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]

July 10, 2009

Executive Summary

We present a hash function family AURORA* which is a revised version of AURORA[1]. The hash function family AURORA* consists of the four algorithms: AURORA-224, AURORA-256, AURORA-384 version 2, AURORA-512 version 2. There is no change in the algorithms of AURORA-224 and AURORA-256.

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 version 2 and AURORA-512 version 2 employ a double block-length (DBL) construction proposed by Hirose. The Hirose's DBL uses two blockciphers in a compression function. This DBL construction enables to make an efficient collision-resistant hash function. Furthermore, the compression function achieves further efficiency by sharing the message scheduling of two underlying blockciphers.

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 version 2 achieves 37.8 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 8.9 Kgates in an area-optimized implementation. In a speed-optimized implementation, AURORA-256 achieves the highest throughput of 10.4 Gbps. For AURORA-512 version 2, the smallest size is 12.4 Kgates and the highest throughput is 6.9 Gbps.

These good performance both in hardware and in software in a single hash function family which is based on the above design techniques makes a clear distinction between the AURORA* family and the SHA-2 family.

[1]The previous version AURORA is a first round candidate for a new cryptographic hash algorithm (SHA-3) family.

Contents

Chapter 1

Introduction

This document describes the algorithm specifications and supporting documentation including design rationale, security, efficient implementation, applications, advantages and limitations of the hash function family AURORA* which is a revised version of AURORA.

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 [39]. 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 as far as possible.
  - 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 too 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 [39] "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 arguments 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 version 2	$512 \times (2^{64} - 1)$	512	512	384
AURORA-512 version 2	$512 \times (2^{64} - 1)$	512	512	512

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 version 2, and AURORA-512 version 2 supporting hash sizes of 224, 256, 384, and 512 bits, respectively. 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 [37, 15]. The compression function is designed based on the well-established design techniques for blockciphers, and uses the Davies-Meyer construction [38, 46, 17]. 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 version 2 and AURORA-512 version 2 employ a double block-length (DBL) construction proposed by Hirose. The Hirose's DBL uses two blockciphers in a compression function [26, 27]. This DBL construction enables to make an efficient collision-resistant hash function. Furthermore, the compression function achieves further efficiency by sharing the message scheduling of two underlying blockciphers.

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 version 2 achieves 37.8 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 version 2, the smallest size is 12.4 Kgates and the highest throughput is 6.9 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 limitations 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.
BD	The Byte Diffusion 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.

CPM	The Chaining value Processing Module.
DR	The Data Rotating function.
CF	The Compression Function for AURORA-224/256.
FF	The Finalization Function for AURORA-224/256.
MS_L and MS_R	The Message Scheduling functions for CF.
MSF_L and MSF_R	The Message Scheduling functions for FF.
CP	The Chaining value Processing function for CF.
CPF	The Chaining value Processing function 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 version 2.

CPM^{512}	The Chaining value Processing Module.
DR^{512}	The Data Rotating function.
$PROTX$	The eXtra Partial ROTating function.
CF^{512}	The Compression Functions for AURORA-384/512 version 2.
FF^{512}	The Finalization Function for AURORA-384/512 version 2.
MS_L^{512}, MS_R^{512} and MS_X^{512}	The Message Scheduling functions for CF^{512}.
MSF_L^{512}, MSF_R^{512} and MSF_X^{512}	The Message Scheduling functions for FF^{512}.
CP_L^{512} and CP_R^{512}	The Chaining value Processing function for CF^{512}.
CPF_L^{512} and CPF_R^{512}	The Chaining value Processing function for FF^{512}.
$CONM_{L,j}^{512}$, $CONM_{R,j}^{512}$ and $CONM_{X,j}^{512}$	The CONstants for MS_L^{512}, MS_R^{512}, MS_X^{512}, MSF_L^{512}, MSF_R^{512} and MSF_X^{512}.
$CONC_{L,j}^{512}$ and $CONC_{R,j}^{512}$	The CONstant for CP_L^{512}, CP_R^{512}, CPF_L^{512} and CPF_R^{512}.

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. 2.2.4. *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. 2.12 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 $\{W_{j\,(32)}\}_{0\le j<68}$.

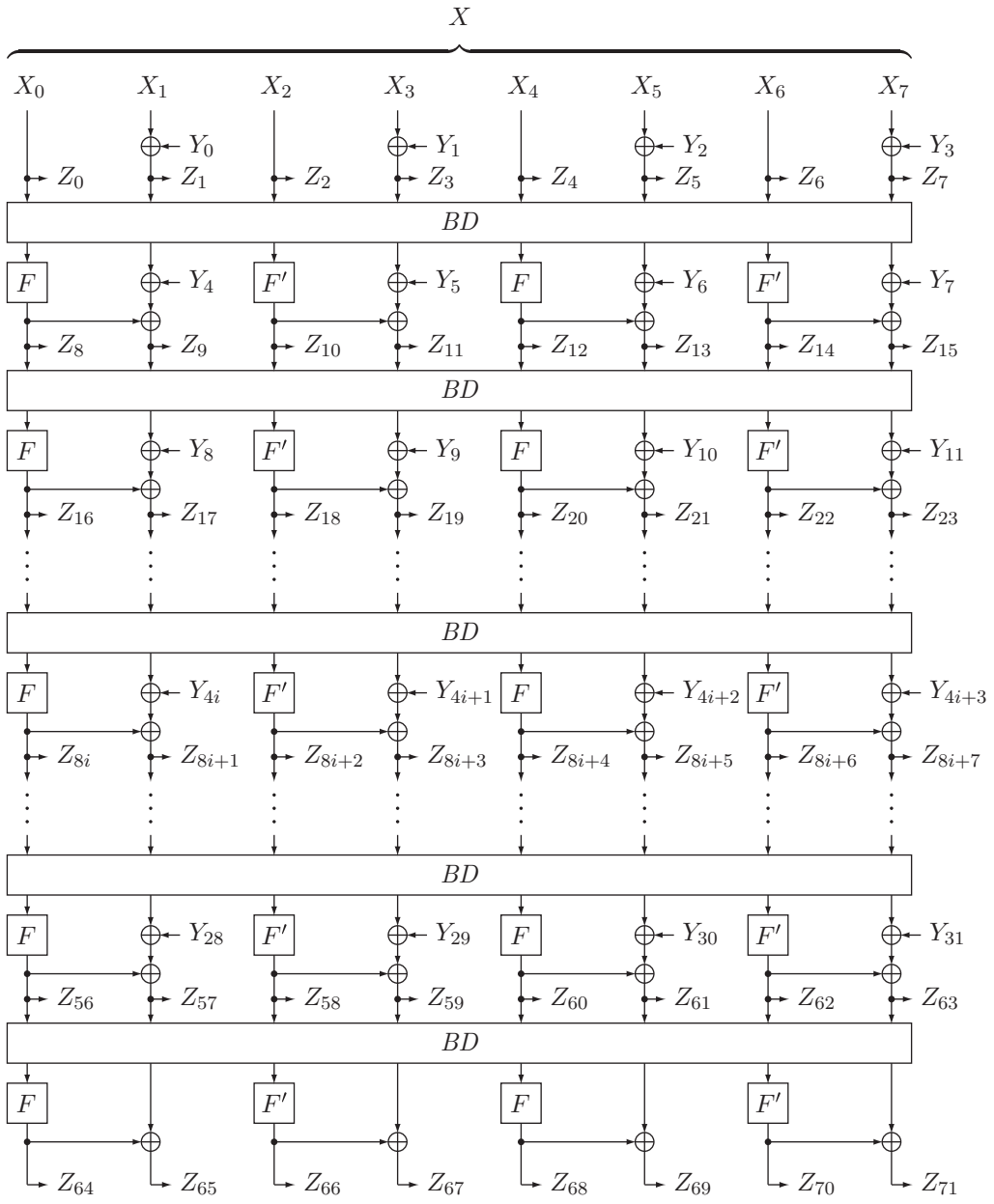

Figure 2.1: $\{Z_{j\,(32)}\}_{0\le j<72} \leftarrow MSM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0\le j<32})$.

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. 2.2.4. 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. 2.13 for a pseudocode.

2.2.3 Chaining Value Processing Module: CPM^{512}

The chaining value processing module, CPM^{512}, takes the following three inputs;

- a bit string $X_{(256)}$,
- a set of bit strings $\{Y_{j\,(32)}\}_{0 \le j < 216}$, and
- a set of bit strings $\{W_{j\,(32)}\}_{0 \le j < 104}$.

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

CPM^{512} internally uses a byte diffusion function *BD*, which is also used in *MSM*, and is defined in Sec. 2.2.4. As with *MSM*, CPM^{512} is parameterized by two functions F and F' over $\{0,1\}^{32}$, and we write $CPM^{512}[F, F']$ when we use functions F and F'.

We now describe the specification of $CPM^{512}[F, F']$.

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. (25 round iterations) The following operations are iterated for $i = 1$ to 25.

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

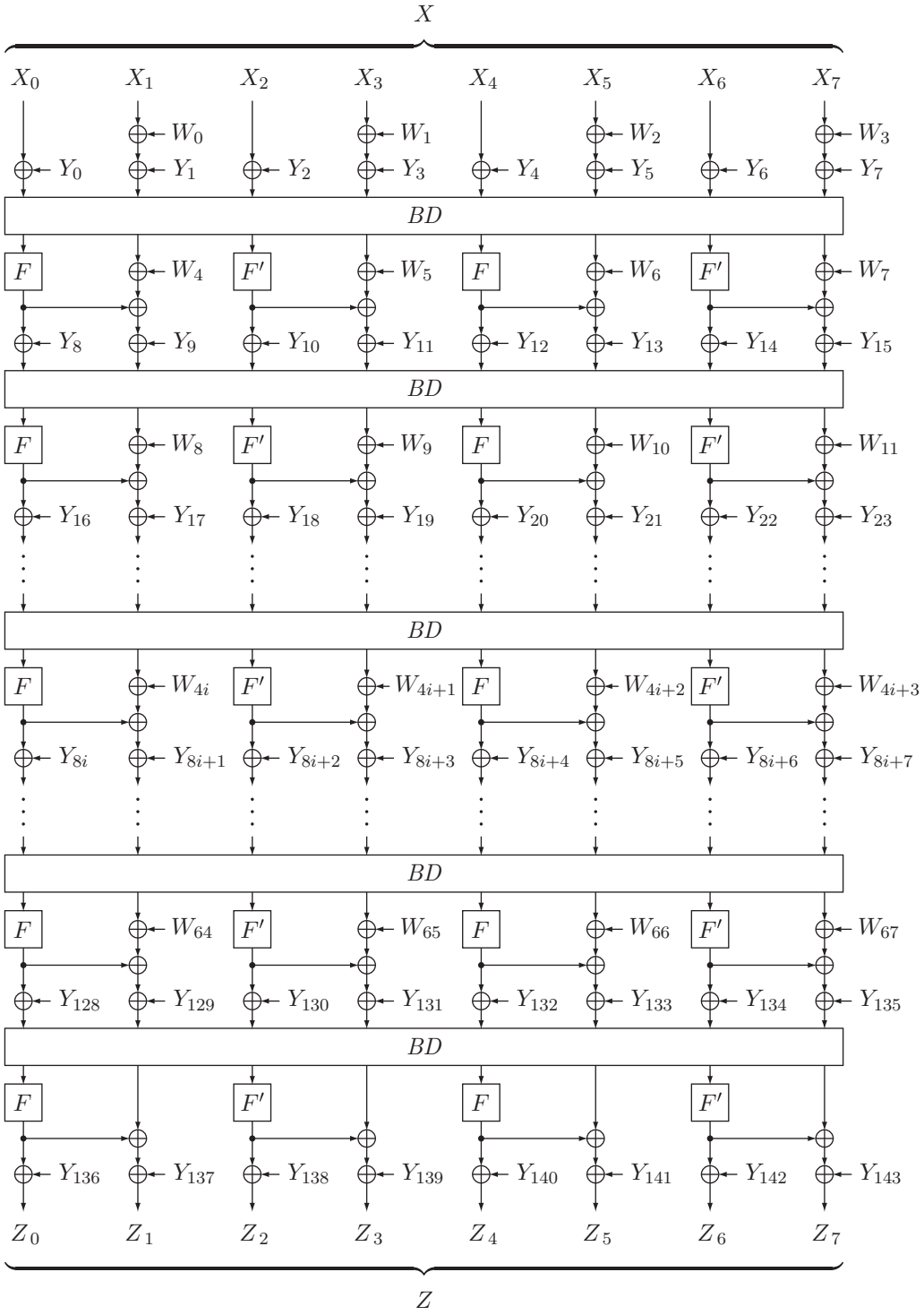

Figure 2.2: $Z_{(256)} \leftarrow CPM[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0 \le j < 144}, \{W_{j\,(32)}\}_{0 \le j < 68})$.

Step 5. (26-th round) Then the following operations are executed.

$$\begin{cases} (X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7) \\ (X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\ (X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\ (X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{208}, Y_{209}, \ldots, Y_{215}) \end{cases}$$

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

See Fig. 2.3 for an illustration and Fig. 2.14 for a pseudocode.

2.2.4 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. 2.4, 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. 2.5 for an illustration and Fig. 2.15 for a pseudocode.

2.2.5 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 $S : \{0,1\}^8 \to \{0,1\}^8$, and four 4×4 matrices, $\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$.

- The S-box $S : x_{(8)} \to y_{(8)}$ 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 $z'^4 + z' + 1$.

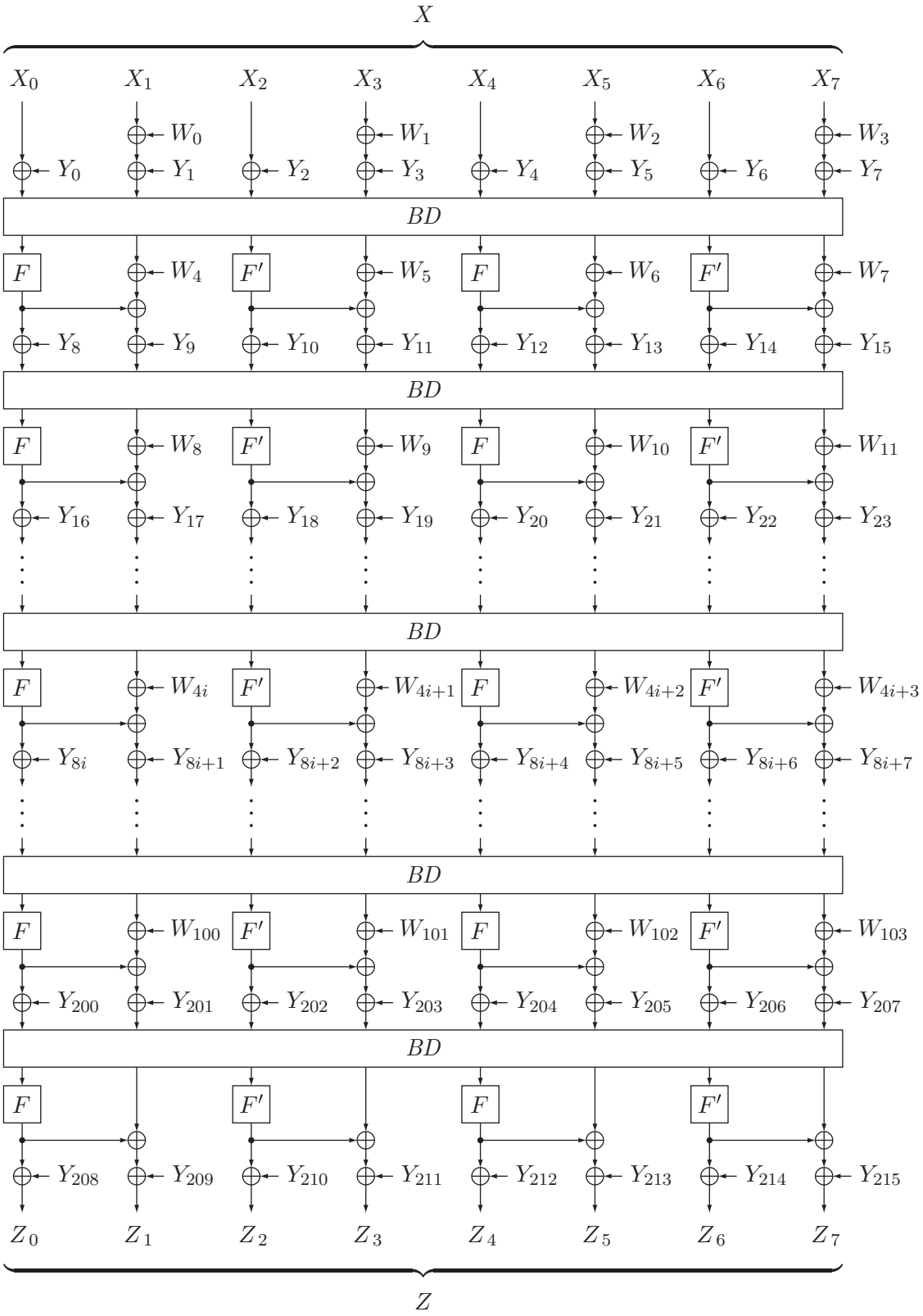

Figure 2.3: $Z_{(256)} \leftarrow CPM^{512}[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0 \le j < 216}, \{W_{j\,(32)}\}_{0 \le j < 104})$.

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

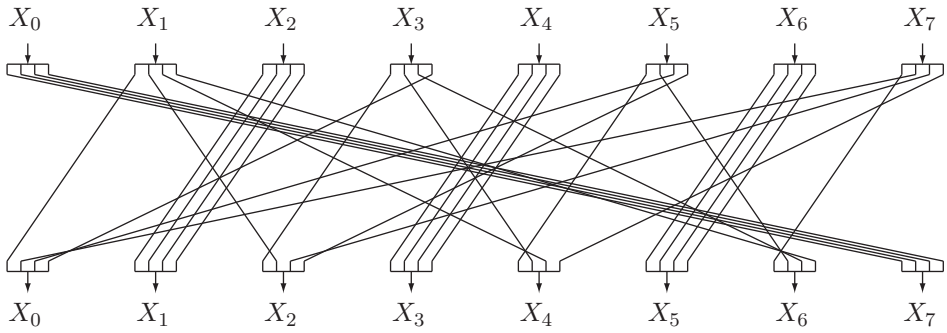

Figure 2.5: $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) \leftarrow BD(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$.

Moreover, $f : x_{(8)} \rightarrow y_{(8)}$ and $g : x_{(8)} \rightarrow y_{(8)}$ are affine transformations over GF(2), which are defined as

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

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.

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

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

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

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

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

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

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

2.2.6 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. 2.6 for an illustration and Fig. 2.16 for a pseudocode.

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

Figure 2.6: $\{Z_{j\,(32)}\}_{0\le j<144} \leftarrow DR(\{X_{j\,(32)}\}_{0\le j<72}, \{Y_{j\,(32)}\}_{0\le j<72})$.

2.2.7 Data Rotating Function: DR^{512}

The data rotating function, DR^{512}, takes the following three inputs;

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

The output is a set of bit strings $\{Z_{j\,(32)}\}_{0\le j<216}$.

DR^{512} uses the following three functions;

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

$PROTL$ and $PROTR$ are defined as (2.8) and (2.9), and $PROTX$ is defined as

$$PROTX(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)}) = (X'_{0\,(32)}, X'_{1\,(32)}, \ldots, X'_{7\,(32)}), \tag{2.10}$$

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

Now DR^{512} works as follows.

Step 1. For inputs $\{X_{j\,(32)}\}_{0\le j<72}$, $\{Y_{j\,(32)}\}_{0\le j<72}$ and $\{W_{j\,(32)}\}_{0\le j<72}$, we define $\{Z_{j\,(32)}\}_{0\le j<216}$ by iterating the following operations for $i = 0$ to 8.

$$\begin{cases} (Z_{24i}, Z_{24i+1}, \ldots, Z_{24i+7}) \leftarrow PROTL(X_{8i}, X_{8i+1}, \ldots, X_{8i+7}) \\ (Z_{24i+8}, Z_{24i+9}, \ldots, Z_{24i+15}) \leftarrow PROTR(Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7}) \\ (Z_{24i+16}, Z_{24i+17}, \ldots, Z_{24i+23}) \leftarrow PROTX(W_{8i}, W_{8i+1}, \ldots, W_{8i+7}) \end{cases}$$

Step 2. The output is $\{Z_{j\,(32)}\}_{0\le j<216}$ defined in the above operations.

See Fig. 2.7 for an illustration and Fig. 2.17 for a pseudocode.

$$\begin{array}{ccccc|ccc|ccccc}
X_0 & X_1 & \cdots & X_7 & \to & (Z_1 \,\|\, Z_3) & \leftarrow & (X_1 \,\|\, X_3) \lll_{64} 1 & \to & Z_0 & Z_1 & \cdots & Z_7 \\
Y_0 & Y_1 & \cdots & Y_7 & \to & (Z_9 \,\|\, Z_{11}) & \leftarrow & (Y_1 \,\|\, Y_3) \ggg_{64} 1 & \to & Z_8 & Z_9 & \cdots & Z_{15} \\
W_0 & W_1 & \cdots & W_7 & \to & (Z_{17} \,\|\, Z_{19}) & \leftarrow & (W_1 \,\|\, W_3) \lll_{64} 2 & \to & Z_{16} & Z_{17} & \cdots & Z_{23} \\
X_8 & X_9 & \cdots & X_{15} & \to & (Z_{25} \,\|\, Z_{27}) & \leftarrow & (X_9 \,\|\, X_{11}) \lll_{64} 1 & \to & Z_{24} & Z_{25} & \cdots & Z_{31} \\
Y_8 & Y_9 & \cdots & Y_{15} & \to & (Z_{33} \,\|\, Z_{35}) & \leftarrow & (Y_9 \,\|\, Y_{11}) \ggg_{64} 1 & \to & Z_{32} & Z_{33} & \cdots & Z_{39} \\
W_8 & W_9 & \cdots & W_{15} & \to & (Z_{41} \,\|\, Z_{43}) & \leftarrow & (W_9 \,\|\, W_{11}) \lll_{64} 2 & \to & Z_{40} & Z_{41} & \cdots & Z_{47} \\
\vdots & \vdots & & \vdots & & & \vdots & & & \vdots & \vdots & & \vdots \\
X_{64} & X_{65} & \cdots & X_{71} & \to & (Z_{193} \,\|\, Z_{195}) & \leftarrow & (X_{65} \,\|\, X_{67}) \lll_{64} 1 & \to & Z_{192} & Z_{193} & \cdots & Z_{199} \\
Y_{64} & Y_{65} & \cdots & Y_{71} & \to & (Z_{201} \,\|\, Z_{203}) & \leftarrow & (Y_{65} \,\|\, Y_{67}) \ggg_{64} 1 & \to & Z_{200} & Z_{201} & \cdots & Z_{207} \\
W_{64} & W_{65} & \cdots & W_{71} & \to & (Z_{209} \,\|\, Z_{211}) & \leftarrow & (W_{65} \,\|\, W_{67}) \lll_{64} 2 & \to & Z_{208} & Z_{209} & \cdots & Z_{215}
\end{array}$$

Figure 2.7: $\{Z_{j\,(32)}\}_{0\le j<216} \leftarrow DR^{512}(\{X_{j\,(32)}\}_{0\le j<72}, \{Y_{j\,(32)}\}_{0\le j<72}, \{W_{j\,(32)}\}_{0\le j<72})$.

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 $Pad(\cdot)$;

$$Pad(M) = M \,\|\, 1 \,\|\, 0^b \,\|\, Len_{64}, \tag{2.11}$$

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. 2.8 for an illustration and Fig. 2.18 for a pseudocode.

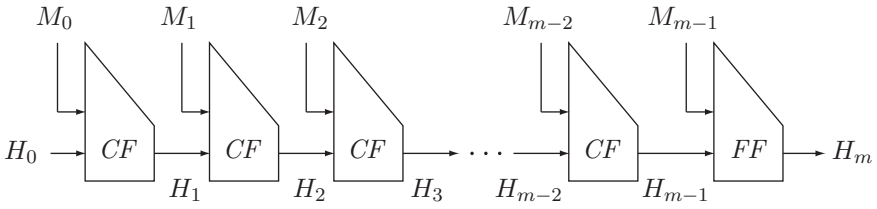

Figure 2.8: AURORA-256.

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

$$MS_L(X) = MSM[F_0, F_1](X, \{CONM_{L,j\,(32)}\}_{0 \le j < 32}), \tag{2.12}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONM_{L,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.7.

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

$$MS_R(X) = MSM[F_2, F_3](X, \{CONM_{R,j\,(32)}\}_{0 \le j < 32}), \tag{2.13}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{R,j\,(32)}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.7.

- DR is the data rotating function defined in Sec. 2.2.6.

- 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

$$CP(X, Y) = CPM[F_1, F_0](X, Y, \{CONC_{j\,(32)}\}_{0 \le j < 68}), \tag{2.14}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONC_{j\,(32)}\}_{0 \le j < 68}$ is the set of constants defined in Sec. 2.7.

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. 2.9 for an illustration and Fig. 2.19 for a pseudocode.

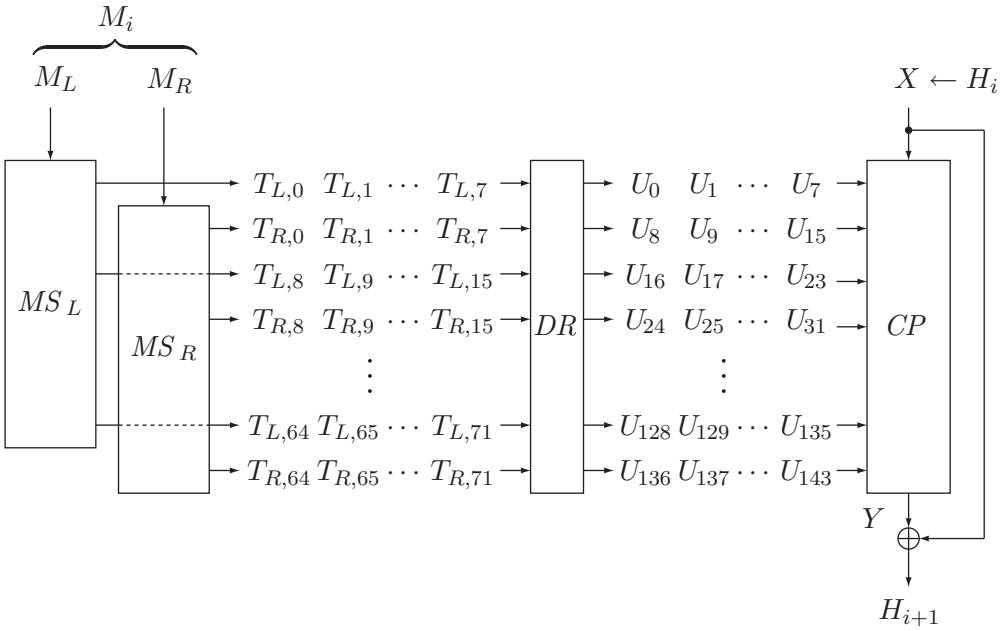

Figure 2.9: $H_{i+1\,(256)} \leftarrow CF(H_{i\,(256)}, M_{i\,(512)})$.

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.

$$\begin{cases} MSF_L(\cdot) : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\ MSF_R(\cdot) : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\ DR(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \rightarrow (\{0,1\}^{32})^{144}, \\ CPF(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} \rightarrow \{0,1\}^{256}. \end{cases} \tag{2.15}$$

These functions are described below.

Components of *FF*

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

$$MSF_L(X) = MSM[F_0, F_1](X, \{CONM_{L,j\,(32)}\}_{32 \le j < 64}), \tag{2.16}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONM_{L,j\,(32)}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.7.

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

$$MSF_R(X) = MSM[F_2, F_3](X, \{CONM_{R,j\,(32)}\}_{32 \le j < 64}), \tag{2.17}$$

  where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{R,j\,(32)}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.7.

- DR is the data rotating function defined in Sec. 2.2.6.
- For any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{144}$, CPF is defined as

$$CPF(X,Y) = CPM[F_1, F_0](X, Y, \{CONC_{j\,(32)}\}_{68 \le j < 136}), \tag{2.18}$$

  where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONC_{j\,(32)}\}_{68 \le j < 136}$ is the set of constants defined in Sec. 2.7.

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. 2.20 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 $RoundC^{(i)}(\cdot)$.

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

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

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

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

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 (2.11), 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. 2.21 for a pseudocode.

2.5 Specification of AURORA-512 version 2

2.5.1 Overall Structure

AURORA-512 version 2 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 a compression function CF^{512} and a finalization function FF^{512}, where

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

The compression function CF^{512} is defined in Sec. 2.5.2 and a finalization function FF^{512} is defined in Sec. 2.5.3.

Now AURORA-512 version 2 works as follows.

Step 1. The input message M is padded with the padding function $Pad(\cdot)$. 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 $H_{0\,(512)} = 0^{512}$, and compute $H_{1\,(512)}, H_{2\,(512)}, \ldots, H_{m-1\,(512)}$ by iterating

$$H_{i+1} \leftarrow CF^{512}(H_i, M_i)$$

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 $H_m \leftarrow FF^{512}(H_{m-1}, M_{m-1})$, and the output is $H_{m\,(512)}$.

See Fig. 2.10 for an illustration and Fig. 2.22 for a pseudocode.

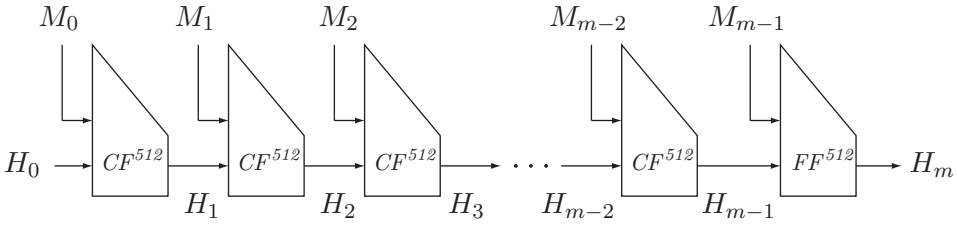

Figure 2.10: AURORA-512 version 2.

2.5.2 Compression Functions: CF^{512}

The compression function, CF^{512}, 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.

It internally uses three message scheduling functions MS_L^{512}, $MS_R^{512}$and MS_X^{512}, a data rotating function DR^{512}, and a chaining value processing functions CP_L^{512} and CP_R^{512}, where

$$\begin{cases} MS_L^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MS_R^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MS_X^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ DR^{512}(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{216}, \\ CP_L^{512}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{216} \to \{0,1\}^{256}, \\ CP_R^{512}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{216} \to \{0,1\}^{256}. \end{cases}$$

These functions are described below.

Components of CF^{512}

- MS_L^{512} is an instance of MSM described in Sec. 2.2.1, and for any $X \in \{0,1\}^{256}$, it is defined as
$$MS_L^{512}(X) = MSM[F_0, F_1](X, \{CONM_{L,j}^{512}\}_{0 \le j < 32}), \tag{2.19}$$
where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONM_{L,j}^{512}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.7.

- Similarly, for any $X \in \{0,1\}^{256}$, MS_R^{512} is defined as
$$MS_R^{512}(X) = MSM[F_2, F_3](X, \{CONM_{R,j}^{512}\}_{0 \le j < 32}), \tag{2.20}$$
where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{R,j}^{512}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.7.

- Also, for any $X \in \{0,1\}^{256}$, MS_X^{512} is defined as
$$MS_X^{512}(X) = MSM[F_0, F_3](X, \{CONM_{X,j}^{512}\}_{0 \le j < 32}), \tag{2.21}$$
where F_0 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{X,j}^{512}\}_{0 \le j < 32}$ is the set of constants defined in Sec. 2.7.

- DR^{512} is the data rotating function defined in Sec. 2.2.7.

- CP_L^{512} is an instance of CPM^{512} described in Sec. 2.2.3, and for any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{216}$, it is defined as
$$CP_L^{512}(X, Y) = CPM^{512}[F_1, F_0](X, Y, \{CONC_{L,j}^{512}\}_{0 \le j < 104}), \tag{2.22}$$
where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONC_{L,j}^{512}\}_{0 \le j < 104}$ is the set of constants defined in Sec. 2.7.

- CP_R^{512} is an instance of CPM^{512} described in Sec. 2.2.3, and for any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{216}$, it is defined as
$$CP_R^{512}(X, Y) = CPM^{512}[F_3, F_2](X, Y, \{CONC_{R,j}^{512}\}_{0 \le j < 104}), \tag{2.23}$$
where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONC_{R,j}^{512}\}_{0 \le j < 104}$ is the set of constants defined in Sec. 2.7.

Specification of CF^{512}

Now we present the specification of CF^{512}.

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

Step 2. Let $\{T_{L,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_L^{512}(M_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_R^{512}(M_{R\,(256)})$.

Step 4. Let $\{T_{X,j\,(32)}\}_{0 \le j < 72} \leftarrow MS_X^{512}(X_{L\,(256)})$.

Step 5. Let $\{U_{j\,(32)}\}_{0 \le j < 216} \leftarrow DR^{512}(\{T_{L,j\,(32)}\}_{0 \le j < 72}, \{T_{R,j\,(32)}\}_{0 \le j < 72}, \{T_{X,j\,(32)}\}_{0 \le j < 72})$.

Step 6. Let $Y_{L(256)} \leftarrow CP_L^{512}(X_{R(256)}, \{U_{j\,(32)}\}_{0 \le j < 216})$.

Step 7. Let $Y_{R(256)} \leftarrow CP_R^{512}(X_{R(256)}, \{U_{j\,(32)}\}_{0 \le j < 216})$.

Step 8. Finally, the output is $H_{i+1\,(512)} \leftarrow (Y_{L(256)} \oplus X_{R(256)} || Y_{R(256)} \oplus X_{R(256)})$.

See Fig. 2.11 for an illustration and Fig. 2.23 for a pseudocode.

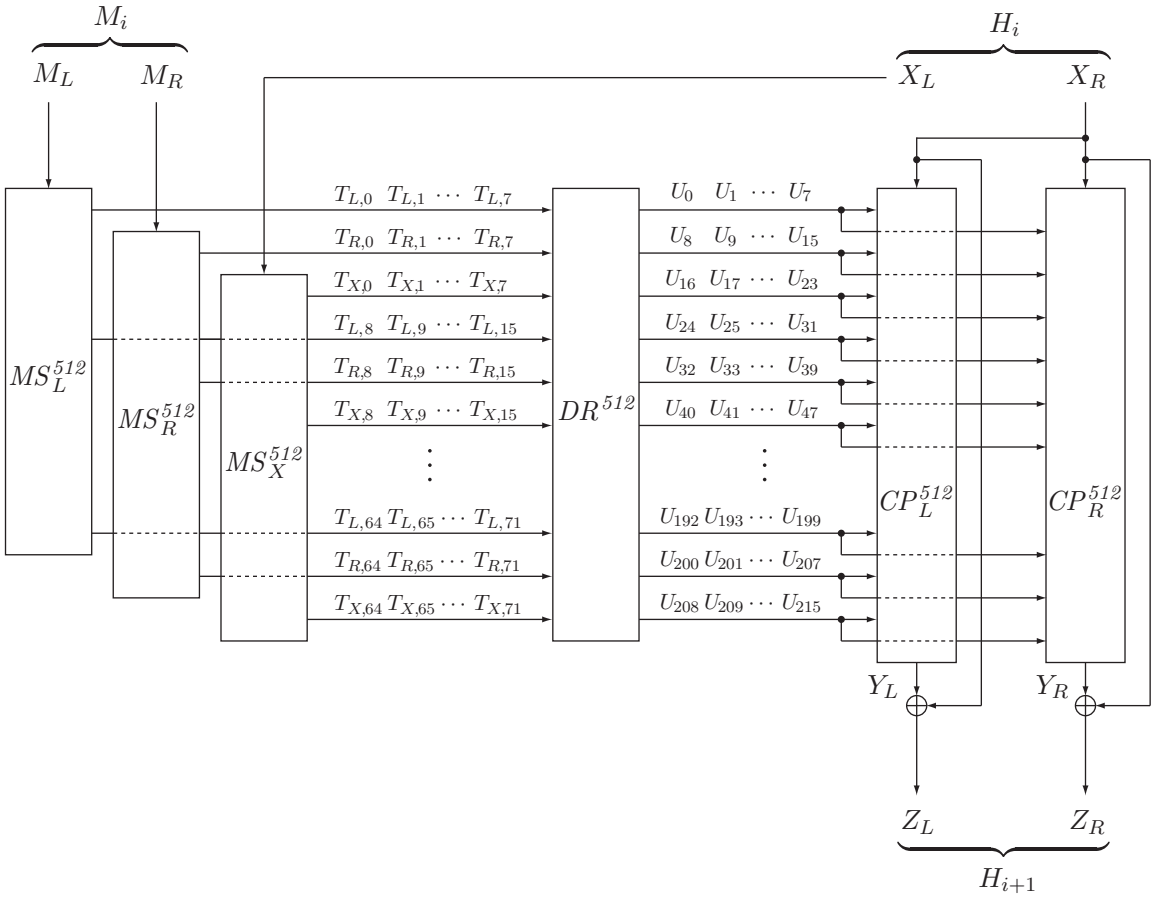

Figure 2.11: $H_{i+1\,(512)} \leftarrow CF^{512}(H_{i\,(512)}, M_{i\,(512)})$.

2.5.3 Finalization Functions: FF^{512}

The finalization function, FF^{512}, takes the chaining value H_{m-1} of 512 bits and the input message block M_{m-1} of 512 bits, and outputs the final hash value H_m of 512 bits.

FF^{512} is structurally equivalent to CF^{512}, and the only difference is the constants used in the components.

It internally uses three message scheduling functions MSF_L^{512}, $MSF_R^{512}$and MSF_X^{512}, a data rotating function DR^{512}, and a chaining value processing functions CPF_L^{512} and CPF_R^{512}, where

$$\begin{cases} MSF_L^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MSF_R^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ MSF_X^{512}(\cdot) : \{0,1\}^{256} \to (\{0,1\}^{32})^{72}, \\ DR^{512}(\cdot,\cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{216}, \\ CPF_L^{512}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{216} \to \{0,1\}^{256}, \\ CPF_R^{512}(\cdot,\cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{216} \to \{0,1\}^{256}. \end{cases}$$

These functions are described below.

Components of FF^{512}

- MSF_L^{512} is an instance of MSM described in Sec. 2.2.1, and for any $X \in \{0,1\}^{256}$, it is defined as
$$MSF_L^{512}(X) = MSM[F_0, F_1](X, \{CONM_{L,j}^{512}\}_{32 \le j < 64}), \tag{2.24}$$
where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONM_{L,j}^{512}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.7.

- Similarly, for any $X \in \{0,1\}^{256}$, MS_R^{512} is defined as
$$MSF_R^{512}(X) = MSM[F_2, F_3](X, \{CONM_{R,j}^{512}\}_{32 \le j < 64}), \tag{2.25}$$
where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{R,j}^{512}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.7.

- Also, for any $X \in \{0,1\}^{256}$, MS_X^{512} is defined as
$$MSF_X^{512}(X) = MSM[F_0, F_3](X, \{CONM_{X,j}^{512}\}_{32 \le j < 64}), \tag{2.26}$$
where F_0 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONM_{X,j}^{512}\}_{32 \le j < 64}$ is the set of constants defined in Sec. 2.7.

- DR^{512} is the data rotating function defined in Sec. 2.2.7.

- CPF_L^{512} is an instance of CPM^{512} described in Sec. 2.2.3, and for any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{216}$, it is defined as
$$CPF_L^{512}(X, Y) = CPM^{512}[F_1, F_0](X, Y, \{CONC_{L,j}^{512}\}_{104 \le j < 208}), \tag{2.27}$$
where F_0 and F_1 are F-functions defined in Sec. 2.2.5, and $\{CONC_{L,j}^{512}\}_{104 \le j < 208}$ is the set of constants defined in Sec. 2.7.

- CPF_R^{512} is an instance of CPM^{512} described in Sec. 2.2.3, and for any $X \in \{0,1\}^{256}$ and $Y \in (\{0,1\}^{32})^{216}$, it is defined as
$$CPF_R^{512}(X, Y) = CPM^{512}[F_3, F_2](X, Y, \{CONC_{R,j}^{512}\}_{104 \le j < 208}), \tag{2.28}$$
where F_2 and F_3 are F-functions defined in Sec. 2.2.5, and $\{CONC_{R,j}^{512}\}_{104 \le j < 208}$ is the set of constants defined in Sec. 2.7.

Specification of FF^{512}

Now we present the specification of FF^{512}.

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

Step 2. Let $\{T_{L,j\,(32)}\}_{0\leq j<72} \leftarrow MSF_L^{512}(M_{L\,(256)})$.

Step 3. Let $\{T_{R,j\,(32)}\}_{0\leq j<72} \leftarrow MSF_R^{512}(M_{R\,(256)})$.

Step 4. Let $\{T_{X,j\,(32)}\}_{0\leq j<72} \leftarrow MSF_X^{512}(X_{L\,(256)})$.

Step 5. Let $\{U_{j\,(32)}\}_{0\leq j<216} \leftarrow DR^{512}(\{T_{L,j\,(32)}\}_{0\leq j<72}, \{T_{R,j\,(32)}\}_{0\leq j<72}, \{T_{X,j\,(32)}\}_{0\leq j<72})$.

Step 6. Let $Y_{L(256)} \leftarrow CPF_L^{512}(X_{R(256)}, \{U_{j\,(32)}\}_{0\leq j<216})$.

Step 7. Let $Y_{R(256)} \leftarrow CPF_R^{512}(X_{R(256)}, \{U_{j\,(32)}\}_{0\leq j<216})$.

Step 8. Finally, the output is $H_{m\,(512)} \leftarrow (Y_{L(256)} \oplus X_{R(256)} || Y_{R(256)} \oplus X_{R(256)})$.

See Fig. 2.24 for a pseudocode.

2.6 Specification of AURORA-384 version 2

AURORA-384 version 2 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^{512}, the finalization function FF^{512} as AURORA-512 version 2 defined in Sec. 2.5.

The difference is that AURORA-384 version 2 uses $H_0 = 1^{512}$ as the initial value, and the output of FF^{512} 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 version 2.

Step 1. The input message M is first padded with $Pad(\cdot)$ in (2.11), 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

$$H_{i+1} \leftarrow CF^{512}(H_i, M_i)$$

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 $H_m \leftarrow FF^{512}(H_{m-1}, M_{m-1})$, and the output is $H'_{m\,(384)} \leftarrow TF_{384}(H_{m\,(512)})$.

See Fig. 2.25 for a pseudocode.

2.7 Constant Values

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

2.7.1 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.

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

2.7.2 Constant Values for AURORA-384/512 version 2

Following constants are used in AURORA-384/512 version 2;

- $\{CONM_{L,j}^{512}\}_{0 \le j < 32}$, $\{CONM_{R,j}^{512}\}_{0 \le j < 32}$, $\{CONM_{X,j}^{512}\}_{0 \le j < 32}$, $\{CONC_{L,j}^{512}\}_{0 \le j < 104}$, $\{CONC_{R,j}^{512}\}_{0 \le j < 104}$ for CF^{512}, and
- $\{CONM_{L,j}^{512}\}_{32 \le j < 64}$, $\{CONM_{R,j}^{512}\}_{32 \le j < 64}$, $\{CONM_{X,j}^{512}\}_{32 \le j < 64}$, $\{CONC_{L,j}^{512}\}_{104 \le j < 208}$, $\{CONC_{R,j}^{512}\}_{104 \le j < 208}$ for FF^{512}.

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 25.

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

Step 3. The following operation is iterated for $i = 0$ to 103.

$$CONC_{R,i}^{512} \leftarrow CONC_{L,i}^{512} \lll_{32} 3$$

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

$$\begin{cases} CONM_{L,4i}^{512} \leftarrow CONC_{L,12i}^{512} \lll_{32} 1 \\ CONM_{L,4i+1}^{512} \leftarrow CONC_{L,12i+1}^{512} \lll_{32} 1 \\ CONM_{L,4i+2}^{512} \leftarrow CONC_{L,12i+2}^{512} \lll_{32} 1 \\ CONM_{L,4i+3}^{512} \leftarrow CONC_{L,12i+3}^{512} \lll_{32} 1 \\ CONM_{R,4i}^{512} \leftarrow CONC_{L,12i+4}^{512} \ggg_{32} 1 \\ CONM_{R,4i+1}^{512} \leftarrow CONC_{L,12i+5}^{512} \ggg_{32} 1 \\ CONM_{R,4i+2}^{512} \leftarrow CONC_{L,12i+6}^{512} \ggg_{32} 1 \\ CONM_{R,4i+3}^{512} \leftarrow CONC_{L,12i+7}^{512} \ggg_{32} 1 \\ CONM_{X,4i}^{512} \leftarrow CONC_{L,12i+8}^{512} \lll_{32} 2 \\ CONM_{X,4i+1}^{512} \leftarrow CONC_{L,12i+9}^{512} \lll_{32} 2 \\ CONM_{X,4i+2}^{512} \leftarrow CONC_{L,12i+10}^{512} \lll_{32} 2 \\ CONM_{X,4i+3}^{512} \leftarrow CONC_{L,12i+11}^{512} \lll_{32} 2 \end{cases}$$

Step 5. The following operations are iterated for $i = 0$ to 25.

$$\begin{cases} CONC^{512}_{L,4i+104} \leftarrow CONC^{512}_{L,4i} \\ CONC^{512}_{L,4i+105} \leftarrow CONC^{512}_{L,4i+1} \\ CONC^{512}_{L,4i+106} \leftarrow CONC^{512}_{L,4i+2} \\ CONC^{512}_{L,4i+107} \leftarrow CONC^{512}_{L,4i+3} \oplus \texttt{0x01010101} \\ CONC^{512}_{R,4i+104} \leftarrow CONC^{512}_{R,4i} \\ CONC^{512}_{R,4i+105} \leftarrow CONC^{512}_{R,4i+1} \\ CONC^{512}_{R,4i+106} \leftarrow CONC^{512}_{R,4i+2} \\ CONC^{512}_{R,4i+107} \leftarrow CONC^{512}_{R,4i+3} \oplus \texttt{0x01010101} \end{cases}$$

Step 6. The following operations are iterated for $i = 0$ to 7.

$$\begin{cases} CONM^{512}_{L,4i+32} \leftarrow CONM^{512}_{L,4i} \\ CONM^{512}_{L,4i+33} \leftarrow CONM^{512}_{L,4i+1} \\ CONM^{512}_{L,4i+34} \leftarrow CONM^{512}_{L,4i+2} \\ CONM^{512}_{L,4i+35} \leftarrow CONM^{512}_{L,4i+3} \\ CONM^{512}_{R,4i+32} \leftarrow CONM^{512}_{R,4i} \\ CONM^{512}_{R,4i+33} \leftarrow CONM^{512}_{R,4i+1} \\ CONM^{512}_{R,4i+34} \leftarrow CONM^{512}_{R,4i+2} \\ CONM^{512}_{R,4i+35} \leftarrow CONM^{512}_{R,4i+3} \\ CONM^{512}_{X,4i+32} \leftarrow CONM^{512}_{X,4i} \\ CONM^{512}_{X,4i+33} \leftarrow CONM^{512}_{X,4i+1} \\ CONM^{512}_{X,4i+34} \leftarrow CONM^{512}_{X,4i+2} \\ CONM^{512}_{X,4i+35} \leftarrow CONM^{512}_{X,4i+3} \end{cases}$$

2.7.3 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 CF of AURORA-224/256, and CF^{512} of AURORA-384/512 version 2. 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 $\{CONM_{L,j}\}_{0 \le j < 32}$							
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 $\{CONM_{R,j}\}_{0 \le j < 32}$							
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 version 2, CF^{512}: $\{CONC^{512}_{L,j}\}_{0 \le j < 104}$							
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	f0f15836	779799d1	a7c9542d	cc5c85f1
022980c4	4ec2cce8	7f3ba6f5	662ebca4	4fa80189	86707e20	fe76eb74	03bf7416
d4aa0312	e2292744	fced7076	b177104f	4a9f368c	041d13a2	c973ee43	d8bbf67b
dec46d18	770709d1	92e77a18	ec5d8561	5e43ea98	4e8a84e8	1567fa9f	f62ebcec
f77cd531	86545a20	2ace53a0	4bbf7432	0d339acb	e23b3544	6534a9ef	9577105d

Constant Values for AURORA-384/512 version 2, CF^{512}: $\{CONC^{512}_{R,j}\}_{0 \le j < 104}$							
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	878ac1b7	bcbcce8b	3e4aa16d	62e42f8e
114c0620	76166742	f9dd37ab	3175e523	7d400c4a	3383f104	f3b75ba7	1dfba0b0
a5501896	11493a27	e76b83b7	8bb8827d	54f9b462	20e89d10	4b9f721e	c5dfb3de
f62368c6	b8384e8b	973bd0c4	62ec2b0f	f21f54c2	74542742	ab3fd4f8	b175e767
bbe6a98f	32a2d104	56729d01	5dfba192	699cd658	11d9aa27	29a54f7b	abb882ec

Constant Values for AURORA-384/512 version 2, CF^{512}: $\{CONM^{512}_{L,j}\}_{0 \le j < 32}$							
d0b1e35c	85e9f4c9	1ca39908	166c6124	92ebb91e	571f021b	46e1db52	fbc8b3d3
d0590a5d	f35548bf	f5a299e0	6e801799	2485d7a8	25e1fd68	28576d3c	052dc12c
938edc1f	cb2f3386	23e0da37	98f12fe3	6a5704e7	c0d2ce8d	fb1823ee	62e6241e
04530188	9d8599d0	fe774dea	cc5d7948	953e6d18	083a2744	92e7dc87	b177ecf7

Constant Values for AURORA-384/512 version 2, CF^{512}: $\{CONM^{512}_{R,j}\}_{0 \le j < 32}$							
cda571ae	6035b2b3	0e511fcb	9a999906	ec885c8f	10676043	23703ee6	3f796954
99c99d7a	eee2653d	e2854ba7	358417d1	b7bf6bd4	5434b387	142b65d1	98f1ad07
3822f65b	a3e16670	89a4ea4c	331e5ad2	c4ce9a27	829e99d1	e5d816a0	cc5cfbad
a7d400c4	43383f10	7f3b75ba	01dfba0b	6f62368c	bb8384e8	4973bd0c	f62ec2b0

Constant Values for AURORA-384/512 version 2, CF^{512}: $\{CONM^{512}_{X,j}\}_{0 \le j < 32}$							
557fd84f	54f0cacc	27b1c60c	6a679d6a	47ad0a5d	c5d9e05e	f5a3d4de	3f450c40
b371d7a8	3fcff5a7	28562002	14b0f654	541abd4e	84f4ce1c	42b0c769	63c74d6e
adc365b6	0bc3f892	9a483eb0	0ed9c258	c3c560db	de5e6745	9f2550b6	317217c7
52a80c4b	88a49d13	f3b5c1db	c5dc413e	790faa61	3a2a13a1	559fea7c	d8baf3b3

2.8 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 2.12: A pseudocode of $MSM : \{0,1\}^{256} \times (\{0,1\}^{32})^{32} \rightarrow (\{0,1\}^{32})^{72}$. *BD* is defined in Sec. 2.2.4. 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 2.13: 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. 2.2.4. F and F' are functions over $\{0,1\}^{32}$.

$CPM^{512}[F, F'](X_{(256)}, \{Y_{j\,(32)}\}_{0\le j<216}, \{W_{j\,(32)}\}_{0\le j<104})$
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 25 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 $(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{208}, Y_{209}, \ldots, Y_{215})$
130 $Z \leftarrow (X_0 \,\|\, X_1 \,\|\, \cdots \,\|\, X_7)$
140 return $Z_{(256)}$

Figure 2.14: A pseudocode of $CPM^{512} : \{0,1\}^{256} \times (\{0,1\}^{32})^{216} \times (\{0,1\}^{32})^{104} \to \{0,1\}^{256}$. BD is defined in Sec. 2.2.4. F and F' are functions over $\{0,1\}^{32}$.

$BD(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$
000 for $i \leftarrow 0$ to 7 do
010 $\quad (x_{4i}, x_{4i+1}, x_{4i+2}, x_{4i+3}) \leftarrow X_i$
020 for $i \leftarrow 0$ to 31 do
030 $\quad x'_{\pi(i)} \leftarrow x_i$
040 for $i \leftarrow 0$ to 7 do
050 $\quad X_i \leftarrow (x'_{4i} \,\|\, x'_{4i+1} \,\|\, x'_{4i+2} \,\|\, x'_{4i+3})$
060 return $(X_{0\,(32)}, X_{1\,(32)}, \ldots, X_{7\,(32)})$

Figure 2.15: A pseudocode of $BD : (\{0,1\}^{32})^8 \to (\{0,1\}^{32})^8$. π is defined in Fig. 2.4.

$DR(\{X_{j\,(32)}\}_{0\le j<72}, \{Y_{j\,(32)}\}_{0\le j<72})$
000 for $i \leftarrow 0$ to 8 do
010 $\quad (Z_{16i}, Z_{16i+1}, \ldots, Z_{16i+7}) \leftarrow PROTL(X_{8i}, X_{8i+1}, \ldots, X_{8i+7})$
020 $\quad (Z_{16i+8}, Z_{16i+9}, \ldots, Z_{16i+15}) \leftarrow PROTR(Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7})$
030 return $\{Z_{j\,(32)}\}_{0\le j<144}$

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.

$DR^{512}(\{X_{j\,(32)}\}_{0\le j<72}, \{Y_{j\,(32)}\}_{0\le j<72}, \{W_{j\,(32)}\}_{0\le j<72})$
000 for $i \leftarrow 0$ to 8 do
010 $\quad (Z_{24i}, Z_{24i+1}, \ldots, Z_{24i+7}) \leftarrow PROTL(X_{8i}, X_{8i+1}, \ldots, X_{8i+7})$
020 $\quad (Z_{24i+8}, Z_{24i+9}, \ldots, Z_{24i+15}) \leftarrow PROTR(Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7})$
030 $\quad (Z_{24i+16}, Z_{24i+17}, \ldots, Z_{24i+23}) \leftarrow PROTX(W_{8i}, W_{8i+1}, \ldots, W_{8i+7})$
040 return $\{Z_{j\,(32)}\}_{0\le j<216}$

Figure 2.17: A pseudocode of $DR^{512} : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \to (\{0,1\}^{32})^{216}$. The functions $PROTL$, $PROTR$ and $PROTX$ are defined in (2.8), (2.9) and (2.10), respectively.

AURORA-256(M)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow Pad(M)$
010 $H_0 \leftarrow 0^{256}$
020 for $i \leftarrow 0$ to $m-2$ do
030 $H_{i+1} \leftarrow CF(H_i, M_i)$
040 $H_m \leftarrow FF(H_{m-1}, M_{m-1})$
050 return $H_{m\,(256)}$

Figure 2.18: A pseudocode of AURORA-256. The padding function, $Pad(\cdot)$, is defined in (2.11), CF is defined in Sec. 2.3.2, and FF is defined in Sec. 2.3.3.

$CF(H_{i\,(256)}, M_{i\,(512)})$
000 $(M_L, M_R) \leftarrow M_i$
010 $X \leftarrow H_i$
020 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS_L(M_L)$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS_R(M_R)$
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 $Y \leftarrow CP(X, \{U_j\}_{0\le j<144})$
060 $H_{i+1} \leftarrow Y \oplus X$
070 return $H_{i+1\,(256)}$

Figure 2.19: 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 (2.12), (2.13), Sec. 2.2.6, and in (2.14), respectively.

$FF(H_{m-1\,(256)}, M_{m-1\,(256)})$
000 $(M_L, M_R) \leftarrow M_{m-1}$
010 $X \leftarrow H_{m-1}$
020 $\{T_{L,j}\}_{0\le j<72} \leftarrow MSF_L(M_L)$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MSF_R(M_R)$
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 $Y \leftarrow CPF(X, \{U_j\}_{0\le j<144})$
060 $H_m \leftarrow Y \oplus X$
070 return $H_{m\,(256)}$

Figure 2.20: 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 (2.16), (2.17), Sec. 2.2.6, and in (2.18), respectively.

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

Figure 2.21: A pseudocode of AURORA-224. Pad, CF, and FF are the same as AURORA-256 and defined in Sec. 2.3.

AURORA-512v2(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 $m-2$ do
030 $\quad H_{i+1} \leftarrow CF^{512}(H_i, M_i)$
040 $H_m \leftarrow FF^{512}(H_{m-1}, M_{m-1})$
050 return $H_{m\,(512)}$

Figure 2.22: A pseudocode of AURORA-512 version 2. The padding function, $Pad(\cdot)$, is defined in (2.11), CF^{512} is defined in Sec. 2.5.2, FF^{512} is defined in Sec. 2.5.3.

$CF^{512}(H_{i\,(512)}, M_{i\,(512)})$

000 $(M_L, M_R) \leftarrow M_i$
010 $(X_L, X_R) \leftarrow H_i$
020 $\{T_{L,j}\}_{0\le j<72} \leftarrow MS_L^{512}(M_L)$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MS_R^{512}(M_R)$
040 $\{T_{X,j}\}_{0\le j<72} \leftarrow MS_X^{512}(X_L)$
050 $\{U_j\}_{0\le j\le 216} \leftarrow DR^{512}(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72}, \{T_{X,j}\}_{0\le j<72})$
060 $Y_L \leftarrow CP_L^{512}(X_R, \{U_j\}_{0\le j<216})$
070 $Y_R \leftarrow CP_R^{512}(X_R, \{U_j\}_{0\le j<216})$
080 $Z_L \leftarrow Y_L \oplus X_R$
090 $Z_R \leftarrow Y_R \oplus X_R$
100 $H_{i+1} \leftarrow (Z_L, Z_R)$
110 return $H_{i+1\,(512)}$

Figure 2.23: A pseudocode of $CF^{512} : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$. MS_L^{512}, MS_R^{512}, MS_X^{512}, DR^{512}, CP_L^{512}, and CP_R^{512} are defined in (2.19), (2.20), (2.21), Sec. 2.2.7, (2.22), and in (2.23), respectively.

$FF^{512}(H_{m-1\,(512)}, M_{m-1\,(512)})$

000 $(M_L, M_R) \leftarrow M_{m-1}$
010 $(X_L, X_R) \leftarrow H_{m-1}$
020 $\{T_{L,j}\}_{0\le j<72} \leftarrow MSF_L^{512}(M_L)$
030 $\{T_{R,j}\}_{0\le j<72} \leftarrow MSF_R^{512}(M_R)$
040 $\{T_{X,j}\}_{0\le j<72} \leftarrow MSF_X^{512}(X_L)$
050 $\{U_j\}_{0\le j\le 216} \leftarrow DR^{512}(\{T_{L,j}\}_{0\le j<72}, \{T_{R,j}\}_{0\le j<72}, \{T_{X,j}\}_{0\le j<72})$
060 $Y_L \leftarrow CPF_L^{512}(X_R, \{U_j\}_{0\le j<216})$
070 $Y_R \leftarrow CPF_R^{512}(X_R, \{U_j\}_{0\le j<216})$
080 $Z_L \leftarrow Y_L \oplus X_R$
090 $Z_R \leftarrow Y_R \oplus X_R$
100 $H_m \leftarrow (Z_L, Z_R)$
110 return $H_{m\,(512)}$

Figure 2.24: A pseudocode of $FF^{512} : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$. MSF_L^{512}, MSF_R^{512}, MSF_X^{512}, DR^{512}, CPF_L^{512}, and CPF_R^{512} are defined in (2.24), (2.25), (2.26), Sec. 2.2.7, (2.27), and in (2.28), respectively.

```
AURORA-384v2(M)
000    (M_0, M_1, ..., M_{m-1}) ← Pad(M)
010    H_0 ← 1^512
020    for i ← 0 to m − 2 do
030        H_{i+1} ← CF^512(H_i, M_i)
040    H_m ← FF^512(H_{m-1}, M_{m-1})
050    H'_m ← TF_384(H_m)
060    return H'_{m (384)}
```

Figure 2.25: A pseudocode of AURORA-384 version 2. Pad, CF^{512}, FF^{512} are the same as AURORA-512 and defined in Sec. 2.5.

2.9 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 defined 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-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-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-384 version 2	
Message	Hash Value
M_1	cb8c22c8 15e3e5a3 8a1691ee f1dc1ad9 15dfea22 9f27a170 455aaaec b4a9f55a 3a372d1e 412d8853 b754ea23 c28a9e12
M_2	b1849343 f6601342 471176d7 bd671692 d3c39ca0 6f5d7a7c dccd802d 47ad5875 b6528095 d51d6be4 4bfb0b0d a5a90099
M_3	b579aa54 199a921d df7a3225 3dc82390 a40eb36d 1b649b8f 79430ef2 75b27f50 595ee272 979eef4d e108d540 b3004556

AURORA-512 version 2	
Message	Hash Value
M_1	51c0c29f d45b4bcf f7f54733 5af4424d 74817faf 1983bf5b e2afafd8 86f830bf b0a49fc2 9f65447b 5336d68c 5793d649 ad19dade 635a84c9 817681e0 1d36acae
M_2	5f2a16e9 99edf233 a9b96f52 1b6e792b bf33ea51 549bc0e7 9f5a62e4 17ceff99 ce7d9592 aae2edf8 1d46ec8e ad8181ec 6cba448e 4170b8cb f0c4ec12 eaceab6f
M_3	d00e5327 16a8dbb7 11aec003 36daeebd dccb72b9 278ad094 2f417e42 b2b09b67 80f18bfc 6d0403e0 6f32ca24 fe4b9b89 43281215 a3a6c560 46d828c5 b6fd6068

Chapter 3

Design Rationale of AURORA*

This chapter describes design rationale of the hash function family AURORA*. 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 and AURORA-512 version 2, then explain the components in the common building blocks.

We describe the design rationale for AURORA-256 and AURORA-512 version 2 as representatives of the AURORA* family. 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 version 2 is applicable to AURORA-384 version 2.

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 [37, 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 often used to instantiate random oracles as well. 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. [34]. 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 [37]: the input message is padded by the padding function $Pad(\cdot)$ in AURORA*. CR-Pr can be proven similarly to the proof in [37]. The pseudo-random 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

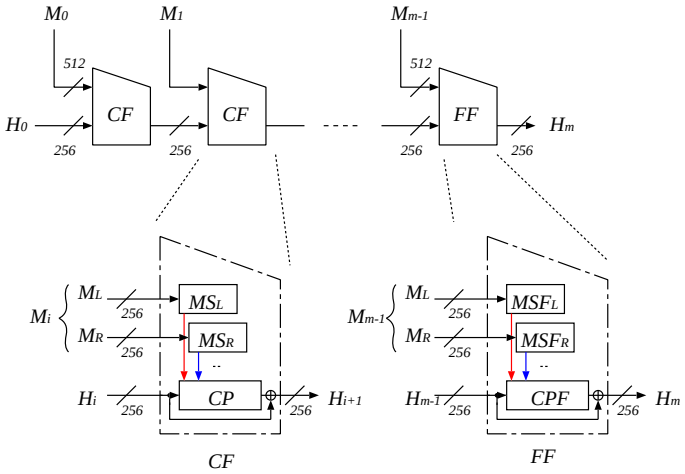

Figure 3.1: AURORA-256: Domain extension and compression function.

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 [36, p.340]. We chose this construction because it is possible to input a longer message than a chaining value to achieve higher throughput, while in the Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions [36, p.340] a message and a chaining value must be the same size. Although the Davies-Meyer construction has a negative property such that fixed points are easily found [38, 46, 17], we attached more importance to achieve higher throughput.

Considering recent attacks on hash functions exploiting simple message scheduling [59, 60, 61], we chose to design more secure (and more heavy) message schedule like Whirlpool [3] and DASH [8]. Each components of the message scheduling function (MS_L, MS_R) is 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 version 2

3.2.1 Domain Extension

AURORA-512 version 2 adopts the strengthened Merkle-Damgård (sMD) transform with a finalization function, which is the same domain extension transform as AURORA-256 (See Fig. 3.2). The sMD transform is widely deployed and its security have been studied extensively. The sMD transform with the finalization function preserves collision-resistance (CR) and indifferentiability (PRO) of the underlying compression function [28].

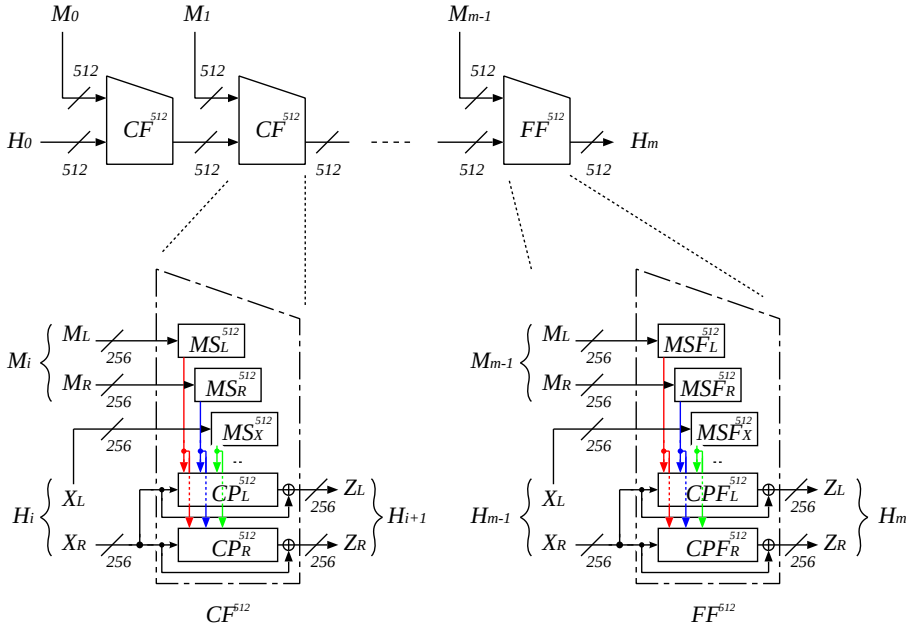

Figure 3.2: AURORA-384/512 version 2: Domain extension and compression function.

Similarly to AURORA-256, the structure of the finalization function FF^{512} is the same as the structure of the compression function CF^{512} except for the constants.

3.2.2 Compression Function – Hirose's DBL construction

The AURORA-512 version 2 compression function CF^{512} is based on one of Hirose's DBL constructions [26]. Let e be a blockcipher with the block size n and the key size k, $e : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$. Then $e(key, \cdot)$ is a permutation for every $key \in \{0,1\}^k$. It is shown that the construction shown in Fig. 3.3 composes collision-resistant hash functions when e_U and e_L are independent ideal blockciphers in the ideal cipher model [26, 27].

As shown in Fig. 3.2, a half of the chaining value X_R is input to two chaining value processing functions CP_L^{512} and CP_R^{512}, which are different 256-bit permutations. The other half of the chaining value X_L is input to the message scheduling function MS_X^{512}. The 512-bit message block M_i is divided into M_L and M_R, and they are input to the message scheduling functions MS_L^{512} and MS_R^{512}, respectively. AURORA-512 version 2 compression function is constructed of $e_0(M_i||X_L, X_R)$ and $e_1(M_i||X_L, X_R)$, where e_0 and e_1 are 256-bit blockciphers with 768-bit keys. Three message scheduling functions MS_L^{512}, MS_R^{512}, MS_X^{512} serve as common key scheduling of e_0 and e_1.

3.3 Components and Constants

The compression functions of the AURORA* family are composed of the common building blocks: the message scheduling module (MSM) and the chaining value processing modules (CPM and CPM^{512}). This section shows the design rationale of the components and constants used in these modules.

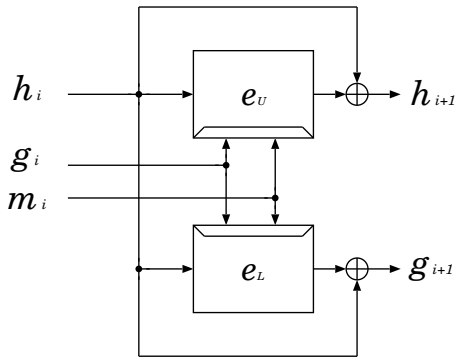

Figure 3.3: A compression function based on one of Hirose's DBL constructions

3.3.1 AURORA Structure

We chose a 256-bit permutation based on byte-oriented operations to construct the structure for both of the message scheduling module (MSM) and the chaining value processing modules (CPM and CPM^{512}). We call it the AURORA structure, which is shown in Fig. 3.4. It can be regarded as a combination of SPN and a generalized Feistel structure.

The AURORA structure itself is novel, but it 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 of the F-function are written in Sec. 3.3.2.

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 of AURORA-256 consists of an underlying 256-bit blockcipher with two message scheduling modules that treat 512-bit messages. 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 each of the 256-bit permutations in MSM and CPM, i.e. $(X_0, X_1, \cdots, X_7) \rightarrow (Z_{64}, Z_{65}, \cdots, Z_{71})$ shown in Fig. 2.1 and $(X_0, X_1, \cdots, X_7) \rightarrow (Z_0, Z_1, \cdots, Z_7)$ shown in Fig. 2.2, so that they have no differential paths with high probability under the assumption that "subkeys" are independent and uniformly distributed.

In choosing the structure among several candidates including the generalized Feistel structure and its variants, we estimated maximum differential characteristic probability obtained by numbers of active S-boxes, and compared estimated performance given by the number of required F-functions. As a result of consideration discussed in Sec. 4.2.2, we chose 8-round AURORA structure for the message scheduling module (MSM) and 17-round AURORA structure for the chaining value processing module (CPM). Similarly, as for AURORA-512 version 2, as a result of consideration discussed in Sec. 4.2.2, we chose the same 8-round AURORA structure for the message scheduling module (MSM) and 26-round AURORA structure for the chaining value processing module (CPM^{512}).

Since (1) AURORA*'s message scheduling module is designed to be secure by itself by using blockcipher design techniques, and (2) AURORA* is 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.

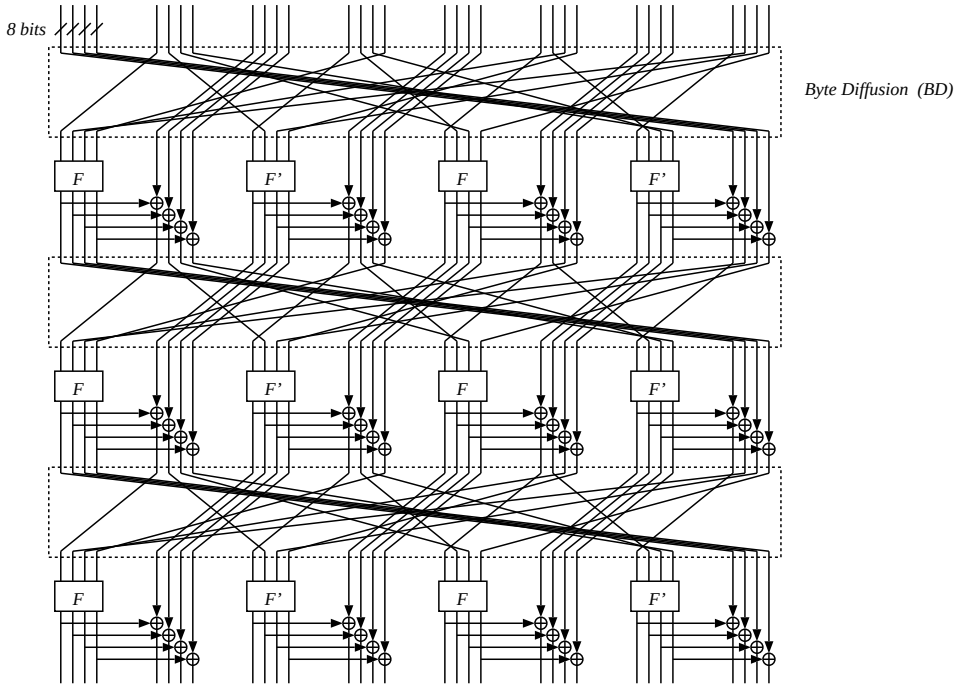

Figure 3.4: AURORA structure.

Byte Diffusion function *BD*. The byte diffusion function *BD* is adopted to enhance diffusion and to destroy 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 drawbacks 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 variants of diffusion on differential characteristic probability to determine the byte diffusion. 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 similar to the ShiftRow transformation in the AES [23], 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.

3.3.2 F-function

The F-function consists of a substitution layer and a permutation layer, where four non-linear byte substitutions (S-boxes) and a 4×4 maximum distance separable (MDS) matrix multiplication over GF(2^8) are operated. The structure and the components of the F-function are chosen to facilitate analysis and to utilize the 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 MS_L, MS_R MS_L^{512}, MS_R^{512}, MS_X^{512}, CP, CP_L^{512}, and CP_R^{512}, 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) [56] 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 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 [23] 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. 2.2.5).

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.

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[1] shown in Table 3.1 according to the pseudocode below:

[1]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 3.1: 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

```
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 [55, 56, 57, 54] and used in the blockcipher CLEFIA [58]. This technique is applicable to the AURORA structure. By using plural different matrices satisfying the conditions as follows, 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 $P : \{0,1\}^{pn} \rightarrow \{0,1\}^{qn}$. The branch number of P is defined as*

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

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 Conditions I and II. Note that the elements of the matrices are in GF(2^8).

Condition I (MDS)

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

Condition II (DSM)

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

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

AURORA-224/256	Function	MS_L	MS_R		CP	
[SBL]	Building block	MSM	MSM		CPM	
Davies-Meyer	# of rounds	8-round	8-round		17-round	
construction	F-functions	F_0,F_1	F_2,F_3		F_1,F_0	
	Matrices	$\mathcal{M}_0$,$\mathcal{M}_1$	$\mathcal{M}_2$,$\mathcal{M}_3$		$\mathcal{M}_1$,$\mathcal{M}_0$	
AURORA-384/512 v2	Function	MS_L^{512}	MS_R^{512}	MS_X^{512}	CP_L^{512}	CP_R^{512}
[DBL]	Building block	MSM	MSM	MSM	CPM^{512}	CPM^{512}
Hirose's	# of rounds	8-round	8-round	8-round	26-round	26-round
construction	F-functions	F_0,F_1	F_2,F_3	F_0,F_3	F_1,F_0	F_3,F_2
	Matrices	$\mathcal{M}_0$,$\mathcal{M}_1$	$\mathcal{M}_2$,$\mathcal{M}_3$	$\mathcal{M}_0$,$\mathcal{M}_3$	$\mathcal{M}_1$,$\mathcal{M}_0$	$\mathcal{M}_3$,$\mathcal{M}_2$

In Condition I, 5 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 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. We call the pair of the matrices satisfying Conditions I and II the "DSM pair". ($\mathcal{M}_0$, $\mathcal{M}_1$) is a DSM pair.

Actually, firstly ($\mathcal{M}_0$, $\mathcal{M}_1$) was chosen according to (3.1) and (3.3). Then $\mathcal{M}_2$ and $\mathcal{M}_3$ were 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:

$$\mathcal{B}_8(\mathcal{M}_0|\mathcal{M}_3) = \mathcal{B}_8({}^t\mathcal{M}_0^{-1}|\,{}^t\mathcal{M}_3^{-1}) = 5 \quad (3.5)$$

$$\mathcal{B}_8(\mathcal{M}_1|\mathcal{M}_2) = \mathcal{B}_8({}^t\mathcal{M}_1^{-1}|\,{}^t\mathcal{M}_2^{-1}) = 5 \quad (3.6)$$

Therefore, the DSM effect is expected to work 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 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.

3.3.3 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 in AURORA-224/256, and DR^{512} in AURORA-384/512 version 2, respectively. The functions DR and DR^{512} are 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.

3.3.4 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 version 2, 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

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-512 v2	$IV_0 = (11^{1/2} - 3)2^{16}$	$mask_0 = (11^{1/3} - 2)2^{16}$	$mask_2 = (11^{1/5} - 1)2^{16}$
	$IV_1 = (13^{1/2} - 3)2^{16}$	$mask_1 = (13^{1/3} - 2)2^{16}$	$mask_3 = (13^{1/5} - 1)2^{16}$

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. 4.2.2.

3.3.5 Constant Generation

Role of Constants in the AURORA* family

AURORA-224/256 and AURORA-384/512 version 2 use 3 and 5 sets of constants, respectively, as listed in Sec. 2.7.3.

The constants play an important role in security. They are used so that each module of *MSM*, *CPM*, and CPM^{512} looks an independent function. Moreover, it is expected that the finalization function *FF* (resp. FF^{512}) behaves as an different function from the compression function *CF* (resp. CF^{512}) 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 [58]. The four 32-bit constant values used in each module of *MSM*, *CPM* and CPM^{512} 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, 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. 2.7, by checking whether the generated sequences pass the statistical test suites: the mono bit test, the poker test, and the runs test [19]. 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 version 2 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 $(r_0, r_1, r_2, r_3) = (8, 8, 8, 9)$.

3.3.6 Initial Value

We consider that the security provided by the structure of the AURORA* 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.

Both of AURORA-256 and AURORA-512 version 2 use the same all-0 constants as the initial value. We don't identify any security problem, because each module used in AURORA-256 and AURORA-512 version 2 are different due to different matrices and constants. Similarly, both of AURORA-224 and AURORA-384 version 2 use the same all-1 constants as the initial value, but we don't identify any security problem.

Chapter 4

Security of AURORA*

4.1 Expected Strength

Each hash function of the AURORA* family of hash size n bits 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 the AURORA* family provide resistance to length-extension attacks (see Sec. 4.3.4).

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 a member of the AURORA* family with hash size n bits is used with HMAC to construct a PRF [24], 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 a member of the AURORA* family with hash size n bits is used in an n-bit randomized hashing scheme [40], it provides approximately $n - k$ bits of security against the following attack. (1) An attacker chooses a message M_1 of length at most 2^k bits, then gets a randomized hash of M_1 with a randomization value r_1 that has been randomly chosen without the attacker's control. (2) Find a second message M_2 and a randomization value r_2 that yield the same randomized hash value.

4.2 Security Argument

4.2.1 Security of HMAC using AURORA*

HMAC-AURORA-224/256 specified in Sec. 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 Sec. 4.2.2, 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].

In a similar manner, HMAC-AURORA-384/512 version 2 employs CF^{512} and FF^{512} as their compression functions and its domain extension is the same as the MD transform. According to the discussion in Sec. 4.2.2, CF^{512} and FF^{512} are expected to be PRFs when keyed via the IV and when keyed via its data input. Therefore HMAC-AURORA-384/512 version 2 is expected to be a good PRF.

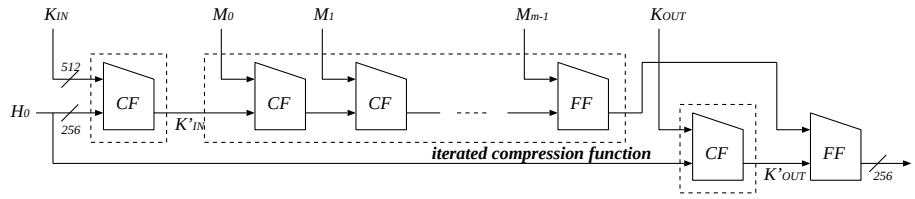

Figure 4.1: HMAC-AURORA-224/256.

4.2.2 Security Properties of AURORA structure

Guaranteed Active S-boxes in AURORA structure

By the recent evolution of research on attacks on hash functions [59, 60, 61], 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. This section investigates a permutation used in AURORA* called "AURORA structure", and discusses security aspect with regard to differential cryptanalysis.

As the specification of AURORA* shows, *MSM*, *CPM*, and CPM^{512} employ 8-round, 17-round, and 26-round AURORA structure, respectively. The AURORA structure is based on 8-bit S-boxes, 4×4 matrices in GF(2^8) and the byte diffusion *BD*, and all components are byte-oriented. Thus, it is natural to evaluate the immunity against differential cryptanalysis by counting the minimum number of active S-boxes of AURORA structure using a blockcipher evaluation method [54, 55, 57, 58].

We used a computer 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 [54]. During the search, the DSM conditions are used to judge whether given truncated paths are valid or not, and behavior of the byte diffusion function *BD* is also taken into consideration.

Table 4.1 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 (*MSM*) does not hold characteristics whose differential probability is higher than $2^{-6\times 26} = 2^{-156}$. Similarly, 17-round AURORA structure (*CPM*) does not hold characteristics with probability higher than $2^{-6\times 56} = 2^{-336}$. Similarly, 26-round AURORA structure (CPM^{512}) does not hold characteristics with probability higher than $2^{-6\times 86} = 2^{-516}$.

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 M, 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 complexity of less than 2^{128}. If the attacker injects a difference into *MSM*, the probability of the differential that follows a specific characteristic which is useful for finding collision is low, which is less than $2^{-156} (< 2^{-128})$. Therefore, it is expected that the attacker fails to find a collision of AURORA* with complexity of less than 2^{128}.

Moreover, we see that the compression function of AURORA-256 uses a 256-bit blockcipher, and the compression function of AURORA-512 version 2 uses two 256-bit blockciphers. The

Table 4.1: Guaranteed Numbers of Active S-boxes in AURORA structure.

Round	♯ of Active S-boxes	Round	♯ of Active S-boxes
1	0	14	46
2	1	15	50
3	5	16	52
4	6	17	<u>56</u>
5	9	18	60
6	15	19	62
7	22	20	66
8	<u>26</u>	21	70
9	30	22	72
10	32	23	76
11	36	24	80
12	40	25	82
13	42	26	<u>86</u>

obtained numbers of active S-boxes shown in Table 4.1 imply that the blockciphers are secure against distinguishing attacks in differential cryptanalytic scenarios, which we believe is a more important requirement than key recovery attacks on hash functions. The 17-round AURORA structure used in AURORA-224/256 does not hold characteristics with probability higher than $2^{-6\times 56} = 2^{-336} (< 2^{-256})$. The 26-round AURORA structure used in AURORA-384/512 version 2 does not hold characteristics with probability higher than $2^{-6\times 86} = 2^{-516} (< 2^{-256})$. As a result, we conclude that the underlying blockcipher does not hold unexpected properties which can be exploited by attackers in differential attack scenarios.

Output Truncation

As stated in Sec. 4.1, any m-bit hash function specified by taking a fixed subset of the AURORA* hash function's output bits is 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 an F-function at the last round. Therefore, we introduced the truncation function TF to avoid such the loss to maximize the effect of F-functions. The output truncation function TF is applied for AURORA-256 with different IVs to generate the output values for AURORA-224. Similarly, TF is applied for AURORA-512 version 2 with different IVs to generate the output values for AURORA-384 version 2. Due to the internal connection of the 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 an F-function. Let $X_{(256)}$ be an output of the 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 an F-function at the last round in CPM or CPM^{512} 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 the AURORA* family are designed to drop byte data at discontinuous positions.

Impossible Differentials in AURORA Structure

Impossible differential is a 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 hash function cases, 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 [52].

We searched for impossible differential paths by considering that the matrices satisfy the DSM conditions (i.e. Conditions I and II described in Sec. 3.3.2). The longest impossible differential paths that we found in the AURORA structure 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 simplified-AURORA structure where byte diffusion function BD is replaced with a "usual" word-wise permutation.

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 modules (CPM and CPM^{512}) employ the 17-round and 26-round AURORA structure, respectively, and the message scheduling modules (MSM) employ the 8-round AURORA structure, it is expected that the compression functions of AURORA-256 and AURORA-512 version 2 have no impossible differentials, which can allow us to distinguish any of AURORA* hash functions 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, multicollision attacks, and slide 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. 4.2).

4.3.1 Collision Attacks

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. 4.2.2 describes differential cryptanalysis of the AURORA structure, and shows that there is no differential characteristic in MSM, CPM, and CPM^{512} 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, 60, 61, 59].

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 4.2. 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.

Table 4.2: 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.

In Table 4.2, 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.

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 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 4.3. The chaining value difference ΔX_i is the difference in the input chaining value X_i of each round. For other symbols, see Fig. 4.2.

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 $\Delta M_L = (x_3, 0, 0, 0, x_1, 0, x_2, 0)$ and $\Delta M_R = (0, 0, 0, x_1, 0, x_2, 0, x_3)$, 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 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. 3.3.2).) 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-256 with a complexity of 2^{16} 3-round AURORA-256 compression function.

Approach II : Application of Peyrin's collision attack on Grindahl. Another approach for finding collisions is a method used in the cryptanalysis of Grindahl [45]. 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

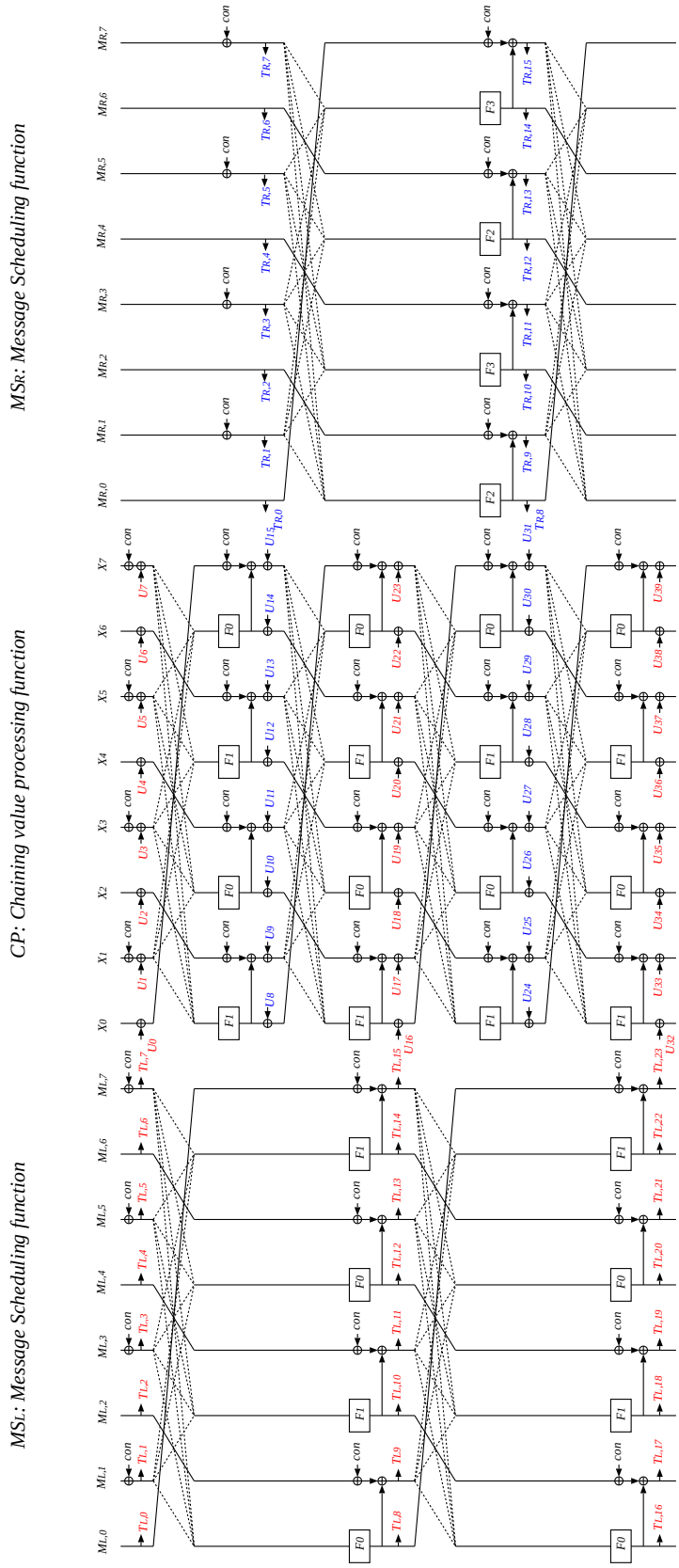

Figure 4.2: Compression function of AURORA-256 (reduced to 4-round).

Table 4.3: 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	–	–	–	–	–	–	–	–

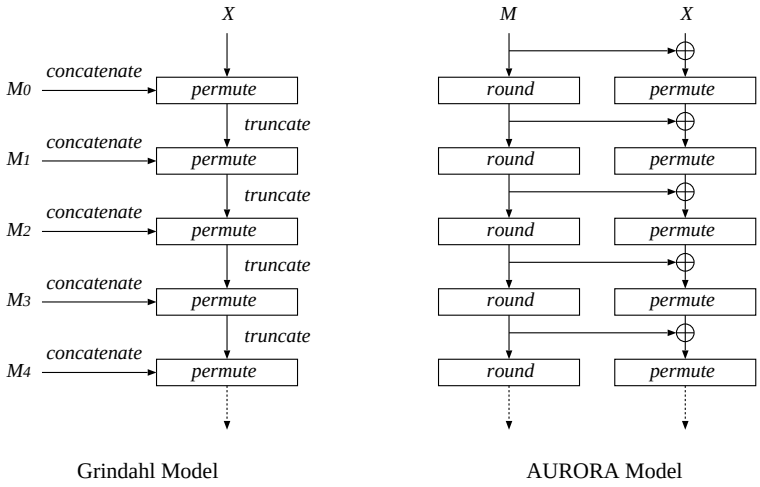

Figure 4.3: Comparison between Grindahl model and AURORA model.

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. 4.3.

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 [45] 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*.

4.3.2 Preimage Attacks

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 [33, 10, 2, 1].

Approach I : Meet-in-the-middle approach. In most of the recent preimage attacks [33, 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 [33] 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 [33] 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 [51].

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 the full-round AURORA*. The meet-in-the-middle approach in [1] works up to 3-round AURORA*. A preimage of a 3-round reduced version of AURORA-256 can be found with the complexity of about 2^{241} 3-round AURORA-256 compression function computation. However, it is difficult to find a preimage of the full-round 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 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 $\mathcal{A}$, which does not contain DR without loss of generality. As a result, the variant $\mathcal{A}$ 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 $\mathcal{A}$ 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 [44]. 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 $\mathcal{B}$, which outputs 128-bit hash values in which 1-round of AURORA

[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).

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 $\mathcal{B}$. 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 has 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. [62], 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. 4.2.2 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 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 [33], in the second preimage attacks on SMASH by Lamberger et al. [32], and the (second) preimage attacks on GOST by Mendel et al. [35].

This approach works if one can find many highly probable differential characteristics for the same special message. According to the analysis in Sec. 4.2.2, 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 [32].

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 [31], 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.

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

Table 4.4: Second preimage resistance for 2^k block messages ($k < 64$) (bits).

AURORA-224	AURORA-256	AURORA-384 v2	AURORA-512 v2
$\min\{224, 256-k\}$	$256-k$	384	$512-k$

Considering this generic long-message second preimage attack, AURORA-256 and AURORA-512 version 2 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 version 2 provides second preimage resistance of 384 bits, because the maximum message block size for the AURORA* family is $2^{64}-1$ blocks, i.e. $k < 64$, and $384 < 512-k$.

Second preimage resistance of the AURORA* is summarized in Table 4.4.

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 and AURORA-384/512 version 2.

4.3.5 Multicollision Attack

Multicollision attack [29], 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 all hash functions in the AURORA* family 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 or AURORA-384/512 version 2 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 [25], the slide attacks also form a potential threat for a certain class of hash functions, e.g., sponge-function like structures. A slide property which is detected with 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 [25] exploits invertibility and self-similarity in the sponge-function like structures.

We believe that 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 CFs and FF, which behaves differently from CF with different constants. In AURORA-384/512 version 2 consists of CF^{512}s and FF^{512}, which behaves differently from CF^{512}. 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 hash function family AURORA*. The first parameter is an iteration number of round functions in AURORA structure used in *MSM*, *CPM* and CPM^{512}. The second parameter is a method to modify the AURORA* family to be able to output digests whose length are other than 224, 256, 384 and 512 bits.

4.4.1 Number of Rounds

Recommended numbers of rounds are 8 for *MSM*, 17 for *CPM*, and 26 for CPM^{512} as described in the specification. The tuning is done so that the following equations should be satisfied: $c_{256} = 2m + 1$ and $c_{512} = 3m + 2$, where m, c_{256}, and c_{512} are the numbers of rounds for *MSM*, *CPM*, and CPM^{512}, respectively. The permissible range for the parameter is $m \in \{8, 9, 10, 11, 12, 13, 14, 15, 16\}$. The greater the parameter is, the security of the hash function increases by paying cost for the performance. We believe that $m > 16$ is too much taking account of the dropping of the performance of implementations.

4.4.2 Variable Hash Size

Current specification of the hash function family AURORA* 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 bits. 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 version 2. 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 version 2 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 version 2 achieves 63.9 cycles/byte; on the NIST 64-bit reference platform, AURORA-256 achieves 15.4 cycles/byte and AURORA-512 version 2 achieves 37.8 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 8.9 Kgates with throughout of 1.1 Gbps, and the highest throughput of AURORA-256 is 10.4 Gbps with area of 35.0 Kgates; the smallest area of AURORA-512 version 2 is 12.4 Kgates with throughout of 467 Mbps, and the highest throughput of AURORA-512 version 2 is 6.9 Gbps with area of 59.7 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 and 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.

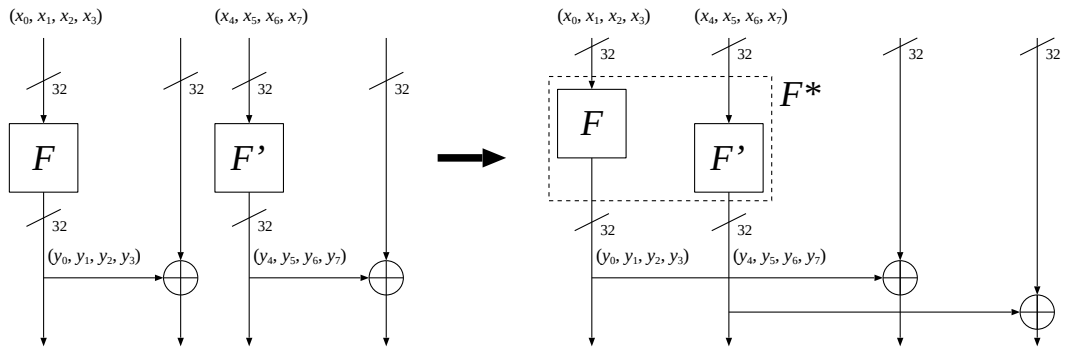

Figure 5.1: F^*-function

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

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

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

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

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	Vendor	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 and AURORA-384/512 version 2 on 32/64 bit processors. As for AURORA-224/256, we also show the evaluation results on 8 bit processors.

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 current evaluation results on performance of AURORA-224/256 and AURORA-384/512 version 2 on 32/64-bit processors. 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 version 2 and AURORA-512 version 2, respectively. All implementation types described in Sec. 5.1.1 are evaluated for each hash function of AURORA*. Bold figures indicate the best results of each AURORA* hash function on each platform. For AURORA-224/256, two types of loop structure 'unroll' and 'looped' are evaluated. In the 'unroll' structure, the round functions of AURORA* are unrolled; in the 'looped' structure, the round functions are implemented by loop function. For AURORA-384/512 version 2, only 'looped' structure is evaluated because 'unrolled' structure not only require more code size but also is even slower than 'looped' structure. As reference, we show the evaluation results of SHA-256 and SHA-512 implemented by Brian Gladman [43] in Tables 5.7 and 5.8, respectively.

8-bit Processors

We present the evaluation results on performance of AURORA-224/256 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. Tables 5.10 and 5.11 represent the evaluation results of AURORA-224 and AURORA-256, 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 version 2 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	(looped)	4,657.4	466.3	91.6	110.3	68.1	4,317.2	71,042
Type-S2	(looped)	4,353.9	441.8	86.1	65.8	**63.9**	4,055.1	69,182
Type-S3	(looped)	6,629.6	666.2	131.3	100.4	97.9	6,175.6	125,704
Type-S4	(looped)	5,631.8	567.0	110.9	84.8	82.7	6,111.5	101,012
Type-S5	(looped)	5,427.9	548.2	107.2	81.9	79.9	5,064.0	88,792
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(looped)	2,994.4	303.8	58.3	44.8	43.6	2,764.2	72,752
Type-S2	(looped)	3,390.9	340.7	66.7	51.2	50.1	3,172.7	71,756
Type-S3	(looped)	2,740.1	274.6	52.9	39.7	38.7	2,441.3	133,132
Type-S4	(looped)	2,651.1	266.3	51.6	38.8	**37.8**	2,403.3	108,440
Type-S5	(looped)	2,895.6	291.1	56.4	42.8	41.7	2,649.6	96,220
		Platform C (Opteron)						
Type-S1	(looped)	4,077.7	410.8	77.2	57.9	56.3	3,568.8	25,460
Type-S2	(looped)	4,463.8	449.6	84.3	63.1	61.3	3,889.5	19,892
Type-S3	(looped)	3,177.0	320.6	59.3	43.5	**42.1**	2,680.1	48,740
Type-S4	(looped)	3,213.8	323.2	59.5	43.8	42.5	2,697.5	32,516
Type-S5	(looped)	3,447.1	347.6	64.6	48.0	46.6	2,964.0	24,484
		Platform D (Pentium 4)						
Type-S1	(looped)	9,861.4	1,001.3	192.1	145.2	**141.4**	9,497.9	26,164
Type-S2	(looped)	10,847.3	1,060.2	204.5	157.5	153.6	9,663.7	20,496
Type-S3	(looped)	16,045.5	1,621.1	310.1	237.8	232.0	14,566.9	61,300
Type-S4	(looped)	15,094.8	1,518.9	292.4	223.1	217.5	13,845.5	43,952
Type-S5	(looped)	14,622.1	1,476.1	283.6	216.3	210.9	13,642.5	34,988

Table 5.6: AURORA-512 version 2 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	-	-
Type-S1	(looped)	4,614.0	464.9	91.4	69.8	68.1	4,317.2	71,042
Type-S2	(looped)	4,350.8	439.1	86.1	65.6	**63.9**	4,055.1	69,182
Type-S3	(looped)	6,490.7	655.8	130.8	100.3	97.9	6,175.6	125,704
Type-S4	(looped)	5,598.6	561.6	110.5	84.8	82.7	6,111.5	101,012
Type-S5	(looped)	5,408.8	543.3	106.7	81.8	79.9	5,064.0	88,792
		Platform B (Core 2 Duo (64-bit))						
Type-S1	(looped)	2,943.8	295.4	57.9	44.6	43.6	2,764.2	72,752
Type-S2	(looped)	3,365.0	336.2	66.3	51.2	50.1	3,172.7	71,756
Type-S3	(looped)	2,667.8	268.5	52.4	39.6	38.6	2,441.3	133,132
Type-S4	(looped)	2,607.1	261.9	50.6	38.8	**37.8**	2,403.3	108,440
Type-S5	(looped)	2,857.5	286.6	55.8	42.8	41.8	2,649.6	96,220
		Platform C (Opteron)						
Type-S1	(looped)	3,895.3	391.7	75.2	57.7	56.2	3,568.8	25,460
Type-S2	(looped)	4,244.2	426.8	82.0	62.8	61.3	3,889.5	19,892
Type-S3	(looped)	2,999.9	302.4	57.5	43.3	**42.3**	2,680.1	48,740
Type-S4	(looped)	3,008.6	303.5	57.5	43.6	42.6	2,697.5	32,516
Type-S5	(looped)	3,253.4	327.8	62.7	47.8	46.5	2,964.0	24,484
		Platform D (Pentium 4)						
Type-S1	(looped)	9,916.8	991.4	191.7	145.0	**141.3**	9,497.9	26,164
Type-S2	(looped)	10,715.7	1,114.7	206.9	157.9	154.4	9,663.7	20,496
Type-S3	(looped)	15,928.9	1,594.4	308.9	237.0	231.6	14,566.9	61,300
Type-S4	(looped)	14,851.3	1,498.3	289.7	222.6	217.3	13,845.5	43,952
Type-S5	(looped)	14,457.9	1,449.4	281.2	215.9	210.7	13,642.5	34,988

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 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

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

5.2 Hardware Implementation

This section describes hardware optimization techniques and several types of hardware implementations for AURORA*, and then shows hardware performance results of AURORA*. Since the implementations of AURORA-224 and AURORA-384 version 2 are basically same as AURORA-256 and AURORA-512 version 2, respectively, except the initial value and truncation of final hash value, we designed and evaluated the implementations of AURORA-256 and AURORA-512 version 2.

5.2.1 Optimization Techniques of F-functions

We introduce optimization techniques of F-functions focusing on an S-box and matrices.

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 [47]:

$$\begin{aligned} b_0 &= a_0\Delta^{-1}, \\ b_1 &= (a_0 + a_1)\Delta^{-1}, \\ \Delta &= (a_0 + a_1)a_1 + \lambda a_0^2. \end{aligned}$$

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.

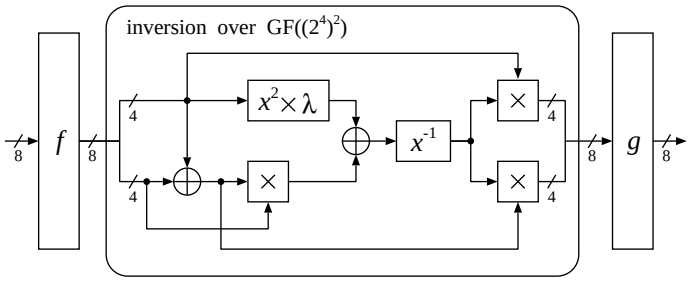

Figure 5.2: Schematic design of S-box implementation

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.

For Type-H5 implementation described in Sec. 5.2.2, the circuits required for multiplication by the matrix $\mathcal{M}_0$ and $\mathcal{M}_1$ are merged. 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 $\mathcal{M}_i$ $(i = 0, 1)$ can be computed through the following equations.

$$\begin{cases} a_0 = \{02\} \times x_0 \\ a_1 = \{02\} \times x_1 \\ a_2 = \{02\} \times x_2 \\ a_3 = \{02\} \times x_3 \end{cases} \begin{cases} b_0 = a_1 \oplus x_0 \\ b_1 = a_2 \oplus x_1 \\ b_2 = a_3 \oplus x_2 \\ b_3 = a_0 \oplus x_3 \end{cases} \begin{cases} c_0 = b_0 \oplus a_3 \\ c_1 = b_1 \oplus a_0 \\ c_2 = b_2 \oplus a_1 \\ c_3 = b_3 \oplus a_2 \end{cases} \begin{cases} d_0 = a_3 \oplus x_0 \\ d_1 = a_0 \oplus x_1 \\ d_2 = a_1 \oplus x_2 \\ d_3 = a_2 \oplus x_3 \end{cases}$$

$$\begin{cases} e_0 = \{04\} \times b_0 \\ e_1 = \{04\} \times b_1 \\ e_2 = \{04\} \times b_2 \\ e_3 = \{04\} \times b_3 \end{cases} \begin{cases} f_0 = (i == 1)?\ e_1 : d_3 \\ f_1 = (i == 1)?\ e_2 : d_0 \\ f_2 = (i == 1)?\ e_3 : d_1 \\ f_3 = (i == 1)?\ e_0 : d_2 \end{cases} \begin{cases} y_0 = c_0 \oplus f_0 \\ y_1 = c_1 \oplus f_1 \\ y_2 = c_2 \oplus f_2 \\ y_3 = c_3 \oplus f_3 \end{cases}$$

The total number of XOR gates and 2:1 selector gates required for multiplication by $\mathcal{M}_0/\mathcal{M}_1$ are 160 and 32.

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.

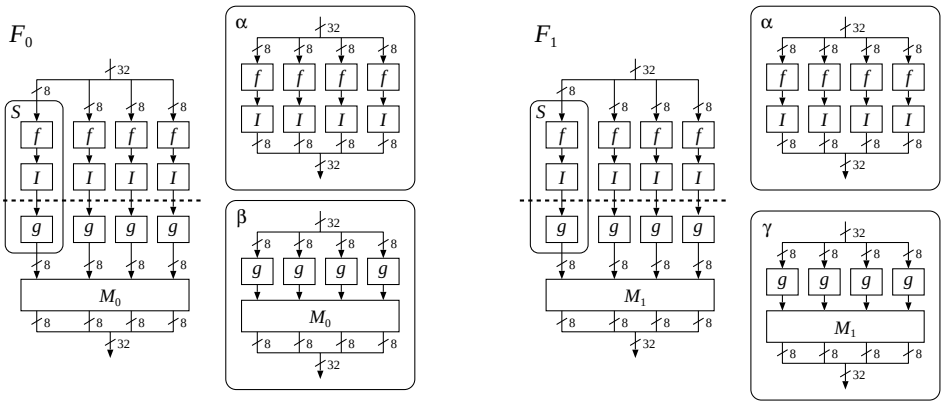

Figure 5.3: Dividing F-functions for pipeline architecture

5.2.2 Implementation Types

This section describes several types of hardware implementations. First, we introduce how to divide F-functions for pipeline architecture. Next, we show five types of hardware implementations for AURORA-256: Type-H1, Type-H2, Type-H3, Type-H4 and Type-H5 implementation. Then, we show three types of hardware implementations for AURORA-512 version 2: Type-H1, Type-H4 and Type-H5 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.

Dividing F-functions for pipeline architecture

In Fig. 5.3, we show the architecture of F-functions F_0 and F_1. 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. We apply pipeline architecture to some types of implementations 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.

AURORA-256 Type-H1

AURORA-256 Type-H1 implementation processes one round of one message scheduling module *MSM* and one chaining value processing module *CPM* in parallel with 1 cycle latency. 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. The 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, the 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 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.

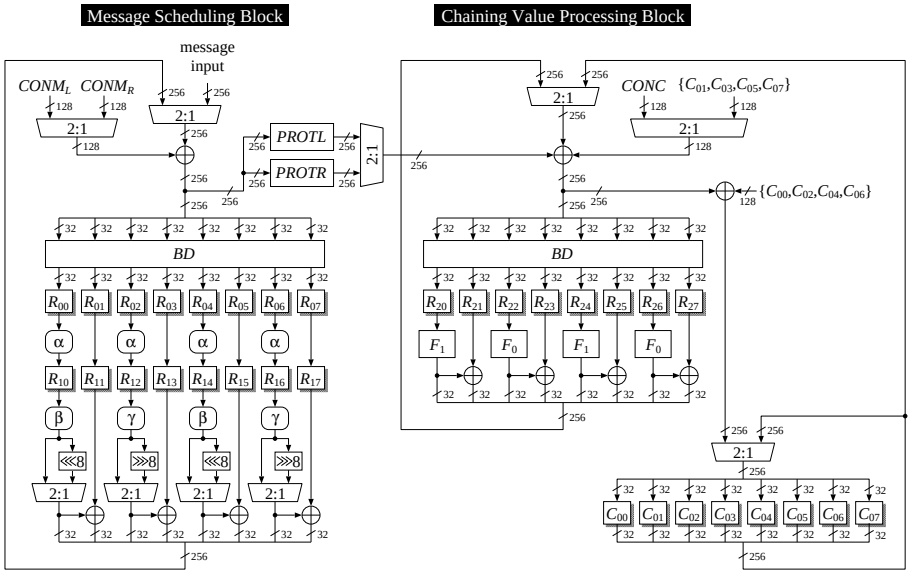

Figure 5.4: Data path architecture of AURORA-256 Type-H1 implementation

We note that the outputs of β and γ are byte-rotated to the left and to the right, respectively, when the 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 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 is 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 one round of one MSM and one CPM in parallel with 2 cycles latency, 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 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

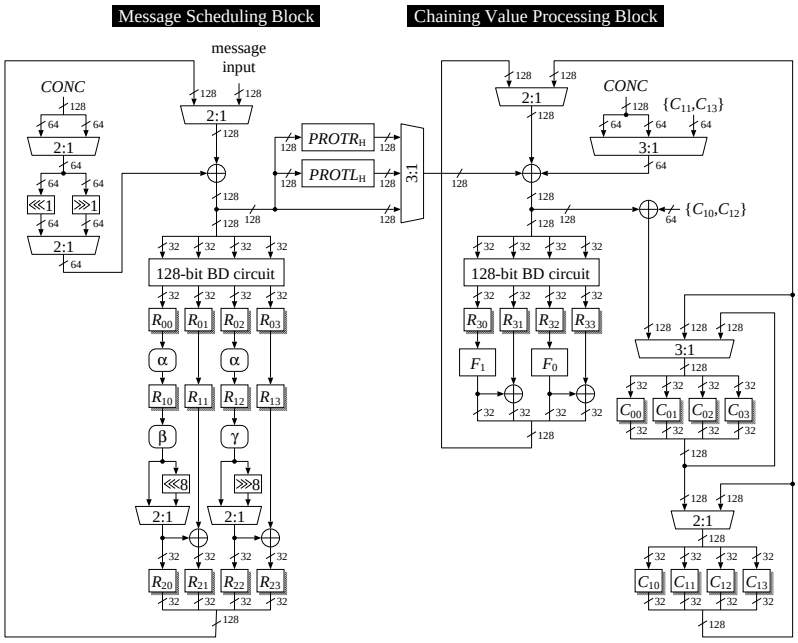

Figure 5.5: Data path architecture of AURORA-256 Type-H2 implementation

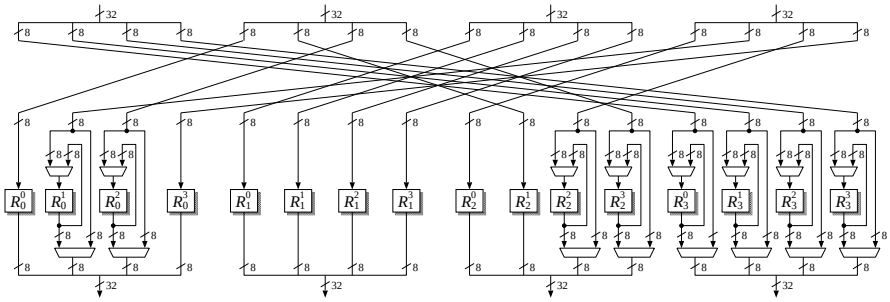

Figure 5.6: 128-bit Byte Diffusion (BD) circuit

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 one round of one MSM or one CPM by turns with 1 cycle latency. 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:

$$MS_L\ (MSF_L) \rightarrow CP\ (CPF) \rightarrow MS_R\ (MSF_R) \rightarrow CP\ (CPF) \rightarrow \cdots$$

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 the 256-bit intermediate values of MS_L (MSF_L) are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$,

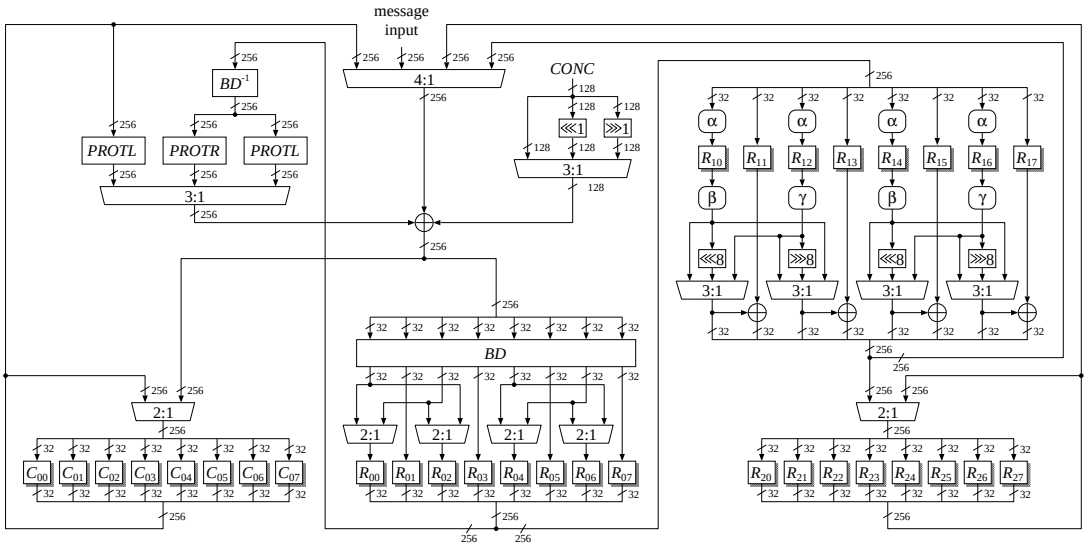

Figure 5.7: Data path architecture of AURORA-256 Type-H3 implementation

$\{R_{10}, \ldots, R_{17}\}$ 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 the 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 the 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 the intermediate values of MS_R (MSF_R), the output of F-functions must be byte-rotated to the left or right. For the 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 the 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 one round of one MSM or one CPM by turns with 2 cycles latency. 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 the intermediate values of MS_L (MSF_L) are stored in registers by repeating the

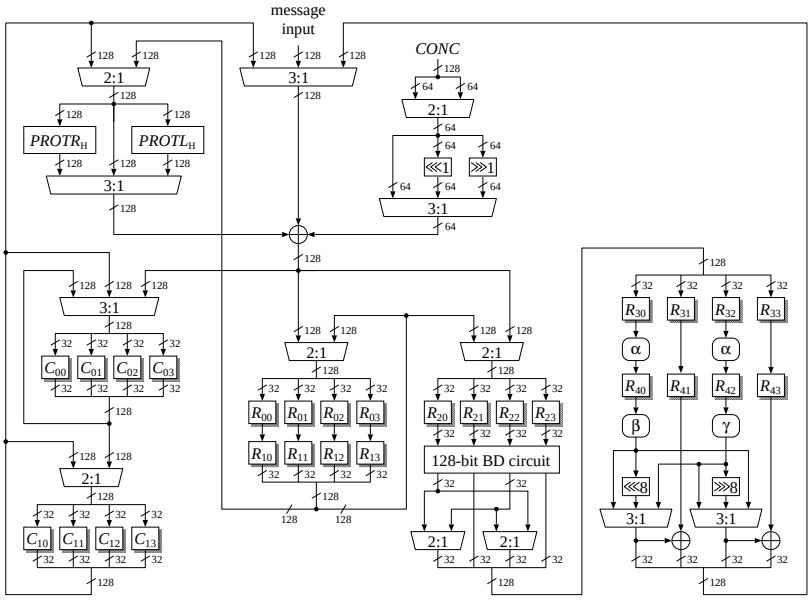

Figure 5.8: Data path architecture of AURORA-256 Type-H4 implementation

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 the 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 the 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 the 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-256 Type-H5

AURORA-256 Type-H5 implementation processes one round of one *MSM* or one *CPM* by turns with 4 cycles latency. It requires only one F-function circuit and takes 144 cycles for both *CF* and *FF*. Fig. 5.9 shows the data path architecture of AURORA-256 Type-H5 implementation, where the data path width is 64 bits. The processing order of the round function as well as basic architecture is the same as Type-H4 implementation except the data path width. The circuits required for multiplication by the matrix $\mathcal{M}_0$ and $\mathcal{M}_1$ are merged by the method described in Sec. 5.2.1.

In a 64-bit data path architecture, the 256-bit byte diffusion function *BD* cannot be implemented only by simple wiring of byte data. We utilize the 64-bit byte diffusion (*BD*) circuit, as shown in Fig. 5.10. It consists of byte wiring, twenty-five 8-bit registers and thirteen 8-bit 2:1 selectors. 256-bit data, which are input into the 64-bit *BD* circuit in four clock cycles, are output in the order corresponding to *BD* by controlling selectors.

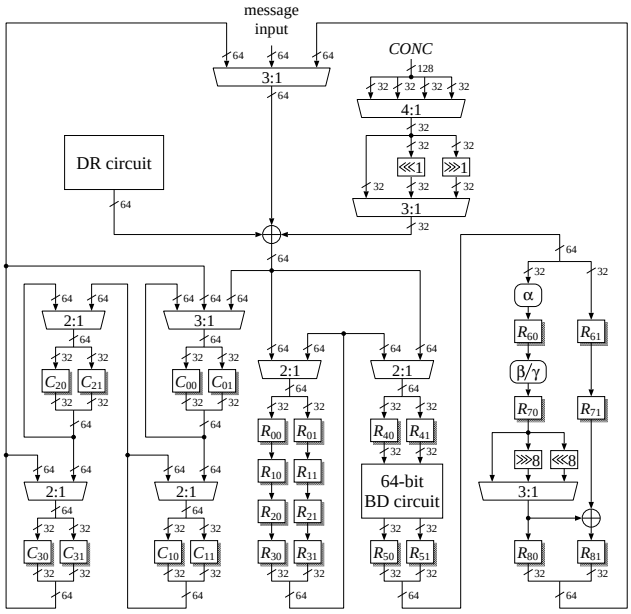

Figure 5.9: Data path architecture of AURORA-256 Type-H5 implementation

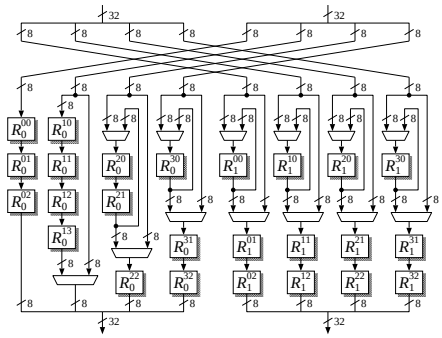

Figure 5.10: 64-bit Byte Diffusion (BD) circuit

The left 256-bit M_L of a 512-bit message block is input in 64-bit blocks from the 1st cycle to the 4th cycle, and then the intermediate values of MS_L (MSF_L) are stored in registers by repeating the following steps:

$$\{R_{00}, R_{01}\} \rightarrow \{R_{10}, R_{11}\} \rightarrow \{R_{20}, R_{21}\} \rightarrow \{R_{30}, R_{31}\} \rightarrow \{R_{00}, R_{01}\} \rightarrow$$
$$\{R_{10}, R_{11}\} \rightarrow \{R_{20}, R_{21}\} \rightarrow \{R_{30}, R_{31}\} \rightarrow \{R_{40}, R_{41}\} \rightarrow \text{64-bit BD circuit} \rightarrow$$
$$\{R_{50}, R_{51}\} \rightarrow \{R_{60}, R_{61}\} \rightarrow \{R_{70}, R_{71}\} \rightarrow \{R_{80}, R_{81}\} \rightarrow \cdots$$

The right 256-bit M_R of a 512-bit message block is input in 64-bit blocks from the 9th cycle to the 12th cycle, and then the 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 two 32-bit registers $\{C_{30}, C_{31}\}$, $\{C_{20}, C_{21}\}$, $\{C_{10}, C_{11}\}$ and $\{C_{00}, C_{01}\}$ is loaded via $\{C_{30}, C_{31}\}$ from the 5th cycle to the 8th cycle, and then the 256-bit intermediate values of CP (CPF) are stored in registers by repeating the following

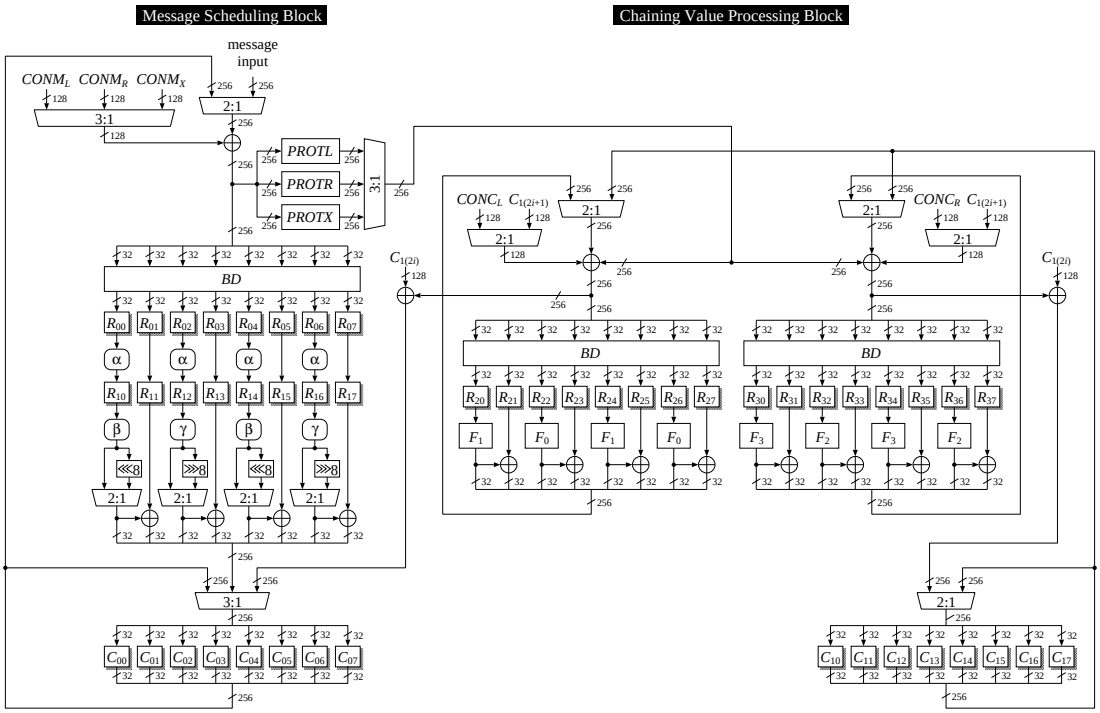

Figure 5.11: Data path architecture of AURORA-512 version 2 Type-H1 implementation

steps:

$$\{R_{40}, R_{41}\} \rightarrow \text{64-bit BD circuit} \rightarrow \{R_{50}, R_{51}\} \rightarrow \{R_{60}, R_{61}\} \rightarrow$$
$$\{R_{70}, R_{71}\} \rightarrow \{R_{80}, R_{81}\} \rightarrow \cdots$$

Note that the chaining value to be fed forward is XORed into the intermediate values of MS_R (MSF_R) in advance from the 137th cycle to the 140th cycle, which can reduce four cycles for updating the chaining value.

AURORA-512 version 2 Type-H1

AURORA-512 version 2 Type-H1 implementation processes one round of one message scheduling module MSM and two chaining value processing modules CPM^{512} in parallel with 1 cycle latency. It requires 12 F-function circuits and takes 27 cycles for both the compression function CF^{512} and the finalization function FF^{512}. Fig. 5.11 shows the data path architecture of AURORA-512 version 2 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 as an input of MS_L^{512} (MSF_L^{512}) and set into eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$ through the byte diffusion circuit BD after being XORed with constant values $CONM_L$. The right 256-bit M_R is input at the 2nd cycle as an input of MS_R^{512} (MSF_R^{512}) and set into $\{R_{00}, \ldots, R_{07}\}$ through BD after being XORed with constant values $CONM_R$. At the 3rd cycle, the left 256-bit of the chaining value X_L stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded as an input of MS_X^{512} (MSF_X^{512}) and set into $\{R_{00}, \ldots, R_{07}\}$ through BD after being XORed with constant values $CONM_X$. The 256-bit intermediate values of MS_L^{512} (MSF_L^{512}), MS_R^{512} (MSF_R^{512}) and MS_X^{512} (MSF_X^{512}) are stored in registers by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \{C_{00}, \ldots, C_{07}\} \rightarrow \cdots$$

The pipeline architecture is introduced into the message scheduling block in order to reduce the rate of increase in area of the message scheduling at high operating frequency. We note that the output of β is byte-rotated to the left, when the 256-bit intermediate values of MS_R^{512} (MSF_R^{512}) are processed; the output of γ is byte-rotated to the right, when the 256-bit intermediate values of MS_R^{512} (MSF_R^{512}) or MS_X^{512} (MSF_X^{512}) are processed.

In the chaining value processing block, the right 256-bit of the chaining value X_R stored in eight 32-bit registers $\{C_{10}, \ldots, C_{17}\}$ is loaded as an input of CP_L^{512} (CPF_L^{512}) and set into eight 32-bit registers $\{R_{20}, \ldots, R_{27}\}$ after being XORed with the data fed from the message scheduling block and constant values $CONC_L$. In parallel, X_R is loaded as an input of CP_R^{512} (CPF_R^{512}) and set into eight 32-bit registers $\{R_{30}, \ldots, R_{37}\}$ after being XORed with the data fed from the message scheduling block and constant values $CONC_R$.

From the 2nd cycle to the 26th cycle, the 256-bit intermediate values of CP_L^{512} (CPF_L^{512}) stored in $\{R_{20}, \ldots, R_{27}\}$ are input to the round function whose F-functions are F_1 and F_0, 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 $CONC_L$. In parallel, the 256-bit intermediate values of CP_R^{512} (CPF_R^{512}) stored in $\{R_{30}, \ldots, R_{37}\}$ are input to the round function whose F-functions are F_3 and F_2, and its output is re-stored into $\{R_{30}, \ldots, R_{37}\}$ through BD after being XORed with the data fed from the message scheduling block and $CONC_R$. The data fed from the message scheduling block pass through the data rotating function $PROTL$, $PROTR$ or $PROTX$

At the 27th cycle, the 256-bit intermediate values of CP_L^{512} (CPF_L^{512}) are input to the round function, and its output are XORed with the data fed from the message scheduling block and X_R stored in $\{C_{10}, \ldots, C_{17}\}$, and then stored into $\{C_{00}, \ldots, C_{07}\}$. In parallel, the 256-bit intermediate values of CP_R^{512} (CPF_R^{512}) are input to the round function, and its output are XORed with the data fed from the message scheduling block and X_R stored in $\{C_{10}, \ldots, C_{17}\}$, and then re-stored into $\{C_{10}, \ldots, C_{17}\}$.

AURORA-512 version 2 Type-H4

AURORA-512 version 2 Type-H4 implementation processes one round of one MSM or one CPM^{512} by turns with 2 cycles latency. It requires 2 F-function circuits and takes 166 cycles for both CF^{512} and FF^{512}: 162 cycle for message scheduling and chaining value processing, and 4 cycles for updating the chaining value. The round function is processed by repeating the following order:

$$
\begin{aligned}
&MS_L^{512}\ (MSF_L^{512}) \rightarrow CP_L^{512}\ (CPF_L^{512}) \rightarrow CP_R^{512}\ (CPF_R^{512}) \rightarrow \\
&MS_R^{512}\ (MSF_R^{512}) \rightarrow CP_L^{512}\ (CPF_L^{512}) \rightarrow CP_R^{512}\ (CPF_R^{512}) \rightarrow \\
&MS_X^{512}\ (MSF_X^{512}) \rightarrow CP_L^{512}\ (CPF_L^{512}) \rightarrow CP_R^{512}\ (CPF_R^{512}) \rightarrow \cdots
\end{aligned}
$$

The pipeline architecture is introduced into the round function circuit, which can improve the maximum operating frequency.

AURORA-512 version 2 Type-H5

AURORA-512 version 2 Type-H5 implementation processes one round of one MSM or one CPM^{512} by turns with 4 cycles latency. It requires only one F-function circuit and takes 332 cycles for both CF^{512} and FF^{512}: 324 cycle for message scheduling and chaining value processing, and 8 cycles for updating the chaining value. The processing order of the round function as well as basic architecture is the same as Type-H5 implementation except the data path width: the data path width of Type-H4 implementation is 128 bits, while that of Type-H5 implementation is 64 bits.

5.2.3 Evaluation Results

We show our current evaluation results on hardware performance of AURORA-256 and AURORA-512 version 2. For AURORA-256, Type-H1, Type-H2, Type-H3, Type-H4 and Type-H5 implementations with S-boxes based on inversion over $\mathrm{GF}((2^4)^2)$ are evaluated. For AURORA-512

version 2, Type-H1, Type-H4 and Type-H5 implementations with S-boxes based on inversion over $GF((2^4)^2)$ are evaluated. In addition, Type-H1 implementation with table-lookup S-boxes is 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 5.12 represents the evaluation results. For each implementation of AURORA-256 and AURORA-512 version 2, 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. "Throughput" is defined as follows:

$$\text{Throughput [Mbps]} = \frac{\text{Frequency [MHz]} \times \text{Block Size (512 [bits])}}{\text{Cycles}}.$$

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. [53]. 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 (8,870 gates) is about 23% smaller with about 28% higher efficiency (122.2 Kbps/gate) than that of SHA-224/256 (11,484 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 version 2, the smallest area (12,389 gates) is about 46% smaller than that of SHA-384/512 (23,146 gates), and the highest throughput (6,901 Mbps) is about 2.37 times higher than that of SHA-224/256 (2,909 Mbps).

The highest efficiency of AURORA-256 (344.3 Kbps/gate) and AURORA-512 version 2 (146.8 Kbps/gate) is about 2.23 times and 1.38 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 5.12: Results on Hardware Performance of AURORA-256 and AURORA-512 version 2

	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
	Type-H5	144	$GF((2^4)^2)$	**8,870**	304.8	1,084	122.2
				9,970	509.3	1,811	181.6
				9,970	509.3	1,811	181.6
SHA-224/256 ($0.13\mu m$) [53]	-	72	-	11,484	154.1	1,096	95.4
				15,329	333.3	2,370	154.6
AURORA-512 version 2 ($0.13\mu m$)	Type-H1	27	$GF((2^4)^2)$	28,968	191.4	3,629	125.3
				41,985	285.6	5,416	129.0
				31,543	244.1	4,629	**146.8**
			Table	42,637	211.4	4,009	94.0
				59,657	363.9	**6,901**	115.7
				47,678	309.3	5,865	123.0
	Type-H4	166	$GF((2^4)^2)$	14,942	310.2	957	64.0
				17,628	504.8	1,557	88.3
				17,012	491.8	1,517	89.2
	Type-H5	332	$GF((2^4)^2)$	**12,389**	302.7	467	37.7
				13,914	509.3	785	56.5
				13,914	509.3	785	56.5
SHA-384/512 ($0.13\mu m$) [53]	-	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 version 2. Bold figures indicate the best results of AURORA-256 and AURORA-512 version 2 in terms of area, throughput and efficiency.

Chapter 6

Applications of AURORA*

6.1 Digital Signature

The digital signature standard (DSS) is specified in FIPS 186-2 [21]. 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 [20]. Due to that the same hash size of SHA-1 is not supported by the AURORA* hash algorithm family, it is not possible to directly replace SHA-1 as a member of the AURORA* family. 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 [22]. 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 [24]. 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 [41]. In this document, minimum bit length of the hash function output is assigned according to the selected parameter set on of FA, FB, FC, EA, EB, EC, ED, and EE.

Table 6.1: The values of L and B.

Algorithm	L	B
AURORA-224	224	512
AURORA-256	256	512
AURORA-384 version 2	384	512
AURORA-512 version 2	512	512

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. Accordingly, AURORA* algorithms can be used when one of the above domain parameters is selected. To be concrete, each 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) [42]. 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 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 version 2 and AURORA-512 version 2, 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 version 2 and AURORA-512 version 2, 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 advantages are the realization of the design goal of AURORA* family. We believe that all advantages 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 [39], 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.

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 the previous version AURORA. Thanks also to external researchers for evaluating AURORA and AURORA* algorithms.

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, sep 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 LNCS, 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 LNCS, 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 LNCS, 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] Niels Ferguson and Stefan Lucks. Attacks on AURORA-512 and the Double-Mix Merkle-Damgaard transform. In *IACR ePrint archive 2009/113*, 2009.

[19] FIPS PUB 140-2. Security requirements for cryptographic modules. Federal Information Processing Standard (FIPS), May 25 2001.

[20] FIPS PUB 180-3. Secure Hash Standard (SHS). Federal Information Processing Standard, Oct 2008.

[21] FIPS PUB 186-2. Digital Signature Standard (DSS). Federal Information Processing Standard, Jan 2000.

[22] FIPS PUB 186-3 Draft. Digital Signature Standard (DSS). Federal Information Processing Standard, March 2006.

[23] FIPS PUB 197. Advanced Encryption Standard. Federal Information Processing Standard (FIPS), November 26 2001.

[24] FIPS PUB 198. The keyed-hash message authentication code (HMAC). Federal Information Processing Standard (FIPS), March 2006.

[25] 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.

[26] Shoichi Hirose. Provably secure double-block-length hash functions in a black-box model. In Choonsik Park and Seongtaek Chee, editors, *Information Security and Cryptology - ICISC 2004, 7th International Conference, Seoul, Korea, December 2-3, 2004, Revised Selected Papers*, volume 3506 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2005.

[27] 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.

[28] Tetsu Iwata, Kyoji Shibutani, Taizo Shirai, Shiho Moriai, and Toru Akishita. AURORA: A cryptographic hash algorithm family. In *Submission to NIST*, 2008.

[29] 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.

[30] Antoine Joux and Stefan Lucks. Improved generic algorithms for 3-collisions. In *IACR ePrint archive 2009/305*, 2009.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[37] 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.

[38] 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.

[39] 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`.

[40] NIST Special Publication 800-106. Draft randomized hashing digital signatures (2nd draft). National Institute of Standards and Technology, August 2008.

[41] 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.

[42] NIST Special Publication 800-90. Recommendation for random number generation using deterministic random bit generators (revised). National Institute of Standards and Technology, March 2007.

[43] Brian Gladman's Home Page. `http://fp.gladman.plus.com/cryptography_technology/sha/sha2-07-01-07.zip`.

[44] The MiniSat Page. `http://minisat.se/`.

[45] 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.

[46] 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.

[47] 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 LNCS, pages 171–184. Springer, 2001.

[48] Yu Sasaki. A 2nd-preimage attack on AURORA-512. In *IACR ePrint archive 2009/112*, 2009.

[49] Yu Sasaki. A collision attack on AURORA-512. In *IACR ePrint archive 2009/106*, 2009.

[50] Yu Sasaki. A full key recovery attack on HMAC-AURORA-512. In *IACR ePrint archive 2009/125*, 2009.

[51] Yu Sasaki and Kazumaro Aoki. Preimage attacks on MD, HAVAL, SHA, and others. CRYPTO2008 rump session, 2008. `http://rump2008.cr.yp.to/efa237568f229268803b82ed02e217ca.pdf`.

[52] 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.

[53] 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.

[54] Taizo Shirai and Kiyomichi Araki. On generalized Feistel structures using the diffusion switching mechanism. *IEICE. Trans. Fundamentals*, E91A(8):2120–2129, 2008.

[55] 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 LNCS, pages 1–15. Springer, 2004.

[56] 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 LNCS, pages 260–278. Springer, 2004.

[57] 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 LNCS, pages 41–56. Springer, 2006.

[58] 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 LNCS, pages 181–195. Springer, 2007.

[59] 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 LNCS, pages 17–36. Springer, 2005.

[60] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Proceedings of EUROCRYPT'05*, number 3494 in LNCS, pages 19–35. Springer, 2005.

[61] 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 LNCS, pages 1–16. Springer, 2005.

[62] 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.

Appendix A

Motivation for the Changes

AURORA is a hash function family which is a first round candidate of SHA-3 competition [39]. AURORA consists of four hash algorithms including AURORA-384 and AURORA-512 which supports 384-bit and 512-bit hash values, respectively.

Since the initial publication of AURORA, a series of analyses on the initial version have been published [49, 48, 18, 50, 30][1] . Collision attacks on AURORA-512 are presented in [49, 18, 30], 2nd-preimage attacks on AURORA-512 are presented in [48, 18], and the key recovery attack on HMAC-AURORA-512 is presented in [50]. Below, we briefly summarize these results.

Collision attacks: [49] shows a collision attack on AURORA-512 with 2^{236} computations of AURORA-512 and $2^{236} \times 512$ bits of memory. [18] shows an attack on AURORA-512 with $2^{234.5}$ computations of AURORA-512 and storing $2^{229.6}$ message blocks. [18] also shows a memory less variant, which requires 2^{249} computations of AURORA-512. [30] shows three collision attacks on AURORA-512. The first one needs 2^{235} computations of AURORA-512 and storing 2^{192} message blocks, the next one needs 2^{242} computations of AURORA-512 and storing 2^{128} message blocks, and the third one needs 2^{245} computations of AURORA-512 and storing 2^{16} message blocks.

2nd-preimage attacks: [48, 18] show 2nd-preimage attacks on AURORA-512. [48] uses 2^{290} computations of AURORA-512 and $2^{288} \times 512$ bits of memory. [18] shows an attack on AURORA-512 with 2^{288} computations of AURORA-512 and storing less than $2^{31.5}$ message blocks. [18] also shows an attack on AURORA-384, which requires 2^{291} computations.

Key recovery on HMAC: [50] shows a key recovery attack on HMAC-AURORA-512. It recovers 512-bit secret key with 2^{257} oracle calls and 2^{259} off-line computations of AURORA-512. The same paper also presents an attack on HMAC-AURORA-384, which recovers the inner-key with 2^{257} oracle calls and 2^{257} off-line computations of AURORA-384, which leads to a universal forgery attacks by combining the inner-key with the 2nd-preimage attacks.

Multi-collision attack: [18] shows a multi-collision attack on AURORA-256M. It creates a 16-collision with $2^{236.5}$ computations, while in the ideal case, creating a 12-collision takes 2^{237} computations. For a memoryless case, finding a 32-collision on AURORA-256M takes $2^{251.38}$ computations, while it takes $2^{251.32}$ computations to find a 31-collision in the ideal case.

All these attacks make use of the structural properties of the DMMD transform (the Double-Mix Merkle-Damgård transform). The DMMD transform does provide a collision resistance beyond $2^{n/2}$, but it is now clear that it does not achieve the optimal resistance of 2^n. Besides, we find that the resistance against 2nd-preimage attacks is much below than the optimal case, and all these attacks motivate us to employ a different mode of operation for our new version of AURORA-384 and AURORA-512.

[1] AURORA-224/256 are also included in the AURORA hash family, but there have not been reported any analytic result by external researchers. Thus we do not modify the specification of AURORA-224/256.

For multicollision attacks, if not optimal, both AURORA-224M and AURORA-256M have better resistance compared to the plain AURORA-224 and AURORA-256. Therefore they may still be advantageous for applications where multicollision resistance beyond $O(2^{112})$ or $O(2^{128})$ is needed.