	δ_2	0	δ_3	0	δ_4	0	δ_1
$i+2$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Note: δ_1, δ_2, δ_3, δ_4 are independent zero or non-zero arbitrary 32-bit values. At least one of δ_i's should be non-zero. Here the message schedule is ignored.

The next step is to form a global collision by analyzing the message schedule. In the case of AURORA, it is difficult to control the message words from the 2nd round due to the heavy message scheduling functions. Considering the message scheduling functions, we have found collision for up to 3-round AURORA-224/256 with complexity less than the birthday bound. The differential characteristic is shown in Table 4.4. The chaining value difference ΔX_i is the difference in the input chaining value X_i of each round. For other symbols, see Fig. 4.6.

Let α be an 8-bit non-zero value, β be an 8-bit non-zero value where the least significant bit is zero, and $\gamma = \beta \ggg_8 1$. Then x_0, x_1, x_2, and x_3 are defined as follows:

x_0: a 32-bit value whose 4th byte is β and the other three bytes are zero. (i.e. 000β)
x_1: a 32-bit value whose 4th byte is γ and the other three bytes are zero. (i.e. 000γ)
x_2: a 32-bit value whose 3rd byte is α and the other three bytes are zero. (i.e. $00\alpha0$)
x_3: a 32-bit value whose 2nd byte is α and the other three bytes are zero. (i.e. $0\alpha00$)

If we set the message difference $M_L = (x_3, 0, 0, 0, x_1, 0, x_2, 0)$ and $M_R = (0, 0, 0, x_1, 0, x_2, 0, x_3)$, the chaining value difference becomes zero at the input of the 2nd round with probability 1. Note that some of the message words are cyclically shifted by the data rotating function *DR* before inputting to the chaining value processing function, e.g., $(U_8||U_{11}) = (T_{R,1}||T_{R,3}) \ggg_{64} 1$. To avoid that the byte difference expands to other bytes by *DR*, we restrict the value of the non-zero byte difference in x_0 and x_1 to β and γ, respectively. Then in the 2nd round, there are differences in three bytes which are input from message words $T_{L,11}$, $T_{L,13}$, and $T_{L,15}$. In the 3rd round, the three byte differences get together to leftmost 32-bit word by the byte diffusion function *BD*. Therefore, there are three active S-boxes in the left F_1. Similarly, there are three active S-boxes in the left F_2 in the message scheduling function MS_R. Under the conditions above, there is a

Table 4.4: A 3-round collision for AURORA-256.

	chaining value difference								message word difference							
round 0	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	$\Delta T_{L,0}$	$\Delta T_{L,1}$	$\Delta T_{L,2}$	$\Delta T_{L,3}$	$\Delta T_{L,4}$	$\Delta T_{L,5}$	$\Delta T_{L,6}$	$\Delta T_{L,7}$
	0	0	0	0	0	0	0	0	x_3	0	0	0	x_1	0	x_2	0
round 1	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	$\Delta T_{R,0}$	$\Delta T_{R,1}$	$\Delta T_{R,2}$	$\Delta T_{R,3}$	$\Delta T_{R,4}$	$\Delta T_{R,5}$	$\Delta T_{R,6}$	$\Delta T_{R,7}$
	x_3	0	0	0	x_1	0	x_2	0	0	0	0	x_0	0	x_2	0	x_3
round 2	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	$\Delta T_{L,8}$	$\Delta T_{L,9}$	$\Delta T_{L,10}$	$\Delta T_{L,11}$	$\Delta T_{L,12}$	$\Delta T_{L,13}$	$\Delta T_{L,14}$	$\Delta T_{L,15}$
	0	0	0	0	0	0	0	0	0	0	0	x_1	0	x_2	0	x_3
round 3	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	$\Delta T_{R,8}$	$\Delta T_{R,9}$	$\Delta T_{R,10}$	$\Delta T_{R,11}$	$\Delta T_{R,12}$	$\Delta T_{R,13}$	$\Delta T_{R,14}$	$\Delta T_{R,15}$
	0	0	0	x_1	0	x_2	0	x_3	y_1	y_1	0	0	0	0	0	0
round 4	ΔX_0	ΔX_1	ΔX_2	ΔX_3	ΔX_4	ΔX_5	ΔX_6	ΔX_7	–	–	–	–	–	–	–	–
	0	0	0	0	0	0	0	0	–	–	–	–	–	–	–	–

possibility that the output differences of F_1 and F_2 cancel. (On the other side, if there are less than five active S-boxes in F_1 and F_2 in total, the output differences of F_1 and F_2 never cancel due to the DSM condition (See Sec. 3.4.2).) When the cancellation occurs, there is a collision in the leftmost 32-bit word ΔX_0, and there is a collision in ΔX_1 at the same time. It is expected that one can find a cancellation in 32-bit output differences by trying 2^{16} message blocks due to birthday paradox. Therefore, it is expected that one can find a collision for 3-round (out of 17-round) AURORA with a complexity of 2^{16} 3-round AURORA compression function.

This attack works for 3-round AURORA-384/512 and AURORA-224M/256M.

Approach II : Application of the collision attack on Grindahl. Another approach for finding collisions is a method used in the cryptanalysis of Grindahl [44]. Although it is very hard to find a low-weight and/or small differential path for Grindahl, Peyrin succeeded in building a truncated differential path starting from an all-difference pair of states. The points for the attack to work on Grindahl include

1. an independent message word concatenated every round, and
2. the truncation at the end of each iteration.

The independent message word was used as control bytes and the truncation was used to erase a truncated difference for no cost. Moreover, in the case of Grindahl, the permutation of each round was not strong enough.

Regarding 1., in the case of AURORA, which is similar to the MDx family, the message words which are input every round are not independent, because they are generated by non-linear round function in a sequential manner. Therefore, it is hard to use message words as control bytes. The difference between Grindahl model and AURORA model is shown in Fig. 4.7.

Regarding 2., in AURORA, a truncated difference can be erased during three operations: the MDS matrix operation, the XOR operation with a message word or the XOR operation after the F-function. Using either of the operations takes high cost (i.e. a truncated difference can be erased with low probability). Therefore, it does not seem that Peyrin's attack on Grindahl [44] works on AURORA.

Remark. The analyses above show that AURORA has a good resistance to existing collision attacks because of its secure message scheduling. Considering the fact that there have been no attacks on Whirlpool [3], which was designed based on a similar philosophy to AURORA, this design strategy using secure message scheduling makes a secure hash function. On the other hand, the MDx family (including SHA-1 and SHA-2) was designed using fast and simple message scheduling, so it is expected that a possibly successful attack on the MDx family is unlikely to be applicable to AURORA.

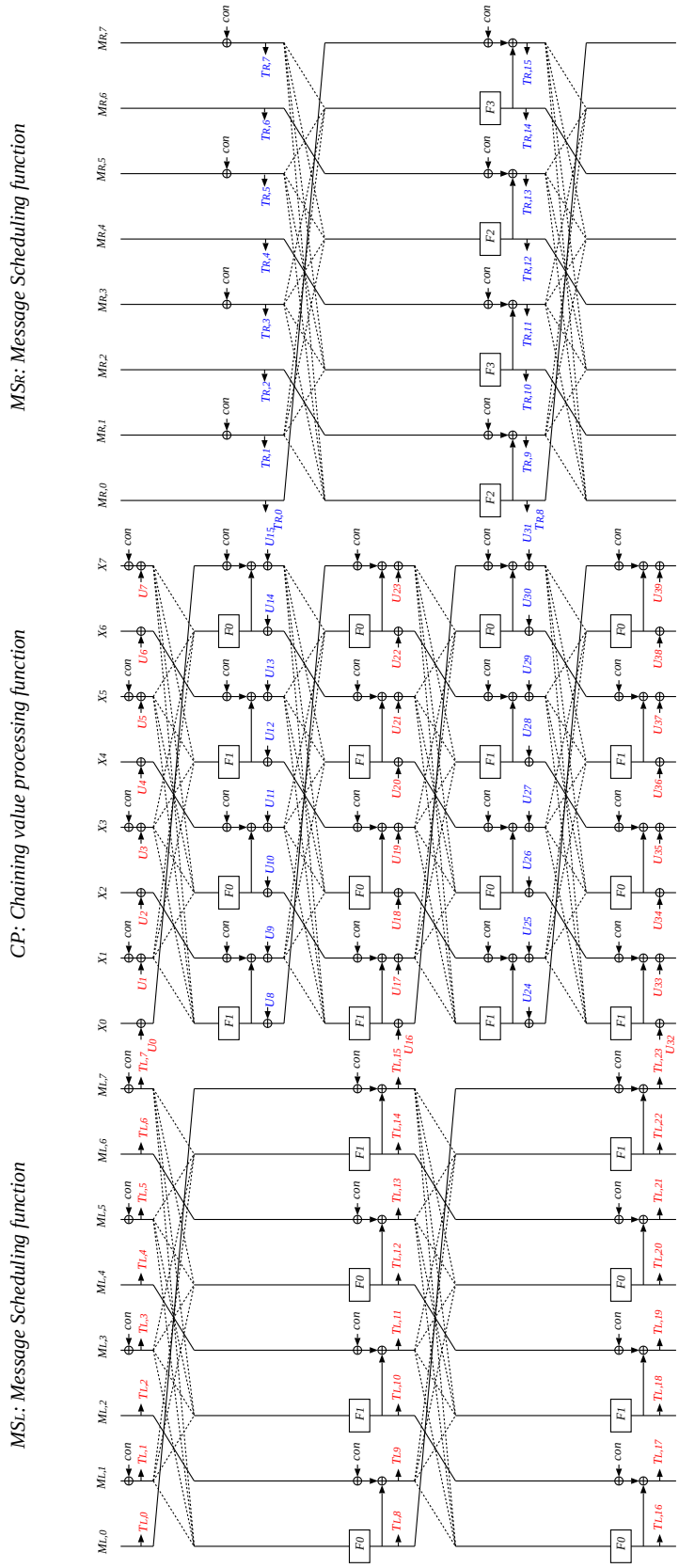

Figure 4.6: Compression function of AURORA-256 (reduced to 4-round).

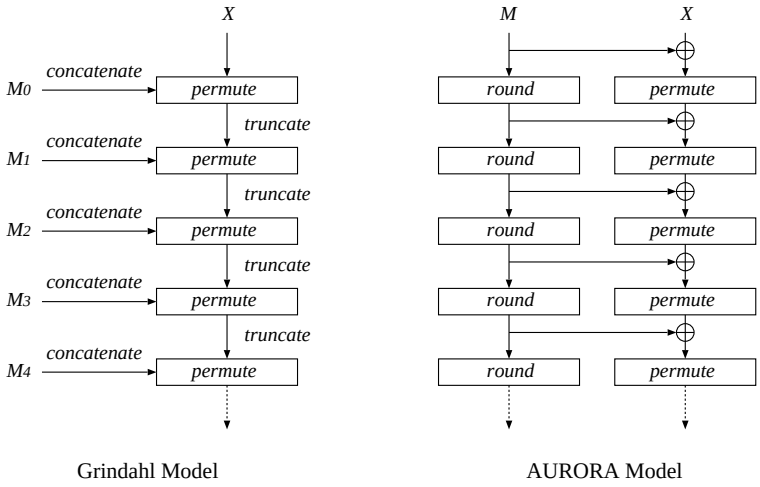

Figure 4.7: Comparison between Grindahl model and AURORA model.

4.3.2 Preimage Attack

Compared with a lot of work on collision resistance, the preimage resistance (i.e., one-wayness) has not been analyzed much. However, there is a steep rise in the study on preimage resistance recently [30, 10, 2, 1].

Approach I : Meet-in-the-middle approach. In most of the recent preimage attacks [30, 10, 2, 1], an attacker first finds a pseudo-preimage, i.e. a preimage on the compression function, then extends it to a preimage attack on the full hash function[1]. Therefore, we start by analysis of the compression function.

Leurent [30] showed the first preimage attack of the full MD4 (also the first preimage attack on a member of the MD4 family), which extensively used its simple step function and message expansion. Therefore, it is difficult to apply the techniques used in Leurent's attack [30] directly to other hash functions. Aoki and Sasaki used the meet-in-the-middle technique in finding pseudo-preimages [1] and succeeded in preimage attacks on many hash functions such as MD4/5, HAVAL-3/4/5, SHA-0/1/2, HAS-160, and RIPEMD [47].

The key idea in the meet-in-the-middle approach in [1] is to divide the attack target into two chunks of steps so that each chunk includes at least one message word that is independent from the other chunk. This strategy was successful for poor message schedules where there is low dependency between message words, but this is not the case for AURORA. For example, it is possible to divide the compression function into two chunks because the message words from the right message scheduling function MS_R are used in odd rounds only, and the message words from the left message scheduling function MS_L are used in even rounds only. However, since two chunks are alternated every round, the meet-in-the-middle approach can not applied to AURORA. Therefore, it is difficult to find a preimage faster than brute-force attack in this approach.

Approach II : Correcting impossible messages. Another approach for finding the preimage was proposed by De Cannière and Rechberger at CRYPTO2008 [10]. The idea is to start with an impossible expanded message that would lead to the required hash value, and then to correct this message until it becomes valid without destroying the preimage property.

This approach has a potential to control a more complex message scheduling, but in the case of AURORA, it is still difficult to correct message words without destroying the preimage property

[1]Meet-in-the-middle-approach is also used for converting pseudo-preimages to a preimage, but in this paragraph we discuss the meet-in-the-middle approach to find pseudo-preimages (i.e. preimages in the compression function).

due to carefully-designed message scheduling functions.

Approach III : SAT-solver approach. De et al. proposed preimage attacks on reduced variants of MD4 and MD5 using SAT-solvers [16]. We describe the preliminary analytic results of preimage attack of AURORA using a SAT-solver. Here, we consider two variants of reduced version of AURORA for the attack.

The first attempt is trying to recover a preimage of 256-bit output value of a 3-round reduced version of *CF* of AURORA-256, called variant A, which does not contain *DR* without loss of generality. As a result, the variant A contains 3-round AURORA structure in *CP* and 1-round AURORA structure both in MS_L and MS_R. The preimage attack for the variant A is non-trivial, and the preimage attack for it can be converted into a SAT problem that contains 384 variables and 58,112 clauses including 3 to 11 literals (avg. 9.15 literals). Then, we tried to solve the 10 instances of the SAT problem using the MiniSat2 [43]. Each problem is executed on a Xeon 2.80GHz processor with 2GB memory. However, after two weeks of calculation effort by the solver, no solutions for these problems are obtained.

The second attempt is to find the shrinking version of 3-round reduced version of *CF* of AURORA-256, called variant B, which outputs 128-bit hash values in which 1-round of AURORA structure is halved to 4 data lines. Thus only two different F-functions are included in a round of the structure. Moreover *DR* is omitted, and *BD* only exchanges two bytes of data. In this case, the SAT-problem contains 192 variables and 29,056 clauses including 3 to 11 literals(avg. 9.15 literals). As a result, we obtained solutions (preimages) of all 10 trials for the variant B. In the trials, the average calculation time for these problems is about 10 hours.

Even though these preliminary results show the resistance of only the variations of AURORA's compression function, but it is sufficient to believe that full *CF* AURORA-256 which contains 8-round, 17-round, and 8-round structure in each module have enough immunity against algebraic attacks using the direct application of SAT-solvers to invert to a preimage within an acceptable duration of time. Also the other compression functions in the AURORA family and hash functions constructed by these compression functions are expected to achieve enough strength against this attack scenario.

4.3.3 Second Preimage Attacks

There are two major directions in second preimage attacks: one is generic long-message second preimage attacks treating the compression function (or the underlying blockcipher) as a black box, and the other is second preimage attacks using certain properties inside the compression function.

Compared with collision resistance, second preimage resistance has not been analyzed much, but we consider possible approaches[2] and how the design of AURORA works to prevent the attacks.

Approach I : Using collision differentials. A straightforward approach for finding second preimages is to use the differential characteristics used in collision attacks by applying the corresponding message difference to the given message. If the characteristic is followed, then this will yield a second preimage. This approach was applied to MD4 by Yu et al. [58], but is has some limitations: one problem is that the success probability of the attack drops by fixing the message. Another problem is that it only works for a small subset of the message space.

According to the discussion in Sec. 4.2.3 and Sec. 4.3.1, there are no differential characteristics that hold with high probability in AURORA, it is expected that this approach is not effective for finding second preimages of AURORA.

Approach II : Using multi-near-collision differentials. Another approach for finding second preimages in the literature is to use multi-near-collision differentials. The idea is to compute

[2]A good summary of possible approaches for finding (second) preimages is written in [10].

Table 4.5: Second preimage resistance for 2^k block messages ($k < 64$) (bits).

AURORA-224	AURORA-256	AURORA-384	AURORA-512	AURORA-224M	AURORA-256M
$\min\{224, 256-k\}$	$256-k$	384	$512-k$	224	256

the hash value for a special message, and try to correct parts of the hash value by applying appropriate differences. This approach was used in the preimage attack on MD4 by Leurent [30], in the second preimage attacks on SMASH by Lamberger et al. [29], and the (second) preimage attacks on GOST by Mendel et al. [33].

This approach works if one can find many highly probable differential characteristics for the same special message. According to the analysis in Sec. 4.2.3, we have not found such differential characteristics in the compression function of AURORA. Furthermore, we have not found any properties in the domain extension transform in the AURORA family, which can be useful in constructing structured messages, e.g. the properties of the SMASH structure used in the second preimage attacks on SMASH [29].

Furthermore, most of the possible known approaches for preimage attacks can be applicable to second preimage attacks. Since no approaches discussed in Sec. 4.3.2 are promising, it is difficult to find second preimage by using those approaches.

Generic long-message second preimage attacks. As Kelsey and Schneier showed in [28], there exists a generic second preimage attack on an n-bit iterated hash functions with the Merkle-Damgård construction, regardless of the compression function used. For a message of 2^k message blocks, a second preimage can be found with about $k \times 2^{\frac{1}{2}+1} + 2^{n-k+1}$ work.

Considering this generic long-message second preimage attack, AURORA-256 and AURORA-512 provide second preimage resistance of about $(256-k)$ bits and $(512-k)$ bits for 2^k-block messages, respectively. AURORA-224 provides second preimage resistance of about $\min\{224, (256-k)\}$ bits, since a brute-force attack is faster for $k < 32$. AURORA-384 provides second preimage resistance of 384 bits, because the maximum message block size for the AURORA family is $2^{64}-1$ blocks, i.e. $k < 64$. AURORA-224M and AURORA-256M have multicollision resistance with the internal chaining value size of 512 bits, therefore, they provide second preimage resistance of about 224 and 256 bits, respectively.

Second preimage resistance of the AURORA family is summarized in Table 4.5.

4.3.4 Length-Extension Attack

Length-extension attack is the attack for hash functions. Given a hash value $h(M)$, the attacker obtains $h(M \,\|\, M')$ without knowing the original message M. AURORA-256 adopts the strengthened Merkle-Damgård (sMD) transform with the finalization function (See Figure 3.1). It is known that it preserves indifferentiability (PRO) of the underlying compression function [6, Lemma 5.1]. In the abstract model, this property ensures that AURORA-256 looks like an ideal random oracle $H : \{0,1\}^* \to \{0,1\}^{256}$, and thus length-extension attack does not work. The same observation holds for AURORA-224. The proof for the pseudorandom oracle preservation (PRO-Pr) is based on the fact that the finalization function is used, and since we follow the same design principle in the DMMD transform, the attack is unlikely to be applicable to AURORA-384/512/224M/256M.

4.3.5 Multicollision Attack

Multicollision attack [26], introduced by Joux, finds the K collision on the classical iterated hash function in time $O(\log K \cdot 2^n)$. We use the classical MD transform in AURORA-224/256, and the attack can be mounted on them. Although the use of the finalization functions, it does not help to increase the security against the attack. Finding K collision for AURORA-224/256 is not much harder than finding ordinary collisions.

Joux also showed how the multicollision attack can be used to get a collision attack on the concatenated hash function. For the DMMD transform, it may be seen as a kind of the concatenated hash function, while the mixing function is used. Since the mixing function inserted frequently, as discussed in detail in Sect. 4.2.2, finding even a single collision is hard for the attacker. Therefore, the attack is unlikely to be applicable to AURORA-224M/256M. However, finding K collision for AURORA-384/512 is not much harder than finding ordinary collisions.

4.3.6 Slide Attacks

Slide attacks have mostly been used for blockcipher cryptanalysis. As shown in [24], slide attacks also form a potential threat for a class of hash functions, e.g., sponge-function like structures. A slide property which can be detected with some significantly high probability can allow up to distinguish a given hash function from a random oracle. Furthermore, certain constructions for hash-function-based MACs, e.g., a MAC with prefix key $\mathrm{MAC}(K, M) = H(K||M)$, can be vulnerable to forgery and even to key recovery attacks.

Slide attacks on blockciphers [9] utilize the self-similarity of the cipher, typically caused by a periodic key schedule. The slide attack on hash functions [24] exploits invertibility and self-similarity in the sponge-function like structures.

The slide attacks are not applicable to AURORA based on the following considerations: (1) The compression function of AURORA is not invertible due to the feed-forward in the Davis-Meyer construction. (2) The structure of AURORA avoids too much self-similarity both in the level of domain extension transform and in the compression function. In the domain extension transform level, AURORA-224/256 consists of *CF*s and *FF*, which behaves differently from CFs. In AURORA-384/512, *CF*s, *MF*s and *MFF* behave differently with different constants and different I/O. In the compression function level, randomly chosen constants avoid a periodic message schedule.

4.4 Tunable Security Parameters

There are two tunable security parameters in the AURORA hash function family. The first is an iteration number of round functions in AURORA structure used in *MSM* and *CPM* for all member AURORA-224/256/384/512/224M/256M . The second is a method to output digests other than 224, 256, 384 and 512 bits.

4.4.1 Number of Rounds

Recommended numbers of round are 8 for *MSM* and 17 for *CPM* as described in the specification. Tuning is done keeping a relationship between these numbers such that $c = 2m + 1$ where m and c are numbers of rounds for *MSM* and *CPM*, respectively. The permissible range for the parameter m is $m \in \{8, 9, 10, 11, 12, 13, 14, 15, 16\}$. The greater the parameter is, the security of the hash function increase by paying cost of the performance. We believe that $m > 16$ is too much taking account of the performance dropping.

4.4.2 Variable Hash Size

Current specification of AURORA hash function family only supports hash sizes of 224, 256, 384, and 512 bits. By setting the initial vectors appropriately, we can also define an alternative hash function family which supports variable hash sizes for the range of from 1-bit to 512-bit. The hash functions for 1-bit to 256-bit output are obtained by modifying AURORA-256, and hash functions for 257-bit to 512-bit output are obtained by modifying AURORA-512. These hash functions are defined as follows.

- l-bit output hash functions for $1 \leq l \leq 256$.

Step. 1 Let $H_{0(256)} \leftarrow 1^l||0^{256-l}$.

Step. 2 Execute the AURORA-256 procedure for a message M, then obtain H_m.

Step. 3 Let $(X_{0(64)}, X_{1(64)}, X_{2(64)}, X_{3(64)}) \leftarrow H_m$.

Step. 4 Let $d = \lfloor l/4 \rfloor$ and $m = l \mod 4$.

Step. 5 Drop the right-most d-bit for all X_i $(0 \leq i \leq 3)$

Step. 6 Additionally, drop the right-most 1-bit for X_i $(0 \leq i \leq m-1)$

Step. 7 Output $X_0||X_1||X_2||X_3$ as an l-bit hash value.

- l-bit output hash functions for $257 \leq l \leq 512$.

Step. 1 Let $H_{0(512)} \leftarrow 1^l||0^{512-l}$.

Step. 2 Execute the AURORA-512 procedure for a message M, then obtain H_m.

Step. 3 Let $(X_{0(64)}, X_{1(64)}, \ldots X_{7(64)}) \leftarrow H_m$.

Step. 4 Let $d = \lfloor l/8 \rfloor$ and $m = l \mod 8$.

Step. 5 Drop the right-most d-bit for all X_i $(0 \leq i \leq 8)$.

Step. 6 Additionally, drop the right-most 1-bit from remaining X_i $(0 \leq i \leq m-1)$.

Step. 7 Output $X_0||X_1||X_2||X_3||X_4||X_5||X_6||X_7$ as an l-bit hash value.

Chapter 5

Efficient Implementation of AURORA

This chapter describes our evaluation results of the hash function family AURORA in both software and hardware implementations.

AURORA can be implemented efficiently in software on various platforms from low-end 8-bit processors to high-end 64-bit processors. On the NIST 32-bit reference platform, AURORA-256 achieves 24.3 cycles/byte and AURORA-512 achieves 46.9 cycles/byte; on the NIST 64-bit reference platform, AURORA-256 achieves 15.4 cycles/byte and AURORA-512 achieves 27.4 cycles/byte. In hardware, AURORA enables a variety of implementations from small-area to high-throughput implementations. In our evaluations using a $0.13\mu m$ CMOS ASIC library, the smallest area of AURORA-256 is 11.1 Kgates with throughout of 2.2 Gbps, and the highest throughput of AURORA-256 is 10.4 Gbps with area of 35.0 Kgates; the smallest area of AURORA-512 is 14.6 Kgates with throughout of 1.2 Gbps, and the highest throughput of AURORA-512 is 9.1 Gbps with area of 56.7 Kgates.

Detailed results of software and hardware implementations are shown in Sec. 5.1 and 5.2, respectively.

5.1 Software Implementation

This section describes the software performance results of AURORA.

5.1.1 Implementation Types

This subsection describes 5 implementation types suitable for either 32-bit or 64-bit processors: 2 types for 32-bit processors and 3 types for 64-bit processors. We only explain the implementation methods for F functions because the performance results are strongly affected by these methods. First, we show the notations used in this section. Next, we present five implementation types either for 32-bit and 64-bit processors. All of these implementation types are implemented in the optimized code we provide. Finally, we describe how to select these implementation types in our optimized codes.

Notations

Let $(x_0^0, x_1^0, x_2^0, x_3^0)$ be an input of F-function F_0 and $(y_0^0, y_1^0, y_2^0, y_3^0)$ be an output of F_0. Similarly, let $(x_0^1, x_1^1, x_2^1, x_3^1)$, $(x_0^2, x_1^2, x_2^2, x_3^2)$ and $(x_0^3, x_1^3, x_2^3, x_3^3)$ be inputs of F_1, F_2 and F_3 , respectively and let $(y_0^1, y_1^1, y_2^1, y_3^1)$, $(y_0^2, y_1^2, y_2^2, y_3^2)$ and $(y_0^3, y_1^3, y_2^3, y_3^3)$ be outputs of F_1, F_2 and F_3 , respectively.

AURORA has the following four different 32-bit input/output F functions. Those notations are used to explain how to implement AURORA on 32-bit processors.

$$F_0 : \begin{pmatrix} y_0^0 \\ y_1^0 \\ y_2^0 \\ y_3^0 \end{pmatrix} = \begin{pmatrix} \texttt{0x01} & \texttt{0x02} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x02} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \end{pmatrix} \begin{pmatrix} S(x_0^0) \\ S(x_1^0) \\ S(x_2^0) \\ S(x_3^0) \end{pmatrix}$$

$$F_1 : \begin{pmatrix} y_0^1 \\ y_1^1 \\ y_2^1 \\ y_3^1 \end{pmatrix} = \begin{pmatrix} \texttt{0x01} & \texttt{0x06} & \texttt{0x08} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x06} & \texttt{0x08} \\ \texttt{0x08} & \texttt{0x02} & \texttt{0x01} & \texttt{0x06} \\ \texttt{0x06} & \texttt{0x08} & \texttt{0x02} & \texttt{0x01} \end{pmatrix} \begin{pmatrix} S(x_0^1) \\ S(x_1^1) \\ S(x_2^1) \\ S(x_3^1) \end{pmatrix}$$

$$F_2 : \begin{pmatrix} y_0^2 \\ y_1^2 \\ y_2^2 \\ y_3^2 \end{pmatrix} = \begin{pmatrix} \texttt{0x03} & \texttt{0x01} & \texttt{0x02} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x02} & \texttt{0x03} \end{pmatrix} \begin{pmatrix} S(x_0^2) \\ S(x_1^2) \\ S(x_2^2) \\ S(x_3^2) \end{pmatrix}$$

$$F_3 : \begin{pmatrix} y_0^3 \\ y_1^3 \\ y_2^3 \\ y_3^3 \end{pmatrix} = \begin{pmatrix} \texttt{0x06} & \texttt{0x08} & \texttt{0x02} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x06} & \texttt{0x08} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x06} & \texttt{0x08} \\ \texttt{0x08} & \texttt{0x02} & \texttt{0x01} & \texttt{0x06} \end{pmatrix} \begin{pmatrix} S(x_0^3) \\ S(x_1^3) \\ S(x_2^3) \\ S(x_3^3) \end{pmatrix}$$

Also, we can consider that AURORA has the following four different 64-bit input/output F functions named F^* functions which have two 32-bit input/output F-functions as internal functions (See Fig. 5.1). Let $(x_0^{0'}, ..., x_7^{0'})$ be an input of F^*-function F_0^* and $(y_0^{0'}, ..., y_7^{0'})$ be an output of F_0^*. Similarly, let $(x_0^{1'}, ..., x_7^{1'})$, $(x_0^{2'}, ..., x_7^{2'})$ and $(x_0^{3'}, ..., x_7^{3'})$ be inputs of F_1^*, $F_{2'}$ and F_3^*, respectively and let $(y_0^{1'}, ..., y_7^{1'})$, $(y_0^{2'}, ..., y_7^{2'})$ and $(y_0^{3'}, ..., y_7^{3'})$ be outputs of F_1^*, F_2^* and F_3^*, respectively. Those notations are used to explain how to implement AURORA on 64-bit processors.

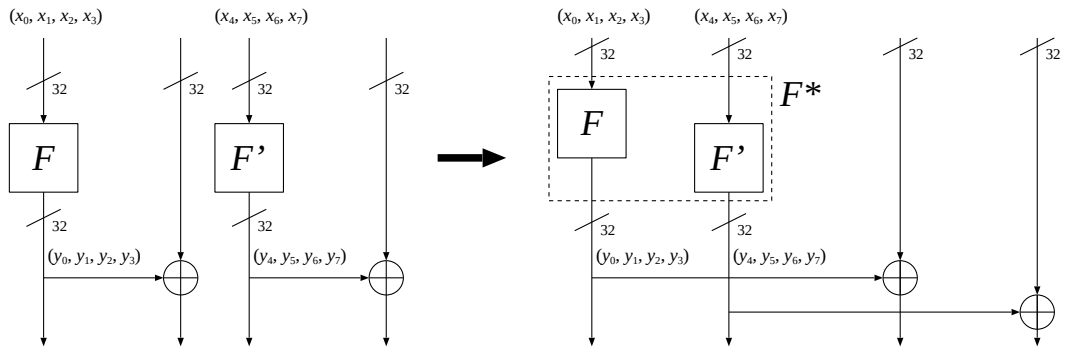

Figure 5.1: F^*-function.

$$F_0^*: \begin{pmatrix} y_0^{0'} \\ y_1^{0'} \\ \vdots \\ y_7^{0'} \end{pmatrix} = \begin{pmatrix} \mathcal{M}_0 & 0 \\ 0 & \mathcal{M}_1 \end{pmatrix} \begin{pmatrix} S(x_0^{0'}) \\ S(x_1^{0'}) \\ \vdots \\ S(x_7^{0'}) \end{pmatrix}$$

$$F_1^*: \begin{pmatrix} y_0^{1'} \\ y_1^{1'} \\ \vdots \\ y_7^{1'} \end{pmatrix} = \begin{pmatrix} \mathcal{M}_1 & 0 \\ 0 & \mathcal{M}_0 \end{pmatrix} \begin{pmatrix} S(x_0^{1'}) \\ S(x_1^{1'}) \\ \vdots \\ S(x_7^{1'}) \end{pmatrix}$$

$$F_2^*: \begin{pmatrix} y_0^{2'} \\ y_1^{2'} \\ \vdots \\ y_7^{2'} \end{pmatrix} = \begin{pmatrix} \mathcal{M}_2 & 0 \\ 0 & \mathcal{M}_3 \end{pmatrix} \begin{pmatrix} S(x_0^{2'}) \\ S(x_1^{2'}) \\ \vdots \\ S(x_7^{2'}) \end{pmatrix}$$

$$F_3^*: \begin{pmatrix} y_0^{3'} \\ y_1^{3'} \\ \vdots \\ y_7^{3'} \end{pmatrix} = \begin{pmatrix} \mathcal{M}_3 & 0 \\ 0 & \mathcal{M}_2 \end{pmatrix} \begin{pmatrix} S(x_0^{3'}) \\ S(x_1^{3'}) \\ \vdots \\ S(x_7^{3'}) \end{pmatrix}$$

Type-S1

Type-S1 is a straight-forward implementation suitable for 32-bit processors. This implementation requires the following eight different 8-bit to 32-bit tables T_0^0, T_1^0, T_2^0, T_3^0, T_0^1, T_1^1, T_2^1 and T_3^1 [14].

$$\begin{aligned}
T_0^0(x) &= (\quad S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x)) \\
T_1^0(x) &= (\{02\} \times S(x), \quad S(x), \{03\} \times S(x), \{02\} \times S(x)) \\
T_2^0(x) &= (\{02\} \times S(x), \{02\} \times S(x), \quad S(x), \{03\} \times S(x)) \\
T_3^0(x) &= (\{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \quad S(x)) \\
T_0^1(x) &= (\quad S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x)) \\
T_1^1(x) &= (\{06\} \times S(x), \quad S(x), \{02\} \times S(x), \{08\} \times S(x)) \\
T_2^1(x) &= (\{08\} \times S(x), \{06\} \times S(x), \quad S(x), \{02\} \times S(x)) \\
T_3^1(x) &= (\{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), \quad S(x))
\end{aligned}$$

The following eight tables can be represented by the previous eight tables.

$$\begin{aligned}
T_0^2(x) &= (\{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \quad S(x)) = T_3^0(x) \\
T_1^2(x) &= (\quad S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x)) = T_0^0(x) \\
T_2^2(x) &= (\{02\} \times S(x), \quad S(x), \{03\} \times S(x), \{02\} \times S(x)) = T_1^0(x) \\
T_3^2(x) &= (\{02\} \times S(x), \{02\} \times S(x), \quad S(x), \{03\} \times S(x)) = T_2^0(x) \\
T_0^3(x) &= (\{06\} \times S(x), \quad S(x), \{02\} \times S(x), \{08\} \times S(x)) = T_1^1(x) \\
T_1^3(x) &= (\{08\} \times S(x), \{06\} \times S(x), \quad S(x), \{02\} \times S(x)) = T_2^1(x) \\
T_2^3(x) &= (\{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), \quad S(x)) = T_3^1(x) \\
T_3^3(x) &= (\quad S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x)) = T_0^1(x)
\end{aligned}$$

The tables T_0^0, T_1^0, T_2^0 and T_3^0 are used for calculating F_0. Similarly, the tables T_0^1, T_1^1, T_2^1 and T_3^1 are for F_1, the tables T_0^2, T_1^2, T_2^2 and T_3^2 are for F_2 , and the tables T_0^3, T_1^3, T_2^3 and T_3^3 are for F_3, respectively. Thus the outputs of F functions can be calculated as follows:

$$
\begin{aligned}
(y_0^0, y_1^0, y_2^0, y_3^0) &= T_0^0(x_0^0) \oplus T_1^0(x_1^0) \oplus T_2^0(x_2^0) \oplus T_3^0(x_3^0) \\
(y_0^1, y_1^1, y_2^1, y_3^1) &= T_0^1(x_0^1) \oplus T_1^1(x_1^1) \oplus T_2^1(x_2^1) \oplus T_3^1(x_3^1) \\
(y_0^2, y_1^2, y_2^2, y_3^2) &= T_0^2(x_0^2) \oplus T_1^2(x_1^2) \oplus T_2^2(x_2^2) \oplus T_3^2(x_3^2) \\
&= T_3^0(x_0^2) \oplus T_0^0(x_1^2) \oplus T_1^0(x_2^2) \oplus T_2^0(x_3^2) \\
(y_0^3, y_1^3, y_2^3, y_3^3) &= T_0^3(x_0^3) \oplus T_1^3(x_1^3) \oplus T_2^3(x_2^3) \oplus T_3^3(x_3^3) \\
&= T_1^1(x_0^3) \oplus T_2^1(x_1^3) \oplus T_3^1(x_2^3) \oplus T_0^1(x_3^3)
\end{aligned}
$$

The required operations for this implementation are estimated as follows:

Size of table (KB):	8
Operations of F_0, F_1, F_2 and F_3	
# of table lookups:	16
# of XORs :	12

Type-S2

Type-S2 uses rotation operations to reduce the table size of Type-S1. This implementation needs two different 8-bit to 32-bit tables. Due to the rotation operations, the number of operations is increased. However, the table size can be reduced to 1/4 compared to Type-S1.

The tables T_1^0, T_2^0, T_3^0, T_1^1, T_2^1, T_3^1 can be replaced as follows:

$$
\begin{aligned}
T_1^0(x) &= T_0^0(x) \ggg 8 \\
T_2^0(x) &= T_0^0(x) \ggg 16 \\
T_3^0(x) &= T_0^0(x) \ggg 24 \\
T_1^1(x) &= T_0^1(x) \ggg 8 \\
T_2^1(x) &= T_0^1(x) \ggg 16 \\
T_3^1(x) &= T_0^1(x) \ggg 24
\end{aligned}
$$

This implementation requires the following operations.

Size of table (KB):	2
Operations of F_0, F_1, F_2 and F_3	
# of table lookups:	16
# of XORs :	12
# of rotations:	12

Type-S3

Type-S3 is a straight-forward implementation suitable for 64-bit processors. This implementation requires the following sixteen different 8-bit to 64-bit tables.

$$
\begin{aligned}
T_0^{0'}(x) &= (\quad S(x),\ \{03\}\times S(x),\ \{02\}\times S(x),\ \{02\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_1^{0'}(x) &= (\{02\}\times S(x),\quad S(x),\ \{03\}\times S(x),\ \{02\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_2^{0'}(x) &= (\{02\}\times S(x),\ \{02\}\times S(x),\quad S(x),\ \{03\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_3^{0'}(x) &= (\{03\}\times S(x),\ \{02\}\times S(x),\ \{02\}\times S(x),\quad S(x),\ 0,\ 0,\ 0,\ 0)\\
T_4^{0'}(x) &= (0,\ 0,\ 0,\ 0,\quad S(x),\ \{02\}\times S(x),\ \{08\}\times S(x),\ \{06\}\times S(x))\\
T_5^{0'}(x) &= (0,\ 0,\ 0,\ 0,\ \{06\}\times S(x),\quad S(x),\ \{02\}\times S(x),\ \{08\}\times S(x))\\
T_6^{0'}(x) &= (0,\ 0,\ 0,\ 0,\ \{08\}\times S(x),\ \{06\}\times S(x),\quad S(x),\ \{02\}\times S(x))\\
T_7^{0'}(x) &= (0,\ 0,\ 0,\ 0,\ \{02\}\times S(x),\ \{08\}\times S(x),\ \{06\}\times S(x),\quad S(x))\\
T_0^{1'}(x) &= (\quad S(x),\ \{02\}\times S(x),\ \{08\}\times S(x),\ \{06\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_1^{1'}(x) &= (\{06\}\times S(x),\quad S(x),\ \{02\}\times S(x),\ \{08\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_2^{1'}(x) &= (\{08\}\times S(x),\ \{06\}\times S(x),\quad S(x),\ \{02\}\times S(x),\ 0,\ 0,\ 0,\ 0)\\
T_3^{1'}(x) &= (\{02\}\times S(x),\ \{08\}\times S(x),\ \{06\}\times S(x),\quad S(x),\ 0,\ 0,\ 0,\ 0)\\
T_4^{1'}(x) &= (0,\ 0,\ 0,\ 0,\quad S(x),\ \{03\}\times S(x),\ \{02\}\times S(x),\ \{02\}\times S(x))\\
T_5^{1'}(x) &= (0,\ 0,\ 0,\ 0,\ \{02\}\times S(x),\quad S(x),\ \{03\}\times S(x),\ \{02\}\times S(x))\\
T_6^{1'}(x) &= (0,\ 0,\ 0,\ 0,\ \{02\}\times S(x),\ \{02\}\times S(x),\quad S(x),\ \{03\}\times S(x))\\
T_7^{1'}(x) &= (0,\ 0,\ 0,\ 0,\ \{03\}\times S(x),\ \{02\}\times S(x),\ \{02\}\times S(x),\quad S(x))
\end{aligned}
$$

The outputs of F^* functions $Y^{0'} = (y_0^{0'}||y_1^{0'}||...||y_7^{0'})$, $Y^{1'} = (y_0^{1'}||y_1^{1'}||...||y_7^{1'})$, $Y^{2'} = (y_0^{2'}||y_1^{2'}||...||y_7^{2'})$ and $Y^{3'} = (y_0^{3'}||y_1^{3'}||...||y_7^{3'})$ can be calculated as follows:

$$
\begin{aligned}
Y^{0'} &= T_0^{0'}(x_0^{0'}) \oplus T_1^{0'}(x_1^{0'}) \oplus T_2^{0'}(x_2^{0'}) \oplus T_3^{0'}(x_3^{0'}) \oplus T_4^{0'}(x_4^{0'}) \oplus T_5^{0'}(x_5^{0'}) \oplus T_6^{0'}(x_6^{0'}) \oplus T_7^{0'}(x_7^{0'})\\
Y^{1'} &= T_0^{1'}(x_0^{1'}) \oplus T_1^{1'}(x_1^{1'}) \oplus T_2^{1'}(x_2^{1'}) \oplus T_3^{1'}(x_3^{1'}) \oplus T_4^{1'}(x_4^{1'}) \oplus T_5^{1'}(x_5^{1'}) \oplus T_6^{1'}(x_6^{1'}) \oplus T_7^{1'}(x_7^{1'})\\
Y^{2'} &= T_0^{2'}(x_0^{2'}) \oplus T_1^{2'}(x_1^{2'}) \oplus T_2^{2'}(x_2^{2'}) \oplus T_3^{2'}(x_3^{2'}) \oplus T_4^{2'}(x_4^{2'}) \oplus T_5^{2'}(x_5^{2'}) \oplus T_6^{2'}(x_6^{2'}) \oplus T_7^{2'}(x_7^{2'})\\
Y^{3'} &= T_0^{3'}(x_0^{3'}) \oplus T_1^{3'}(x_1^{3'}) \oplus T_2^{3'}(x_2^{3'}) \oplus T_3^{3'}(x_3^{3'}) \oplus T_4^{3'}(x_4^{3'}) \oplus T_5^{3'}(x_5^{3'}) \oplus T_6^{3'}(x_6^{3'}) \oplus T_7^{3'}(x_7^{3'})
\end{aligned}
$$

This implementation requires the following operations.

Size of table (KB):	32
Operations of F_0^*, F_1^*, F_2^* and F_3^*	
# of table lookups:	32
# of XORs :	28

Type-S4

Type-S4 uses two rotation operations to reduce the table size of Type-S3. Since $T_0^{1'}, ..., T_7^{1'}$ can be implemented by using $T_0^{0'}, ..., T_7^{0'}$ with two rotations, the table size can be reduced to half compared to Type-S3.

$$
\begin{aligned}
Y^{0'} &= T_0^{0'}(x_0^{0'}) \oplus T_1^{0'}(x_1^{0'}) \oplus \cdots \oplus T_7^{0'}(x_7^{0'})\\
Y^{1'} &= \left(T_0^{1'}(x_0^{1'}) \oplus T_1^{1'}(x_1^{1'}) \oplus \cdots \oplus T_7^{1'}(x_7^{1'})\right) \ggg 32\\
Y^{2'} &= T_0^{2'}(x_0^{2'}) \oplus T_1^{2'}(x_1^{2'}) \oplus \cdots \oplus T_7^{2'}(x_7^{2'})\\
Y^{3'} &= \left(T_0^{3'}(x_0^{3'}) \oplus T_1^{3'}(x_1^{3'}) \oplus \cdots \oplus T_7^{3'}(x_7^{3'})\right) \ggg 32
\end{aligned}
$$

This implementation requires the following operations.

Size of table (KB):	16
Operations of F_0^*, F_1^*, F_2^* and F_3^*	
# of table lookups:	32
# of XORs :	28
# of rotations:	2

Type-S5

Type-S5 aims to reduce the table size of Type-S4 by half. It requires the following four different 8-bit to 64-bit tables.

$$
\begin{aligned}
T_0(x) = (\ & S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \\
& S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x)) \\
T_1(x) = (\ & \{02\} \times S(x), S(x), \{03\} \times S(x), \{02\} \times S(x), \\
& \{06\} \times S(x), S(x), \{02\} \times S(x), \{08\} \times S(x)) \\
T_2(x) = (\ & \{02\} \times S(x), \{02\} \times S(x), S(x), \{03\} \times S(x), \\
& \{08\} \times S(x), \{06\} \times S(x), S(x), \{02\} \times S(x)) \\
T_3(x) = (\ & \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), S(x), \\
& \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), S(x))
\end{aligned}
$$

$$
\begin{aligned}
Y^{0'} &= \big((T_0(x_0^{0'}) \oplus T_1(x_1^{0'}) \oplus T_2(x_2^{0'}) \oplus T_3(x_3^{0'}))\&\texttt{0xffffffff00000000}\big) \oplus \\
&\quad \big((T_0(x_4^{0'}) \oplus T_1(x_5^{0'}) \oplus T_2(x_6^{0'}) \oplus T_3(x_7^{0'}))\&\texttt{0x00000000ffffffff}\big) \\
Y^{1'} &= \big((T_0(x_0^{1'}) \oplus T_1(x_1^{1'}) \oplus T_2(x_2^{1'}) \oplus T_3(x_3^{1'})) \ll 32\big) \oplus \\
&\quad \big((T_0(x_4^{1'}) \oplus T_1(x_5^{1'}) \oplus T_2(x_6^{1'}) \oplus T_3(x_7^{1'})) \gg 32\big) \\
Y^{2'} &= \big((T_3(x_0^{2'}) \oplus T_0(x_1^{2'}) \oplus T_1(x_2^{2'}) \oplus T_2(x_3^{2'}))\&\texttt{0xffffffff00000000}\big) \oplus \\
&\quad \big((T_1(x_4^{2'}) \oplus T_2(x_5^{2'}) \oplus T_3(x_6^{2'}) \oplus T_0(x_7^{2'}))\&\texttt{0x00000000ffffffff}\big) \\
Y^{3'} &= \big((T_3(x_0^{3'}) \oplus T_0(x_1^{3'}) \oplus T_1(x_2^{3'}) \oplus T_2(x_3^{3'})) \ll 32\big) \oplus \\
&\quad \big((T_1(x_4^{3'}) \oplus T_2(x_5^{3'}) \oplus T_3(x_6^{3'}) \oplus T_0(x_7^{3'})) \gg 32\big)
\end{aligned}
$$

This implementation requires the following operations.

Size of table (KB):	8
Operations of F_0^*, F_1^*, F_2^* and F_3^*	
# of table lookups:	32
# of XORs :	28
# of ANDs :	4
# of shift operations :	4

Selecting Implementation Types in the Optimized Codes

We explain how to choose the implementation types described in the previous section from the optimized codes. In default, Type-S1 for 32-bit processors and Type-S3 for 64-bit processors are selected. When '_USE_ROT' is defined in preprocessor, Type-S2 for 32-bit processors is chosen. Similarly, when '_USE_SHIFT' is defined, Type-S4 is selected and when '_SHARE_TABLE' is defined, Type-S5 is selected.

Table 5.1: 32/64-bit Processors.

Platform	Processor	Clock speed [GHz]	Memory [GB]	OS	Compiler
A	Core 2 Duo	2.4	2.0	Windows Vista Ultimate (32-bit)	Visual Studio 2005 Professional Edition
B	Core 2 Duo	2.4	2.0	Windows Vista Ultimate (64-bit)	Visual Studio 2005 Professional Edition
C	Opteron	2.6	16.0	Linux kernel 2.4	gcc 3.2.3 (x64)
D	Pentium 4	2.26	1.0	Red Hat Linux 7.3	gcc 2.96

Table 5.2: 8-bit Processors.

Platform	Vender	Processor	Compiler	IDE
E	ATMEL	megaAVR family	gcc-4.3.0 (WinAVR 20080610)	AVR Studio 4.1.4 build 589
F	RENESAS	H8/300 family, 3217 Group	ch38 V.6.02.00.000	HEW 4.03.00.001 (+H8/300 tool chain 6.2.0)

5.1.2 Evaluation Results

This section shows the evaluation results of AURORA-224/256/384/512 on 8/32/64-bit processors. We omit the results of AURORA-224M/256M. As mentioned in Sec. 2.7 and 2.8, AURORA-224M/256M is structurally very similar to AURORA-384/512, except for constants and final mixing function. These differences affect the performance results little. Thus the evaluation results of AURORA-224M/256M can be deduced from those of AURORA-384/512.

The number of cycles/byte for 1 byte message on each table implicate the minimum number of clock cycles to generate one message digest. For instance, the number of clock cycles of AURORA-224 implemented by Type-S1 (unroll) to generate one message digest of 1 byte message is 1848 cycles on the Platform A. Since there is no calculation for setting up the algorithms in the optimized code (e.g., build internal tables), the results on the tables are precise clock cycles to generate hash values.

32/64-bit Processors

We present the evaluation results on performance of AURORA-224/256/384/512 on 32/64-bit processors at the present. The platforms used for the evaluation are shown in Table 5.1. We use cycle counters included in 'cycle.h' [13]. This code provides machine dependent cycle counters.

Tables 5.3, 5.4, 5.5 and 5.6 represent the evaluation results of AURORA-224, AURORA-256, AURORA-384 and AURORA-512, respectively. All implementation types described in Sec. 5.1.1 are evaluated for each AURORA hash function. Also, two types of loop structure 'unroll' and 'looped' are evaluated. In the 'unroll' implementation, the round functions of AURORA are unrolled. Similarly, in the 'looped' implementation, the round functions are implemented by loop function. Besides the results of AURORA hash functions, the evaluation results of SHA-256 and SHA-512 implemented by Brian Gladman [42] are shown in Tables 5.7 and 5.8 by using the same evaluation method to compare the performances.

8-bit Processors

We present the evaluation results on performance of AURORA-224/256/384/512 on 8-bit processors at the present. The platforms used for the evaluation are shown in Table 5.2. Table 5.9 shows

the evaluation results of the compression function for AURORA-224/256 and AURORA-384/512. Tables 5.10, 5.11, 5.12 and 5.13 represent the evaluation results of AURORA-224, AURORA-256, AURORA-384 and AURORA-512, respectively.

Table 5.3: AURORA-224 on 32/64-bit processors.

		Hash Function [cycles/byte]					1 CF call [cycles]	code size [bytes]
message size [bytes]		1	10	100	1,000	10,000	-	-
		Platform A (Core 2 Duo (32-bit))						
Type-S1	(unroll)	1,847.4	188.4	36.6	26.2	25.3	1,598.1	142,926
	(looped)	1,860.8	190.2	36.8	26.6	25.6	1,616.9	64,996
Type-S2	(unroll)	1,788.9	183.2	35.7	25.3	24.3	1,534.8	179,662
	(looped)	1,929.4	195.8	38.1	27.3	26.3	1,662.1	60,172
Type-S3	(unroll)	3,117.2	317.1	62.4	45.8	44.5	2,821.4	198,002
	(looped)	2,586.0	265.6	51.8	37.2	35.8	2,285.9	121,034
Type-S4	(unroll)	2,803.6	283.9	55.4	41.0	39.8	2,535.4	174,070
	(looped)	2,625.7	265.4	51.7	37.9	36.8	2,334.0	96,262
Type-S5	(unroll)	2,686.2	272.6	52.9	39.0	37.7	2,396.4	163,270
	(looped)	2,477.9	251.3	48.7	35.7	34.6	2,193.2	83,990
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(unroll)	1,270.9	125.6	23.9	17.3	16.8	1,066.8	149,072
	(looped)	1,412.5	140.1	26.7	19.6	19.0	1,204.9	66,326
Type-S2	(unroll)	1,490.4	147.8	28.2	20.9	20.3	1,288.1	189,792
	(looped)	1,608.2	159.6	30.4	22.6	22.0	1,397.8	62,126
Type-S3	(unroll)	1,155.4	119.0	22.5	15.9	15.4	980.7	205,626
	(looped)	1,308.2	132.7	25.3	18.2	17.6	1,119.1	128,490
Type-S4	(unroll)	1,177.8	119.3	22.6	16.2	15.7	995.1	181,694
	(looped)	1,262.2	128.7	24.3	17.7	17.1	1,086.2	103,718
Type-S5	(unroll)	1,342.9	134.9	25.5	18.7	18.2	1,156.4	170,894
	(looped)	1,421.3	142.8	27.0	20.0	19.4	1,233.9	91,446
		Platform C (Opteron)						
Type-S1	(unroll)	2,742.1	276.1	50.2	36.4	35.3	2,246.4	57,305
	(looped)	2,912.1	292.1	54.0	39.5	38.3	2,455.0	21,641
Type-S2	(unroll)	2,972.8	299.7	55.3	40.6	39.3	2,521.9	51,241
	(looped)	3,091.9	311.5	57.8	42.6	41.3	2,654.4	15,625
Type-S3	(unroll)	2,196.4	221.9	40.0	28.9	27.9	1,773.9	83,609
	(looped)	1,590.4	161.1	28.1	19.3	18.5	1,179.0	46,169
Type-S4	(unroll)	2,114.6	213.8	38.7	27.7	26.7	1,702.2	70,073
	(looped)	1,611.0	164.8	28.7	19.8	19.0	1,197.1	30,073
Type-S5	(unroll)	2,173.0	220.0	39.7	28.4	27.4	1,748.5	60,537
	(looped)	1,709.0	173.3	30.0	20.1	19.2	1,234.9	21,881
		Platform D (Pentium 4)						
Type-S1	(unroll)	4,299.5	436.3	79.5	53.7	51.4	3,279.8	59,772
	(looped)	4,197.9	428.0	78.9	52.0	49.7	2,930.5	22,092
Type-S2	(unroll)	5,069.7	498.5	94.7	67.3	65.2	4,142.2	55,172
	(looped)	5,093.4	525.4	100.3	68.8	66.6	3,236.1	16,560
Type-S3	(unroll)	10,748.6	1,082.7	199.8	148.4	143.9	9,155.1	127,828
	(looped)	7,504.9	755.5	143.9	106.1	103.0	6,588.6	55,436
Type-S4	(unroll)	10,486.6	1,051.0	194.2	143.6	139.4	8,905.0	103,356
	(looped)	7,471.2	762.0	142.0	103.7	100.4	6,438.1	38,536
Type-S5	(unroll)	10,005.3	1,010.8	188.4	139.6	135.3	8,579.3	89,528
	(looped)	7,213.3	725.8	136.8	99.1	96.3	6,210.2	29,580

Table 5.4: AURORA-256 on 32/64-bit processors.

		Hash Function [cycles/byte]					1 CF call [cycles]	code size [bytes]
message size [bytes]		1	10	100	1,000	10,000	-	-
		Platform A (Core 2 Duo (32-bit))						
Type-S1	(unroll)	1,836.3	185.7	36.3	26.2	25.4	1,598.1	142,926
	(looped)	1,837.4	188.2	36.5	26.5	25.6	1,616.9	64,996
Type-S2	(unroll)	1,770.2	179.9	35.0	25.1	24.3	1,534.8	179,662
	(looped)	1,902.6	193.6	37.6	27.2	26.3	1,662.1	60,172
Type-S3	(unroll)	3,069.0	311.5	61.6	45.8	44.3	2,821.4	198,002
	(looped)	2,544.0	259.1	50.9	36.9	35.8	2,285.9	121,034
Type-S4	(unroll)	2,787.6	280.9	55.1	41.0	39.8	2,535.4	174,070
	(looped)	2,585.4	260.7	51.2	37.9	36.7	2,334.0	96,262
Type-S5	(unroll)	2,649.7	267.4	52.4	38.9	37.7	2,396.4	163,270
	(looped)	2,447.1	248.2	48.4	35.7	34.6	2,193.2	83,990
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(unroll)	1,235.3	123.4	23.6	17.3	16.8	1,066.8	149,072
	(looped)	1,374.9	137.5	26.4	19.5	19.0	1,204.9	66,326
Type-S2	(unroll)	1,459.8	145.1	28.0	20.8	20.2	1,288.1	189,792
	(looped)	1,576.1	156.3	30.4	22.6	22.0	1,397.8	62,126
Type-S3	(unroll)	1,142.2	115.4	22.3	15.9	15.4	980.7	205,626
	(looped)	1,273.7	130.1	25.0	18.1	17.6	1,119.1	128,490
Type-S4	(unroll)	1,154.8	117.2	22.3	16.2	15.7	995.1	181,694
	(looped)	1,247.7	126.0	24.1	17.7	17.1	1,086.2	103,718
Type-S5	(unroll)	1,315.3	132.6	25.2	18.7	18.2	1,156.4	170,894
	(looped)	1,392.6	140.4	26.8	20.0	19.4	1,233.9	91,446
		Platform C (Opteron)						
Type-S1	(unroll)	2,575.6	262.1	48.9	36.3	35.2	2,246.4	57,305
	(looped)	2,792.2	280.1	52.8	39.3	38.2	2,455.0	21,641
Type-S2	(unroll)	2,848.6	286.9	54.0	40.5	39.3	2,521.9	51,241
	(looped)	2,978.0	299.8	56.6	42.4	41.3	2,654.4	15,625
Type-S3	(unroll)	2,074.4	209.9	38.8	28.8	27.9	1,773.9	83,609
	(looped)	1,476.0	149.7	27.0	19.2	18.5	1,179.0	46,169
Type-S4	(unroll)	2,005.7	202.5	37.6	27.6	26.7	1,702.2	70,073
	(looped)	1,492.9	153.5	27.6	19.7	19.0	1,197.1	30,073
Type-S5	(unroll)	2,065.0	213.5	39.0	28.3	27.4	1,748.5	60,537
	(looped)	1,534.7	155.9	28.2	19.9	19.2	1,234.9	21,881
		Platform D (Pentium 4)						
Type-S1	(unroll)	4,036.5	422.3	77.9	53.5	52.2	3,279.8	59,772
	(looped)	3,963.2	403.6	76.6	52.2	49.6	2,930.5	22,092
Type-S2	(unroll)	5,069.7	498.5	94.7	67.3	65.2	4,142.2	55,172
	(looped)	5,318.5	515.0	101.0	69.2	66.5	3,236.1	16,560
Type-S3	(unroll)	10,475.5	1,045.4	196.9	148.2	143.8	9,155.1	127,828
	(looped)	7,297.2	736.2	142.1	106.0	102.8	6,588.6	55,436
Type-S4	(unroll)	10,055.7	1,016.1	191.1	143.3	139.4	8,905.0	103,356
	(looped)	7,256.2	735.6	139.7	103.5	100.4	6,438.1	38,536
Type-S5	(unroll)	9,712.1	984.7	185.5	139.1	134.9	8,579.3	89,528
	(looped)	6,992.4	709.0	134.6	98.9	96.3	6,210.2	29,580

Table 5.5: AURORA-384 on 32/64-bit processors.

		Hash Function [cycles/byte]					1 CF call [cycles]	code size [bytes]
message size [bytes]		1	10	100	1,000	10,000	-	-
		Platform A (Core 2 Duo (32-bit))						
Type-S1	(unroll)	5,709.0	574.9	86.1	48.8	47.6	2,666.9	142,926
	(looped)	6,160.2	628.3	92.5	52.8	51.4	2,875.5	64,996
Type-S2	(unroll)	5,724.6	574.7	85.0	48.2	46.8	2,612.9	179,662
	(looped)	5,743.4	580.9	86.4	49.0	47.7	2,666.1	60,172
Type-S3	(unroll)	21,527.2	2,153.0	320.7	187.3	183.1	10,343.1	198,002
	(looped)	8,686.9	875.5	130.6	74.3	72.4	4,043.8	121,034
Type-S4	(unroll)	20,457.5	2,048.5	305.3	178.6	174.7	9,842.3	174,070
	(looped)	7,603.6	764.1	113.7	64.9	63.4	3,554.4	96,262
Type-S5	(unroll)	21,207.6	2,118.5	315.0	183.7	179.8	10,131.3	163,270
	(looped)	7,270.0	732.0	108.8	62.1	60.6	3,395.7	83,990
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(unroll)	3,786.4	378.0	55.4	31.7	30.7	1,711.8	149,072
	(looped)	4,079.6	407.6	59.7	34.0	33.2	1,853.4	66,326
Type-S2	(unroll)	4,267.8	426.4	62.6	35.7	34.9	1,962.1	189,792
	(looped)	4,572.6	456.8	67.2	38.4	37.5	2,099.1	62,126
Type-S3	(unroll)	3,455.2	346.4	50.8	28.2	27.4	1,521.4	205,626
	(looped)	4,002.3	403.3	59.1	33.1	32.2	1,774.7	128,490
Type-S4	(unroll)	3,506.9	352.0	51.6	28.9	28.1	1,566.7	181,694
	(looped)	3,694.3	371.2	54.6	30.7	29.9	1,670.8	103,718
Type-S5	(unroll)	3,803.9	382.6	55.9	31.8	31.0	1,725.2	170,894
	(looped)	4,057.3	404.4	59.3	33.7	32.9	1,837.9	91,446
		Platform C (Opteron)						
Type-S1	(unroll)	7,943.1	796.6	115.7	65.6	63.9	3,587.9	57,305
	(looped)	7,212.3	723.9	104.7	59.2	57.7	3,253.4	21,641
Type-S2	(unroll)	8,864.7	886.1	129.1	74.1	72.3	4,060.8	51,241
	(looped)	8,103.2	816.5	118.6	67.3	65.6	3,675.1	15,625
Type-S3	(unroll)	8,427.3	844.4	122.8	70.1	68.4	3,844.2	83,609
	(looped)	4,214.4	422.6	59.7	32.6	31.5	1,755.7	46,169
Type-S4	(unroll)	8,938.8	895.7	130.9	75.5	73.9	4,139.6	70,073
	(looped)	4,223.2	423.1	60.0	32.9	31.9	1,777.1	30,073
Type-S5	(unroll)	7,850.4	787.3	114.6	65.4	63.8	3,577.0	60,537
	(looped)	4,380.0	439.5	62.5	34.4	33.3	1,854.6	21,881
		Platform D (Pentium 4)						
Type-S1	(unroll)	13,832.6	1,410.5	204.5	106.5	98.3	5,483.4	59,772
	(looped)	13,475.2	1,342.5	200.5	111.2	107.9	5,811.4	22,092
Type-S2	(unroll)	16,059.2	1,600.5	233.7	135.6	126.2	7,090.0	55,172
	(looped)	17,817.9	1,778.9	263.5	150.9	147.4	6,420.3	16,560
Type-S3	(unroll)	29,331.6	2,949.2	433.6	247.9	241.8	13,570.7	127,828
	(looped)	21,529.5	2,144.9	316.4	181.0	176.5	9,930.2	55,436
Type-S4	(unroll)	28,286.8	2,839.3	416.8	237.6	232.0	13,046.8	103,356
	(looped)	20,501.7	2,051.2	301.4	171.2	167.1	9,451.8	38,536
Type-S5	(unroll)	26,879.0	2,703.9	395.7	225.2	219.7	12,367.1	89,528
	(looped)	20,039.9	2,000.5	295.0	167.4	162.8	9,348.4	29,580

Table 5.6: AURORA-512 on 32/64-bit processors.

		Hash Function [cycles/byte]					1 CF call [cycles]	code size [bytes]
message size [bytes]		1	10	100	1,000	10,000	-	-
		Platform A (Core 2 Duo (32-bit))						
Type-S1	(unroll)	5,733.8	577.0	86.1	48.9	47.6	2,666.9	142,926
	(looped)	6,166.8	619.4	92.4	52.9	51.5	2,875.5	64,996
Type-S2	(unroll)	5,737.9	573.2	85.4	48.1	46.9	2,612.9	179,662
	(looped)	5,779.8	582.3	86.4	49.0	47.9	2,666.1	60,172
Type-S3	(unroll)	21,441.6	2,147.6	319.8	186.2	183.3	10,343.1	198,002
	(looped)	8,647.4	869.0	129.8	74.2	72.4	4,043.8	121,034
Type-S4	(unroll)	20,521.7	2,056.0	305.6	177.5	174.8	9,842.3	174,070
	(looped)	7,587.3	759.8	113.2	64.9	63.3	3,554.4	96,262
Type-S5	(unroll)	20,975.2	2,099.8	312.5	183.7	179.0	10,131.3	163,270
	(looped)	7,233.3	728.5	108.5	62.1	60.6	3,395.7	83,990
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(unroll)	3,743.1	372.1	54.9	31.4	30.7	1,711.8	149,072
	(looped)	4,028.9	401.5	59.2	34.0	33.2	1,853.4	66,326
Type-S2	(unroll)	4,210.4	421.0	61.9	35.7	34.9	1,962.1	189,792
	(looped)	4,523.4	451.6	66.9	38.5	37.6	2,099.1	62,126
Type-S3	(unroll)	3,377.2	340.3	50.2	28.1	27.4	1,521.4	205,626
	(looped)	3,928.2	394.6	58.5	33.0	32.2	1,774.7	128,490
Type-S4	(unroll)	3,440.3	346.1	51.0	28.8	28.1	1,566.7	181,694
	(looped)	3,653.1	366.0	54.2	30.7	29.9	1,670.8	103,718
Type-S5	(unroll)	3,751.1	376.4	55.4	31.7	30.9	1,725.2	170,894
	(looped)	3,992.3	400.0	58.8	33.6	32.9	1,837.9	91,446
		Platform C (Opteron)						
Type-S1	(unroll)	7,766.1	776.1	113.5	65.3	63.8	3,587.9	57,305
	(looped)	7,023.7	702.6	102.6	59.0	57.7	3,253.4	21,641
Type-S2	(unroll)	8,639.4	865.4	127.1	73.9	72.3	4,060.8	51,241
	(looped)	7,861.8	788.1	115.6	67.1	65.6	3,675.1	15,625
Type-S3	(unroll)	8,133.3	813.6	119.7	69.8	68.4	3,844.2	83,609
	(looped)	3,967.2	398.1	57.3	32.3	31.5	1,755.7	46,169
Type-S4	(unroll)	8,744.3	875.6	129.0	75.3	74.0	4,139.6	70,073
	(looped)	4,032.2	404.1	58.1	32.7	31.9	1,777.1	30,073
Type-S5	(unroll)	7,636.7	764.4	112.4	65.2	63.8	3,577.0	60,537
	(looped)	4,161.0	416.8	60.2	34.1	33.3	1,854.6	21,881
		Platform D (Pentium 4)						
Type-S1	(unroll)	13,666.9	1,370.1	198.9	108.5	98.2	5,483.4	59,772
	(looped)	13,098.6	1,321.6	195.9	110.9	107.9	5,811.4	22,092
Type-S2	(unroll)	15,518.0	1,557.4	229.4	129.7	128.8	7,090.0	55,172
	(looped)	17,203.8	1,729.2	259.2	150.6	147.2	6,420.3	16,560
Type-S3	(unroll)	29,074.0	2,915.4	430.1	247.3	241.7	13,570.7	127,828
	(looped)	20,826.0	2,087.1	310.3	180.0	176.3	9,930.2	55,436
Type-S4	(unroll)	27,901.7	2,800.7	413.4	237.3	231.9	13,046.8	103,356
	(looped)	20,095.9	2,030.2	298.1	171.0	166.9	9,451.8	38,536
Type-S5	(unroll)	26,513.4	2,662.9	391.9	224.8	219.6	12,367.1	89,528
	(looped)	19,608.8	1,978.8	291.8	166.4	162.5	9,348.4	29,580

Table 5.7: SHA-256 on 32/64-bit processors.

	Hash Function [cycles/byte]					1 CF call [cycles]	code size [bytes]
message size [bytes]	1	10	100	1,000	10,000	-	-
	Platform A (Core 2 Duo (32-bit))						
	1,609.3	162.2	31.0	23.1	22.5	1,302.1	43,802
	Platform B (Core 2 Duo (64-bit))						
	1,376.1	138.6	26.9	20.5	20.2	1,198.1	44,452
	Platform C (Opteron)						
	1,686.0	169.4	31.9	24.0	23.3	1,403.0	13,745
	Platform D (Pentium 4)						
	3,084.2	311.4	57.2	42.5	41.2	2,390.3	23,668

Table 5.8: SHA-512 on 32/64-bit processors.

	Hash Function [cycles/byte]					CF call [cycles]	code size [bytes]
message size [bytes]	1	10	100	1,000	10,000	-	-
	Platform A (Core 2 Duo (32-bit))						
	6,191.1	621.2	61.2	43.6	42.5	5,118.4	43,802
	Platform B (Core 2 Duo (64-bit))						
	1,805.1	181.5	19.0	13.6	13.3	1,512.6	44,452
	Platform C (Opteron)						
	2,237.0	224.7	22.7	15.4	14.9	1,779.5	13,745
	Platform D (Pentium 4)						
	15,873.8	1,684.2	176.5	120.7	107.8	13,269.1	23,668

Table 5.9: Compression functions for AURORA-224/256 and AURORA-384/512 on 8-bit processors.

CF	Platform	1 CF call [cycles/byte]	code size [bytes]	stack [bytes]
AURORA-224/256	Platform E	446,675	6,158	204
	Platform F	3,410,460	4,596	216
AURORA-384/512	Platform E	676,814	6,158	240
	Platform F	5,152,644	4,596	250

Table 5.10: AURORA-224 on 8-bit processors.

	Hash Function [cycles/byte]					code size [bytes]	stack [bytes]
message size [bytes]	1	10	100	400	1,000	-	-
Platform E	451,055	45,255.0	9,147.8	8,002.6	7,326.3	6,158	442
Platform F	3,428,682	343,170.6	68,803.2	60,169.4	55,024.0	4,596	320

Table 5.11: AURORA-256 on 8-bit processors.

	Hash Function [cycles/byte]					code size [bytes]	stack [bytes]
message size [bytes]	1	10	100	400	1,000	-	-
Platform E	450,601	45,209.6	9,143.3	8,001.5	7,325.9	6,158	442
Platform F	3,425,578	342,922.6	68,767.1	60,158.0	55,022.9	4,596	300

Table 5.12: AURORA-384 on 8-bit processors.

	Hash Function [cycles/byte]					code size [bytes]	stack [bytes]
message size [bytes]	1	10	100	400	1,000	-	-
Platform E	1,358,852	136,034.7	20,527.6	13,724.9	12,363.7	6,158	503
Platform F	10,331,178	1,033,537.6	155,252.8	103,566.0	93,225.5	4,596	352

Table 5.13: AURORA-512 on 8-bit processors.

	Hash Function [cycles/byte]					code size [bytes]	stack [bytes]
message size [bytes]	1	10	100	400	1,000	-	-
Platform E	1,358,098	135,959.3	20,520.1	13,723.0	12,363.0	6,158	486
Platform F	10,324,052	1,032,861.2	155,179.1	103,546.4	93,217.6	4,596	300

5.2 Hardware Implementation

This section describes the hardware optimization techniques and performance results of AURORA. Since the implementations of AURORA-224 and AURORA-384 are basically same as AURORA-256 and AURORA-512, respectively, except the initial value and truncation of final hash value, we designed and evaluated the implementations of AURORA-256 and AURORA-512 in this section.

5.2.1 Optimization Techniques of F-functions

We introduce optimization techniques of F-functions focusing on an S-box, matrices and a pipeline architecture in hardware implementation.

S-box

The 8-bit S-box of AURORA consists of three layers: affine transformation f, inversion over $\mathrm{GF}((2^4)^2)$ and affine transformation g. In Fig. 5.2 we show the schematic design of our S-box implementation. The inversion is performed in $\mathrm{GF}((2^4)^2)$ defined by the following polynomials:

$$\begin{cases} \mathrm{GF}(2^4) & : p(x) = x^4 + x + 1 \\ \mathrm{GF}((2^4)^2) & : q(x) = x^2 + x + \lambda \quad (\lambda = \{1001\} \in \mathrm{GF}(2^4)) \end{cases}.$$

For an arbitrary element $a_0\beta + a_1$ over $\mathrm{GF}((2^4)^2)$ where $a_0, a_1 \in \mathrm{GF}(2^4)$ and β is a root of $q(x)$, the inversion $b_0\beta + b_1 = (a_0\beta + a_1)^{-1}$ $(b_0, b_1 \in \mathrm{GF}(2^4))$ is computed as follows [46]:

$$\begin{aligned} b_0 &= a_0\Delta^{-1}, \\ b_1 &= (a_0 + a_1)\Delta^{-1}, \\ \Delta &= (a_0 + a_1)a_1 + \lambda a_0^2. \end{aligned}$$

These arithmetics except an inversion over $\mathrm{GF}(2^4)$, which is automatically generated by logic synthesis tool according to 16 entries $\times$ 4 bits table, can be implemented using NAND logic gates and XOR logic gates.

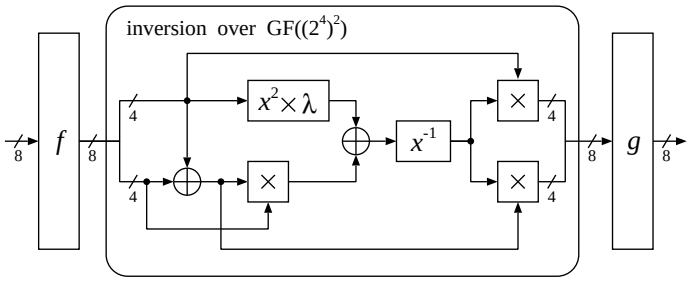

Figure 5.2: Schematic design of S-box implementation.

In Sec. 5.2.3, we apply not only this type of S-box implementation to all the hardware designs of AURORA but also table-lookup S-box implementation using 256 entries $\times$ 8 bits table to Type-H1 implementation described in Sec. 5.2.2 for higher throughput.

Matrices $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$

The 4×4 matrices $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ are multiplied to the outputs of S-boxes as a linear $(4, 4)$ multipermutation over $\mathrm{GF}(2^8)$ which is defined by an irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$. An addition of two elements in $\mathrm{GF}(2^8)$, denoted by $\oplus$, is equivalent to a bitwise XOR operation of their representations as an 8-bit binary string, which costs 8 XOR logic gates. A multiplication in

GF(2^8), denoted by $\times$, corresponds to a multiplication of polynomials modulo $x^8+x^4+x^3+x^2+1$. For an element a in GF(2^8), $\{02\} \times a$, $\{04\} \times a$ and $\{08\} \times a$ require 3, 5 and 8 XOR logic gates, respectively.

The matrix $\mathcal{M}_0$ can be decomposed into the following form by using the common term.

$$\begin{pmatrix} 01 & 02 & 02 & 03 \\ 03 & 01 & 02 & 02 \\ 02 & 03 & 01 & 02 \\ 02 & 02 & 03 & 01 \end{pmatrix} = \begin{pmatrix} 01 & 00 & 00 & 01 \\ 01 & 01 & 00 & 00 \\ 00 & 01 & 01 & 00 \\ 00 & 00 & 01 & 01 \end{pmatrix} \begin{pmatrix} 01 & 00 & 02 & 00 \\ 00 & 01 & 00 & 02 \\ 02 & 00 & 01 & 00 \\ 00 & 02 & 00 & 01 \end{pmatrix} + \begin{pmatrix} 00 & 00 & 00 & 02 \\ 02 & 00 & 00 & 00 \\ 00 & 02 & 00 & 00 \\ 00 & 00 & 02 & 00 \end{pmatrix}$$

For an input vector (x_0, x_1, x_2, x_3) and an output vector (y_0, y_1, y_2, y_3), the multiplication by $\mathcal{M}_0$ can be computed through the following equations.

$$\begin{cases} a_0 = \{02\} \times x_0 \\ a_1 = \{02\} \times x_1 \\ a_2 = \{02\} \times x_2 \\ a_3 = \{02\} \times x_3 \end{cases} \begin{cases} b_0 = a_2 \oplus x_0 \\ b_1 = a_3 \oplus x_1 \\ b_2 = a_0 \oplus x_2 \\ b_3 = a_1 \oplus x_3 \end{cases} \begin{cases} y_0 = a_3 \oplus b_0 \oplus b_3 \\ y_1 = a_0 \oplus b_1 \oplus b_0 \\ y_2 = a_1 \oplus b_2 \oplus b_1 \\ y_3 = a_2 \oplus b_3 \oplus b_2 \end{cases}$$

The total number and the maximum delay of XOR gates required for multiplication by $\mathcal{M}_0$ are 112 and 4, respectively.

The matrix $\mathcal{M}_1$ can be decomposed into the following form by using the common term.

$$\begin{pmatrix} 01 & 06 & 08 & 02 \\ 02 & 01 & 06 & 08 \\ 08 & 02 & 01 & 06 \\ 06 & 08 & 02 & 01 \end{pmatrix} = \begin{pmatrix} 01 & 04 & 00 & 00 \\ 00 & 01 & 04 & 00 \\ 00 & 00 & 01 & 04 \\ 04 & 00 & 00 & 01 \end{pmatrix} \begin{pmatrix} 01 & 02 & 00 & 00 \\ 00 & 01 & 02 & 00 \\ 00 & 00 & 01 & 02 \\ 02 & 00 & 00 & 01 \end{pmatrix} + \begin{pmatrix} 00 & 00 & 00 & 02 \\ 02 & 00 & 00 & 00 \\ 00 & 02 & 00 & 00 \\ 00 & 00 & 02 & 00 \end{pmatrix}$$

For an input vector (x_0, x_1, x_2, x_3) and an output vector (y_0, y_1, y_2, y_3), the multiplication by $\mathcal{M}_1$ can be computed through the following equations.

$$\begin{cases} a_0 = \{02\} \times x_0 \\ a_1 = \{02\} \times x_1 \\ a_2 = \{02\} \times x_2 \\ a_3 = \{02\} \times x_3 \end{cases} \begin{cases} b_0 = a_1 \oplus x_0 \\ b_1 = a_2 \oplus x_1 \\ b_2 = a_3 \oplus x_2 \\ b_3 = a_0 \oplus x_3 \end{cases} \begin{cases} c_0 = \{04\} \times b_0 \\ c_1 = \{04\} \times b_1 \\ c_2 = \{04\} \times b_2 \\ c_3 = \{04\} \times b_3 \end{cases} \begin{cases} y_0 = a_3 \oplus b_0 \oplus c_1 \\ y_1 = a_0 \oplus b_1 \oplus c_2 \\ y_2 = a_1 \oplus b_2 \oplus c_3 \\ y_3 = a_2 \oplus b_3 \oplus c_0 \end{cases}$$

The total number and the maximum delay of XOR gates required for multiplication by $\mathcal{M}_1$ are 128 and 4, respectively.

The matrices $\mathcal{M}_2$ and $\mathcal{M}_3$ are composed of the common row vectors to $\mathcal{M}_0$ and $\mathcal{M}_1$. Therefore, the multiplications by $\mathcal{M}_2$ and $\mathcal{M}_3$ are computed by substituting elements of an output vector of the multiplication by $\mathcal{M}_0$ and $\mathcal{M}_1$, respectively.

Dividing F-functions for pipeline architecture

In Fig. 5.3, we show the circuits of F-functions F_0 and F_1. The characters f, I and g in the figure represent the circuit of the function f, the inverse function over GF($(2^4)^2$) and the function g in the S-box S, respectively. In Sec. 5.2.2, we apply the pipeline architecture to both Type-H3 and Type-H4 implementations of AURORA-256 and AURORA-512 in order to achieve higher throughput. By dividing the circuit F_0 into the two parts α and β and inserting registers between α and β, we can shorten the critical path of the designs and improve the maximum operating frequency. Similarly, the circuit F_1 is divided into the two parts α and γ.

5.2.2 Data Path Architectures

For both AURORA-256 and AURORA-512, we designed four types of hardware implementations: Type-H1, Type-H2, Type-H3 and Type-H4 implementation. All the implementations do not include padding function; we assume that an input message is padded and divided into message blocks of 512 bits. We give the data path architecture of each implementation, where all registers represented by a box with shadow are composed of registers without enable signal.

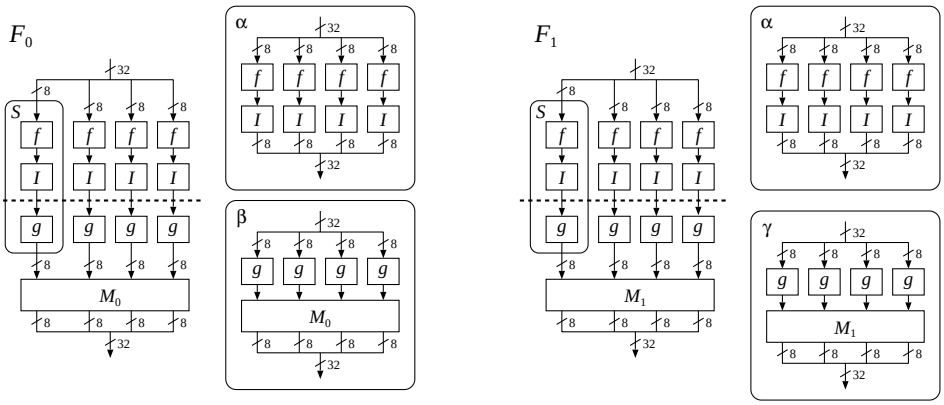

Figure 5.3: Dividing F-functions for pipeline architecture.

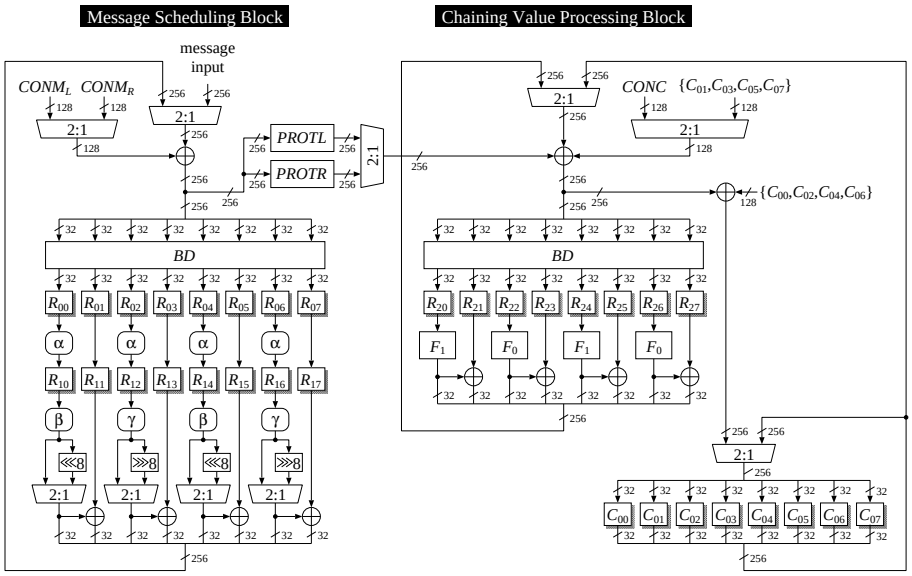

Figure 5.4: Data path architecture of AURORA-256 Type-H1 implementation.

AURORA-256 Type-H1

AURORA-256 Type-H1 implementation processes a round of the AURORA architecture both in one of the message scheduling module *MSM* and in the chaining value processing module *CPM* simultaneously in one clock cycle. It requires 8 F-function circuits and takes 18 cycles for both the compression function *CF* and the finalization function *FF*. Fig. 5.4 shows the data path architecture of AURORA-256 Type-H1 implementation. It is divided into two blocks: the message scheduling block and the chaining value processing block.

In the message scheduling block, a 512-bit message block is input in two cycles; the left 256-bit M_L is input at the 1st cycle and the right 256-bit M_R is input at the 2nd cycle. 256-bit intermediate values of MS_L (MSF_L) are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$ at the cycle of even order and stored in eight 32-bit registers $\{R_{10}, \ldots, R_{17}\}$ at the cycle of odd order. On the other hand, 256-bit intermediate values of MS_R (MSF_R) are stored in $\{R_{10}, \ldots, R_{17}\}$ at the cycle of even order and stored in $\{R_{00}, \ldots, R_{07}\}$ at the cycle of odd order. The pipeline architecture described in Sec. 5.2.1 is introduced into the message scheduling block; 32-bit registers are inserted between

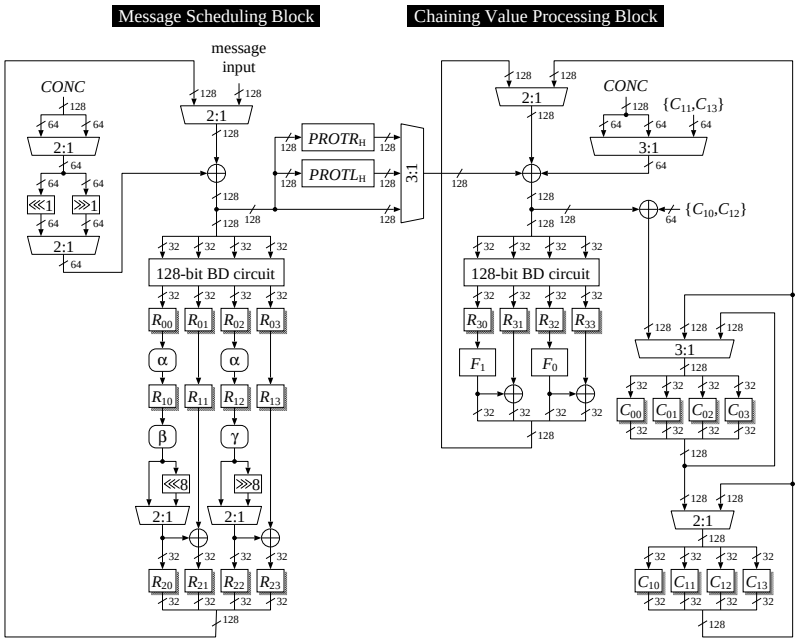

Figure 5.5: Data path architecture of AURORA-256 Type-H2 implementation.

α and β, and between α and γ. The architecture cannot shorten the critical path of the whole circuit because the longer paths exist in the chaining value processing block, but can reduce the rate of increase in area of the message scheduling block at high operating frequency. We note that the outputs of β and γ are byte-rotated to the left and to the right, respectively, when 256-bit intermediate values of MS_R (MSF_R) are processed.

In the chaining value processing block, the chaining value stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded and set into eight 32-bit registers $\{R_{20}, \ldots, R_{27}\}$ through the byte diffusion circuit BD after being XORed with the data fed from the message scheduling block and constant values $CONC$. BD can be implemented by simple wiring of byte data without any transistors. From the 2nd cycle to the 17th cycle, the data stored in $\{R_{20}, \ldots, R_{27}\}$ are input to the round function, and its output is re-stored into $\{R_{20}, \ldots, R_{27}\}$ through BD after being XORed with the data fed from the message scheduling block and $CONC$. The data fed from the message scheduling block pass through the data rotating function $PROTL$ at the cycle of odd order and $PROTR$ at the cycle of even order, respectively. At the 18th cycle, the output of the round function are XORed with the data fed from the message scheduling block and the chaining value stored in $\{C_{00}, \ldots, C_{07}\}$, and then re-stored into $\{C_{00}, \ldots, C_{07}\}$. The 128-bit XOR gates required for updating $\{C_{01}, C_{03}, C_{05}, C_{07}\}$ can be merged with those for $CONC$ by appending a 128-bit 2:1 selector.

AURORA-256 Type-H2

AURORA-256 Type-H2 implementation processes a round of the AURORA architecture both in one of MSM and in CPM simultaneously in two clock cycles, when the left 128-bit data are processed first. It requires 4 F-function circuits and takes 36 cycles for both CF and FF. Fig. 5.5 shows the data path architecture of AURORA-256 Type-H2 implementation, where the data path width is 128 bits. A 512-bit message block is input in 128-bit blocks using 4 cycles. $PROTL_H$ and $PROTR_H$ in the figure show the functions whose input and output are the left 128-bit of the input and output of the data rotating function $PROTL$ and $PROTR$, respectively. The number of F-functions and XOR gates are reduced to half compared to those in Type-H1 implementation. The pipeline architecture is introduced into the message scheduling block in order to reduce the

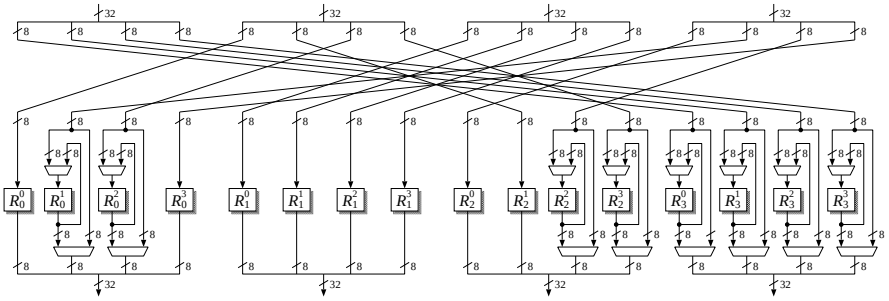

Figure 5.6: 128-bit Byte Diffusion (BD) circuit.

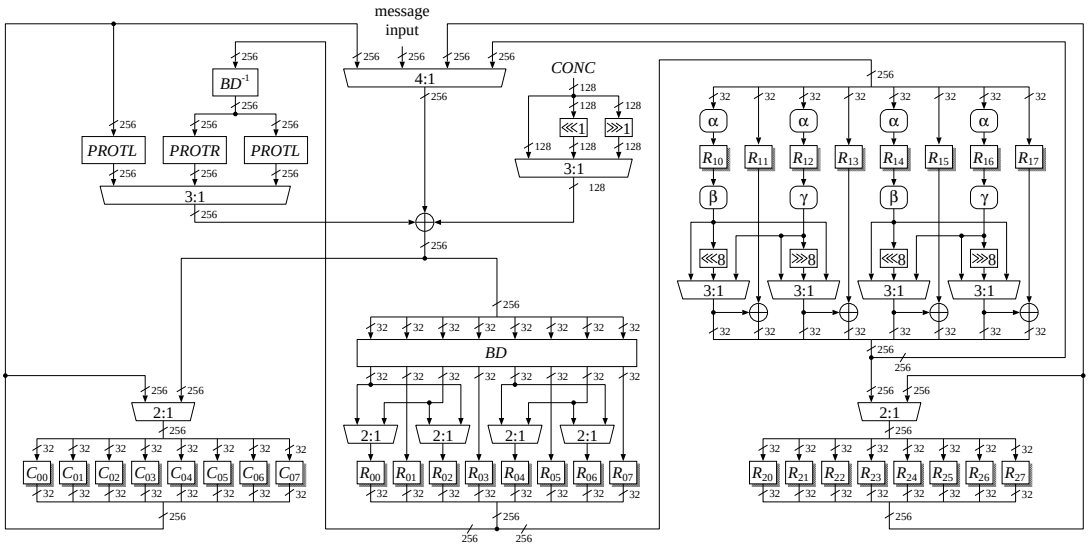

Figure 5.7: Data path architecture of AURORA-256 Type-H3 implementation.

rate of increase in area of the message scheduling at high operating frequency.

In a 128-bit data path architecture such as Type-H2 implementation, the byte diffusion function BD cannot be implemented only by simple wiring of byte data; generally it requires a 256-bit 2:1 selector. In our implementations, we utilize the 128-bit byte diffusion (BD) circuit, as shown in Fig. 5.6. The 128-bit BD circuit consists of byte wiring, sixteen 8-bit registers and sixteen 8-bit 2:1 selectors, where selectors of 128 bits can be reduced. The 256-bit data, which are input into the 128-bit BD circuit in two clock cycles, are output in the order corresponding to BD by controlling selectors.

AURORA-256 Type-H3

AURORA-256 Type-H3 implementation processes a round of the AURORA architecture either in one of MSM or in CPM mutually in every one clock cycle. It requires 4 F-function circuits and takes 36 cycles for both CF and FF. Fig. 5.7 shows the data path architecture of AURORA-256 Type-H3 implementation. Unlike AURORA-256 Type-H1 and Type-H2 implementation, the round function circuit is shared for MSM and CPM. The round function is processed by repeating the following order:

$$MS_L\ (MSF_L) \rightarrow CP\ (CPF) \rightarrow MS_R\ (MSF_R) \rightarrow CP\ (CPF) \rightarrow \cdots$$

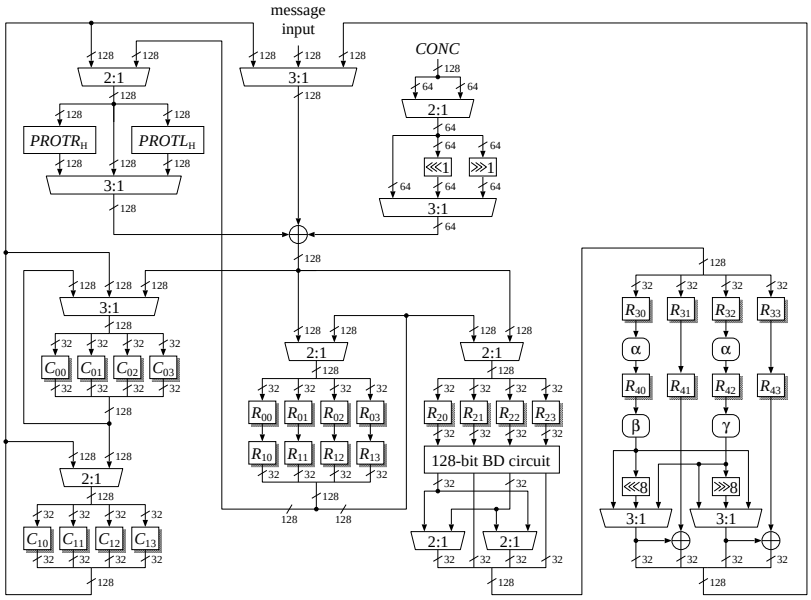

Figure 5.8: Data path architecture of AURORA-256 Type-H4 implementation.

We can shorten the critical path of the whole circuit and improve the maximum operating frequency by applying the pipeline architecture into the round function circuit.

The left 256-bit M_L of a 512-bit message block is input at the 1st cycle, and then 256-bit intermediate values of MS_L (MSF_L) are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$, $\{R_{10}, \ldots, R_{17}\}$ or $\{R_{20}, \ldots, R_{27}\}$ by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \cdots$$

The right 256-bit M_R of a 512-bit message block is input at the 3rd cycle, and then intermediate values of MS_R (MSF_R) are stored in registers by repeating the same order as MS_L. On the other hand, the chaining value stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded at the 2nd cycle, and then 256-bit intermediate values of CP (CPF) are stored in $\{R_{00}, \ldots, R_{07}\}$ or $\{R_{10}, \ldots, R_{17}\}$ by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \cdots$$

The input and output of F-functions must be adequately selected because either the kind or the positioning of F-functions among MS_L (MSF_L), MS_R (MSF_R) and CP (CPF) is different; for intermediate values of MS_R (MSF_R), the output of F-functions must be byte-rotated to the left or right. For intermediate values of CP (CPF), both of the 1st and 3rd 32-bit line, and the 5th and 7th 32-bit line of the input and output of F-functions must be swapped. Note that the chaining value to be fed forward is XORed into intermediate values of MS_R (MSF_R) through $PROTL$ in advance at the 35th cycle, which can reduce one cycle for updating the chaining value.

AURORA-256 Type-H4

AURORA-256 Type-H4 implementation is hybrid of Type-H2 and Type-H3 implementation; it processes a round of the AURORA architecture either in one of *MSM* or in *CPM* mutually in every two clock cycles. It requires 2 F-function circuits and takes 72 cycles for both *CF* and *FF*. Fig. 5.8 shows the data path architecture of AURORA-256 Type-H4 implementation, where the data path width is 128 bits. The round function circuit is shared for *MSM* and *CPM* in the same way as Type-H3 implementation. The processing order of the round function is also the same as

AURORA-256 Type-H3 implementation, but it requires two clock cycles for each round function. The pipeline architecture is introduced into the round function circuit, which can improve the maximum operating frequency.

The left 256-bit M_L of a 512-bit message block is input in 128-bit blocks at the 1st and 2nd cycle, and then intermediate values of MS_L (MSF_L) are stored in registers by repeating the following order:

$$\{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{20}, \ldots, R_{23}\} \rightarrow$$
$$\text{128-bit BD circuit} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \cdots$$

The right 256-bit M_R of a 512-bit message block is input in 128-bit blocks at the 5th and 6th cycle, and then intermediate values of MS_R (MSF_R) are stored in registers by repeating the same order as MS_L. On the other hand, the chaining value stored in four 32-bit registers $\{C_{10}, \ldots, C_{13}\}$ and $\{C_{00}, \ldots, C_{03}\}$ is loaded via $\{C_{10}, \ldots, C_{13}\}$ at the 3rd and 4th cycle, and then 256-bit intermediate values of CP (CPF) are stored in registers by repeating the following order:

$$\{R_{20}, \ldots, R_{23}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \cdots$$

Note that the chaining value to be fed forward is XORed into intermediate values of MS_R (MSF_R) in advance at the 69th and 70th cycle, which can reduce two cycles for updating the chaining value.

AURORA-512 Type-H1

AURORA-512 Type-H1 implementation processes a round of the AURORA architecture both in one of the message scheduling module *MSM* and in the two chaining value processing modules *CPM* simultaneously in one clock cycle. It requires 12 F-function circuits and takes 18 cycles for the compression functions CF_s ($0 \leq s \leq 7$), the mixing functions *MF* and the mixing function for finalization *MFF*. The data path architecture of AURORA-512 Type-H1 implementation can be constructed by appending another chaining value processing block to that of AURORA-256 Type-H1 implementation. In addition, two 256-bit paths from the eight 32-bit chaining value registers in both of the two chaining value processing blocks to the message scheduling block must be appended to process *MF* and *MFF*. The two chaining value processing blocks are basically same except constant values and the F-functions circuits; one block arranges the F-function circuits of F_1 and F_0, and the other arranges those of F_3 and F_2.

AURORA-512 Type-H2

AURORA-512 Type-H2 implementation processes a round of the AURORA architecture both in one of *MSM* and in two *CPM* simultaneously in two clock cycles. It requires 6 F-function circuits and takes 36 cycles for CF_s, *MF* and *MFF*. The data path architecture of AURORA-512 Type-H2 implementation can be constructed by appending another chaining value processing block to that of AURORA-256 Type-H2 implementation. In addition, two 128-bit paths from the four 32-bit chaining value registers in both of the two chaining value processing blocks to the message scheduling block must be appended to process *MF* and *MFF*.

AURORA-512 Type-H3

AURORA-512 Type-H3 implementation processes a round of the AURORA architecture either in one of *MSM* or in one of *CPM* mutually in one clock cycle. It requires 4 F-function circuits and takes 56 cycles for CF_s, *MF* and *MFF*: 54 cycle for message scheduling and chaining value processing, and 2 cycles for updating the chaining value. Fig. 5.9 shows the data path architecture of AURORA-512 Type-H3 implementation. The round function is processed by repeating the following order:

$$MS_{L,i} \rightarrow CP_{L,i} \rightarrow CP_{R,i} \rightarrow MS_{R,i} \rightarrow CP_{L,i} \rightarrow CP_{R,i} \rightarrow \cdots$$

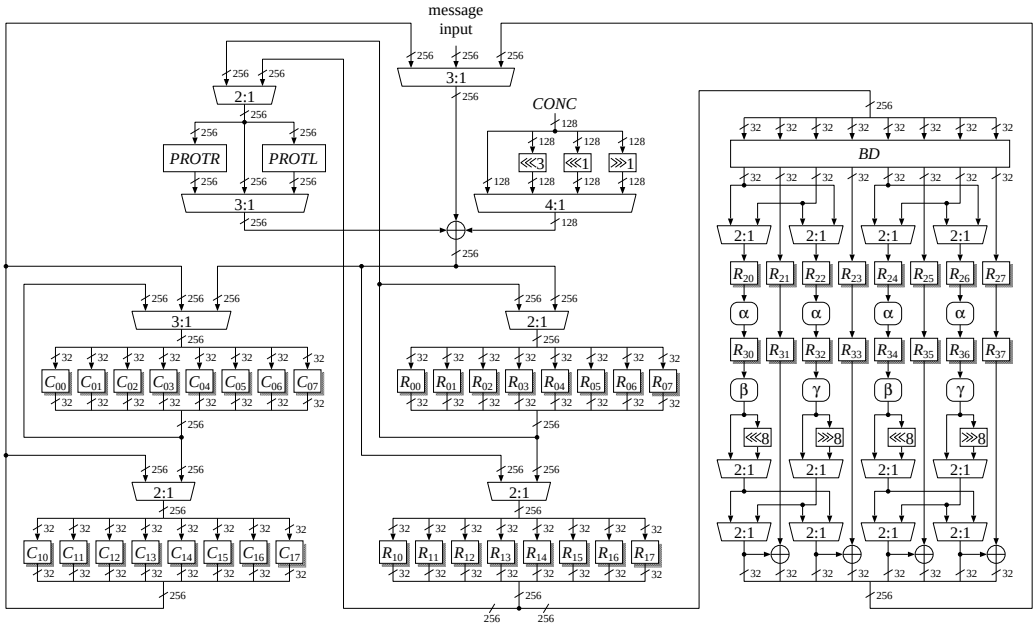

Figure 5.9: Data path architecture of AURORA-512 Type-H3 implementation.

for $0 \leq i \leq 9$. The pipeline architecture is introduced into the round function circuit, which can improve the maximum operating frequency.

For CF_s, the left 256-bit M_L of a 512-bit message block is input at the 1st cycle. For MF (MFF), the chaining value stored in eight 32-bit registers $\{C_{10}, \ldots, C_{17}\}$ is loaded at the 1st cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{L,i}$ are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$, $\{R_{10}, \ldots, R_{17}\}$, $\{R_{20}, \ldots, R_{27}\}$ or $\{R_{30}, \ldots, R_{37}\}$ by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \{R_{30}, \ldots, R_{37}\} \rightarrow \cdots$$

For CF_s, the right 256-bit M_R of a 512-bit message block is input at the 4th cycle. For MF (MFF), the chaining value stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded via $\{C_{10}, \ldots, C_{17}\}$ at the 4th cycle as the input of $MS_{R,8}$ ($MS_{R,9}$). Intermediate values of $MS_{R,i}$ are stored in registers by repeating the same order as $MS_{L,i}$.

On the other hand, the chaining value stored in $\{C_{10}, \ldots, C_{17}\}$ is loaded at the 2nd cycle as the input of $CP_{L,i}$, and then intermediate values of $CP_{L,i}$ are stored in registers by repeating he following order:

$$\{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \{R_{30}, \ldots, R_{37}\} \rightarrow \cdots$$

The chaining value stored in $\{C_{00}, \ldots, C_{07}\}$ is loaded via $\{C_{10}, \ldots, C_{17}\}$ at the 3rd cycle as the input of $CP_{R,i}$, and then intermediate values of $CP_{R,i}$ are stored in registers by repeating the same order as $CP_{L,i}$.

The input and output of F-functions must be adequately selected because the kind or the positioning of F-functions among $MS_{L,i}$, $MS_{R,i}$, $CP_{L,i}$ and $CP_{R,i}$ is different; for intermediate values of $MS_{R,i}$ and $CP_{R,i}$, the output of F-functions must be byte-rotated to the left or right. For intermediate values of $CP_{L,i}$ and $CP_{R,i}$, both of the 1st and 3rd 32-bit line, and the 5th and 7th 32-bit line of the input and output of F-functions must be swapped.

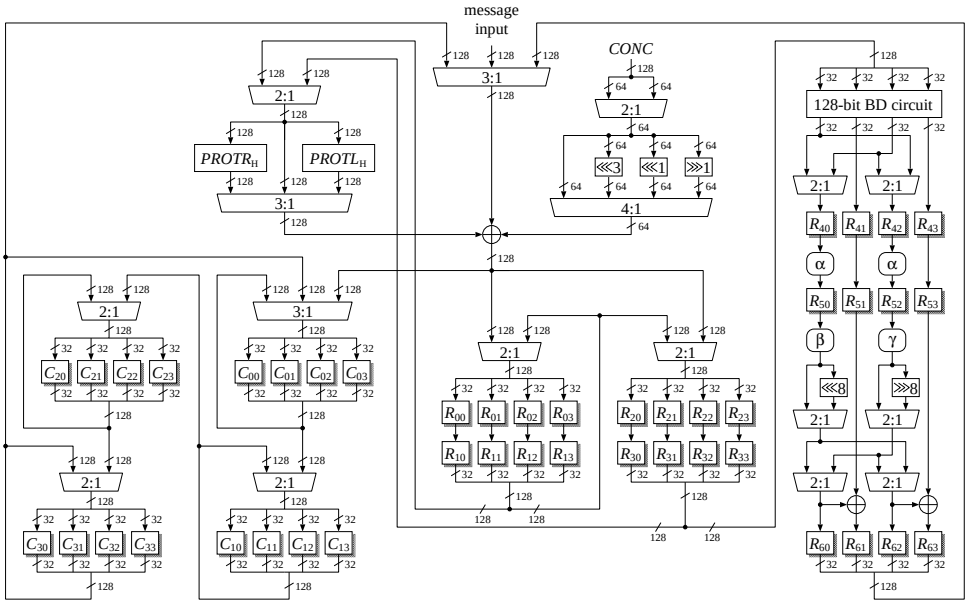

Figure 5.10: Data path architecture of AURORA-512 Type-H4 implementation.

AURORA-512 Type-H4

AURORA-512 Type-H4 implementation processes a round of the AURORA architecture in one of *MSM* or in one of *CPM* mutually in two clock cycle. It requires 2 F-function circuits and takes 112 cycles for CF_s, *MF* and *MFF*: 108 cycle for message scheduling and chaining value processing, and 4 cycles for updating the chaining value. Fig. 5.10 shows the data path architecture of AURORA-512 Type-H4 implementation, where the data path width is 128 bits. The processing order of the round function is the same as AURORA-512 Type-H3 implementation, but it requires two clock cycles for each round function.

For CF_s, the left 256-bit M_L of a 512-bit message block is input in 128-bit blocks at the 1st and 2nd cycle. For *MF* (*MFF*), the chaining value stored in four 32-bit registers $\{C_{30}, \ldots, C_{33}\}$ and $\{C_{20}, \ldots, C_{23}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 1st and 2nd cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{L,i}$ are stored in registers by repeating the following order:

$$\{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{03}\} \rightarrow$$
$$\{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{20}, \ldots, R_{23}\} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow$$
$$\{R_{50}, \ldots, R_{53}\} \rightarrow \{R_{60}, \ldots, R_{63}\} \rightarrow \cdots$$

For CF_s, the right 256-bit M_R of a 512-bit message block is input at the 7th and 8th cycle. For *MF* (*MFF*), the chaining value stored in four 32-bit registers $\{C_{10}, \ldots, C_{13}\}$ and $\{C_{00}, \ldots, C_{03}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 7th and 8th cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{R,i}$ are stored in registers by repeating the same order as $MS_{L,i}$.

On the other hand, the chaining value stored in $\{C_{30}, \ldots, C_{33}\}$ and $\{C_{20}, \ldots, C_{23}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 3rd and 4th cycle as the input of $CP_{L,i}$, and then intermediate values of $CP_{L,i}$ are stored in registers by repeating the following order:

$$\{R_{20}, \ldots, R_{23}\} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \{R_{50}, \ldots, R_{53}\} \rightarrow$$
$$\{R_{60}, \ldots, R_{63}\} \rightarrow \cdots$$

The chaining value stored in $\{C_{10}, \ldots, C_{13}\}$ and $\{C_{00}, \ldots, C_{03}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 5th and 6th cycle as the input of $CP_{R,i}$, and then intermediate values of $CP_{R,i}$ are stored in registers by repeating the same order as $CP_{L,i}$.

5.2.3 Evaluation Results

We show our evaluation results on hardware performance of AURORA-256 and AURORA-512 at the present. For both AURORA-256 and AURORA-512, Type-H1, Type-H2, Type-H3 and Type-H4 implementations with S-boxes based on inversion over $\mathrm{GF}((2^4)^2)$ are evaluated. In addition, Type-H1 implementation with table-lookup S-boxes is also evaluated in order to achieve higher throughput. Control signals for all selectors and constant values are generated in a controller module which is included in each implementation.

The environment of our hardware design and evaluation is as follows.

Language	Verilog-HDL
Design library	0.13 μm CMOS ASIC library
Simulator	VCS version 2006.06
Logic synthesis	Design Compiler version 2007.03-SP3

One gate is equivalent to a 2-way NAND and speed is evaluated under the worst-case conditions. Table 5.14 represents the evaluation results. For each implementation of AURORA-256 and AURORA-512, two types of circuits are synthesized by specifying either area or speed optimization. In the addition, we investigate the condition to maximize "Efficiency" that indicates "Throughput" per area, which we call efficiency optimization. For AURORA-256 implementations, "Throughput" is defined as follows:

$$\text{Throughput [Mbps]} = \frac{\text{Frequency [MHz]} \times \text{Block Size (512 [bits])}}{\text{Cycles}}.$$

On the other hand, for AURORA-512 implementations, "Throughput" is defined as follows:

$$\text{Throughput [Mbps]} = \frac{\text{Frequency [MHz]} \times \text{Block Size (512 [bits])}}{\text{Cycles}} \times \frac{8}{9},$$

because the mixing functions MF are inserted after every 8 compression functions $CF_0, CF_1, \cdots, CF_7$.

We also show, for comparison, the best known results of hardware performance of SHA-2 using a 0.13 μm CMOS ASIC library by Satoh et al. [49]. The performance of AURORA cannot be directly compared with them because different design libraries and different logic synthesis tools were used. However, AURORA enables a variety of implementations from small-area to high-throughput implementations; for AURORA-256, the smallest area (11,111 gates) is about 3% smaller with about 2.06 times higher efficiency (196.1 Kbps/gate) than that of SHA-224/256 (11,184 gates, 95.4 Kbps/gate), and the highest throughput (10,352 Mbps) is about 4.37 times higher than that of SHA-224/256 (2,370 Mbps). For AURORA-512, the smallest area (14,613 gates) is about 37% smaller with about 30% higher efficiency (81.5 Kbps/gate) than that of SHA-384/512 (23,146 gates, 62.8 Kbps/gate), and the highest throughput (9,132 Mbps) is about 3.14 times higher than that of SHA-224/256 (2,909 Mbps).

The highest efficiency of AURORA-256 (344.3 Kbps/gate) and AURORA-512 (194.9 Kbps/gate) is about 2.23 times and 1.83 times higher than that of SHA-224/256 (154.6 Kbps/gate) and SHA-384/512 (106.6 Kbps/gate), respectively, which indicates that AURORA is highly efficient hash function family in hardware implementation.

Table 5.14: Results on Hardware Performance of AURORA-256 and AURORA-512.

	Data Path Architecture	Cycles	S-box	Area [gates]	Frequency [MHz]	Throughput [Mbps]	Efficiency [Kbps/gate]
AURORA-256 ($0.13\mu m$)	Type-H1	18	$GF((2^4)^2)$	18,883	194.3	5,528	292.7
				24,645	287.9	8,189	332.3
				20,825	252.1	7,171	344.3
			Table	27,854	213.2	6,065	217.7
				35,016	363.9	10,352	295.6
				32,997	345.9	9,838	298.2
	Type-H2	36	$GF((2^4)^2)$	13,446	189.2	2,691	200.1
				17,797	293.9	4,180	234.9
				15,523	266.2	3,786	243.9
	Type-H3	36	$GF((2^4)^2)$	15,173	260.7	3,707	244.3
				23,490	464.3	6,603	281.1
				17,064	360.9	5,132	300.8
	Type-H4	72	$GF((2^4)^2)$	11,111	306.4	2,179	196.1
				14,255	475.3	3,380	237.1
				12,257	423.6	3,012	245.7
SHA-224/256 ($0.13\mu m$) [49]	-	72	-	11,484	154.1	1,096	95.4
				15,329	333.3	2,370	154.6
AURORA-512 ($0.13\mu m$)	Type-H1	18	$GF((2^4)^2)$	29,235	195.5	4,943	169.1
				40,219	285.4	7,217	179.4
				31,746	244.7	6,187	194.9
			Table	42,691	213.2	5,391	126.3
				56,748	361.2	9,132	160.9
				48,337	317.1	8,018	165.9
	Type-H2	36	$GF((2^4)^2)$	20,685	185.7	2,347	113.5
				28,358	286.3	3,619	127.6
				22,731	244.7	3,093	136.1
	Type-H3	56	$GF((2^4)^2)$	19,335	236.4	1,921	99.4
				25,915	455.8	3,705	143.0
				22,129	406.2	3,302	149.2
	Type-H4	112	$GF((2^4)^2)$	14,613	293.1	1,191	81.5
				16,969	504.2	2,049	120.7
				16,670	496.7	2,018	121.1
SHA-384/512 ($0.13\mu m$) [49]	-	88	-	23,146	125.0	1,455	62.8
				27,297	250.0	2,909	106.6

For each implementation, the 1st row and the 2nd row show the results of the synthesized circuits by area and speed optimization, respectively. The 3rd row also shows the results by efficiency optimization for each implementation of AURORA-256 and AURORA-512.

Chapter 6

Application of AURORA

6.1 Digital Signature

The digital signature standard (DSS) is specified in FIPS 186-2 [20]. In this standard, the hash function SHA-1 specified in FIPS 180-1 (FIPS 180-3) is used in many occasions including the generation of a message digest, the generation and the verification of parameters [19]. Due to that the same hash size of SHA-1 is not supported by AURORA hash algorithm family, it is not possible to replace SHA-1 as a member of AURORA directly. However, if we want to use a 160-bit output hash function, an appropriate truncation function may be applied to AURORA hash function.

Moreover, there is a draft of the digital signature standard which is available as FIPS 186-3 [21]. In the draft, usages of SHA-2 algorithm family are specified. Thus, our AURORA algorithm can be used as a replacement of corresponding SHA-2 algorithm which has the same hash size.

6.2 Keyed-Hash Message Authentication Code (HMAC)

In FIPS 198, the keyed hash message authentication code (HMAC) is standardized [23]. From the definition of HMAC that any hash function can be applicable in principle, any algorithm of AURORA family can be used as a base hash function for it. The output length L and the block length B should be selected according to the specification of a considered hash function. Table 6.1 summarizes the actual values of L and B for each AURORA hash algorithm.

6.3 Key Establishment Schemes Using Discrete Logarithm Cryptography

The pair-wise key establishment schemes using discrete logarithm cryptography is described in NIST SP800-56A [40]. In this document, minimum bit length of the hash function output is

Table 6.1: The values of L and B.

Algorithm	L	B
AURORA-224	224	512
AURORA-256	256	512
AURORA-384	384	512
AURORA-512	512	512
AURORA-224M	224	512
AURORA-256M	256	512

assigned according to the selected parameter set on of FA,FB,FC, EA, EB, EC, ED, and EE. Among them FB and EB require 224-bit output, FC and EC require 256-bit output. ED and EE require 384-bit and 512-bit output, respectively. AURORA algorithms can be used when one of the above domain parameters is selected. To be concrete, AURORA algorithm is used as a hash function H in the concatenation key derivation function or the ASN.1 key derivation function use a hash function in the document.

6.4 Random Number Generation Using Deterministic Random Bit Generators

NIST SP800-90 specifies the recommendation for random number generation using deterministic random generators (DRBG) [41]. There are three DRBGs that use a secure hash function. HMAC_DRBG uses the aforementioned HMAC scheme, thus AURORA algorithms can be applied by following the rule of the HMAC. Hash_DRBG and Dual_EC_DRBG employ a derivation function using a hash function called Hash_df which call one of SHA-1 and four SHA-2 algorithms. Accordingly, one of AURORA algorithms can be used as a replacement for one of SHA-2 algorithm called in Hash_df. It may be helpful to note that the seed length for Hash_DRBG is 440-bit when using AURORA-384 and AURORA-512 , on the other hand the seed length is 888-bit when using SHA-384 and SHA-512. This is due to the block length for these AURORA algorithms are 512-bit not 1024-bit. However this is consistent with the specification because it is required that minimum entropy for seed and reseed are 192-bit and 256-bit for AURORA-384 and AURORA-512, respectively. The specified seed length 440-bit apparently exceeds these minimum required entropy.

Chapter 7

Advantages and Limitations

The hash function family AURORA has the following advantages and limitations. The following advantages are the realization of the design goal of AURORA family. We believe that the fact that all advantages are achieved in one hash function family draw a line between AURORA and other hash functions.

- **High and Well-balanced Performance on Variety of Platforms**
  To meet the requirements of SHA-3 announced by NIST [38], we defined one of our design goal of a new hash function family that the new hash functions must achieve good performance on a variety of platforms including software for desktop PCs, servers, micro processors and hardware implementations for ASIC and FPGAs. This design goal was also demanded in the AES competition, and finally selected algorithm Rijndael actually satisfied the design goal [14]. The consequences of the design goal can be found in the selected components such as S-box, matrices, byte oriented architecture, reuse of common structure. As a result, we confirmed that AURORA's performance on a variety of platforms is competitive with other known hash functions. On the other hand there is limitation due to such the design goal of AURORA. It is possible to design a hash function which is very fast when it is implemented only on a specific platform by scarifying the well-balanced performance on multi platform implementations. But as explained above, we did not aim for the excellent performance only on specific platform.

- **Sufficient Security Arguments**
  Moreover, as for the security evaluation, we tried to adopt well-studied components to construct AURORA, otherwise newly developed components are employed if reasonable security arguments are provided for the components. For the AURORA structure, the strength against differential cryptanalysis and impossible differential cryptanalysis can be evaluated in a relatively reasonable way. For the new domain extension Double Mix Merkle-Damgaad (DMMD) transform for the longer output, the expected security proof has been provided.

- **Multicollision Resistance with Low Additional Cost**
  Furthermore, we adopted the DMMD transform to offer multicollision resistance by combining parallel compression functions and mixing functions together. As a result, we can provide AURORA-256M, which is an almost identical hash function with AURORA-512, and additional implementation cost from AURORA-256 is limited. This fact emphases that the AURORA hash function family has good consistency among hash functions in the family.

Acknowledgments

We would like to express our deep appreciation to Asami Mizuno, Satoshi Higano, and Eiji Fujii for their kind support for this hash design project. Thanks also to Kazuya Kamio, Tadaoki Yamamoto and Hiroyuki Abe for evaluating performance of AURORA algorithms. We would also like to thank Koichi Sakumoto for his support for analysis of AURORA.

Bibliography

[1] Kazumaro Aoki and Yu Sasaki. Preimage attacks on one-block MD4 and full-round MD5. In *Workshop Records of Selected Areas in Cryptography – SAC 2008*, pages 82–98, 2008.

[2] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage attacks on 3-pass HAVAL and step-reduced MD5. In *Workshop Records of Selected Areas in Cryptography – SAC 2008*, pages 99–114, 2008.

[3] Paulo. S. L. M. Barreto and Vincent. Rijmen. The Whirlpool hashing function. Primitive submitted to NESSIE, September 2000. Available at http://www.cryptonessie.org/.

[4] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In C. Dwork, editor, *Proceedings of CRYPTO '06*, number 4117 in Lecture Notes in Computer Science, pages 602–619. Springer, 2006.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Proceedings of CRYPTO '96*, number 1109 in Lecture Notes in Computer Science, pages 1–15. Springer, 1996.

[6] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In *IACR ePrint archive 2006/399*, 2006. A preliminary version appears in Xuejia Lai and Kefei Chen, editors, Proceedings of ASIACRYPT 2006, number 4284 in Lecture Notes in Computer Science, pages 299-314. Springer, 2006.

[7] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In J. Stern, editor, *Proceedings of Eurocrypt'99*, number 1592 in Lecture Notes in Computer Science, pages 12–23. Springer, 1999.

[8] Olivier Billet, Matthew J. B. Robshaw, Yannick Seurin, and Yiqun Lisa Yin. Looking back at a new hash function. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *Proceedings of Information Security and Privacy – ACISP 2008*, number 5107 in Lecture Notes in Computer Science, pages 239–253. Springer, 2008.

[9] Alex Biryukov and David Wagner. Slide attack. In L. R. Knudsen, editor, *Proceedings of Fast Software Encryption – FSE'99*, number 1636 in Lecture Notes in Computer Science, pages 245–259. Springer, 1999.

[10] Christophe De Cannière and Christian Rechberger. Preimages for reduced SHA-0 and SHA-1. In David Wagner, editor, *Proceedings of CRYPTO 2008*, number 5157 in Lecture Notes in Computer Science, pages 179–202. Springer, 2008.

[11] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Hugo Krawczyk, editor, *Proceedings of CRYPTO '98*, number 1462 in Lecture Notes in Computer Science, pages 56–71. Springer, 1998.

[12] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Proceedings of CRYPTO 2005*, number 3621 in Lecture Notes in Computer Science, pages 430–448. Springer, 2005.

[13] cycle.h. available at `http://www.fftw.org/cycle.h`.

[14] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard (Information Security and Cryptography)*. Springer, 2002.

[15] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Proceedings of CRYPTO '89*, number 435 in Lecture Notes in Computer Science, pages 416–427. Springer, 1989.

[16] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Proceedings of Theory and Applications of Satisfiability Testing – SAT 2007*, number 4501 in Lecture Notes in Computer Science, pages 377–382. Springer, 2007.

[17] Richard Drews Dean. Formal aspects of mobile code security. Ph.D Dissertation, Princeton University, January 1999.

[18] FIPS PUB 140-2. Security requirements for cryptographic modules. Federal Information Processing Standard (FIPS), May 25 2001.

[19] FIPS PUB 180-3. Secure Hash Standard (SHS). Federal Information Processing Standard, October 2008.

[20] FIPS PUB 186-2. Digital Signature Standard (DSS). Federal Information Processing Standard, January 2000.

[21] FIPS PUB 186-3 Draft. Digital Signature Standard (DSS). Federal Information Processing Standard, March 2006.

[22] FIPS PUB 197. Advanced Encryption Standard. Federal Information Processing Standard (FIPS), November 26 2001.

[23] FIPS PUB 198. The keyed-hash message authentication code (HMAC). Federal Information Processing Standard (FIPS), March 2006.

[24] Michael Gorski, Stefan Lucks, and Thomas Peyrin. Slide attacks on a class of hash functions. In *Proceedings of ASIACRYPT 2008*, Lecture Notes in Computer Science. Springer, to be published.

[25] Shoichi Hirose. Some plausible constructions of double-block-length hash functions. In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006.

[26] Antoine Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In Matthew K. Franklin, editor, *Proceedings of CRYPTO 2004*, number 3152 in Lecture Notes in Computer Science, pages 306–316. Springer, 2004.

[27] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In Serge Vaudenay, editor, *Proceedings of EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[28] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *Proceedings EUROCRYPT 2005*, number 3494 in Lecture Notes in Computer Science, pages 474–490. Springer, 2005.

[29] Mario Lamberger, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Second preimages for SMASH. In Masayuki Abe, editor, *Proceedings of the Cryptographers' Track at the RSA Conference 2007 – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 101–111. Springer, 2007.

[30] Gaëtan Leurent. MD4 is not one-way. In Kaisa Nyberg, editor, *Proceedings of Fast Software Encryption – FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 412–428. Springer, 2008.

[31] Stefan Lucks. A failure-friendly design principle for hash functions. In Bimal K. Roy, editor, *Proceedings of ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[32] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *Proceedings Theory of Cryptography – TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

[33] Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (second) preimage attack on the gost hash function. In Kaisa Nyberg, editor, *Proceedings of FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 224–234. Springer, 2008.

[34] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[35] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Proceedings of CRYPTO '89*, number 435 in Lecture Notes in Computer Science, pages 428–446. Springer, 1989.

[36] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. Confirmation that some hash functions are not collision free. In Ivan Damgård, editor, *Proceedings of EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 1990.

[37] Mridul Nandi. Towards optimal double-length hash functions. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Proceedings of INDOCRYPT 2005*, number 3797 in Lecture Notes in Computer Science, pages 77–89. Springer, 2005.

[38] National Institute of Standards and Technology. Announcement request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Docket No.:070911510-7512-01, 2007. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[39] NIST Special Publication 800-106. Draft randomized hashing digital signatures (2nd draft). National Institute of Standards and Technology, August 2008.

[40] NIST Special Publication 800-56A. Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised). National Institute of Standards and Technology, March 2007.

[41] NIST Special Publication 800-90. Recommendation for random number generation using deterministic random bit generators (revised). National Institute of Standards and Technology, March 2007.

[42] Brian Gladman's Home Page. `http://fp.gladman.plus.com/cryptography_technology/sha/sha2-07-01-07.zip`.

[43] The MiniSat Page. `http://minisat.se/`.

[44] Thomas Peyrin. Cryptanalysis of Grindahl. In Kaoru Kurosawa, editor, *Proceedings of ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer, 2007.

[45] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Douglas R. Stinson, editor, *Proceedings of CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

[46] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In Ç. Koç, D. Naccache, and C. Paar, editors, *Proceedings of Cryptographic Hardware and Embedded Systems – CHES 2001*, number 2162 in Lecture Notes in Computer Science, pages 171–184. Springer, 2001.

[47] Yu Sasaki and Kazumaro Aoki. Preimage attacks on MD, HAVAL, SHA, and others. CRYPTO2008 rump session, 2008. `http://rump2008.cr.yp.to/efa237568f229268803b82ed02e217ca.pdf`.

[48] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Password recovery on challenge and response: Impossible differential attack on hash function. In Serge Vaudenay, editor, *Proceedings of AFRICACRYPT 2008*, number 5023 in Lecture Notes in Computer Science, pages 290–307. Springer, 2008.

[49] Akashi Satoh and Tadanobu Inoue. ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS. Integration, the VLSI Journal, 40(1):3–10, 2007.

[50] Taizo Shirai and Kiyomichi Araki. On generalized Feistel structures using the diffusion switching mechanism. *IEICE. Trans. Fundamentals*, E91A(8):2120–2129, 2008.

[51] Taizo Shirai and Bart Preneel. On Feistel ciphers using optimal diffusion mappings across multiple rounds. In Pil Joong Lee, editor, *Proceedings of Asiacrypt'04*, number 3329 in Lecture Notes in Computer Science, pages 1–15. Springer, 2004.

[52] Taizo Shirai and Kyoji Shibutani. Improving immunity of Feistel ciphers against differential cryptanalysis by using multiple MDS matrices. In Bimal Roy and Willi Meier, editors, *Proceedings of Fast Software Encryption – FSE'04*, number 3017 in Lecture Notes in Computer Science, pages 260–278. Springer, 2004.

[53] Taizo Shirai and Kyoji Shibutani. On Feistel structures using a diffusion switching mechanism. In M.J.B. Robshaw, editor, *Proceedings of Fast Software Encryption – FSE'06*, number 4047 in Lecture Notes in Computer Science, pages 41–56. Springer, 2006.

[54] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA. In A. Biryukov, editor, *Proceedings of Fast Software Encryption – FSE'07*, number 4593 in Lecture Notes in Computer Science, pages 181–195. Springer, 2007.

[55] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Proceedings of CRYPTO'05*, number 3621 in Lecture Notes in Computer Science, pages 17–36. Springer, 2005.

[56] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Proceedings of EUROCRYPT'05*, number 3494 in Lecture Notes in Computer Science, pages 19–35. Springer, 2005.

[57] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attack on SHA-0. In Victor Shoup, editor, *Proceedings of CRYPTO'05*, number 3621 in Lecture Notes in Computer Science, pages 1–16. Springer, 2005.

[58] Hongbo Yu, Gaoli Wang, Guoyan Zhang, and Xiaoyun Wang. The second-preimage attack on MD4. In Yvo Desmedt, Huaxiong Wang, Yi Mu, and Yongqing Li, editors, *Proceedings of Cryptology and Network Security – CANS 2005*, number 3810 in Lecture Notes in Computer Science, pages 1–12. Springer, 2005.