AURORA:  
A Cryptographic Hash Algorithm Family

Submitters:  
Sony Corporation\textsuperscript{1} and Nagoya University\textsuperscript{2}

Algorithm Designers:  
Tetsu Iwata\textsuperscript{2}, Kyoji Shibutani\textsuperscript{1}, Taizo Shirai\textsuperscript{1}, Shiho Moriai\textsuperscript{1}, Toru Akishita\textsuperscript{1}

January 9, 2009
Executive Summary

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

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

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

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

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

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

1 Introduction 7

2 Specification of AURORA 11
   2.1 Notation .................................................. 11
   2.2 Building Blocks ......................................... 13
      2.2.1 Message Scheduling Module: MSM .............. 13
      2.2.2 Chaining Value Processing Module: CPM .... 13
      2.2.3 Byte Diffusion Function: BD ..................... 15
      2.2.4 F-Functions: F₀, F₁, F₂, and F₃ ............ 17
      2.2.5 Data Rotating Function: DR ..................... 19
   2.3 Specification of AURORA-256 ......................... 21
      2.3.1 Overall Structure .................................. 21
      2.3.2 Compression Function: CF ......................... 21
      2.3.3 Finalization Function: FF .......................... 23
      2.3.4 Alternate Method for Computing CF and FF .... 25
   2.4 Specification of AURORA-224 ........................... 27
   2.5 Specification of AURORA-512 ............................ 28
      2.5.1 Overall Structure .................................. 28
      2.5.2 Compression Functions: CF₀, CF₁, ..., CF₇ .... 28
      2.5.3 Mixing Function: MF ................................ 30
      2.5.4 Mixing Function for Finalization: MFF .......... 32
   2.6 Specification of AURORA-384 ............................ 34
   2.7 Specification of AURORA-256M (optional) ............. 35
      2.7.1 Overall Structure .................................. 35
      2.7.2 Compression Functions: CF₀M, CF₁M, ..., CF₇M .... 35
      2.7.3 Mixing Function: MFₘ ................................. 37
      2.7.4 Mixing Function for Finalization: MFFₘ ........ 37
   2.8 Specification of AURORA-224M (optional) ............ 40
   2.9 Constant Values .......................................... 41
      2.9.1 Constant Values for AURORA-224/256 ......... 41
      2.9.2 Constant Values for AURORA-384/512 .......... 42
      2.9.3 Constant Values for AURORA-224M/256M ....... 43
      2.9.4 List of Constant Values ............................. 44
   2.10 Pseudocodes ............................................. 47
   2.11 AURORA Examples ....................................... 53

3 Design Rationale of AURORA 55
   3.1 AURORA-256 ............................................ 55
      3.1.1 Domain Extension ................................. 55
      3.1.2 Compression Function ............................. 56
   3.2 AURORA-512 ............................................. 56
      3.2.1 Domain Extension – Double-Mix Merkle-Damgard transform 56
3.2.2 Compression Function. .................................................. 58
3.3 AURORA-256M ................................................................. 58
  3.3.1 Domain Extension ...................................................... 58
  3.3.2 Compression Function ................................................ 59
3.4 Components and Constants .............................................. 59
  3.4.1 AURORA Structure .................................................... 59
  3.4.2 F-function ............................................................... 60
  3.4.3 Data Rotating Function .............................................. 63
  3.4.4 Truncation Functions ................................................. 63
  3.4.5 Constant Generation ................................................ 64
  3.4.6 Initial Value ............................................................ 65
4 Security of AURORA .............................................................. 67
  4.1 Expected Strength ......................................................... 67
  4.2 Security Argument ....................................................... 67
    4.2.1 Security of HMAC using AURORA ............................... 67
    4.2.2 Security Proofs of DMMD Transform ......................... 69
    4.2.3 Security Properties of AURORA structure .................. 82
  4.3 Algorithm Analysis ..................................................... 84
    4.3.1 Collision Attacks .................................................. 84
    4.3.2 Preimage Attacks .................................................. 87
    4.3.3 Second Preimage Attacks ...................................... 89
    4.3.4 Length-Extension Attack ...................................... 90
    4.3.5 Multicollision Attack ........................................... 90
    4.3.6 Slide Attacks ...................................................... 90
  4.4 Tunable Security Parameters .......................................... 91
    4.4.1 Number of Rounds ................................................ 91
    4.4.2 Variable Hash Size .............................................. 91
5 Efficient Implementation of AURORA ...................................... 93
  5.1 Software Implementation .............................................. 93
    5.1.1 Implementation Types .......................................... 93
    5.1.2 Evaluation Results .............................................. 99
  5.2 Hardware Implementation ............................................. 107
    5.2.1 Optimization Techniques of F-functions .................... 107
    5.2.2 Data Path Architectures ...................................... 108
    5.2.3 Evaluation Results .............................................. 116
6 Applications of AURORA ...................................................... 119
  6.1 Digital Signature ....................................................... 119
  6.2 Keyed-Hash Message Authentication Code (HMAC) ................ 119
  6.3 Key Establishment Schemes Using Discrete Logarithm Cryptography ................................................. 120
  6.4 Random Number Generation Using Deterministic Random Bit Generators ................................................. 120
7 Advantages and Limitations ............................................... 121
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 we submit as a candidate for a new cryptographic hash algorithm (SHA-3) family.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>max. message size (bits)</th>
<th>message block size (bits)</th>
<th>chaining value size (bits)</th>
<th>hash size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AURORA-224</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>256</td>
<td>224</td>
</tr>
<tr>
<td>AURORA-256</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>AURORA-384</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>512</td>
<td>384</td>
</tr>
<tr>
<td>AURORA-512</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>optional instances</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AURORA-224M</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>AURORA-256M</td>
<td>$512 \times (2^{64} - 1)$</td>
<td>512</td>
<td>256</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

Moreover, the AURORA family achieves high efficiency on many platforms. In software implementation on the NIST reference platform (64-bit), AURORA-256 achieves 15.4 cycles/byte and AURORA-512 achieves 27.4 cycles/byte. Also, AURORA shows good performance across a variety of platforms, because it uses platform-independent operations. In hardware implementation, AURORA enables a variety of implementations, from high-speed to area-restricted implementations. Using a 0.13µm CMOS ASIC library, AURORA-256 can be implemented with only 11.1 Kgates in an area-optimized implementation. In a speed-optimized implementation, AURORA-256 achieves the highest throughput of 10.4 Gbps. For AURORA-512, the smallest size is 14.6 Kgates and the highest throughput is 9.1 Gbps.
Organization of the document. This document is organized as follows: Chapter 2 describes the specification of the AURORA family. Chapter 3 provides the design rationale. Chapter 4 explains all aspects of security: security argument and algorithm analysis. Chapter 5 shows efficient implementation results of AURORA. Chapter 6 describes the usage of AURORA in important applications. Finally, AURORA’s advantages and 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 $y$, $x ← y$ means that the bit string $x$ is updated by the bit string $y$. For an $nl$-bit $x$, we write $(x_0(n), x_1(n), \ldots, x_{l-1}(n)) ← x(nl)$ 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 $l$, $x \ll_n l$ is the $l$-bit left cyclic shift of $x$, and $x \gg_n l$ is the $l$-bit right cyclic shift of $x$.
- For bit strings $x_0, x_1, \ldots, x_{n-1}$, $\{x_j\}_{0 \leq j < n}$ 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 $\alpha$ is a root of an irreducible polynomial, $x_{n-1}||x_{n-2}||\ldots||x_1||x_0$ denotes the bit representation of the polynomial.
Following variables and symbols have specific meanings.

- $M$: The input message.
- $M_i$: The $i$-th block of the message (after the padding).
- $m$: The length of $M$ in blocks (after the padding).
- $H_i$: The $i$-th chaining value.
- $MSM$: The Message Scheduling Module.
- $CPM$: The Chaining value Processing Module.
- $BD$: The Byte Diffusion function.
- $DR$: The Data Rotating function.
- $PROTL$: The Partial ROTating Left function.
- $PROTR$: The Partial ROTating Right function.
- $Pad$: The Padding function.
- $Len_n$: The Length of the input message in blocks encoded into $n$ bits.
- $TF_n$: The Truncation Function that outputs $n$ bits.
- $F_0$, $F_1$, $F_2$, and $F_3$: The F-Functions.
- $M_0$, $M_1$, $M_2$, and $M_3$: The matrices used in the F-functions.
- $S$: The S-box.

Following symbols are used for AURORA-224/256.

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

Following symbols are used for AURORA-384/512.

- $CF_0$, $CF_1$, $CF_2$, $CF_3$, $CF_4$, $CF_5$, $CF_6$, and $CF_7$: The Compression Functions for AURORA-384/512.
- $MS_{L,s}$ and $MS_{R,s}$: The Message Scheduling functions for $CF_s$ ($0 \leq s \leq 7$), $MF$ ($s = 8$), and $MFF$ ($s = 9$).
- $CP_{L,s}$ and $CP_{R,s}$: The Chaining value Processing functions for $CF_s$ ($0 \leq s \leq 7$), $MF$ ($s = 8$), and $MFF$ ($s = 9$).
- $CONM_{L,s,j}$ and $CONM_{R,s,j}$: The CONstants used in $MS_{L,s}$ and $MS_{R,s}$, respectively.
- $CONC_{L,s,j}$ and $CONC_{R,s,j}$: The CONstants used in $CP_{L,s}$ and $CP_{R,s}$, respectively.

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

- $CF^M_0$, $CF^M_1$, $CF^M_2$, $CF^M_3$, $CF^M_4$, $CF^M_5$, $CF^M_6$, and $CF^M_7$: The Compression Functions for AURORA-224M/256M.
- $MF^M$: The Mixing Function for AURORA-224M/256M.
- $MFF^M$: The Mixing Function for Finalization for AURORA-224M/256M.
- $MS_{L,s}^M$ and $MS_{R,s}^M$: The Message Scheduling functions for $CF^M_s$ ($0 \leq s \leq 7$), $MF^M$ ($s = 8$), and $MFF^M$ ($s = 9$).
- $CP_{L,s}^M$ and $CP_{R,s}^M$: The Chaining value Processing functions for $CF^M_s$ ($0 \leq s \leq 7$), $MF^M$ ($s = 8$), and $MFF^M$ ($s = 9$).
- $CONM_{L,s,j}^M$ and $CONM_{R,s,j}^M$: The CONstants used in $MS_{L,s}^M$ and $MS_{R,s}^M$, respectively.
- $CONC_{L,s,j}^M$ and $CONC_{R,s,j}^M$: The CONstants used in $CP_{L,s}^M$ and $CP_{R,s}^M$, respectively.
2.2 Building Blocks

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

2.2.1 Message Scheduling Module: MSM

The message scheduling module, MSM, takes the following two inputs:

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

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

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

\[
\begin{align*}
F : \{0,1\}^{32} &\rightarrow \{0,1\}^{32}, \\
F' : \{0,1\}^{32} &\rightarrow \{0,1\}^{32}.
\end{align*}
\]

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 \( 7 \).

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

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

\[
\begin{align*}
(X_0, X_1, \ldots, X_7) &\leftarrow BD(X_0, X_1, \ldots, X_7) \\
(X_0, X_2, X_4, X_6) &\leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\
(X_1, X_3, X_5, X_7) &\leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\
(Z_{64}, Z_{65}, \ldots, Z_{71}) &\leftarrow (X_0, X_1, \ldots, X_7)
\end{align*}
\]

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

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

2.2.2 Chaining Value Processing Module: CPM

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

- a bit string \( X_{(256)} \),
- a set of bit strings \( \{Y_{j(32)}\}_{0 \leq j < 144} \), and
- a set of bit strings \( \{W_{j(32)}\}_{0 \leq j < 68} \).
Figure 2.1: \( \{Z_{j(32)}\}_{0 \leq j < 72} \leftarrow MSM[F, F'](X_{(256)}, \{Y_{j(32)}\}_{0 \leq j < 32}) \).
The output is a bit string $Z_{(256)}$.

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

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

**Step 1.** Let $(X_0(32), X_1(32), \ldots, X_7(32)) \leftarrow (X_{(256)})$.

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

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

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

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

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

\[
\begin{align*}
(X_0, X_1, \ldots, X_7) &\leftarrow BD(X_0, X_1, \ldots, X_7) \\
(X_0, X_2, X_4, X_6) &\leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6)) \\
(X_1, X_3, X_5, X_7) &\leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\
(X_0, X_1, \ldots, X_7) &\leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{136}, Y_{137}, \ldots, Y_{143})
\end{align*}
\]

**Step 6.** Finally, the output is $Z_{(256)} \leftarrow (X_0(32) \parallel X_1(32) \parallel \cdots \parallel X_7(32))$.

See Fig. 2.22 for an illustration and Fig. 2.14 for a pseudocode.

### 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 $\pi$ defined in Fig. 2.23 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 $X_i(32) = (x'_{4i}(8) \parallel x'_{4i+1}(8) \parallel x'_{4i+2}(8) \parallel x'_{4i+3}(8))$, and the output is $(X_0(32), X_1(32), \ldots, X_7(32))$.

See Fig. 2.24 for an illustration and Fig. 2.15 for a pseudocode.
Figure 2.2: $Z_{(256)} \leftarrow CPM[F, F'][X_{(256)}; \{Y_j(32)\}_{0 \leq j < 144}, \{W_j(32)\}_{0 \leq j < 68}]$. 

---

<table>
<thead>
<tr>
<th>X</th>
<th>BD</th>
<th>BD</th>
<th>BD</th>
<th>BD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>$W_0$</td>
<td>$Y_0$</td>
<td>$F$</td>
<td>$Y_8$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>$W_1$</td>
<td>$Y_1$</td>
<td>$F'$</td>
<td>$Y_9$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>$Y_2$</td>
<td>$Y_10$</td>
<td>$F$</td>
<td>$Y_{18}$</td>
</tr>
<tr>
<td>$X_3$</td>
<td>$Y_3$</td>
<td>$Y_{11}$</td>
<td>$F'$</td>
<td>$Y_{19}$</td>
</tr>
<tr>
<td>$X_4$</td>
<td>$W_2$</td>
<td>$Y_4$</td>
<td>$F$</td>
<td>$Y_{20}$</td>
</tr>
<tr>
<td>$X_5$</td>
<td>$W_3$</td>
<td>$Y_5$</td>
<td>$F'$</td>
<td>$Y_{21}$</td>
</tr>
<tr>
<td>$X_6$</td>
<td>$Y_6$</td>
<td>$Y_7$</td>
<td>$F$</td>
<td>$Y_{22}$</td>
</tr>
<tr>
<td>$X_7$</td>
<td>$W_0$</td>
<td>$Y_8$</td>
<td>$W_4$</td>
<td>$Y_{23}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z</th>
<th>BD</th>
<th>BD</th>
<th>BD</th>
<th>BD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Z_0$</td>
<td>$Z_1$</td>
<td>$Z_2$</td>
<td>$Z_3$</td>
<td>$Z_4$</td>
</tr>
<tr>
<td>$Z_5$</td>
<td>$Z_6$</td>
<td>$Z_7$</td>
<td>$Z_8$</td>
<td>$Z_9$</td>
</tr>
</tbody>
</table>

$Y_{8i} = Y_{8i+1} + Y_{8i+2} + Y_{8i+3} + Y_{8i+4} + Y_{8i+5} + Y_{8i+6} + Y_{8i+7}$
2.2.4 F-Functions: $F_0$, $F_1$, $F_2$, and $F_3$

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

Before defining these F-functions, we first define the S-box $S : \{0, 1\}^8 \rightarrow \{0, 1\}^8$, and four $4 \times 4$ matrices, $M_0, M_1, M_2$ and $M_3$.

- The S-box $S : x(8) \rightarrow y(8)$ is defined as follows.

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

The inverse function is performed in 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 $z^4 + z^4 + 1$. Moreover, $f : x(8) \rightarrow y(8)$ and $g : x(8) \rightarrow y(8)$ are affine transformations over GF(2), which are defined as

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

17
Table 2.1: S

<table>
<thead>
<tr>
<th>.0</th>
<th>.1</th>
<th>.2</th>
<th>.3</th>
<th>.4</th>
<th>.5</th>
<th>.6</th>
<th>.7</th>
<th>.8</th>
<th>.9</th>
<th>.a</th>
<th>.b</th>
<th>.c</th>
<th>.d</th>
<th>.e</th>
<th>.f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>d9</td>
<td>dc</td>
<td>d3</td>
<td>69</td>
<td>bd</td>
<td>00</td>
<td>4d</td>
<td>eb</td>
<td>02</td>
<td>24</td>
<td>57</td>
<td>c2</td>
<td>b8</td>
<td>5d</td>
<td>b7</td>
</tr>
<tr>
<td>1.</td>
<td>f5</td>
<td>40</td>
<td>37</td>
<td>4e</td>
<td>19</td>
<td>d8</td>
<td>64</td>
<td>62</td>
<td>9d</td>
<td>34</td>
<td>0f</td>
<td>7c</td>
<td>ec</td>
<td>ce</td>
<td>94</td>
</tr>
<tr>
<td>2.</td>
<td>d1</td>
<td>8a</td>
<td>74</td>
<td>fb</td>
<td>e7</td>
<td>87</td>
<td>12</td>
<td>23</td>
<td>b5</td>
<td>5c</td>
<td>1a</td>
<td>bb</td>
<td>42</td>
<td>49</td>
<td>18</td>
</tr>
<tr>
<td>3.</td>
<td>11</td>
<td>46</td>
<td>0d</td>
<td>71</td>
<td>67</td>
<td>8f</td>
<td>c6</td>
<td>50</td>
<td>58</td>
<td>fd</td>
<td>4b</td>
<td>a4</td>
<td>cd</td>
<td>51</td>
<td>6c</td>
</tr>
<tr>
<td>4.</td>
<td>ad</td>
<td>63</td>
<td>c9</td>
<td>6b</td>
<td>f7</td>
<td>28</td>
<td>9f</td>
<td>65</td>
<td>2f</td>
<td>5f</td>
<td>61</td>
<td>73</td>
<td>3d</td>
<td>7e</td>
<td>90</td>
</tr>
<tr>
<td>5.</td>
<td>33</td>
<td>e0</td>
<td>ac</td>
<td>26</td>
<td>a1</td>
<td>e3</td>
<td>f3</td>
<td>82</td>
<td>83</td>
<td>75</td>
<td>44</td>
<td>90</td>
<td>13</td>
<td>a9</td>
<td>01</td>
</tr>
<tr>
<td>6.</td>
<td>1e</td>
<td>2d</td>
<td>89</td>
<td>39</td>
<td>e6</td>
<td>9c</td>
<td>0a</td>
<td>54</td>
<td>0c</td>
<td>51</td>
<td>6c</td>
<td>43</td>
<td>ae</td>
<td>db</td>
<td>53</td>
</tr>
<tr>
<td>7.</td>
<td>a6</td>
<td>f4</td>
<td>06</td>
<td>da</td>
<td>e2</td>
<td>78</td>
<td>1d</td>
<td>29</td>
<td>30</td>
<td>e1</td>
<td>35</td>
<td>fc</td>
<td>ed</td>
<td>bc</td>
<td>47</td>
</tr>
<tr>
<td>8.</td>
<td>c0</td>
<td>ab</td>
<td>cc</td>
<td>a8</td>
<td>80</td>
<td>2b</td>
<td>09</td>
<td>b0</td>
<td>93</td>
<td>d4</td>
<td>c5</td>
<td>b3</td>
<td>d0</td>
<td>df</td>
<td>a9</td>
</tr>
<tr>
<td>9.</td>
<td>a0</td>
<td>ba</td>
<td>b6</td>
<td>6e</td>
<td>5e</td>
<td>be</td>
<td>7e</td>
<td>20</td>
<td>f9</td>
<td>22</td>
<td>66</td>
<td>05</td>
<td>d2</td>
<td>cb</td>
<td>c3</td>
</tr>
<tr>
<td>a.</td>
<td>7a</td>
<td>36</td>
<td>2a</td>
<td>d6</td>
<td>b2</td>
<td>f4</td>
<td>e2</td>
<td>78</td>
<td>1d</td>
<td>29</td>
<td>30</td>
<td>e1</td>
<td>35</td>
<td>fc</td>
<td>ed</td>
</tr>
<tr>
<td>b.</td>
<td>fe</td>
<td>76</td>
<td>bf</td>
<td>c4</td>
<td>f2</td>
<td>3e</td>
<td>4a</td>
<td>0b</td>
<td>10</td>
<td>14</td>
<td>f1</td>
<td>ef</td>
<td>a7</td>
<td>27</td>
<td>e5</td>
</tr>
<tr>
<td>c.</td>
<td>de</td>
<td>9b</td>
<td>8d</td>
<td>3c</td>
<td>5b</td>
<td>78</td>
<td>5c</td>
<td>60</td>
<td>8a</td>
<td>79</td>
<td>ee</td>
<td>a5</td>
<td>31</td>
<td>2e</td>
<td>77</td>
</tr>
<tr>
<td>d.</td>
<td>0f</td>
<td>fc</td>
<td>00</td>
<td>fe</td>
<td>5b</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>e.</td>
<td>72</td>
<td>3a</td>
<td>7b</td>
<td>84</td>
<td>f6</td>
<td>32</td>
<td>86</td>
<td>03</td>
<td>b4</td>
<td>38</td>
<td>6f</td>
<td>b9</td>
<td>c1</td>
<td>45</td>
<td>88</td>
</tr>
<tr>
<td>f.</td>
<td>ba</td>
<td>b6</td>
<td>6e</td>
<td>5e</td>
<td>be</td>
<td>7e</td>
<td>20</td>
<td>f9</td>
<td>22</td>
<td>66</td>
<td>05</td>
<td>d2</td>
<td>cb</td>
<td>c3</td>
<td>cf</td>
</tr>
</tbody>
</table>

and

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

(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.

\[
M_0 = \begin{pmatrix} 0x01 & 0x02 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x02 & 0x02 \\ 0x02 & 0x03 & 0x01 & 0x02 \\ 0x02 & 0x02 & 0x03 & 0x01 \end{pmatrix},
\]

(2.4)

\[
M_1 = \begin{pmatrix} 0x01 & 0x06 & 0x08 & 0x02 \\ 0x02 & 0x01 & 0x06 & 0x08 \\ 0x08 & 0x02 & 0x01 & 0x06 \\ 0x06 & 0x08 & 0x02 & 0x01 \end{pmatrix},
\]

(2.5)

\[
M_2 = \begin{pmatrix} 0x03 & 0x01 & 0x02 & 0x02 \\ 0x02 & 0x03 & 0x01 & 0x02 \\ 0x02 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x02 & 0x02 & 0x03 \end{pmatrix},
\]

(2.6)

\[
M_3 = \begin{pmatrix} 0x06 & 0x08 & 0x02 & 0x01 \\ 0x01 & 0x06 & 0x08 & 0x02 \\ 0x02 & 0x01 & 0x06 & 0x08 \\ 0x08 & 0x02 & 0x01 & 0x06 \end{pmatrix}.
\]

(2.7)
Multiplications are operated over $\text{GF}(2^8)$ 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 $Y(32) = (y_0(8) \parallel y_1(8) \parallel y_2(8) \parallel y_3(8))$, where

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

### 2.2.5 Data Rotating Function: DR

The data rotating function, $DR$, takes the following two inputs:

- a set of bit strings $\{X_j(32)\}_{0 \leq j < 72}$, and
- a set of bit strings $\{Y_j(32)\}_{0 \leq j < 72}$.

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

$DR$ uses the following two functions:

$$
\begin{align*}
\text{PROTL} : \{(0, 1)^{32}\}^8 &
\rightarrow \{(0, 1)^{32}\}^8, \\
\text{PROTR} : \{(0, 1)^{32}\}^8 &
\rightarrow \{(0, 1)^{32}\}^8,
\end{align*}
$$

which we define as

$$
\text{PROTL}(X_0(32), X_1(32), \ldots, X_7(32)) = (X_0'(32), X_1'(32), \ldots, X_7'(32)),
$$

where $X'_i = X_i$ for $i = 0, 2, 4, 5, 6, 7$, and $(X'_i \parallel X'_j) = (X_i \parallel X_j) \ll 64$ 1.

Similarly, we define

$$
\text{PROTR}(X_0(32), X_1(32), \ldots, X_7(32)) = (X_0'(32), X_1'(32), \ldots, X_7'(32)),
$$

where $X'_i = X_i$ for $i = 0, 2, 4, 5, 6, 7$, and $(X'_i \parallel X'_j) = (X_i \parallel X_j) \gg 64$ 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 $\{X_j(32)\}_{0 \leq j < 72}$ and $\{Y_j(32)\}_{0 \leq j < 72}$, we define $\{Z_j(32)\}_{0 \leq j < 144}$ by iterating the following operations for $i = 0$ to 8.

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

**Step 2.** The output is $\{Z_j(32)\}_{0 \leq j < 144}$ defined in the above operations.

See Fig. 2.5 for an illustration and Fig. 2.16 for a pseudocode.
| $X_0$ | $X_1$ | $\cdots$ | $X_7$ | $Z_0$ | $Z_1$ | $\cdots$ | $Z_7$ |
| $Y_0$ | $Y_1$ | $\cdots$ | $Y_7$ | $Z_0$ | $Z_1$ | $\cdots$ | $Z_7$ |
| $X_8$ | $X_9$ | $\cdots$ | $X_{15}$ | $Z_8$ | $Z_9$ | $\cdots$ | $Z_{15}$ |
| $Y_8$ | $Y_9$ | $\cdots$ | $Y_{15}$ | $Z_8$ | $Z_9$ | $\cdots$ | $Z_{15}$ |
| $X_{16}$ | $X_{17}$ | $\cdots$ | $X_{23}$ | $Z_{24}$ | $Z_{25}$ | $\cdots$ | $Z_{31}$ |
| $Y_{16}$ | $Y_{17}$ | $\cdots$ | $Y_{23}$ | $Z_{40}$ | $Z_{41}$ | $\cdots$ | $Z_{47}$ |
| $X_{64}$ | $X_{65}$ | $\cdots$ | $X_{71}$ | $Z_{128}$ | $Z_{129}$ | $\cdots$ | $Z_{135}$ |
| $Y_{64}$ | $Y_{65}$ | $\cdots$ | $Y_{71}$ | $Z_{136}$ | $Z_{137}$ | $\cdots$ | $Z_{143}$ |

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

\[
\begin{align*}
CF(\cdot, \cdot) : \{0, 1\}^{256} \times \{0, 1\}^{512} & \rightarrow \{0, 1\}^{256}, \\
FF(\cdot, \cdot) : \{0, 1\}^{256} \times \{0, 1\}^{512} & \rightarrow \{0, 1\}^{256}.
\end{align*}
\]

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 \(Pad(\cdot)\);

\[
Pad(M) = M \mid 1 \mid 0^b \parallel Len_{64},
\]

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 \(|M|/512\) in 64-bit string. That is, \(Len_{64}\) is the length of \(M\) in blocks, where a partial block counts for one block, and \(b\) is the minimal integer such that the total length of \(Pad(M)\) is a multiple of 512 bits. Then \(Pad(M)\) is divided into blocks \(M_0, M_1, \ldots, M_{m-1}\) each of length 512 bits, i.e., we let

\[
(M_0(512), M_1(512), \ldots, M_{m-1}(512)) \leftarrow Pad(M).
\]

**Step 2.** Let \(H_0(256) = 0^{256}\), and compute \(H_1(256), H_2(256), \ldots, H_{m-1}(256)\) by iterating

\[
H_{i+1} \leftarrow CF(H_{i}, M_{i})
\]

for \(i = 0\) to \(m - 2\).

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

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

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

![Figure 2.6: AURORA-256.](image)

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

\[
\begin{align*}
MS_L(\cdot) & : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\
MS_R(\cdot) & : \{0,1\}^{256} \rightarrow (\{0,1\}^{32})^{72}, \\
DR(\cdot, \cdot) & : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} \rightarrow (\{0,1\}^{32})^{144}, \\
CP(\cdot, \cdot) & : (\{0,1\}^{256} \times (\{0,1\}^{32})^{144} \rightarrow (0,1)^{256}.
\end{align*}
\]

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
  \[MS_L(X) = MSM[F_0, F_1](X, \{CONM_{L,j}\}_{0 \leq j < 32}),\]
  \hfill (2.11)
  where $F_0$ and $F_1$ are F-functions defined in Sec. 2.2.4 and $\{CONM_{L,j}\}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.2.4.

- Similarly, for any $X \in \{0,1\}^{256}$, $MS_R$ is defined as
  \[MS_R(X) = MSM[F_2, F_3](X, \{CONM_{R,j}\}_{0 \leq j < 32}),\]
  \hfill (2.12)
  where $F_2$ and $F_3$ are F-functions defined in Sec. 2.2.4 and $\{CONM_{R,j}\}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.2.4.

- $DR$ is the data rotating function defined in Sec. 2.2.4.

- $CP$ 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
  \[CP(X,Y) = CPM[F_1, F_0](X,Y, \{CONC_{j}\}_{0 \leq j < 68}),\]
  \hfill (2.13)
  where $F_0$ and $F_1$ are F-functions defined in Sec. 2.2.4 and $\{CONC_{j}\}_{0 \leq j < 68}$ is the set of constants defined in Sec. 2.2.4.

**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 $\{T_{L,j}(32)\}_{0 \leq j < 72} \leftarrow MS_L(M_L(256))$.

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

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

**Step 5.** Let $Y(256) \leftarrow CP(X(256), \{U_j(32)\}_{0 \leq j < 144})$.

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

See Fig. 2.27 for an illustration and Fig. 2.18 for a pseudocode.
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.

\[
\begin{align*}
MSF_L(\cdot) : \{0, 1\}^{256} & \rightarrow (\{0, 1\}^{32})^{72}, \\
MSF_R(\cdot) : \{0, 1\}^{256} & \rightarrow (\{0, 1\}^{32})^{72}, \\
DR(\cdot, \cdot) : (\{0, 1\}^{32})^{72} \times (\{0, 1\}^{32})^{72} & \rightarrow (\{0, 1\}^{32})^{144}, \\
CPF(\cdot, \cdot) : \{0, 1\}^{256} \times (\{0, 1\}^{32})^{144} & \rightarrow \{0, 1\}^{256}.
\end{align*}
\] (2.14)

These functions are described below.

Components of FF

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

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

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

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

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

where \( F_2 \) and \( F_3 \) are F-functions defined in Sec. 2.2.4 and \( \{CONM_{R,j}(32)\}_{32 \leq j < 64} \) is the set of constants defined in Sec. 2.9.
• \( DR \) is the data rotating function defined in Sec. 2.2.5.

• For any \( X \in \{0,1\}^{256} \) and \( Y \in \{(0,1)^{32}\}^{144} \), \( CPF \) is defined as

\[
CPF(X,Y) = \text{CPM}\left[F_1, F_0\right](X,Y, \{\text{CONC}_j^{(32)}\}_{68 \leq j < 136}).
\]  

\[ (2.17) \]

where \( F_0 \) and \( F_1 \) are F-functions defined in Sec. 2.2.4 and \( \{\text{CONC}_j^{(32)}\}_{68 \leq j < 136} \) is the set of constants defined in Sec. 2.9.

**Specification of FF**

Now the finalization function \( FF \) works as follows.

**Step 1.** Let \((M_L^{(256)}, M_R^{(256)}) \leftarrow M_{m-1}^{(512)}\), and let \(X^{(256)} \leftarrow H^{(m-1)(256)}\).

**Step 2.** Let \(\{T_{L,j}^{(32)}\}_{0 \leq j < 72} \leftarrow \text{MSF}_L(M_L^{(256)})\).

**Step 3.** Let \(\{T_{R,j}^{(32)}\}_{0 \leq j < 72} \leftarrow \text{MSF}_R(M_R^{(256)})\).

**Step 4.** Let \(\{U_j^{(32)}\}_{0 \leq j \leq 144} \leftarrow \text{DR}(\{T_{L,j}^{(32)}\}_{0 \leq j < 72}, \{T_{R,j}^{(32)}\}_{0 \leq j < 72})\).

**Step 5.** Let \(Y^{(256)} \leftarrow CPF(X^{(256)}, \{U_j^{(32)}\}_{0 \leq j < 144})\).

**Step 6.** Finally, the output is \(H^{(m)(256)} \leftarrow Y^{(256)} \oplus X^{(256)}\).

See Fig. 2.19 for a pseudocode.
2.3.4 Alternate Method for Computing $CF$ and $FF$

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

**Components $RoundC$, $RoundM_L$ and $RoundM_R$**

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

$$RoundC^{(i)}(X_0, X_1, \ldots, X_T) :$$

\[
\begin{align*}
(X_0, X_1, \ldots, X_T) & \leftarrow BD(X_0, X_1, \ldots, X_T) \\
(X_0, X_2, X_4, X_6) & \leftarrow (F_0(X_0), F_0(X_2), F_1(X_4), F_0(X_6)) \\
\text{If } i \neq 17, \text{ do the following line} \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONC_4i, CONC_{4i+1}, CONC_{4i+2}, CONC_{4i+3}) \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\
\text{Output } (X_0, X_1, \ldots, X_T)
\end{align*}
\]

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

$$RoundM^{(i)}_L(X_0, X_1, \ldots, X_T) :$$

\[
\begin{align*}
(X_0, X_1, \ldots, X_T) & \leftarrow BD(X_0, X_1, \ldots, X_T) \\
(X_0, X_2, X_4, X_6) & \leftarrow (F_0(X_0), F_1(X_2), F_0(X_4), F_1(X_6)) \\
\text{If } i \neq 8, \text{ do the following line} \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM^{L}_{4i}, CONM^{L}_{4i+1}, CONM^{L}_{4i+2}, CONM^{L}_{4i+3}) \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\
\text{Output } (X_0, X_1, \ldots, X_T)
\end{align*}
\]

$$RoundM^{(i)}_R(X_0, X_1, \ldots, X_T) :$$

\[
\begin{align*}
(X_0, X_1, \ldots, X_T) & \leftarrow BD(X_0, X_1, \ldots, X_T) \\
(X_0, X_2, X_4, X_6) & \leftarrow (F_0(X_0), F_0(X_2), F_2(X_4), F_0(X_6)) \\
\text{If } i \neq 8, \text{ do the following line} \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM^{R}_{4i}, CONM^{R}_{4i+1}, CONM^{R}_{4i+2}, CONM^{R}_{4i+3}) \\
(X_1, X_3, X_5, X_7) & \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6) \\
\text{Output } (X_0, X_1, \ldots, X_T)
\end{align*}
\]

Alternative Specification of $CF$

Now we present an alternative computation method of $CF$.

**Step 1.** Initialize input values.

\[
\begin{align*}
\{X_0^{(32)}, X_1^{(32)}, \ldots, X_7^{(32)}, Y_0^{(32)}, Y_1^{(32)}, \ldots, Y_7^{(32)}\} & \leftarrow M_i^{(512)} \\
\{Z_0^{(32)}, Z_1^{(32)}, \ldots, Z_7^{(32)}\} & \leftarrow H_i^{(256)}
\end{align*}
\]

**Step 2.** Add constant values to the initial values.

\[
\begin{align*}
\{X_1, X_3, X_5, X_7\} & \leftarrow (X_1, X_3, X_5, X_7) \oplus (CONM^{L}_{0}, CONM^{L}_{1}, CONM^{L}_{2}, CONM^{L}_{3}) \\
\{Y_1, Y_3, Y_5, Y_7\} & \leftarrow (Y_1, Y_3, Y_5, Y_7) \oplus (CONM^{R}_{0}, CONM^{R}_{1}, CONM^{R}_{2}, CONM^{R}_{3}) \\
\{Z_1, Z_3, Z_5, Z_7\} & \leftarrow (Z_1, Z_3, Z_5, Z_7) \oplus (CONC_{0}, CONC_{1}, CONC_{2}, CONC_{3})
\end{align*}
\]

25
Step 3. Do the first round function.

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

Step 4. The following operations are iterated for \( j = 1 \) to 8.

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

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 \( (X'_1 \parallel X'_3) = (X_1 \parallel X_3) \ll 64 \) and \( (Y'_1 \parallel Y'_3) = (Y_1 \parallel Y_3) \gg 64 \).

Alternative Specification of \( FF \)

An alternative specification of \( FF \) is obtained by replacing constants in the specification of \( CF \) as

\[
\begin{align*}
CONC_j &\leftarrow CONC_{j+32} \\
CONM_{L,j} &\leftarrow CONM_{L,j+32} \\
CONM_{R,j} &\leftarrow CONM_{R,j+32}
\end{align*}
\]
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 \( \text{Pad} \), the compression function \( \text{CF} \), and the finalization function \( \text{FF} \) 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 \( \text{FF} \) is truncated to 224 bits by the truncation function \( \text{TF}_{224} \).

The truncation function, \( \text{TF}_{224}(\cdot) : \{0,1\}^{256} \rightarrow \{0,1\}^{224} \), 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

\[
\begin{align*}
    m'_i &= m_i & \text{for } 0 \leq i \leq 6 \\
    m'_i &= m_{i+1} & \text{for } 7 \leq i \leq 13 \\
    m'_i &= m_{i+2} & \text{for } 14 \leq i \leq 20 \\
    m'_i &= m_{i+3} & \text{for } 21 \leq i \leq 27
\end{align*}
\]

Now we describe the specification of AURORA-224.

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

\[
(M_0(512), M_1(512), \ldots, M_{m-1}(512)) \leftarrow \text{Pad}(M).
\]

**Step 2.** Let \( H_0(256) = 1^{256} \), and compute \( H_1(256), H_2(256), \ldots, H_{m-1}(256) \) by iterating

\[
H_{i+1} \leftarrow \text{CF}(H_i, M_i)
\]

for \( i = 0 \) to \( m - 2 \).

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

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

See Fig. 2.20 for a pseudocode.
2.5 Specification of AURORA-512

2.5.1 Overall Structure

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

\[
\begin{align*}
&CF_s(\cdot, \cdot) : \{0, 1\}^{512} \times \{0, 1\}^{512} \to \{0, 1\}^{512} \text{ for } s \in \{0, 1, \ldots, 7\}, \\
&MF(\cdot) : \{0, 1\}^{512} \to \{0, 1\}^{512}, \\
&MFF(\cdot) : \{0, 1\}^{512} \to \{0, 1\}^{512}.
\end{align*}
\]

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

Now we describe the specification of AURORA-512.

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

\[
(M_0(512), M_1(512), \ldots, M_{m-1}(512)) \leftarrow Pad(M).
\]

Step 2. Now let \(H_0(512) \leftarrow 0^{512}\). Then compute \(H_1(512), H_2(512), \ldots, H_m(512)\) by iterating the following operations for \(i = 0\) to \(m - 1\).

\[
\begin{align*}
H_{i+1} &\leftarrow CF_{i \mod 8}(H_i, M_i) \\
&\text{if } (0 < i < m - 1) \wedge (i \mod 8 = 7) \text{ then } H_{i+1} \leftarrow MF(H_{i+1})
\end{align*}
\]

Step 3. Finally, the output is \(H_m(512) \leftarrow MFF(H_m(512))\).

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

2.5.2 Compression Functions: \(CF_0, CF_1, \ldots, CF_7\)

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

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

\[
\begin{align*}
&MS_{L,s}(\cdot) : \{0, 1\}^{256} \to \{(0, 1)^{32}\}^{72}, \\
&MS_{R,s}(\cdot) : \{0, 1\}^{256} \to \{(0, 1)^{32}\}^{72}, \\
&DR(\cdot, \cdot) : \{(0, 1)^{32}\}^{72} \times \{(0, 1)^{32}\}^{72} \to \{(0, 1)^{32}\}^{144}, \\
&CP_{L,s}(\cdot, \cdot) : \{0, 1\}^{256} \times \{(0, 1)^{32}\}^{144} \to \{0, 1\}^{256}, \\
&CP_{R,s}(\cdot, \cdot) : \{0, 1\}^{256} \times \{(0, 1)^{32}\}^{144} \to \{0, 1\}^{256}.
\end{align*}
\]

These functions are defined below.

Components of \(CF_0, CF_1, \ldots, CF_7\)

\[\text{• For any } X \in \{0, 1\}^{256}, \text{ } MS_{L,s}(X) \text{ is defined as}\]

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

where \(F_0\) and \(F_1\) are F-functions defined in Sec. 2.2.4, and \(\{CONM_{L,s,j}(32)\}_{0 \leq j < 32}\) is the set of constants defined in Sec. 2.4.
Figure 2.8: AURORA-512, where $l = m \mod 8$. 
For any $X \in \{0,1\}^{256}$, $MS_{R,s}$ is defined as

$$MS_{R,s}(X) = MSM[F_2, F_3](X, \{CONM_{R,s,j}(32)\}_{0 \leq j < 32}),$$  \hspace{1cm} (2.19)

where $F_2$ and $F_3$ are F-functions defined in Sec. 2.2.4 and $\{CONM_{R,s,j}(32)\}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.7.

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

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

$$CP_{L,s}(X,Y) = CPM[F_0, F_1](X,Y, \{CONC_{L,s,j}(32)\}_{0 \leq j < 68}),$$ \hspace{1cm} (2.20)

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

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

$$CP_{R,s}(X,Y) = CPM[F_3, F_2](X,Y, \{CONC_{R,s,j}(32)\}_{0 \leq j < 68}),$$ \hspace{1cm} (2.21)

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

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

Now the compression function $CF_s$ works as follows.

**Step 1.** Let $(M_L(256), M_R(256)) \leftarrow M_i(512)$, and $(X_L(256), X_R(256)) \leftarrow H_i(512)$.

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

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

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

**Step 5.** Let $Y_L(256) \leftarrow CP_{L,s}(X_L(256), \{U_j(32)\}_{0 \leq j < 144})$.

**Step 6.** Let $Y_R(256) \leftarrow CP_{R,s}(X_R(256), \{U_j(32)\}_{0 \leq j < 144})$.

**Step 7.** Finally, the output is $H_{i+1}(512) \leftarrow (Y_L(256) \oplus X_L(256) \parallel Y_R(256) \oplus X_R(256))$.

See Fig. 2.39 for an illustration and Fig. 2.22 for a pseudocode.

### 2.5.3 Mixing Function: $MF$

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

\[
\begin{align*}
MS_{L,s}(\cdot) : \{0,1\}^{256} &\rightarrow (\{0,1\}^{32})^{72}, \\
MS_{R,s}(\cdot) : \{0,1\}^{256} &\rightarrow (\{0,1\}^{32})^{72}, \\
DR(\cdot, \cdot) : (\{0,1\}^{32})^{72} \times (\{0,1\}^{32})^{72} &\rightarrow (\{0,1\}^{32})^{144}, \\
CP_{L,s}(\cdot, \cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} &\rightarrow \{0,1\}^{256}, \\
CP_{R,s}(\cdot, \cdot) : \{0,1\}^{256} \times (\{0,1\}^{32})^{144} &\rightarrow \{0,1\}^{256}.
\end{align*}
\]

These functions are defined below.
Components of $MF$

- For any $X \in \{0, 1\}^{256}$, $MS_{L,s}$ is defined as
  \[
  MS_{L,s}(X) = MSM[F_0, F_1](X, \{CONM_{L,s,j}(32)\}_{0 \leq j < 32}),
  \]
  where $F_0$ and $F_1$ are F-functions defined in Sec. 2.2.4 and $\{CONM_{L,s,j}(32)\}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0, 1\}^{256}$, $MS_{R,s}$ is defined as
  \[
  MS_{R,s}(X) = MSM[F_2, F_3](X, \{CONM_{R,s,j}(32)\}_{0 \leq j < 32}),
  \]
  where $F_2$ and $F_3$ are F-functions defined in Sec. 2.2.4 and $\{CONM_{R,s,j}(32)\}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.9.

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

- For any $X \in \{0, 1\}^{256}$ and $Y \in \{(0,1)^{32}\}^{144}$, $CP_{L,s}$ is defined as
  \[
  CP_{L,s}(X,Y) = CPM[F_0, F_1](X,Y, \{CONC_{L,s,j}(32)\}_{0 \leq j < 68}),
  \]
  where $F_0$ and $F_1$ are F-functions defined in Sec. 2.2.4 and $\{CONC_{L,s,j}(32)\}_{0 \leq j < 68}$ is the set of constants defined in Sec. 2.9.

- For any $X \in \{0, 1\}^{256}$ and $Y \in \{(0,1)^{32}\}^{144}$, $CP_{R,s}$ is defined as
  \[
  CP_{R,s}(X,Y) = CPM[F_2, F_3](X,Y, \{CONC_{R,s,j}(32)\}_{0 \leq j < 68}),
  \]
  where $F_2$ and $F_3$ are F-functions defined in Sec. 2.2.4 and $\{CONC_{R,s,j}(32)\}_{0 \leq j < 68}$ is the set of constants defined in Sec. 2.9.
Specification of MF

Now we describe the specification of MF.

Step 1. Let \((X_L(256), X_R(256)) \leftarrow H_i(512)\).

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

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

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

Step 5. Let \(Y_L(256) \leftarrow CP_{L,8}(X_L(256), \{U_j(32)\}_{0 \leq j < 144})\).

Step 6. Let \(Y_R(256) \leftarrow CP_{R,8}(X_R(256), \{U_j(32)\}_{0 \leq j < 144})\).

Step 7. Finally, the output is \(H_i(512) \leftarrow (Y_L(256) \oplus X_L(256)) || Y_R(256) \oplus X_R(256))\).

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

![Diagram](image)

Figure 2.10: \(H_i(512) \leftarrow MF(H_i(512))\).

2.5.4 Mixing Function for Finalization: MFF

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

\[
\begin{align*}
MS_{L,9} : \{0,1\}^{256} \times \{(0,1)^{32}\}^{32} &\rightarrow \{(0,1)^{32}\}^{72}, \\
MS_{R,9} : \{0,1\}^{256} \times \{(0,1)^{32}\}^{32} &\rightarrow \{(0,1)^{32}\}^{72}, \\
DR : \{(0,1)^{32}\}^{72} &\rightarrow \{(0,1)^{32}\}^{144}, \\
CP_{L,9} : \{0,1\}^{256} \times \{(0,1)^{32}\}^{144} \times \{(0,1)^{32}\}^{68} &\rightarrow \{0,1\}^{256}, \\
CP_{R,9} : \{0,1\}^{256} \times \{(0,1)^{32}\}^{144} \times \{(0,1)^{32}\}^{68} &\rightarrow \{0,1\}^{256}.
\end{align*}
\]  

These functions are defined below.

**Components of MFF**

- For any $X \in \{0,1\}^{256}$, $MS_{L,9}$ is defined as
\[
MS_{L,9}(X) = MSM[F_0, F_1|(X, \{CONM_{L,9,j}(32)\}_{0 \leq j < 32})],
\]  

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

- Similarly, for any $X \in \{0,1\}^{256}$, $MS_{R,9}$ is defined as
\[
MS_{R,9}(X) = MSM[F_2, F_3|(X, \{CONM_{R,9,j}(32)\}_{0 \leq j < 32})],
\]  

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

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

- For any $X \in \{0,1\}^{256}$ and $Y \in \{(0,1)^{32}\}^{144}$, $CP_{L,9}$ is defined as
\[
CP_{L,9}(X, Y) = CPM[F_0, F_1|(X, Y, \{CONC_{L,9,j}(32)\}_{0 \leq j < 68})],
\]  

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

- For any $X \in \{0,1\}^{256}$ and $Y \in \{(0,1)^{32}\}^{144}$, $CP_{R,9}$ is defined as
\[
CP_{R,9}(X, Y) = CPM[F_3, F_2|(X, Y, \{CONC_{R,9,j}(32)\}_{0 \leq j < 68})],
\]  

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

**Specification of MFF**

Now we describe the specification of MFF.

**Step 1.** Let $(X_L(256), X_R(256)) \leftarrow H_m(512)$.

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

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

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

**Step 5.** Let $Y_L(256) \leftarrow CP_{L,9}(X_L(256), \{U_{j}(32)\}_{0 \leq j < 144})$.

**Step 6.** Let $Y_R(256) \leftarrow CP_{R,9}(X_R(256), \{U_{j}(32)\}_{0 \leq j < 144})$.

**Step 7.** Finally, the output is $H_m(512) \leftarrow (Y_L(256) \oplus X_L(256) \parallel Y_R(256) \oplus X_R(256))$.

See Fig. 2.24 for a pseudocode.

33
2.6 Specification of AURORA-384

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

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

The truncation function, $TF_{384}(\cdot) : \{0, 1\}^{512} \rightarrow \{0, 1\}^{384}$, 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

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

Now we describe the specification of AURORA-384.

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

\[
(M_0(512), M_1(512), \ldots, M_{m-1}(512)) \leftarrow Pad(M).
\]

**Step 2.** Let $H_0(512) = 1^{512}$, and compute $H_1(512), H_2(512), \ldots, H_m(512)$ by iterating

\[
\begin{align*}
  H_{i+1} &\leftarrow CF_{i \mod 8}(H_i, M_i) \\
  \text{if } (0 < i < m - 1) \land (i \mod 8 = 7) \text{ then } H_{i+1} &\leftarrow MF(H_{i+1})
\end{align*}
\]

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

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

See Fig. 2.26 for a pseudocode.
2.7 Specification of AURORA-256M (optional)

2.7.1 Overall Structure

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

\[
\begin{align*}
CF_s^M(\cdot) : & \{0,1\}^{512} \times \{0,1\}^{512} \rightarrow \{0,1\}^{512} \text{ for } s \in \{0,1,\ldots,7\}, \\
MF^M(\cdot) : & \{0,1\}^{512} \rightarrow \{0,1\}^{512}, \\
MFF^M(\cdot) : & \{0,1\}^{512} \rightarrow \{0,1\}^{512}.
\end{align*}
\]

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

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

Now we describe the specification of AURORA-256M.

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

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

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

\[
\begin{align*}
H_{i+1} \leftarrow & CF_{i \mod 8}(H_i, M_i) \\
& \text{if } (0 < i < m - 1) \land (i \mod 8 = 7) \text{ then } H_{i+1} \leftarrow MF^M(H_{i+1})
\end{align*}
\]

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

See Fig. 2.11 for an illustration and Fig. 2.20 for a pseudocode.

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

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

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

- $CONM_{L,s,j}^M(32)$ for $MS_{L,s}^M$ instead of $CONM_{L,s,j}^M(32)$ for $MS_{L,s}$,
- $CONM_{R,s,j}^M(32)$ for $MS_{R,s}^M$ instead of $CONM_{R,s,j}^M(32)$ for $MS_{R,s}$,
- $CONC_{L,s,j}^M(32)$ for $CP_{L,s}^M$ instead of $CONC_{L,s,j}^M(32)$ for $CP_{L,s}$, and
- $CONC_{R,s,j}^M(32)$ for $CP_{R,s}^M$ instead of $CONC_{R,s,j}^M(32)$ for $CP_{R,s}$.
Figure 2.11: AURORA-256M, where $l = m \mod 8$. 
The constants, $CONM_{L,s,j}^M(32)$, $CONM_{R,s,j}^M(32)$, $CONC_{L,s,j}^M(32)$, and $CONC_{R,s,j}^M(32)$ are all defined in Sec. 2.9. Below, we present the specification, and show the pseudocode in Fig. 2.27 for completeness.

\begin{align}
MSM_{L,s}^M(X) &= MSM[F_0, F_1][X, \{CONM_{L,s,j}^M(32)\}_{0 \leq j < 32}], \quad (2.31) \\
MSM_{R,s}^M(X) &= MSM[F_2, F_3][X, \{CONM_{R,s,j}^M(32)\}_{0 \leq j < 32}], \quad (2.32) \\
CPM_{L,s}^M(X, Y) &= CPM[F_1, F_0][X, Y, \{CONC_{L,s,j}^M(32)\}_{0 \leq j < 68}], \quad (2.33) \\
CPM_{R,s}^M(X, Y) &= CPM[F_3, F_2][X, Y, \{CONC_{R,s,j}^M(32)\}_{0 \leq j < 68}]. \quad (2.34)
\end{align}

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

### 2.7.3 Mixing Function: $MFM^M$

The mixing function $MFM^M$ is used to mix the chaining values every after eight calls of $CPM^M_s$. It takes the chaining value $H_i$ of 512 bits and outputs the updated chaining value $H_{i+1}$ of 512 bits. It internally uses two message scheduling functions $MSM_{L,8}$ and $MSM_{R,8}$, a data rotating function $DR$, and two chaining value processing functions $CPM_{L,8}^M$ and $CPM_{R,8}^M$. These functions are equivalent to the corresponding functions in Sec. 2.5.3 for AURORA-512, where we use

- $CONM_{L,8,j}^M(32)$ for $MSM_{L,8}^M$ instead of $CONM_{L,8,j}^M(32)$ for $MSM_{L,8}$,
- $CONM_{R,8,j}^M(32)$ for $MSM_{R,8}^M$ instead of $CONM_{R,8,j}^M(32)$ for $MSM_{R,8}$,
- $CONC_{L,8,j}^M(32)$ for $CPM_{L,8}^M$ instead of $CONC_{L,8,j}^M(32)$ for $CPM_{L,8}$, and
- $CONC_{R,8,j}^M(32)$ for $CPM_{R,8}^M$ instead of $CONC_{R,8,j}^M(32)$ for $CPM_{R,8}$.

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

\begin{align}
MSM_{L,8}^M(X) &= MSM[F_0, F_1][X, \{CONM_{L,8,j}^M(32)\}_{0 \leq j < 32}], \quad (2.35) \\
MSM_{R,8}^M(X) &= MSM[F_2, F_3][X, \{CONM_{R,8,j}^M(32)\}_{0 \leq j < 32}], \quad (2.36) \\
CPM_{L,8}^M(X, Y) &= CPM[F_1, F_0][X, Y, \{CONC_{L,8,j}^M(32)\}_{0 \leq j < 68}], \quad (2.37) \\
CPM_{R,8}^M(X, Y) &= CPM[F_3, F_2][X, Y, \{CONC_{R,8,j}^M(32)\}_{0 \leq j < 68}]. \quad (2.38)
\end{align}

### 2.7.4 Mixing Function for Finalization: $MFF^M$

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

\begin{equation}
\begin{cases}
MSM_{R,9}^M : \{0, 1\}^{256} \times \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}, \\
DR : \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}, \\
CPM_{L,9}^M : \{0, 1\}^{256} \times \{0, 1\}^{32} \times \{0, 1\}^{32} \rightarrow \{0, 1\}^{256}.
\end{cases} \quad (2.39)
\end{equation}

These functions are defined below.
Components of MFF\textsuperscript{M}

- For any $X \in \{0, 1\}^{256}$, $MS_{L,9}^{M}$ is defined as
  \[
  MS_{R,0}^{M}(X) = MSM[F_{2}, F_{3}](X, \{ CONM_{R,9,j}^{M} \}_{0 \leq j < 32}),
  \]
  where $F_{2}$ and $F_{3}$ are F-functions defined in Sec. 2.2.4 and $\{ CONM_{R,9,j}^{M} \}_{0 \leq j < 32}$ is the set of constants defined in Sec. 2.9.

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

- For any $X \in \{0, 1\}^{256}$ and $Y \in (\{0, 1\}^{32})^{144}$, $CP_{L,9}^{M}$ is defined as
  \[
  CP_{L,9}^{M}(X, Y) = CPM[F_{1}, F_{0}](X, Y, \{ CONC_{L,9,j}^{M} \}_{0 \leq j < 68}),
  \]
  where $F_{0}$ and $F_{1}$ are F-functions defined in Sec. 2.2.4 and $\{ CONC_{L,9,j}^{M} \}_{0 \leq j < 68}$ is the set of constants defined in Sec. 2.9.

Specification of MFF\textsuperscript{M}

Now we describe the specification of MFF\textsuperscript{M}.

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

**Step 2.** Let $T_{L,j}^{\leq 32}$ for $0 \leq j < 72$.

**Step 3.** Let $\{ T_{R,j}^{\leq 32} \}_{0 \leq j < 72} \leftarrow MS_{R,0}^{M}(Y_{(256)})$.

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

**Step 5.** Let $Z_{(256)} \leftarrow CP_{L,9}^{M}(X_{(256)}, \{ U_{j}^{(32)} \}_{0 \leq j < 144})$.

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

See Fig. 2.12 for an illustration and Fig. 2.29 for a pseudocode.
Figure 2.12: $H'_m(256) \leftarrow MFF^M(H_m(512))$. Note that $T_{L,j} = 0^{32}$ for $0 \leq j < 72$. 
2.8 Specification of AURORA-224M (optional)

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

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

Now we describe the specification of AURORA-224M.

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

$$(M_0(512), M_1(512), \ldots, M_{m-1}(512)) \leftarrow Pad(M).$$

**Step 2.** Now let $H_0(512) \leftarrow 1^{512}$. Then compute $H_1(512), H_2(512), \ldots, H_m(512)$ by iterating the following operations for $i = 0$ to $m - 1$.

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

**Step 3.** Let $H'_m(256) \leftarrow MFF^M(H_m(512))$.

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

See Fig. 2.30 for a pseudocode.
2.9 Constant Values

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

2.9.1 Constant Values for AURORA-224/256

Following constants are used in AURORA-224/256;

- \( \{CONM_{L,j}\}_{0 \leq j < 32} \), \( \{CONM_{R,j}\}_{0 \leq j < 32} \), \( \{CONC_j\}_{0 \leq j < 68} \) for CF, and
- \( \{CONM_{L,j}\}_{32 \leq j < 64} \), \( \{CONM_{R,j}\}_{32 \leq j < 64} \), \( \{CONC_j\}_{68 \leq j < 136} \) 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 \( x^{16} + x^{15} + x^{11} + x^5 + x^4 + 1 \), which is 0x1a831.

**Step 1.** Let \( IV_0 \), \( IV_1 \), \( mask_0 \), \( mask_1 \), \( mask_2 \) and \( mask_3 \) be the following values.

\[
\begin{align*}
IV_0 &\leftarrow (2^{1/2} - 1)2^{16} = 0x6a09 \\
IV_1 &\leftarrow (3^{1/2} - 1)2^{16} = 0xbb67 \\
mask_0 &\leftarrow (2^{1/3} - 1)2^{16} = 0x428a \\
mask_1 &\leftarrow (3^{1/3} - 1)2^{16} = 0x7137 \\
mask_2 &\leftarrow (2^{1/5} - 1)2^{16} = 0x2611 \\
mask_3 &\leftarrow (3^{1/5} - 1)2^{16} = 0x3ee8
\end{align*}
\]

**Step 2.** The following operations are iterated for \( i = 0 \) to 16.

\[
\begin{align*}
T_{0,i} &\leftarrow IV_0 \cdot 0x0002^i \\
T_{1,i} &\leftarrow IV_1 \cdot 0x0002^{-i} \\
CONC_{4i} &\leftarrow (T_{0,i} \oplus mask_0) \| T_{0,i} \lll 16 \ 8) \\
CONC_{4i+1} &\leftarrow (T_{1,i} \oplus mask_1) \| T_{1,i} \lll 16 \ 8) \\
CONC_{4i+2} &\leftarrow (T_{0,i} \lll 16 \ 8 \| T_{0,i} \oplus mask_2) \\
CONC_{4i+3} &\leftarrow (T_{1,i} \lll 16 \ 9 \| T_{1,i} \oplus mask_3)
\end{align*}
\]

**Step 3.** The following operations are iterated for \( i = 0 \) to 7.

\[
\begin{align*}
CONM_{L,4i} &\leftarrow CONC_{8i} \lll 32 \ 1 \\
CONM_{L,4i+1} &\leftarrow CONC_{8i+1} \lll 32 \ 1 \\
CONM_{L,4i+2} &\leftarrow CONC_{8i+2} \lll 32 \ 1 \\
CONM_{L,4i+3} &\leftarrow CONC_{8i+3} \lll 32 \ 1 \\
CONM_{R,4i} &\leftarrow CONC_{8i+4} \ggg 32 \ 1 \\
CONM_{R,4i+1} &\leftarrow CONC_{8i+5} \ggg 32 \ 1 \\
CONM_{R,4i+2} &\leftarrow CONC_{8i+6} \ggg 32 \ 1 \\
CONM_{R,4i+3} &\leftarrow CONC_{8i+7} \ggg 32 \ 1
\end{align*}
\]

**Step 4.** The following operations are iterated for \( i = 0 \) to 16.

\[
\begin{align*}
CONC_{4i+68} &\leftarrow CONC_{4i} \\
CONC_{4i+69} &\leftarrow CONC_{4i+1} \\
CONC_{4i+70} &\leftarrow CONC_{4i+2} \\
CONC_{4i+71} &\leftarrow CONC_{4i+3} \oplus 0x01010101
\end{align*}
\]
Step 5. The following operations are iterated for $i = 0$ to $7$.

\[
\begin{align*}
CONM_{L,4i+32} & \leftarrow CONM_{L,4i} \\
CONM_{L,4i+33} & \leftarrow CONM_{L,4i+1} \\
CONM_{L,4i+34} & \leftarrow CONM_{L,4i+2} \\
CONM_{L,4i+35} & \leftarrow CONM_{L,4i+3} \\
CONM_{R,4i+32} & \leftarrow CONM_{R,4i} \\
CONM_{R,4i+33} & \leftarrow CONM_{R,4i+1} \\
CONM_{R,4i+34} & \leftarrow CONM_{R,4i+2} \\
CONM_{R,4i+35} & \leftarrow CONM_{R,4i+3}
\end{align*}
\]

### 2.9.2 Constant Values for AURORA-384/512

Following constants are used in AURORA-384/512:

- \{CONM_{L,s,j}\}_{0\leq j<32}, \{CONM_{R,s,j}\}_{0\leq j<68}, \{CONC_{L,s,j}\}_{0\leq j<68} for CF_{s}$, where $s = 0, 1, \ldots, 7$,
- \{CONM_{L,s,j}\}_{0\leq j<32}, \{CONM_{R,s,j}\}_{0\leq j<68}, \{CONC_{L,s,j}\}_{0\leq j<68} for MF$,
- \{CONM_{L,s,j}\}_{0\leq j<32}, \{CONM_{R,s,j}\}_{0\leq j<68}, \{CONC_{L,s,j}\}_{0\leq j<68} for MFF$.

These constants are generated with the procedure described below.

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

\[
\begin{align*}
IV_0^{512} & \leftarrow (11^{1/2} - 3)2^{16} = 0x510e \\
IV_1^{512} & \leftarrow (13^{1/2} - 3)2^{16} = 0x9b05 \\
mask_0^{512} & \leftarrow (11^{1/2} - 2)2^{16} = 0x3956 \\
mask_1^{512} & \leftarrow (13^{1/2} - 2)2^{16} = 0x59f1 \\
mask_2^{512} & \leftarrow (11^{1/2} - 1)2^{16} = 0x9d8a \\
mask_3^{512} & \leftarrow (13^{1/2} - 1)2^{16} = 0xab97
\end{align*}
\]

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

\[
\begin{align*}
T_0^{512} & \leftarrow IV_0^{512} \cdot 0x0002^i \\
T_1^{512} & \leftarrow IV_1^{512} \cdot 0x0002^{-i} \\
CONC_{L,0,i} & \leftarrow (T_{0,i}^{512} \oplus mask_0^{512}) || T_{0,i}^{512} \ll 16 \ll 8) \\
CONC_{L,0,i+1} & \leftarrow (T_{1,i}^{512} \oplus mask_1^{512}) || T_{1,i}^{512} \ll 16 \ll 8) \\
CONC_{L,0,i+2} & \leftarrow (T_{0,i}^{512} \ll 16 \ll 8 || T_{0,i}^{512} \oplus mask_2^{512}) \\
CONC_{L,0,i+3} & \leftarrow (T_{1,i}^{512} \ll 16 \ll 9 || T_{1,i}^{512} \oplus mask_3^{512})
\end{align*}
\]

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

\[
CONC_{R,0,i} \leftarrow CONC_{L,0,i} \lll 32
\]

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

\[
\begin{align*}
CONM_{L,0,i} & \leftarrow CONC_{L,0,8i} \lll 32 \\
CONM_{L,0,i+1} & \leftarrow CONC_{L,0,8i+1} \lll 32 \\
CONM_{L,0,i+2} & \leftarrow CONC_{L,0,8i+2} \lll 32 \\
CONM_{L,0,i+3} & \leftarrow CONC_{L,0,8i+3} \lll 32 \\
CONM_{R,0,i} & \leftarrow CONC_{L,0,8i+4} \ggg 32 \\
CONM_{R,0,i+1} & \leftarrow CONC_{L,0,8i+5} \ggg 32 \\
CONM_{R,0,i+2} & \leftarrow CONC_{L,0,8i+6} \ggg 32 \\
CONM_{R,0,i+3} & \leftarrow CONC_{L,0,8i+7} \ggg 32
\end{align*}
\]
Step 5. The following operations are iterated for $i = 0$ to 16 and for $s = 1$ to 9.

\[
\begin{align*}
& CONC_{L,s,4i} \leftarrow CONC_{L,0,4i} \\
& CONC_{L,s,4i+1} \leftarrow CONC_{L,0,4i+1} \\
& CONC_{L,s,4i+2} \leftarrow CONC_{L,0,4i+2} \\
& CONC_{L,s,4i+3} \leftarrow CONC_{L,0,4i+3} \oplus CONS_s \\
& CONC_{R,s,4i} \leftarrow CONC_{R,0,4i} \\
& CONC_{R,s,4i+1} \leftarrow CONC_{R,0,4i+1} \\
& CONC_{R,s,4i+2} \leftarrow CONC_{R,0,4i+2} \\
& CONC_{R,s,4i+3} \leftarrow CONC_{R,0,4i+3} \oplus CONS_s \\
\end{align*}
\]

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

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

\[
\begin{align*}
& CONM_{L,s,4i} \leftarrow CONM_{L,0,4i} \\
& CONM_{L,s,4i+1} \leftarrow CONM_{L,0,4i+1} \\
& CONM_{L,s,4i+2} \leftarrow CONM_{L,0,4i+2} \\
& CONM_{L,s,4i+3} \leftarrow CONM_{L,0,4i+3} \\
& CONM_{R,s,4i} \leftarrow CONM_{R,0,4i} \\
& CONM_{R,s,4i+1} \leftarrow CONM_{R,0,4i+1} \\
& CONM_{R,s,4i+2} \leftarrow CONM_{R,0,4i+2} \\
& CONM_{R,s,4i+3} \leftarrow CONM_{R,0,4i+3} \\
\end{align*}
\]

2.9.3 Constant Values for AURORA-224M/256M

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

- $\{CONM^M_{L,s,j}\}_{0 \leq j < 32}$, $\{CONM^M_{R,s,j}\}_{0 \leq j < 32}$, $\{CONC^M_{L,s,j}\}_{0 \leq j < 68}$, $\{CONC^M_{R,s,j}\}_{0 \leq j < 68}$ for $CF^M_s$, where $s = 0, 1, \ldots, 7$,

- $\{CONM^M_{L,8,j}\}_{0 \leq j < 32}$, $\{CONM^M_{R,8,j}\}_{0 \leq j < 32}$, $\{CONC^M_{L,8,j}\}_{0 \leq j < 68}$, $\{CONC^M_{R,8,j}\}_{0 \leq j < 68}$ for $MF^M$, and

- $\{CONM^M_{R,9,j}\}_{0 \leq j < 32}$, $\{CONC^M_{L,9,j}\}_{0 \leq j < 68}$ for $MF^M$.

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

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

\[
\begin{align*}
& IV^M_0 \leftarrow (5^{1/2} - 2)2^{16} = 0x3c6e \\
& IV^M_1 \leftarrow (7^{1/2} - 2)2^{16} = 0xa54f \\
& mask^M_0 \leftarrow (5^{1/3} - 1)2^{16} = 0xb5c0 \\
& mask^M_1 \leftarrow (7^{1/3} - 1)2^{16} = 0xe9b5 \\
& mask^M_2 \leftarrow (5^{1/5} - 1)2^{16} = 0x6135 \\
& mask^M_3 \leftarrow (7^{1/5} - 1)2^{16} = 0x79cc \\
\end{align*}
\]

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

\[
\begin{align*}
& T^M_{0,i} \leftarrow IV^M_0 \cdot 0x0002^i \\
& T^M_{1,i} \leftarrow IV^M_1 \cdot 0x0002^{-i} \\
& CONC^M_{L,0,4i} \leftarrow (T^M_{0,i} \oplus mask^M_0) \parallel T^M_{1,i} \ll (16, 8) \\
& CONC^M_{L,0,4i+1} \leftarrow (T^M_{1,i} \oplus mask^M_1) \parallel T^M_{1,i} \ll (16, 8) \\
& CONC^M_{L,0,4i+2} \leftarrow (T^M_{0,i} \ll (16, 8) \parallel T^M_{0,i} \oplus mask^M_2) \\
& CONC^M_{L,0,4i+3} \leftarrow (T^M_{1,i} \ll (16, 9) \parallel T^M_{1,i} \oplus mask^M_3) \\
\end{align*}
\]
Step 3. The following operation is iterated for \(i = 0\) to 67.

\[ CONC_{R,0,i}^M \leftarrow CONC_{L,0,i}^M \ll 32 \]

Step 4. The following operations are iterated for \(i = 0\) to 7.

\[
\begin{align*}
CONC_{L,0,4i}^M & \leftarrow CONC_{L,0,4i+1}^M \ll 32 \\
CONC_{L,0,4i+1}^M & \leftarrow CONC_{L,0,4i+2}^M \ll 32 \\
CONC_{L,0,4i+2}^M & \leftarrow CONC_{L,0,4i+3}^M \ll 32 \\
CONC_{R,0,4i}^M & \leftarrow CONC_{R,0,4i+1}^M \gg 32 \\
CONC_{R,0,4i+1}^M & \leftarrow CONC_{R,0,4i+2}^M \gg 32 \\
CONC_{R,0,4i+2}^M & \leftarrow CONC_{R,0,4i+3}^M \gg 32 \\
\end{align*}
\]

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

\[
\begin{align*}
CONC_{L,s,4i}^M & \leftarrow CONC_{L,0,4i}^M \\
CONC_{L,s,4i+1}^M & \leftarrow CONC_{L,0,4i+1}^M \\
CONC_{L,s,4i+2}^M & \leftarrow CONC_{L,0,4i+2}^M \\
CONC_{L,s,4i+3}^M & \leftarrow CONC_{L,0,4i+3}^M \oplus CONS_s \\
CONC_{R,s,4i}^M & \leftarrow CONC_{R,0,4i}^M \\
CONC_{R,s,4i+1}^M & \leftarrow CONC_{R,0,4i+1}^M \\
CONC_{R,s,4i+2}^M & \leftarrow CONC_{R,0,4i+2}^M \\
CONC_{R,s,4i+3}^M & \leftarrow CONC_{R,0,4i+3}^M \oplus CONS_s \\
\end{align*}
\]

Each \(CONS_s\) is the same as in AURORA-384/512.

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

\[
\begin{align*}
CONC_{L,s,4i}^M & \leftarrow CONC_{L,0,4i}^M \\
CONC_{L,s,4i+1}^M & \leftarrow CONC_{L,0,4i+1}^M \\
CONC_{L,s,4i+2}^M & \leftarrow CONC_{L,0,4i+2}^M \\
CONC_{L,s,4i+3}^M & \leftarrow CONC_{L,0,4i+3}^M \\
CONC_{R,s,4i}^M & \leftarrow CONC_{R,0,4i}^M \\
CONC_{R,s,4i+1}^M & \leftarrow CONC_{R,0,4i+1}^M \\
CONC_{R,s,4i+2}^M & \leftarrow CONC_{R,0,4i+2}^M \\
CONC_{R,s,4i+3}^M & \leftarrow CONC_{R,0,4i+3}^M \\
\end{align*}
\]

2.9.4 List of Constant Values

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

<table>
<thead>
<tr>
<th>Constant Values for AURORA-224/256 CF {CONS_{i}}_{0 \leq j \leq 68}</th>
</tr>
</thead>
<tbody>
<tr>
<td>2883f695</td>
</tr>
<tr>
<td>429f1eaf</td>
</tr>
<tr>
<td>42deabff</td>
</tr>
<tr>
<td>43daafe</td>
</tr>
<tr>
<td>47cabff</td>
</tr>
<tr>
<td>578affea</td>
</tr>
<tr>
<td>16afffbb</td>
</tr>
<tr>
<td>babbe07</td>
</tr>
<tr>
<td>f22c59df</td>
</tr>
</tbody>
</table>

44
## Constant Values for AURORA-224/256 CF $CONM_{L,j}$ for $0 \leq j < 32$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5107ed2a</td>
<td>5107ed2a</td>
</tr>
<tr>
<td>85bd57fe</td>
<td>85bd57fe</td>
</tr>
<tr>
<td>8f957f4f</td>
<td>8f957f4f</td>
</tr>
<tr>
<td>2d15ff56</td>
<td>2d15ff56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9ad5d386</td>
<td>9ad5d386</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-224/256 CF $CONM_{R,j}$ for $0 \leq j < 32$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb4c7695</td>
<td>cb4c7695</td>
</tr>
<tr>
<td>a112bf0f</td>
<td>a112bf0f</td>
</tr>
<tr>
<td>a4053f5a</td>
<td>a4053f5a</td>
</tr>
<tr>
<td>f5457f6b</td>
<td>f5457f6b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ad533a</td>
<td>89ad533a</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-384/512 CF $CONC_{L,0,j}$ for $0 \leq j < 68$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42c78d73</td>
<td>42c78d73</td>
</tr>
<tr>
<td>66e3a5f1</td>
<td>66e3a5f1</td>
</tr>
<tr>
<td>6f7ed7a9</td>
<td>6f7ed7a9</td>
</tr>
<tr>
<td>899d344f</td>
<td>899d344f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ad533a</td>
<td>89ad533a</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-384/512 CF $CONC_{R,0,j}$ for $0 \leq j < 68$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d0b1e35c</td>
<td>d0b1e35c</td>
</tr>
<tr>
<td>b21f372f</td>
<td>b21f372f</td>
</tr>
<tr>
<td>59b9ebd4</td>
<td>59b9ebd4</td>
</tr>
<tr>
<td>938edc1f</td>
<td>938edc1f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>382ff6b5</td>
<td>382ff6b5</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-384/512 CF $CONM_{L,0,j}$ for $0 \leq j < 32$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db01e35c</td>
<td>db01e35c</td>
</tr>
<tr>
<td>b21f372f</td>
<td>b21f372f</td>
</tr>
<tr>
<td>59b9ebd4</td>
<td>59b9ebd4</td>
</tr>
<tr>
<td>938edc1f</td>
<td>938edc1f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>382ff6b5</td>
<td>382ff6b5</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-224M/256M CF $CONM_{L,0,j}$ for $0 \leq j < 68$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-224M/256M CF $CONM_{R,0,j}$ for $0 \leq j < 68$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
</tbody>
</table>

## Constant Values for AURORA-224M/256M CF $CONM_{M,0,j}$ for $0 \leq j < 68$

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>89ae91c3</td>
<td>89ae91c3</td>
</tr>
<tr>
<td>4478470e</td>
<td>4478470e</td>
</tr>
<tr>
<td>2342d7e9</td>
<td>2342d7e9</td>
</tr>
<tr>
<td>179a4b5d</td>
<td>179a4b5d</td>
</tr>
<tr>
<td>aec5f3e4</td>
<td>aec5f3e4</td>
</tr>
<tr>
<td>ada19ee7</td>
<td>ada19ee7</td>
</tr>
<tr>
<td>Constant Values for AURORA-224M/256M $CF^n_{0}$, ${CONM^n_{R,j}}_{0 \leq j &lt; 68}$</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td></td>
</tr>
<tr>
<td>4d748e1c 67d582d2 71e2eadd 8a56e41c 68e11c3e 785203cb e3c0cf4e f86ff9b9</td>
<td></td>
</tr>
<tr>
<td>23c23872 f795c343 c78c846d 7977745c f40df5a7 b0722303 0a5953a2 b9ff33bb</td>
<td></td>
</tr>
<tr>
<td>1a13eb49 9381d323 14b7bdbc 59bf104c 87ae53d1 82782b33 ac2f2001 a9ff01b7</td>
<td></td>
</tr>
<tr>
<td>bccd22e8 2a40159d dd161772 d4c9a98f ca3bc09a de9cc868 3f646d94 6f245d56</td>
<td></td>
</tr>
<tr>
<td>27fe0476 84366434 fb888051 379007fb bd4f08ea 09a3f0b8 f7141a5b 1e8c8a68</td>
<td></td>
</tr>
<tr>
<td>c869949e 4f6d3afa 6b606fc6 8a06c95a 6d23293c ccce9d7d c5014f02</td>
<td></td>
</tr>
<tr>
<td>762fd725 0df4be8 28dd180 62828ed1 1e57ae4c  e6d77a7f5 1b2b9f8 b1436e38</td>
<td></td>
</tr>
<tr>
<td>cea75c9e bd471109 a3606908 dde53e89 6f46b93b 951f4a22 465c8e9b 6bb616d1</td>
<td></td>
</tr>
<tr>
<td>6d0cf73d a1f3a515 08c3caa3 b5d92238</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant Values for AURORA-224M/256M $CF^n_{0}$, ${CONM^n_{R,j}}_{0 \leq j &lt; 32}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>135d2387 99f560b4 dc78bab8 3e95b907 88f08e1c fde570d0 71e32111 b1e5ddd17</td>
</tr>
<tr>
<td>4684fad2 e4e074c8 052def6f 166fc413 2f3748ba 4a900567 b74586dc f5326a63</td>
</tr>
<tr>
<td>89ff811d 2109d900 7e22014 cde401fe b21a6527 93db4ebe 9a81bf1 6281b329</td>
</tr>
<tr>
<td>5db8ff5c9 93e7f3af 0a3e2446 58a0a3b4 b3a9d727 6f51c442 28d81a42 77794fa2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant Values for AURORA-224M/256M $CF^n_{0}$, ${CONM^n_{R,j}}_{0 \leq j &lt; 32}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>e68e11c3 b785203c ee3c0cf4 bf86ff9b 7f40df5a 3b072230 20a5953a bb9ff33b</td>
</tr>
<tr>
<td>187ae53d 382782b3 1ac2f200 7a9ff01b aca3bc09 8de9cc86 43f646d9 66f245d5</td>
</tr>
<tr>
<td>abdf408e 809a3f0b bf7141a5 81e8c8a6 362d3293 dcccc9d7 c6c6c577 2c5014f0</td>
</tr>
<tr>
<td>c1e57ae4 fed77a75 851b29bf 8b1436e3 b6f46b93 2951f4a2 946c5c8e 16bb616d</td>
</tr>
</tbody>
</table>

46
2.10 Pseudocodes

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

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

<table>
<thead>
<tr>
<th>CPM $[F, F']$</th>
<th>$(X_{j(256)}, {Y_{j(32)}}<em>{0 \leq j &lt; 144}, {W</em>{j(32)}}_{0 \leq j &lt; 68})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow X$</td>
</tr>
<tr>
<td>010</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_0, Y_1, Y_2, Y_3)$</td>
</tr>
<tr>
<td>020</td>
<td>$(Z_0, Z_1, \ldots, Z_7) \leftarrow (X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>for $i \leftarrow 1$ to $16$ do</td>
<td></td>
</tr>
<tr>
<td>040</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>050</td>
<td>$(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$</td>
</tr>
<tr>
<td>060</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_{4i+1}, Y_{4i+2}, Y_{4i+3})$</td>
</tr>
<tr>
<td>070</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$</td>
</tr>
<tr>
<td>080</td>
<td>$(Z_{64}, Z_{65}, \ldots, Z_{71}) \leftarrow (X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>090</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>100</td>
<td>$(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$</td>
</tr>
<tr>
<td>110</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$</td>
</tr>
<tr>
<td>120</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7})$</td>
</tr>
<tr>
<td>030</td>
<td>for $i \leftarrow 1$ to $16$ do</td>
</tr>
<tr>
<td>040</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>050</td>
<td>$(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$</td>
</tr>
<tr>
<td>060</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7})$</td>
</tr>
<tr>
<td>070</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$</td>
</tr>
<tr>
<td>080</td>
<td>$(Z_{64}, Z_{65}, \ldots, Z_{71}) \leftarrow (X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>090</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow BD(X_0, X_1, \ldots, X_7)$</td>
</tr>
<tr>
<td>100</td>
<td>$(X_0, X_2, X_4, X_6) \leftarrow (F(X_0), F'(X_2), F(X_4), F'(X_6))$</td>
</tr>
<tr>
<td>110</td>
<td>$(X_1, X_3, X_5, X_7) \leftarrow (X_1, X_3, X_5, X_7) \oplus (X_0, X_2, X_4, X_6)$</td>
</tr>
<tr>
<td>120</td>
<td>$(X_0, X_1, \ldots, X_7) \leftarrow (X_0, X_1, \ldots, X_7) \oplus (Y_{136}, Y_{137}, \ldots, Y_{143})$</td>
</tr>
<tr>
<td>130</td>
<td>$Z \leftarrow (X_0 | X_1 | \cdots | X_7)$</td>
</tr>
<tr>
<td>140</td>
<td>return $Z_{(256)}$</td>
</tr>
</tbody>
</table>

**Figure 2.14:** A pseudocode of $CPM : \{0, 1\}^{256} \times (\{0, 1\}^{32})^{144} \rightarrow (\{0, 1\}^{32})^{68} \rightarrow \{0, 1\}^{256}$. $BD$ is defined in Sec. 2.2.3. $F$ and $F'$ are functions over $\{0, 1\}^{32}$.
A pseudocode of AURORA-256. The padding function, \( CF \), is defined in Sec. 2.10.

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

\[
\begin{align*}
BD(X_0(32), X_1(32), \ldots, X_7(32)) \\
000 & \text{ for } i \leftarrow 0 \text{ to } 7 \text{ do} \\
010 & (x_{4i}, x_{4i+1}, x_{4i+2}, x_{4i+3}) \leftarrow X_i \\
020 & \text{ for } i \leftarrow 0 \text{ to } 31 \text{ do} \\
030 & x'_{\pi(i)} \leftarrow x_i \\
040 & \text{ for } i \leftarrow 0 \text{ to } 7 \text{ do} \\
050 & X_i \leftarrow (x'_{4i}, x'_{4i+1}, x'_{4i+2}, x'_{4i+3}) \\
060 & \text{return } (X_0(32), X_1(32), \ldots, X_7(32))
\end{align*}
\]

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

\[
\begin{align*}
DR(\{X_j(32)\}_{0 \leq j < 72}, \{Y_j(32)\}_{0 \leq j < 72}) \\
000 & \text{ for } i \leftarrow 0 \text{ to } 8 \text{ do} \\
010 & (Z_{16i}, Z_{16i+1}, \ldots, Z_{16i+7}) \leftarrow PROTL(X_{8i}, X_{8i+1}, \ldots, X_{8i+7}) \\
020 & (Z_{16i+8}, Z_{16i+9}, \ldots, Z_{16i+15}) \leftarrow PROTR(Y_{8i}, Y_{8i+1}, \ldots, Y_{8i+7}) \\
030 & \text{return } \{Z_j(32)\}_{0 \leq j < 144}
\end{align*}
\]

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

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

Figure 2.18: A pseudocode of \( CF : \{0, 1\}^{256} \times \{0, 1\}^{512} \rightarrow \{0, 1\}^{256} \). \( MS_L, MS_R, DR, \) and \( CP \) are defined in (2.11), (2.12), Sec. 2.2.3, and in (2.13), respectively.

\[
\begin{align*}
CF(H_i(256), M_i(512)) \\
000 & (M_L, M_R) \leftarrow M_i \\
010 & X \leftarrow H_i \\
020 & (T_{L,j})_{0 \leq j < 72} \leftarrow MS_L(M_L) \\
030 & (T_{R,j})_{0 \leq j < 72} \leftarrow MS_R(M_R) \\
040 & (U_j)_{0 \leq j \leq 144} \leftarrow DR((T_{L,j})_{0 \leq j < 72}, (T_{R,j})_{0 \leq j < 72}) \\
050 & Y \leftarrow CP(X, (U_j)_{0 \leq j \leq 144}) \\
060 & H_{i+1} \leftarrow Y \oplus X \\
070 & \text{return } H_{i+1}(256)
\end{align*}
\]

48
Figure 2.19: A pseudocode of $FF : \{0,1\}^{256} \times \{0,1\}^{512} \rightarrow \{0,1\}^{256}$. $MSF_L$, $MSF_R$, $DR$, and $CPF$ are defined in (2.15), (2.16), Sec. 2.2.5 and in (2.17), respectively.

AURORA-224($M$)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow \text{Pad}(M)$
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 2.20: A pseudocode of AURORA-224. Pad, CF, and FF are the same as AURORA-256 and defined in Sec. 2.3.

AURORA-512($M$)
000 $(M_0, M_1, \ldots, M_{m-1}) \leftarrow \text{Pad}(M)$
010 $H_0 \leftarrow 0^{512}$
020 for $i \leftarrow 0$ to $m-1$ do
030 $H_{i+1} \leftarrow CF_i \text{mod} \, 8(H_i, M_i)$
040 if $(0 < i < m - 1) \land (i \text{ mod } 8 = 7)$ then
050 $H_{i+1} \leftarrow MF(H_{i+1})$
060 $H_m \leftarrow MFF(H_m)$
070 return $H_m(512)$

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

49
A pseudocode of

\[ CF_i(H_{i(512)}, M_{i(512)}) \]

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

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

\[ MF(H_{i(512)}) \]

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

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

\[ MFF(H_{m(512)}) \]

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

Figure 2.24: A pseudocode of \( MFF : \{0, 1\}^{512} \rightarrow \{0, 1\}^{512} \), \( MS_{L,9}, MS_{R,9}, DR, CP_{L,9}, \) and \( CP_{R,9} \) are defined in (2.27), (2.28), Sec. 2.2.5, (2.29), and in (2.30), respectively.
Figure 2.25: A pseudocode of AURORA-384. Pad, CF₂, MF, and MFF are the same as AURORA-512 and defined in Sec. 2.5.

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

Figure 2.27: A pseudocode of CF^M: \{0, 1\}^{512} \times \{0, 1\}^{512} \rightarrow \{0, 1\}^{512}, MS^M_{L,s}; MS^M_{R,s}, DR, CP^M_{L,s}, and CP^M_{R,s} are defined in (2.31), (2.32), Sec. 2.2.5 (2.33), and in (2.34), respectively.
$MF^M (H_{i(512)})$
000 $(X_L, X_R) \leftarrow H_i$
010 $(T_{L,j})_{0 \leq j < 72} \leftarrow MS_{L,8}^M(X_L)$
020 $(T_{R,j})_{0 \leq j < 72} \leftarrow MS_{R,8}^M(X_R)$
030 $(U_j)_{0 \leq j < 144} \leftarrow DR((T_{L,j})_{0 \leq j < 72}, (T_{R,j})_{0 \leq j < 72})$
040 $Y_L \leftarrow CP_{L,8}^M(X_L, (U_j)_{0 \leq j < 144})$
050 $Y_R \leftarrow CP_{R,8}^M(X_R, (U_j)_{0 \leq j < 144})$
060 $Z_L \leftarrow Y_L \oplus X_L$
070 $Z_R \leftarrow Y_R \oplus X_R$
080 $H_i \leftarrow (Z_L, Z_R)$
090 return $H_i(512)$

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

$MFF^M (H_m(512))$
000 $(X, Y) \leftarrow H_m$
010 for $j \leftarrow 0$ to 71 do
020 $T_{L,j} \leftarrow 0^{32}$
030 $(T_{R,j})_{0 \leq j < 72} \leftarrow MS_{R,9}^M(Y)$
040 $Z \leftarrow CP_{L,9}^M(X, (T_{L,j})_{0 \leq j < 144})$
050 $H_m \leftarrow Z \oplus X$
060 return $H_m(256)$

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

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

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

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

“abcdbcdecdefgefghfghighijhijklmklmnlmnomnopnopq”.

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.

<table>
<thead>
<tr>
<th>AURORA-256</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>3e0c31c1 8ef5c404 33844fac 2d4acdf4 9e390962 797821a4 9e3553f3</td>
<td>8189917e</td>
</tr>
<tr>
<td>$M_2$</td>
<td>21621069 e64ec45a eccf140a d881c684 44c30081 32a3b2d0</td>
<td>e9ad9611 2dc034f</td>
</tr>
<tr>
<td>$M_3$</td>
<td>ec8cede6 3fd1bd3b c6de6702 b6ed25e8 d80f5efa b5433912</td>
<td>446aaefc db026b5f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-224</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>50fddc1c 776012c2 c01cc258 ecc6a10</td>
<td>37646235 860da74b 6e0280af</td>
</tr>
<tr>
<td>$M_2$</td>
<td>05874948 064d42ca e0fffa86 45034160</td>
<td>8d571731 f9581ca8 b8ea1890</td>
</tr>
<tr>
<td>$M_3$</td>
<td>7977bc32 b66d7b05 6b215153</td>
<td>1545668d 5f3d1c6c 42a48334 5ab31f70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-512</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>6a4cf6d1 18619abd e8c920d5 9806e483</td>
<td>cc90616f 8d1b4db6 b98abab7 00c4ec47</td>
</tr>
<tr>
<td>$M_2$</td>
<td>cfb432c3 01103535 f0cf0027 efe2c0c6</td>
<td>2046414e 6128ec83 bbd0cccf 7425f908</td>
</tr>
<tr>
<td>$M_3$</td>
<td>577e573e d9fbfcb1 80bce8a8 2d1e4441</td>
<td>89d31fe0 7cda57d3 a2c8ad00 9800f6ae</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-384</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>cb7a330f 33ab55ec 98698f49 4ace5996</td>
<td>3dcecc8e2 bdfaf12df1 f8db22fc 1bb5591e</td>
</tr>
<tr>
<td>$M_2$</td>
<td>f16bb878 ddee5e5f 51994078 61aee1c</td>
<td>b23c63fb 649f3f8d fbecf41e cf24805f</td>
</tr>
<tr>
<td>$M_3$</td>
<td>c18722f8 d9e0fe10 de818d07 8e667346</td>
<td>c23532ee 7d1d9968 18f60ab0 3950b416</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-256M</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>46c5db6a cfdf333b 7cfb4242 8fe9345e</td>
<td>a0882acb c10c5694 9c248501 b156c457</td>
</tr>
<tr>
<td>$M_2$</td>
<td>3c3353d9 67d30005 de02caae e38a1205</td>
<td>11e3b3a8 3d90438e 5694df40 2bdc9588</td>
</tr>
<tr>
<td>$M_3$</td>
<td>cd97a51f 79cb722a c2c33a46</td>
<td>62502b10 a13565b4 1f662699 11b9b438 f9fe81fb</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-224M</th>
<th>Message</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>d6eaa68 02030670 3e7d6301 74bd2f9b</td>
<td>607a1e95 b6620ba2 5d2a3248</td>
</tr>
<tr>
<td>$M_2$</td>
<td>587879d0 6e6bb1da 87b5de9e 06ed0bdf</td>
<td>24e5fbaa d98bc0dd 1257ad26</td>
</tr>
<tr>
<td>$M_3$</td>
<td>c78f12a4 308821ab 3d312f6d 9df6408</td>
<td>5496a44e a1ae3b6d a734166c</td>
</tr>
</tbody>
</table>

54
Chapter 3

Design Rationale of AURORA

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

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

3.1 AURORA-256

3.1.1 Domain Extension

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

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

We chose the sMD transform with the finalization function, because it preserves CR and indistinguishability (PRO) of the underlying compression function. The collision resistance preservation (CR-Pr) is ensured by the MD strengthening [35]: the input message is padded by the padding function Pad(·) in AURORA. CR-Pr can be proven similarly to the proof in [35]. The pseudorandom oracle preservation (PRO-Pr) is due to the finalization function. The finalization function works to envelope the internal MD iteration as the enveloping mechanism used in NMAC/HMAC constructions [5] and the EMD transform [5]. PRO-Pr can be easily proven from Lemma 5.1 in [5], 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
is expected that $FF$ behaves as a different function from $CF$. On the other hand, $FF$ can be efficiently implemented by using the same module as $CF$.

3.1.2 Compression Function

The AURORA-256 compression function $CF$ uses two message scheduling functions $MS_L$ and $MS_R$, and the chaining value processing function $CP$, as shown in the below of Fig. 3.1. It is regarded as the Davies-Meyer construction [34, p.340]. We chose this construction because it is possible to input a longer message than a chaining value to achieve higher throughput, while in the Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions [35, 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 [36, 45, 17], we attached more importance to achieving higher throughput.

Considering recent attacks on hash functions exploiting simple message scheduling [55, 56, 57], we chose to design more secure (and more heavy) message schedule like Whirlpool [3] and DASH [8]. Each component 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

3.2.1 Domain Extension – Double-Mix Merkle-Damgård transform

In order to achieve an efficient 512-bit hash function, a novel domain extension transform, called the Double-Mix Merkle-Damgård (DMMMD) transform is introduced. The DMMMD transform consists of double lines of compression functions and whole state mixing functions inserted every 8 blocks as in Fig. 3.2 (a similar method was proposed by Nandi [37]). The DMMMD transform enables
an efficient collision-resistant construction for double length hash functions\footnote{They are also called as double-block-length (DBL) hash functions, but we use the term “double length hash functions” following\cite{37}.} which outputs $2n$-bit hash values using component functions with $n$-bit output. We adopted this approach because (1) the same compression function can be used in all the AURORA family, and because (2) the message scheduling functions can be shared between two compression functions by making the best use of the structure of the AURORA compression function.

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

**Security of the DMMD transform.** The collision resistance (CR) and preimage resistance (Pre) of the DMMD transform can be achieved with appropriate assumptions on the underlying components (see Sec.\ref{sec:security} for the proofs).

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

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

**Mixing functions.** In the DMMD transform, the mixing function is inserted at intervals of several calls of the compression functions. The purpose of the mixing function is to mix the two $n$-bit chaining values. The number of blocks the mixing function is inserted effects the security bound. In AURORA-512, the mixing function $\text{MF}$ is inserted every 8 blocks.
Independent instances of compression functions. In order to prove that the DMMD transform has collision resistance, each of the (eight) compression functions between the mixing functions is expected to behave as an independent function. To specify independent compression function instances with limited implementation cost, we use the same components for all compression functions with different sets of constants. For the security proofs of the DMMD transform, see Sec. 4.2.2.

3.2.2 Compression Function

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

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

3.3 AURORA-256M

3.3.1 Domain Extension

AURORA-256M, which outputs 256-bit hash values, is an optional instance with multi-collision resistance (“M” means multi-collision resistance). AURORA-256M has the same structure as AURORA-512 except the final mixing function. Therefore, it has the almost same performance as AURORA-512, which is only about 50% additional cost to AURORA-256, i.e. less than double cost of AURORA-256. Thus AURORA-256M achieves multi-collision resistance very efficiently.
It is known that many iterated hash functions including the Merkle-Damgård construction and its variants allow Joux’s multi-collision attack \cite{26}, Kelsey-Schneier’s second preimage attack \cite{28}, and Kelsey-Kohno’s Herding attack \cite{27}. In particular, Kelsey-Schneier’s second preimage attack on \( n \)-bit iterated hash functions finds a second preimage for a message of \( 2^k \) message blocks with about \( 2^{n-k+1} \) work. (Note that the security requirement for SHA-3 regarding the second preimage resistance is approximately \( n-k \) bits for any message shorter than \( 2^k \) bits, so we understand that multi-collision resistance is not a mandatory requirement.)

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

### 3.3.2 Compression Function

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

### 3.4 Components and Constants

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

#### 3.4.1 AURORA Structure

As is known in blockcipher design and analysis, security evaluation tends to be difficult or infeasible as the block/input size increases, because the required computational complexity increases. To facilitate analysis, choice of the structure and its components is important. 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 module \((CPM)\). 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 \( 4 \times 4 \) matrix multiplication in \( \text{GF}(2^8) \) are operated. Details are written in Sec. 3.4.2. In order that the hash function family AURORA has desirable security properties including the collision resistance and indifferentiability, it should be guaranteed that the underlying compression function has no differential paths with high probability that are exploitable in collision-finding attacks or distinguishing attacks. The compression function consists of an underlying 256-bit blockcipher with two message scheduling. Since it is computationally infeasible to estimate maximum differential probability of the overall compression function \( CF : \{0,1\}^{256} \times \{0,1\}^{512} \rightarrow \{0,1\}^{256} \), we designed so that each of the 256-bit permutation from the 256-bit input \( X \) to the 256-bit output \( Z \) in \((MSM)\) and \((CPM)\) (For \( X \) and \( Z \), see Sec. 2.2.1 and 2.2.2) has no differential paths with high probability under the assumption that “subkeys” (i.e., constants in \((MSM)\) and expanded messages in \((CPM)\)) are independent and uniformly distributed.

In choosing the structure, we estimated maximum differential characteristic probability obtained by numbers of active S-boxes, and compared estimated performance given by the number of required F-functions among several candidates including the generalized Feistel structure and its variants. As a result of consideration discussed in Sec. 4.2.3, we chose 8-round AURORA.
structure for the message scheduling module and 17-round AURORA structure for the chaining value processing module.

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

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

3.4.2 F-function

The F-function consists of a substitution layer and a permutation layer, where four non-linear byte substitutions (S-boxes) and a $4 \times 4$ maximum distance separable (MDS) matrix multiplication over GF($2^8$) are operated. The S-boxes provide confusion, and the matrix multiplication provides local
diffusion in the F-function. The structure and the components of the F-function are chosen to facilitate analysis and to utilize the well-established techniques for blockcipher design and analysis. AURORA uses four F-Functions $F_0$, $F_1$, $F_2$, and $F_3$ with different diffusion matrices. Each of the building blocks $CP_L$, $CP_R$, $MS_L$, and $MS_R$ uses two different F-Functions chosen out of four (see Table 3.2). We chose four diffusion matrices so that the Diffusion Switching Mechanism (DSM) 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 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 $8 \times 8$ matrices in $GF(2)$, and $c_f$ and $c_g$ are constant vectors in $GF(2)$ (See (2.2) and (2.3) in Sec. 2.2.4).

**Conditions on $M_f$ and $M_g$**

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

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

**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.
Table 3.1: Security properties of the S-box.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum differential probability</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>maximum linear probability</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>minimum degree of Boolean polynomial</td>
<td>7</td>
</tr>
<tr>
<td>minimum number of terms in polynomial over GF($2^8$)</td>
<td>252</td>
</tr>
<tr>
<td>length of cycle</td>
<td>255</td>
</tr>
</tbody>
</table>

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 shown in Table 3.1 according to the pseudocode below:

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

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

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

### Diffusion Matrices

AURORA employs four different diffusion matrices $M_0, M_1, M_2$ and $M_3$ to improve the immunity against differential (and linear) attacks by using the Diffusion Switching Mechanism (DSM). The concept of DSM was first proposed by Shirai and Shibutani in 2004, followed by extended works $[51, 52, 53, 50]$ and used in the blockcipher CLEFIA $[54]$. This technique is applicable to the AURORA structure. By using plural different matrices, we can prevent difference cancellations which can happen at the XOR operations in the structure. As a result the guaranteed number of active S-boxes is increased.

Let $B_n(M)$ be the branch number of matrix $M$, which is defined as follows:

**Definition 1** Let $x \in \{0,1\}^m$ represented as $x = [x_0,x_1,\ldots,x_{p-1}]$ 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 $P : \{0,1\}^m \rightarrow \{0,1\}^m$. The branch number of $P$ is defined as

$$B_n(P) = \min_{a \neq 0} \{w_n(a) + w_n(P(a))\}.$$  

To utilize the DSM technique, AURORA uses two pairs of diffusion matrices $(M_0, M_1)$, and $(M_2, M_3)$ which satisfy the conditions I and II. Note that the elements of the matrices are in GF($2^8$).

---

2The 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.
Diffusion matrices used in each building block of AURORA family.

<table>
<thead>
<tr>
<th>Building Block</th>
<th>Diffusion Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>AURORA-224/256</td>
<td>$F_0,F_1$</td>
</tr>
<tr>
<td></td>
<td>$M_0,M_1$</td>
</tr>
<tr>
<td>AURORA-384/512</td>
<td>$F_0,F_1$</td>
</tr>
<tr>
<td></td>
<td>$M_0,M_1$</td>
</tr>
<tr>
<td>AURORA-224M/256M</td>
<td>$F_0,F_1$</td>
</tr>
<tr>
<td></td>
<td>$M_0,M_1$</td>
</tr>
</tbody>
</table>

### 3.1 CP building block

### 3.2 M building block

### 3.3 CP building block

### 3.4 F building block

Condition I (MDS)

$$B_8(M_0) = B_8(M_1) = 5$$ (3.1)

$$B_8(M_2) = B_8(M_3) = 5$$ (3.2)

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

Besides the condition I, the branch numbers of the concatenated matrices $M_0|M_1$, $M_0^{-1}|M_1^{-1}$, $M_2|M_3$, and $M_2^{-1}|M_3^{-1}$ should be optimal.

Condition II (DSM)

$$B_8(M_0|M_1) = B_8(M_1|^{t}M_0^{-1}|^{t}M_1^{-1}) = 5$$ (3.3)

$$B_8(M_2|M_3) = B_8(M_3|^{t}M_2^{-1}|^{t}M_3^{-1}) = 5$$ (3.4)

We call the pair of the matrices satisfying these conditions the “DSM pair”. $(M_0, M_1)$ is a DSM pair.

Actually, $(M_0, M_1)$ is chosen according to Table 3.2 and Table 3.3. $M_2$ and $M_3$ are obtained by cyclically shifting each column of $M_0$ and $M_1$, respectively. It is easily proven that $(M_2, M_3)$ is a DSM pair, i.e. Table 3.2 and Table 3.3 hold for $M_2$ and $M_3$ obtained in this way. Moreover, it is also shown that $(M_0, M_3)$ and $(M_1, M_2)$ are DSM pairs. Therefore, the DSM technique works not only in the single building block but also across the building blocks such as $CP$, $MS_L$, and $MS_R$.

Table 3.2 shows diffusion matrices used in each building block of the AURORA family.

Since there are huge number of matrices satisfying the conditions I and II, we chose $(M_0, M_1)$ 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 $GF(2^8)$. $M_2$ and $M_3$ are obtained by cyclically shifting each column of $M_0$ and $M_1$, respectively.

### 3.4.3 Data Rotating Function

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

### 3.4.4 Truncation Functions

In AURORA-224, the 224-bit hash value is obtained by truncating the 256-bit final hash value by the truncation function $TF_{224}$. Similarly, in AURORA-384, the 384-bit hash value is obtained
Table 3.3: Initial values and parameters in constant generation procedure.

<table>
<thead>
<tr>
<th>AURORA-256</th>
<th>$IV_0 = (2^{1/2} - 1)2^{16}$</th>
<th>$mask_0 = (2^{1/3} - 1)2^{16}$</th>
<th>$mask_2 = (2^{1/5} - 1)2^{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$IV_1 = (3^{1/2} - 1)2^{16}$</td>
<td>$mask_1 = (3^{1/3} - 1)2^{16}$</td>
<td>$mask_3 = (3^{1/5} - 1)2^{16}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-256M</th>
<th>$IV_0 = (5^{1/2} - 2)2^{16}$</th>
<th>$mask_0 = (5^{1/3} - 1)2^{16}$</th>
<th>$mask_2 = (5^{1/5} - 1)2^{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$IV_1 = (7^{1/2} - 2)2^{16}$</td>
<td>$mask_1 = (7^{1/3} - 1)2^{16}$</td>
<td>$mask_3 = (7^{1/5} - 1)2^{16}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AURORA-512</th>
<th>$IV_0 = (11^{1/2} - 3)2^{16}$</th>
<th>$mask_0 = (11^{1/3} - 2)2^{16}$</th>
<th>$mask_2 = (11^{1/5} - 1)2^{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$IV_1 = (13^{1/2} - 3)2^{16}$</td>
<td>$mask_1 = (13^{1/3} - 2)2^{16}$</td>
<td>$mask_3 = (13^{1/5} - 1)2^{16}$</td>
</tr>
</tbody>
</table>

by truncating the 512-bit final hash value by the truncation function $TF_{384}$. These truncation functions do not just drop right-most bytes like the SHA-2 family, but drop bytes equally from every 64-bit block to make effective use of all the outputs from the $F$-functions in the last round of the compression function. See also Sec. 2.2.3

### 3.4.5 Constant Generation

**Role of Constants in the AURORA family**

AURORA-224/256, AURORA-384/512, and AURORA-224M/256M, use 3, 4, and 4 sets of constants, respectively, as listed in Sec. 2.9.3.

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

**Design of Constant Generation Procedure**

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

The constant generation procedure is designed to generate pseudorandom sequences by using simple operations such as XOR, bit-rotations, and so on. The design strategy is similar to the constant generator of the blockcipher CLEFIA \[51\]. The four 32-bit constant values used in each module of $CPM$ and $MSM$ in one round are generated from 16-bit values $T_{0,i}$ and $T_{1,i}$. $T_{0,i}$ and $T_{1,i}$ are updated every round by multiplication by $x$ or $x^{-1}$ in GF($2^{16}$), respectively, where the primitive polynomial is $x^{16} + x^{15} + x^{13} + x^{11} + x^5 + x^4 + 1 = 0x1a831$. This primitive polynomial is also used in CLEFIA, and the choosing strategy is as follows. The lower 16-bit value is defined as $0xa831 = (\sqrt[16]{101} - 4) \cdot 2^{16}$. “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 $mask_0$, $mask_1$, $mask_2$, and $mask_3$ as the first 16 bits of the fractional parts of the square/cube/fifth roots of prime numbers 2, 3, 5, 7, 11, and 13 as Table 3.3 shows. This is an evidence that there is no trapdoor in these values.

We selected the amounts of rotation $(r_0, r_1, r_2, r_3) = (8, 8, 8, 9)$ in Step 2 in the generation procedure of the constants, which is described in Sec. 2.4 by checking whether the generated sequences pass the statistical test suites: the mono bit test, the poker test, and the runs test \[18\].

In details, we checked the pseudorandomness of the first 20,000 bits of the following sequences for all the combinations of the amounts of rotation $(r_0, r_1, r_2, r_3)$:
• Sequences of constants for AURORA-224/256
  - a sequence generated based on $T_{0,i}$: \{CONC$_{4i}$, CONC$_{4i+2}$, CONC$_{4i+4}$, \ldots \}
  - a sequence generated based on $T_{1,i}$: \{CONC$_{4i+1}$, CONC$_{4i+3}$, CONC$_{4i+5}$, \ldots \}
  - a sequence of constants used in CP: \{CONC$_{4i}$, CONC$_{4i+1}$, CONC$_{4i+2}$, \ldots \}
  - a sequence of constants used in $MS_L$: \{CONM$_{L,4i}$, CONM$_{L,4i+1}$, CONM$_{L,4i+2}$, \ldots \}
  - a sequence of constants used in $MS_R$: \{CONM$_{R,4i}$, CONM$_{R,4i+1}$, CONM$_{R,4i+2}$, \ldots \}
  - a sequence of constants used in CPF: \{CONC$_{4i+68}$, CONC$_{4i+69}$, CONC$_{4i+70}$, \ldots \}
  - a sequence of constants used in $MSF_L$: \{CONM$_{L,4i+32}$, CONM$_{L,4i+33}$, CONM$_{L,4i+34}$, \ldots \}
  - a sequence of constants used in $MSF_R$: \{CONM$_{R,4i+32}$, CONM$_{R,4i+33}$, CONM$_{R,4i+34}$, \ldots \}
• Sequences of constants for AURORA-384/512 in a similar manner to above
• Sequences of constants for AURORA-224M/256M in a similar manner to above

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

3.4.6 Initial Value

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

All of AURORA-256, AURORA-512, and AURORA-256M use the same all-0 constants. We don’t identify any security problem, because each module used in AURORA-256, AURORA-512, and AURORA-256M are different due to different matrices and constants. Similarly, all of AURORA-224, AURORA-384, and AURORA-224M use the same all-1 constants, but we don’t identify any security problem.
Chapter 4

Security of AURORA

4.1 Expected Strength

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

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

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

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

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

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

4.2 Security Argument

4.2.1 Security of HMAC using AURORA

HMAC-AURORA-224/256 specified in 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. 4.2.3, CF and FF are expected to be pseudorandom functions (PRFs) when keyed via the IV. They are also expected to be PRFs when keyed via its data input. HMAC using the MD transform was proved to be a PRF when keyed via the IV assuming that the underlying compression function is a PRF when keyed via the IV and when keyed via its data input. Therefore HMAC-AURORA-224/256 is expected to be a good PRF.

Fig. 4.2 shows a structure of HMAC-AURORA-384/512 specified in Sec. 6.2. It can be regarded that the iterated compression function of HMAC-AURORA-384/512 consists of the following 17 compression functions.
\[
\begin{align*}
CFMFF^1(X, CV) &= MFF(CF_1(X, CV)), \\
& \vdots \\
CFMFF^7(X_0|||X_6) &= MFF(CF_7(X_6, CF_6(X_5, \ldots, CF_1(X_0, CV)\ldots))), \\
CFMFF^0(X, CV) &= MFF(CF_0(X, CV)), \\
& \vdots \\
CFMF^0(X_0|||X_7, CV) &= MFF(CF_7(X_7, CF_6(X_6, \ldots, CF_0(X_0, CV)\ldots))), \\
CFMF^0(X_0|||X_7, CV) &= MF(CF_7(X_7, CF_6(X_6, \ldots, CF_0(X_0, CV)\ldots))).
\end{align*}
\]

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

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

- We derive the success probability of the collision finding adversary against the DMMD transform, as a hash function, in the random oracle model, and
- We show that the DMMD transform is preimage resistant if \( MFF \) is preimage resistant.

Collision resistance of the DMMD transform. We first restate the transform to fix the notation. Let \( f_0, f_1, \ldots, f_{l-1}, f_l : \{0,1\}^{m+n} \rightarrow \{0,1\}^n \), \( F_0 : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n} \), and \( F_1 : \{0,1\}^{2n} \rightarrow \{0,1\}^c \) be functions. The DMMD transform internally uses \( f_0, f_1, \ldots, f_{l-1}, f_l, F_0, \) and \( F_1 \). Two initial values \( H_0, \hat{H}_0 \in \{0,1\}^n \) are fixed constants. Without loss of generality, we assume it takes an (already padded) message \( M = (M_0, \ldots, M_{\mu-1}) \in (\{0,1\}^m)^\ast \) as input. The block length, \( \mu \), may vary across the messages. As we assume that the padding is properly done, the last block, \( M_{\mu-1} \), contains the length of the original message, and therefore \( \mu \geq 1 \). The output is \( H_{\mu+1} \in \{0,1\}^c \). It works as in Fig. 4.3.

![Algorithm of DMMD](image)

Figure 4.3: Algorithm of \( \text{DMMD}^{f_0, f_1, \ldots, f_{l-1}, f_l, F_0, F_1}(M) \).

Now we describe our collision finding adversary \( A_1 \). \( A_1 \) has oracle access to \( f_0, f_1, \ldots, f_l, f_{l-1}, F_0 \), and \( F_1 \), which are all random oracles, and outputs \( M, M' \in \{0,1\}^\ast \) such that \( M \neq M' \). \( A_1 \) makes \( q \) queries to each of \( f_0, f_1, \ldots, f_{l-1}, f_l, F_0, \) and \( F_1 \). We say \( A_1 \) wins if \( \text{DMMD}(M) = \text{DMMD}(M') \). \( A_1 \) may access the oracles in an arbitrary order.

Now \( A_1 \)'s advantage is defined as

\[
\text{Adv}^{\text{coll}}_{\text{DMMD}}(A_1) = \text{Pr}(A_1^{f_0, f_1, \ldots, f_{l-1}, f_l, F_0, F_1} \text{ wins}),
\]

where the probability is taken over the choices of \( f_0, f_1, \ldots, f_{l-1}, f_l, F_0, F_1 \) and \( A_1 \)'s coin (if any). \( A_1 \) is assumed to know \( \text{DMMD}(M) = \text{DMMD}(M') \) holds when \( A_1 \) outputs \( M \) and \( M' \).

Function \( G \). We next define a function \( G \), which corresponds to “one loop” of the DMMD transform. It internally uses \( f_0, f_1, \ldots, f_{l-1}, f_l, F_0 \) and \( F_1 \). It takes two initial values \( H_0, \hat{H}_0 \in \{0,1\}^n \), and a message \( M = (M_0, \ldots, M_{\mu-1}) \) of at most \( l \) blocks (i.e., \( \mu \leq l \)) as inputs, and produces the output \( (H_\mu, \hat{H}_\mu) \in (\{0,1\}^n)^2 \). It works as in Fig. 4.4.

![Algorithm of G](image)

Figure 4.4: Algorithm of \( G^{f_0, f_1, \ldots, f_{l-1}, f_l, F_0}(H_0, \hat{H}_0, M) \), where \( M = (M_0, \ldots, M_{\mu-1}) \) and \( \mu \leq l \).
We next describe our collision finding adversary $A_2$ against $G$. $A_2$ has access to $l + 1$ oracles, $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$, and $F_0$. $(f_i(\cdot), \hat{f}_i(\cdot))$ takes $(M, H, \hat{H})$ as input, and the output is $(h, \hat{h}) = (f_i(M, H), \hat{f}_i(M, \hat{H}))$. The $2l + 1$ functions, $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, and $F_0$ are random oracles. $A_2$ may access the oracles in an arbitrary order, and outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where $M$ and $M'$ are at most $l$ blocks. $A_2$ makes $2q$ queries to each of $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$, and $q$ queries to $F_0$. We say $A_2$ wins if $G((H, \hat{H}), M) = G((H', \hat{H}'), M')$. $A_2$'s advantage is defined as

$$\text{Adv}^\text{coll}_G(A_2) = \Pr(A^\text{coll}_2(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1}), F_0 \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}, F_0$ and $A_2$'s coin (if any).

$A_2$ is assumed to know $G((H, \hat{H}), M) = G((H', \hat{H}'), M')$ holds when $A_2$ outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$.

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

**Algorithm $F$.**

```
for $i \leftarrow 0$ to $\mu - 1$
    $(H_{i+1}, \hat{H}_{i+1}) \leftarrow (f_i(M_i, H_i), \hat{f}_i(M_i, \hat{H}_i))$
return $(H_\mu, \hat{H}_\mu)$
```

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

$F$ is the same as $G$ without the initial computation of $F_0$. We note that, trivially, $F$ is not collision resistant as a 2n-bit compression function. However, by making a restriction on the chaining values, then it is possible to show its collision resistance.

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

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

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

(4.1)

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

Now $A_3$’s advantage is defined as

$$\text{Adv}^\text{coll}_F(A_3) = \Pr(A^\text{coll}_3(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1}) \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$ and $A_3$’s coin (if any). $A_3$ is assumed to know $F((H, \hat{H}), M) = F((H', \hat{H}'), M')$ holds when $A_3$ outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$. 70
We next describe another adversary $A_4$. $A_4$ tries to make a collision between $G$ and $F$. Now $A_4$ has access to $l + 1$ oracles, $(f_0, f_0), \ldots, (f_{l-1}, f_{l-1})$, and $F_0$, as $A_2$. $A_4$ may access the oracles in an arbitrary order, and outputs $((H, H), M)$ and $M'$, where $M$ and $M'$ are at most $l$ blocks in lengths. $A_4$ makes $2q$ queries to each of $(f_0, f_0), \ldots, (f_{l-1}, f_{l-1})$, and $q$ queries to $F_0$. We say $A_4$ wins if $G((H, H), M) = F((H_0, H_0), M')$, where $(H_0, H_0)$ is the fixed initial value of the DMMD transform.

Now $A_4$’s advantage is defined as
\[ \text{Adv}_{G,F}^{\text{coll}}(A_4) = \Pr(A_4^{f_0, f_0, \ldots, f_{l-1}, f_{l-1}, F_0} \text{ wins}), \]
where the probability is taken over the choices of $f_0, f_0, \ldots, f_{l-1}, f_{l-1}, F_0$ and $A_4$’s coin (if any). $A_4$ is assumed to know $G((H, H), M) = F((H_0, H_0), M')$ holds when $A_4$ outputs $((H, H), M)$ and $M'$.

Now we have the following result.

**Theorem 1 (Collision resistance of the DMMD transform)** Let $A_1$, $A_2$, $A_3$, and $A_4$ be adversaries, described as above. Then we have
\[ \text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq \text{Adv}_{G}^{\text{coll}}(A_2) + \text{Adv}_{F}^{\text{coll}}(A_3) + \text{Adv}_{G,F}^{\text{coll}}(A_4) + \frac{q^2}{2^{c+1}}. \]

**Proof.** Let $A'_4$ be an adversary, exactly the same as $A_1$, but outputs $M$ and $M'$ such that $\text{DMMD}'(M) = \text{DMMD}'(M')$ and $M \neq M'$, where $\text{DMMD}'$ is the same as $\text{DMMD}$ but without the final $F_1$ function. First, we claim that
\[ \text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq \text{Adv}_{\text{DMMD}'}^{\text{coll}}(A'_4) + \frac{q^2}{2^{c+1}}, \quad (4.2) \]
since without finding a collision against $\text{DMMD}'$, $A_1$ is forced to find a collision against $F_1$, i.e., a random oracle of $c$-bit output.

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

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

Now there are three cases for the length of messages that $A'_4$ outputs. Let $M = (M_0, \ldots, M_{\mu-1})$ and $M' = (M'_0, \ldots, M'_{\mu-1})$ be the messages;

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

In case 1, $A_2$ can simulate $A'_4$’s oracles and by computing $\text{DMMD}(M)$ and $\text{DMMD}(M')$, $A_2$ obtains the desired $(H, H), (H', H')$, and $\text{cut}(M)$ and $\text{cut}(M')$ correspond to the messages that $A_2$ outputs. Similarly, in case 2, $A_3$ can output $((H_0, H_0), M)$ and $((H_0, H_0), M')$, where $(H_0, H_0)$ is the fixed initial value of the DMMD transform. In case 3, $A_4$ can compute $(H, H)$ by computing $\text{DMMD}(M)$, and $\text{cut}(M)$ and $M'$ itself are the messages that $A_4$ outputs.

We note that $A'_4$ makes $q$ queries to $f_i$ and $q$ queries to $f_i$, while $A_2$, $A_3$, and $A_4$ make $2q$ queries to $(f_i, f_i)$ oracle. Therefore $A_2, A_3,$ and $A_4$ can simulate $A'_4$’s oracles.

We next show that $A_2$ and $A_4$ are essentially equivalent to $A_3$. We show the following result.
Let \( \mathcal{A} \) be re-written as \( A \).

We describe our collision finding adversary \( A \).

There are two cases that

1. \( A \) wins, case \(|M| = |M'|\) and case \(|M| \neq |M'|\).

Consider the case where \( A \) wins with \( M \) and \( M' \) such that \(|M| = \mu m \) and \(|M'| = \mu' m \), but \( \mu \neq \mu' \).

Then \( A \) must have found the collision between the outputs of the \((f_{\mu-1}, \hat{f}_{\mu-1})\) oracle and the \((f_{\mu'-1}, \hat{f}_{\mu'-1})\) oracle, i.e., a collision between 2n bit independent random strings, or the output of the \((f_{\mu-1}, \hat{f}_{\mu-1})\) oracle is \((H_0, \hat{H}_0)\), the fixed initial value for the DMMD transform. Since there are \( l \) intermediate values, and since \( A_3 \) makes \( 2q \) oracle calls, we have the bound.

Next, we further relax the assumption.

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

\[
\text{Adv}_G^\text{coll}(A_2) \leq \text{Adv}_F^\text{coll}(A_3) + \frac{2qK}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}
\]

and

\[
\text{Adv}_G^\text{coll}(A_4) \leq \text{Adv}_F^\text{coll}(A_3) + \frac{2q^2K}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}.
\]

Proof. \( A_2 \) has a random oracle \( F_0 \), where the \( F \) function is followed. Now since the output of \( F_0 \) is a 2n-bit truly random string, we may give all answers to \( A_2 \), before \( A_2 \) has oracle access to \((f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})\). That is, we let \( A_2 \) know the response before making queries, and let \( A_2 \) choose the corresponding input values. Clearly, this does not decrease the success probability of \( A_2 \).

Now we give \( q \) random strings to \( A_2 \). Let \( \{(H_1, \hat{H}_1), \ldots, (H_q, \hat{H}_q)\}, (H_i, \hat{H}_i) \in (\{0,1\}^n)^2 \), be the \( q \) random strings. Now since we have

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

we have

\[
\text{Adv}_G^\text{coll}(A_2) \leq \text{Adv}_F^\text{coll}(A_3) + \frac{2qK}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}.
\]

By the same argument, we have a bound for \( A_4 \).

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

\[
\text{Adv}_G^\text{coll}(A_1) \leq 3\text{Adv}_F^\text{coll}(A_3) + \frac{4qK}{2^{n(K-1)}} + \frac{q^2}{2^{2n+1}}.
\]

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

Adversary \( A_6 \). We describe our collision finding adversary \( A_6 \) against \( F \). \( A_6 \) is exactly the same as \( A_3 \), but the output of \( A_6 \) is \((\hat{H}, H)(M)\) and \((\hat{H}', H')(M')\) such that \((\hat{H}, H)(M) \neq (\hat{H}', H')(M')\), where \( M \) and \( M' \) are at most \( 2n \) blocks, and \(|M| = |M'|\).

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

We have the following result.

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

\[
\text{Adv}_G^\text{coll}(A_3) \leq \text{Adv}_F^\text{coll}(A_6) + \frac{(2q + 1)^2}{2^{2n+1}}.
\]

Proof. There are two cases that \( A_3 \) wins, case \(|M| = |M'|\) and case \(|M| \neq |M'|\). Consider the case where \( A_3 \) wins with \( M \) and \( M' \) such that \(|M| = \mu m \) and \(|M'| = \mu' m \), but \( \mu \neq \mu' \).

Then \( A_3 \) must have found the collision between the outputs of the \((f_{\mu-1}, \hat{f}_{\mu-1})\) oracle and the \((f_{\mu'-1}, \hat{f}_{\mu'-1})\) oracle, i.e., a collision between 2n bit independent random strings, or the output of the \((f_{\mu-1}, \hat{f}_{\mu-1})\) oracle is \((H_0, \hat{H}_0)\), the fixed initial value for the DMMD transform. Since there are \( l \) intermediate values, and since \( A_3 \) makes \( 2q \) oracle calls, we have the bound.

Next, we further relax the assumption.

72
**Adversary $A_7$.** We describe our collision finding adversary $A_7$ against $\mathcal{F}$. $A_7$ is exactly the same as $A_6$, but it takes $\mu \leq l$ as the input, and the output of $A_7$ is $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where $M$ and $M'$ are exactly $\mu$ blocks. Notice that the output message length of $A_6$ may be adaptively chosen during the oracle access, $A_7$ has to output $\mu$ blocks of messages.

We have the following result.

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

$$\text{Adv}_F^{\text{coll}}(A_6) \leq (\text{Adv}_F^{\text{coll}}(A_7)).$$

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

Overall, the bound on $A_1$ is thus

$$\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq 3l\text{Adv}_F^{\text{coll}}(A_7) + \frac{3(2lq + 1)^2}{2^{2n+1}} + \frac{4qK}{2n(K-1)} + \frac{q^2}{2^{2n}} + \frac{q^2}{2^{n+l}}.$$

To show that the DMMD transform is secure against collision attacks, it is enough to show that $\text{Adv}_F^{\text{coll}}(A_7)$ is small enough. To show this, we consider another adversary $A'_7$ that works exactly the same as $A_7$, but is restricted in the order of oracle access. $A'_7$ has to access to oracles $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$ in this order.

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

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

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

- the multi-set $\mathcal{H}^s = \{S_1, \ldots, S_s\}$ is $K'$-COLL-free, and
- the multi-set $\mathcal{H}^t = \{T_1, \ldots, T_s\}$ is $K'$-COLL-free.

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

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

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

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

- $\mathcal{L}_i^{M_i,U,V}$ consists of $(M_i, U_i, V_i)$ such that $(M_i, U_i, V_i, X_i, Y_i) \in \mathcal{L}_i$ for some $(X_i, Y_i)$.
- $\mathcal{U}_i^{M_i,U}$ consists of $(M_i, U_i)$ such that $(M_i, U_i) \in \mathcal{U}_i$ for some $X_i$.
- $\mathcal{V}_i^{M_i,V}$ consists of $(M_i, V_i)$ such that $(M_i, V_i) \in \mathcal{V}_i$ for some $Y_i$.
We also use $L^{X,Y}_i$, $L^X_i$, $L^Y_i$, $U^X_i$, and $V^Y_i$, which are defined in an obvious way.

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

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

- $L^X_q$ is $K'$-COLL,
- $L^Y_q$ is $K'$-COLL, or
- $L^{X,Y}_q$ is 2-COLL.

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

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

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

We present four lemmata to prove Lemma 4. Let $\text{win}_i$, $0 \leq i \leq q'$, be the event that $A_S$ wins at the $i$-th query, and $\overline{\text{win}}_i$ be its complement event. Then we have

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

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

$$
\Pr(A_S^{(f_0, f_1)} \text{ wins}) \leq \sum_{0 \leq i \leq q'-1} \Pr(L^X_{i+1} \text{ is } K'-K\text{-COLL} | \overline{\text{WIN}}_i) \quad (4.3)
$$

$$
+ \sum_{0 \leq i \leq q'-1} \Pr(L^Y_{i+1} \text{ is } K'-K\text{-COLL} | \overline{\text{WIN}}_i) \quad (4.4)
$$

$$
+ \sum_{0 \leq i \leq q'-1} \Pr(L^{X,Y}_{i+1} \text{ is } 2\text{-COLL} | \overline{\text{WIN}}_i). \quad (4.5)
$$

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

**Lemma 5**

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

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

- $S_j = S_{j_1} = \cdots = S_{j_{K'-2}}$,
- $(M_{i+1}, S_j) = (M_{i+1}, S_{j_1}) = \cdots = (M_{i+1}, S_{j_{K'-2}}) \not\in U_i^{M,U}$, and
- $(M_{i+1}, T_j), (M_{i+1}, T_{j_1}), \ldots, (M_{i+1}, T_{j_{K'-2}}) \in V_i^{M,V}$.

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

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

and we have the claimed bound.

By exactly the same argument, we have the following lemma for $L_i$.

**Lemma 6**

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

Before analyzing $\Pr(L_{i+1}^{X,Y} \text{ is 2-COLL} \mid \text{WIN}_i)$, we define the events $\text{bad}_i^U$ and $\text{bad}_i^V$, $0 \leq i \leq q'$, as follows.

- We say $\text{bad}_i^U$ occurs if, the $i$-th query $(M_i, U_i, V_i)$ satisfies $(M_i, U_i) \notin U_{i-1}^{M,U}$ where $(U_i, V_i) = (S_i, T_i)$, and there exists $(M_j, U_j, X_j)$ such that
  - $(M_j, U_j, X_j) \in U_{i-1}$,
  - $U_j = S_{i+1}$, and
  - $M_i = M_j$, $S_i \neq S_{j+1}$, $T_i = T_{j+1}$, and $X_i = X_j$.

- Similarly, we say $\text{bad}_i^V$ occurs if, the $i$-th query $(M_i, U_i, V_i)$ satisfies $(M_i, V_i) \notin V_{i-1}^{M,V}$ where $(U_i, V_i) = (S_i, T_i)$, and there exists $(M_j, V_j, Y_j)$ such that
  - $(M_j, V_j, Y_j) \in V_{i-1}$,
  - $V_j = T_{i+1}$, and
  - $M_i = M_j$, $S_i = S_{i+1}$, $T_i \neq T_{j+1}$, and $Y_i = Y_j$.

We show that these bad events rarely occur.

**Lemma 7**

$$\Pr(\text{bad}_i^U) \leq \frac{K'}{2^n} \text{ and } \Pr(\text{bad}_i^V) \leq \frac{K'}{2^n}.$$ 

**Proof.** We first consider $\Pr(\text{bad}_i^U)$. We claim that there are at most $K'$ choices for $(M_j, U_j, X_j)$. To see this, our $H'$ contains only $K' - 1$ collisions, and let $(S_1, T_1), \ldots, (S_{K'-1}, T_{K'-1})$ be the elements of $H$ such that $T_1 = \cdots = T_{K'-1}$. Now we see that the probability of $\text{bad}_i^U$ is maximized when $A_8$ has already obtained $f_0(M_i, S_1), \ldots, f_0(M_i, S_{K'-2})$, and let $U_i = S_{K'-1}$. In this case, $A_8$ has $K' - 2 < K'$ target values for a collision. Now since the returned value $X_i$ is a random $n$-bit string, we have the claimed bound.

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

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

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

**Lemma 8**

$$\Pr(L_{i+1}^{X,Y} \text{ is 2-COLL} \mid \text{WIN}_i) \leq \frac{2K'i}{2^n} + \frac{2(K')^2K}{2^n} + \frac{6K'}{2^n}.$$
Proof. From the assumption on \( H \), for the \((i+1)\)-st query \((M_{i+1}, U_{i+1}, V_{i+1})\) that \( A_8 \) makes, at most \(2K'\) elements are added to \( \mathcal{L}_i^{X,Y} \). Suppose that \( A_{8} \) makes a query \((M_{i+1}, U_{i+1}, V_{i+1})\) such that \((U_{i+1}, V_{i+1}) = (S_{j}, T_{j})\), where

- \( S_{j} = S_{j_{1}} = \cdots = S_{j_{K' - 2}} \),
- \( T_{j} = T_{t_{1}} = \cdots = T_{t_{K' - 2}} \),
- \( \{j_{1}, \ldots, j_{K' - 2}\} \cap \{t_{1}, \ldots, t_{K' - 2}\} = \emptyset \),
- \( (M_{i+1}, S_{j}) = (M_{i+1}, S_{j_{1}}) = \cdots = (M_{i+1}, S_{j_{K' - 2}}) \notin \mathcal{U}_{i}^{M,U} \),
- \( (M_{i+1}, S_{t_{1}}), \ldots, (M_{i+1}, S_{t_{K' - 2}}) \in \mathcal{U}_{i}^{M,U} \),
- \( (M_{i+1}, T_{j}), \ldots, (M_{i+1}, T_{t_{K' - 2}}) \in \mathcal{V}_{i}^{M,V} \), and
- \( (M_{i+1}, T_{j}) = (M_{i+1}, T_{t_{1}}) = \cdots = (M_{i+1}, T_{t_{K' - 2}}) \notin \mathcal{V}_{i}^{M,V} \).

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

We divide the elements that are added to \( \mathcal{L}_i \) into the following three multi-sets, Type\(_1\), Type\(_2\), and Type\(_3\):

- **Type\(_1\)** consists of \((M, U, V, X, Y)\) such that \(X\) and \(Y\) are both randomly chosen at this query.
- **Type\(_2\)** consists of \((M, U, V, X, Y)\) such that \(A_8\) already knows \(X\), and only \(Y\) is randomly chosen at this query. Note that the added elements share the same random \(Y\).
- **Type\(_3\)** consists of \((M, U, V, X, Y)\) such that \(A_8\) already knows \(Y\), and only \(X\) is randomly chosen at this query. The added elements have the same random \(X\).

If the \((i+1)\)-st query \((M_{i+1}, U_{i+1}, V_{i+1})\) satisfies \((M_{i+1}, U_{i+1}) \notin \mathcal{U}_{i}^{M,U} \) and \((M_{i+1}, V_{i+1}) \notin \mathcal{V}_{i}^{M,V} \), then the element in Type\(_1\) is added, and elements in Type\(_2\) and Type\(_3\) may also be added to \( \mathcal{L}_i \). Similarly, if the \((i+1)\)-st query satisfies \((M_{i+1}, U_{i+1}) \in \mathcal{U}_{i}^{M,U} \) and \((M_{i+1}, V_{i+1}) \notin \mathcal{V}_{i}^{M,V} \), then only elements in Type\(_2\) are added to \( \mathcal{L}_i \), and if the \((i+1)\)-st query satisfies \((M_{i+1}, U_{i+1}) \notin \mathcal{U}_{i}^{M,U} \) and \((M_{i+1}, V_{i+1}) \in \mathcal{V}_{i}^{M,V} \), then only elements in Type\(_3\) are added. \( A_8 \) does not make a query \((M_{i+1}, U_{i+1}, V_{i+1})\) if \((M_{i+1}, U_{i+1}) \in \mathcal{U}_{i}^{M,U} \) and \((M_{i+1}, V_{i+1}) \in \mathcal{V}_{i}^{M,V} \). Therefore, when \((M, U, V, X, Y)\) is added to \( \mathcal{L}_i \), \(X\) or \(Y\) (or both) are randomly chosen. Also, observe that Type\(_1\) has at most one element, and Type\(_2\) and Type\(_3\) have at most \(K'\) elements, respectively.

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

Now 2-COLL can occur in the following cases:

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

\[
\Pr(\text{Type}_{i}^{X,Y} \cap \mathcal{L}_{i}^{X,Y} \neq \emptyset | \text{WIN}_i) \leq \frac{2K'}{2^{2n}}.
\]

**Case 2:** Type\(_{2}^{X,Y} \cap \mathcal{L}_{i}^{X,Y} \neq \emptyset\), or Type\(_{3}^{X,Y} \cap \mathcal{L}_{i}^{X,Y} \neq \emptyset\).

Consider Type\(_{2}^{X,Y} \cap \mathcal{L}_{i}^{X,Y} \neq \emptyset\). We know that at most \(K'\) elements are added to \( \mathcal{L}_i \). Let

\[
(M_{t_{1}}, U_{t_{1}}, V_{t_{1}}, X_{t_{1}}, Y_{t_{1}}) : \cdots : (M_{t_{K'}}, U_{t_{K'}}, V_{t_{K'}}, X_{t_{K'}}, Y_{t_{K'}})
\]

be corresponding elements, where \(Y_{t_{1}} = \cdots = Y_{t_{K'}}\) is a random \(n\)-bit string.
Now for each \((M_{\ell}, U_{\ell}, V_{\ell}, X_{\ell}, Y_{\ell})\), there are at most \(K'K\) elements in \(L_{X,Y}^{X,Y}\) for the collision, as \(L_{i}^{X}\) has most one \(K'K\) collision that share the same \(X_{\ell}\) with \((M_{\ell}, U_{\ell}, V_{\ell}, X_{\ell}, Y_{\ell})\).

Therefore, at most \(K'\) elements are added, each added element has at most \(K'\) elements in \(L_{i}^{X,Y}\) for a collision, and each elements collide with probability \(\frac{1}{2^n}\), and we thus have

\[
\Pr(\text{Type}_{2}^{X,Y} \cap L_{i}^{X,Y} \neq \emptyset \mid \text{WIN}_{i}) \leq \frac{(K')^2 K}{2^n}.
\]

Similarly, we have the same bound for \(\text{Type}_{3}^{X,Y} \cap L_{i}^{X,Y} \neq \emptyset\).

**Case 3:** \(\text{Type}_{1}^{X,Y} \cap \text{Type}_{2}^{X,Y} \neq \emptyset\), or \(\text{Type}_{1}^{X,Y} \cap \text{Type}_{3}^{X,Y} \neq \emptyset\).

Consider \(\text{Type}_{1}^{X,Y} \cap \text{Type}_{2}^{X,Y} \neq \emptyset\). The elements in \(\text{Type}_{1}^{X,Y}\) and \(\text{Type}_{2}^{X,Y}\) share the same \(Y_{\ell}\). Since \(X_{\ell}\) in \(\text{Type}_{1}^{X,Y}\) is randomly chosen, and \(\text{Type}_{2}^{X,Y}\) has at most \(K'\) elements, we have

\[
\Pr(\text{Type}_{1}^{X,Y} \cap \text{Type}_{2}^{X,Y} \neq \emptyset \mid \text{WIN}_{i}) \leq \frac{K'}{2^n}.
\]

Similarly, we have the same bound for \(\text{Type}_{1}^{X,Y} \cap \text{Type}_{3}^{X,Y} \neq \emptyset\).

**Case 4:** \(\text{Type}_{2}^{X,Y} \cap \text{Type}_{3}^{X,Y} \neq \emptyset\). We may have 2-COLL only if the randomly chosen \(Y_{\ell}\) for \(\text{Type}_{2}^{X,Y}\) collides with the \(Y_{\ell}\) in \(\text{Type}_{3}^{X,Y}\), or the randomly chosen \(X_{\ell}\) for \(\text{Type}_{3}^{X,Y}\) collides with the \(X_{\ell}\) in \(\text{Type}_{2}^{X,Y}\). Since \(\text{Type}_{2}^{X,Y}\) and \(\text{Type}_{3}^{X,Y}\) have at most \(K'\) elements,

\[
\Pr(\text{Type}_{2}^{X,Y} \cap \text{Type}_{3}^{X,Y} \neq \emptyset \mid \text{WIN}_{i}) \leq \frac{2K'}{2^n}.
\]

**Case 5:** Two elements in \(\text{Type}_{2}^{X,Y}\) collide, or two elements in \(\text{Type}_{3}^{X,Y}\) collide. Now in order for two elements in \(\text{Type}_{2}^{X,Y}\) to collide, \(\text{bad}^{U}_{i}\) has to occur. Therefore, from Lemma 7,

\[
\Pr(\text{Two elements in } \text{Type}_{2}^{X,Y} \text{ collide } \mid \text{WIN}_{i}) \leq \frac{K'}{2^n}.
\]

We have the same bound for \(\text{Type}_{3}^{X,Y}\).

Overall, we have

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

and this completes the proof. \(\Box\)

Now we present the proof of Lemma 4.

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

\[
\sum_{0 \leq i \leq q' - 1} \Pr(\mathcal{L}_{i+1}^{X,Y} \text{ is 2-COLL } \mid \text{WIN}_{i}) \leq \sum_{0 \leq i \leq q' - 1} \frac{2K'i}{2^n} + \frac{2(K')^2 K}{2^n} + \frac{6K'}{2^n} \leq \frac{q'^2 K'}{2^n} + \frac{2q'(K')^2 K}{2^n} + \frac{6q'K'}{2^n},
\]

and therefore, we have the claimed bound. \(\Box\)
Collision resistance of $\mathcal{F}$ against $A'_7$. Now we return to $A'_7$. We first recall its definition. Let $K \geq 1$ and $s \geq 1$ be fixed integers and let $\mathcal{H}_0 = \{(H_0, \hat{H}_0), \ldots, (H_{s-1}, \hat{H}_{s-1})\}, (H_i, \hat{H}_i) \in (\{0,1\}^n)^2$, be a fixed set of strings such that

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

The adversary $A'_7$ takes $\mu \leq l$ as the input. It has access to $l$ oracles, $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$, in this order, that is, $A'_7$ first makes queries to $(f_0, \hat{f}_0)$ and then $(f_1, \hat{f}_1)$, until $(f_{l-1}, \hat{f}_{l-1})$ (but the last $l - \mu$ oracles are irrelevant). $(f_i(\cdot, \cdot), \hat{f}_i(\cdot, \cdot))$ takes $(M, H, \hat{H})$ as the input, and the output is $(h, \hat{h}) = (f_i(M, H), \hat{f}_i(M, \hat{H}))$. The 2l functions, $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$, are random oracles. $A'_7$ outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$ such that $((H, \hat{H}), M) \neq ((H', \hat{H}'), M')$, where $M$ and $M'$ are both $\mu$ blocks. We say $A'_7$ wins if $\mathcal{F}((H, \hat{H}), M) = \mathcal{F}((H', \hat{H}'), M')$, where $(H, \hat{H}), (H', \hat{H}') \in \mathcal{H}_0$ must hold.

Now we restate the advantage of $A'_7$, which is defined as

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A'_7) = \Pr(A'_7(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1}) \text{ wins}),$$

where the probability is taken over the choices of $f_0, \hat{f}_0, \ldots, f_{l-1}, \hat{f}_{l-1}$ and $A'_7$’s coin (if any). $A'_7$ is assumed to know $\mathcal{F}((H, \hat{H}), M) = \mathcal{F}((H', \hat{H}'), M')$ holds when $A'_7$ outputs $((H, \hat{H}), M)$ and $((H', \hat{H}'), M')$.

We have the following result.

**Theorem 2 (Collision resistance of $\mathcal{F}$ against $A'_7$)** Let $A'_7$ be a collision finding adversary, described as above, that makes $2q$ queries to each of $(f_0, \hat{f}_0), \ldots, (f_{l-1}, \hat{f}_{l-1})$. Then for any integer $K \geq 2$,

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A'_7) \leq \frac{4q^2K^{l+1}}{2^n} + \frac{4qK^{2l+2}}{2^n} + \frac{12qK^{l+1}}{2^n} + \frac{l^2K^{l+3}q^K}{2^n(K-1)}.$$

**Proof.** Let $\mathcal{H}_i, 1 \leq i \leq l$, be a multi-set consists of all the strings of $(M, H, \hat{H}, h, \hat{h})$ such that $A'_7$ knows $(h, \hat{h}) = (f_i(M, H), \hat{f}_i(M, \hat{H}))$. $\mathcal{H}_i^{h, \hat{h}}, \mathcal{H}_i^{h},$ and $\mathcal{H}_i^{\hat{h}}$ are also defined in the natural way.

For notational simplicity, let $E_i$ be the event that

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

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

We have

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A'_7) \leq \Pr(E_1 \mid \overline{E}_0) \leq \frac{4q^2K}{2^n} + \frac{4qK^3}{2^n} + \frac{12qK}{2^n} + \frac{2K^{l+1}q^K}{2^{n(K-1)}}$$

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

$$\Pr(E_1 \mid \overline{E}_0 \land \cdots \land \overline{E}_{i-1}) \leq \frac{4q^2K^i}{2^n} + \frac{4qK^{2i+1}}{2^n} + \frac{12qK^i}{2^n} + \frac{2K^{l+1}q^K}{2^{n(K-1)}}.$$

Therefore, we have

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A'_7) \leq \Pr(E_l) \leq \sum_{1 \leq i \leq l} \Pr(E_i \mid \overline{E}_0 \land \cdots \land \overline{E}_{i-1})$$

$$\leq \sum_{1 \leq i \leq l} \frac{4q^2K^i}{2^n} + \frac{4qK^{2i+1}}{2^n} + \frac{12qK^i}{2^n} + \frac{2K^{l+1}q^K}{2^{n(K-1)}}$$

$$\leq \frac{4q^2K^{l+1}}{2^n} + \frac{4qK^{2l+2}}{2^n} + \frac{12qK^{l+1}}{2^n} + \frac{l^2K^{l+3}q^K}{2^n(K-1)}.$$

This completes the proof. \(\square\)
Analysis of adaptivity. Here we consider two consecutive compression functions of the DMMD transform. Obviously, the adversary can "connect" the chaining value if the output the first compression function is used as the input of the second compression function. Consider otherwise, i.e., the adversary does not use the output of the first compression function as the input of the second compression function. In this case, we show that the adversary has a low chance in connecting the chaining values.

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

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

We consider the following adversary $A_9$ that has access to two oracles $(f_0(\cdot, \cdot), f_1(\cdot, \cdot))$ and $(f_2(\cdot, \cdot), f_3(\cdot, \cdot))$ that, on input $(M, U, V)$, returns $(X, Y) = (f_0(M, U), f_1(M, V))$, or on input $(m, u, v)$, returns $(x, y) = (f_2(m, u), f_3(m, v))$. $f_0, \ldots, f_3$ are random oracles.

Let us define multi-sets $\mathcal{L}_i$, $\mathcal{I}_i$, and a set $\mathcal{I}'_i$, $0 \leq i \leq q'$, as follows. $\mathcal{L}_i$ is the multi-set of $(M, U, V, X, Y)$ such that $A_9$ knows $(X, Y) = (f_0(M, U), f_1(M, V))$ for some $(M, U, V)$ after the $i$-th query to the $(f_0, f_1)$ oracle (thus $\mathcal{L}_0 = \emptyset$), and $\mathcal{I}_i$ is the multi-set of $(m, u, v, x, y)$ such that $A_9$ knows $(x, y) = (f_2(m, u), f_3(m, v))$ for some $(m, u, v)$ after the $i$-th query to the $(f_2, f_3)$ oracle. Note that we count the number of queries to the $(f_0, f_1)$ oracle when we consider $\mathcal{I}_i$, $\mathcal{I}'_i$ is the set of $(u, v)$ induced by $\mathcal{I}_i$, i.e., $\mathcal{I}'_i = \mathcal{I}_i^{u,v}$, but the same bit strings are treated as one element.

We consider $A_9$ with the following constraint: For the $i$-th query $(M_i, U_i, V_i)$ that $A_9$ makes to the $(f_0, f_1)$ oracle, $(U_i, V_i)$ has to be chosen from the set $\mathcal{H}$. Suppose that $A_9$ made $i$ queries to the $(f_0, f_1)$ oracle. $\mathcal{L}_i$ is thus now defined. At this point suppose that $A_9$ makes the $j$-th query $(m_j, u_j, v_j)$ to the $(f_2, f_3)$ oracle. Now $(u_j, v_j)$ must not be chosen from $\mathcal{L}_j^{X,Y}$. We assume $A_9$ makes $q'$ queries to $(f_0, f_1)$ and $q'$ queries to $(f_2, f_3)$.

We say $A_9$ wins at the $(i+1)$-st query (to the $(f_0, f_1)$ oracle) if

$$\mathcal{L}_i^{X,Y} \cap (\mathcal{I}_i)^{u,v} = \emptyset \text{ and } (\mathcal{L}_{i+1}^{X,Y} \setminus \mathcal{L}_i^{X,Y}) \cap (\mathcal{I}_i)^{u,v} \neq \emptyset,$$

and we say $A_9$ wins if there exists $0 \leq i \leq q' - 1$ such that $A_9$ wins at the $(i+1)$-st query.

We show that $A_9$ has a low probability in winning the game.

**Lemma 9 (Success probability of $A_9$)** Let $A_9$ be the adversary, described as above. Assume $A_9$ makes $q'$ queries to each of $(f_0, f_1)$ and $(f_2, f_3)$ oracles. Let $K \geq 2$ be any integer. Then we have

$$\Pr(A_9^{(f_0, f_1) \rightarrow (f_2, f_3)}) \leq \Pr(A_9^{(f_0, f_1)}) + \frac{2q'K'K}{2^n} + \frac{2q'(K')^2K}{2^n}.$$  

**Proof.** As we have seen in the proof of Lemma 8, the size of $\mathcal{L}_q'$ is at most $2q'K'$. Let

$$(M_1, U_1, V_1, X_1, Y_1) \ldots (M_{2q'K'}, U_{2q'K'}, V_{2q'K'}, X_{2q'K'}, Y_{2q'K'})$$

be $\mathcal{L}_q'$. Also, observe that the size of $\mathcal{T}_q'$ is at most $q'$. Let

$$(u_1, v_1) \ldots (u_{q'}, v_{q'})$$

be the set $\mathcal{T}_q'$. We consider the event $((\mathcal{L}_{i+1}^{X,Y} \setminus \mathcal{L}_i^{X,Y}) \cap \mathcal{I}_i)^{u,v} \neq \emptyset$ instead of $((\mathcal{L}_{i+1}^{X,Y} \setminus \mathcal{L}_i^{X,Y}) \cap \mathcal{I}_i)^{u,v} \neq \emptyset$ since these events are equivalent. For $1 \leq j \leq 2q'K'$, let $Q_X$ be the size of the set $\{i \mid 1 \leq i \leq q', u_i = X_j\}$, and let $Q_{Y_j}$ be the size of the set $\{i \mid 1 \leq i \leq q', v_i = Y_j\}$. Now we assume that $\mathcal{L}_q^X$
is $K'K$-COLL-free, and $F'_q$ is $K'K$-COLL-free, since otherwise $A_8$ wins the game. Observe that, in this case, $Q_{X_1} + \cdots + Q_{X_{2qK'}} \leq q'K'K$ and $Q_{Y_1} + \cdots + Q_{Y_{2qK'}} \leq q'K'K$ hold.

Now we fix $1 \leq j \leq 2q'K'$, and consider $(M_j, U_j, V_j, X_j, Y_j)$. For $(M_j, U_j, V_j, X_j, Y_j)$, we see that either $X_j$ or $Y_j$, or both are randomly chosen. Then the probability that $A_9$ wins with this $(M_j, U_j, V_j, X_j, Y_j)$ is at most

$$\frac{Q_{X_j} + Q_{Y_j}}{2^n} + \frac{q'}{2^{2n}},$$

since if $X_j$ is randomly chosen, and $A_9$ knows that the oracle will return $Y_j$, then the probability that $A_9$ wins is $Q_{Y_j}/2^n$, or if $Y_j$ is randomly chosen, and $A_9$ knows that the oracle will return $X_j$, then the probability that $A_9$ wins is $Q_{X_j}/2^n$, or the probability is $q'/2^{2n}$ if $X_j$ and $Y_j$ are both randomly chosen. Now we have the claimed bound since

$$\sum_{1 \leq j \leq 2q'K'} \frac{Q_{X_j} + Q_{Y_j}}{2^n} + \frac{q'}{2^{2n}} = 2q'K'K/2^n + 2(q')^2K'.$$

Collision resistance of $\mathcal{F}$ against $A_7$. We now consider the collision resistance of $\mathcal{F}$ against $A_7$. We have the following theorem.

**Theorem 3 (Collision resistance of $\mathcal{F}$ against $A_7$)** Let $A_7$ be a collision finding adversary, as previously described, that makes $2q$ queries to each of $(f_0, f_0), \ldots, (f_{l-1}, f_{l-1})$. Then for any integer $K \geq 2$,

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A_7) \leq \frac{12q^2K2^{l+1}}{2^n} + \frac{36qK2^{l+2}}{2^n} + \frac{12K^2qK}{2(nK-1)}.$$

**Proof.** Let $\mathcal{H}_i, 1 \leq i \leq l$, be a multi-set consists of all the strings of $(M, H, \bar{H}, h, \bar{h})$ such that $A_7$ knows $(h, \bar{h}) = (f_{i-1}(M, H), f_{i-1}(M, H))$. For $0 \leq i \leq l-2$, let $D_i$ be the event that

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

or if $A_9$ wins with $K' = K^{i+1}$,

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

Observe that we have

$$\text{Adv}^{\text{coll}}_{\mathcal{F}}(A_7) \leq \text{Adv}^{\text{coll}}_{\mathcal{F}}(A'_7) + \sum_{0 \leq i \leq l-2} \Pr(D_i \mid \overline{D}_0 \land \cdots \land \overline{D}_{i-1}),$$

since without $A_9$ winning the game, $A_7$ is the same as $A'_7$.

We have

$$\Pr(D_0) \leq \frac{4q^2K}{2^n} + \frac{4qK^3}{2^n} + \frac{12qK^2}{2^n} + \frac{2^{K+1}qK}{2(nK-1)} + \frac{4qK^2}{2^n} + \frac{4qK^3}{2^n}$$

from Lemma 3 by setting $q' = 2q$ and $K' = K$, and

$$\Pr(D_i \mid \overline{D}_0) \leq \frac{4q^2K^2}{2^n} + \frac{4qK^5}{2^n} + \frac{12qK^2}{2^n} + \frac{2^{K+1}qK}{2(nK-1)} + \frac{4qK^3}{2^n} + \frac{4qK^5}{2^n}.$$

Now in general, for $0 \leq i \leq l-2$, $\Pr(D_i \mid \overline{D}_0 \land \cdots \land \overline{D}_{i-1})$ is at most

$$\frac{4q^2K^{i+1}}{2^n} + \frac{4qK^{2i+3}}{2^n} + \frac{12qK^{i+1}}{2^n} + \frac{2^{K+1}qK}{2(nK-1)} + \frac{4qK^{i+2}}{2^n} + \frac{4qK^{2i+3}}{2^n}.$$
Table 4.1: Numerical examples of the bound on $\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1)$ for $l = 8$ and $n = 256$. The numbers in $q$ column denote the minimum number of $q$ that the bound reaches 1, ignoring the last term. Since the bound holds for any $K$, $A_1$ needs at least $O(2^{201})$ queries.

<table>
<thead>
<tr>
<th>$K = 4$</th>
<th>bound on $\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1)$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{q^2}{2^{566}} + \frac{q}{2^{519}} + \frac{q^2}{2^{514}} + \frac{q^2}{2^{513}}$</td>
<td>$2^{188}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$K = 5$</th>
<th>bound on $\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1)$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{q^2}{2^{566}} + \frac{q}{2^{514}} + \frac{q^2}{2^{513}} + \frac{q^2}{2^{511}}$</td>
<td>$2^{201}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$K = 6$</th>
<th>bound on $\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1)$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{q^2}{2^{566}} + \frac{q}{2^{519}} + \frac{q^2}{2^{514}} + \frac{q^2}{2^{513}}$</td>
<td>$2^{199}$</td>
<td></td>
</tr>
</tbody>
</table>

Therefore, we have

$$\sum_{0 \leq i \leq l-2} \Pr(D_i \mid D_0 \land \cdots \land D_{i-1}) \leq \sum_{0 \leq i \leq l-2} \frac{4q^2K^{i+1}}{2^{2n}} + \frac{4qK^{2i+3}}{2^n} + \frac{12qK^{i+1}}{2^n} + \frac{2K^{i+1}q^K}{2^{n(K-1)}} + \frac{4qK^{i+2}}{2^n} + \frac{4qK^{2i+3}}{2^{2n}}$$

Now we have the bound since

$$\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq \frac{4q^2K^{i+1}}{2^{2n}} + \frac{4qK^{2i+2}}{2^n} + \frac{12qK^{i+1}}{2^n} + \frac{2K^{i+1}q^K}{2^{n(K-1)}} + \frac{4qK^{i+2}}{2^n} + \frac{4qK^{2i+3}}{2^{2n}}$$

This completes the proof.

Overall, the bound on $A_1$ is thus

$$\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq 3\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_7) + \frac{3(2q + 1)^2}{2^{2n+1}} + \frac{4q^2}{2^{2n}} + \frac{q^2}{2^{c+1}}$$

If we set $l = 8$ and $n = 256$,

$$\text{Adv}_{\text{DMMD}}^{\text{coll}}(A_1) \leq \frac{4q^2K^{17}}{2^{2n-12}} + \frac{qK^{18}}{2^{n-10}} + \frac{2K^{10}q^K}{2^{n(K-1)}} + \frac{q^2}{2^{c+1}}$$

See Table 4.1 for examples of concrete values.

We note that although the bound does not reach the level of collision resistance for a 2n bit hash function, for a reasonable size of $K$, say 4 or 5, the bound is much better than the standard birthday bound, and also, the bound does not necessarily imply the existence of the actual attack. Besides, there are many places in our proofs that the success probability is overestimated. We believe that the actual attack against the DMMD transform is much harder than the bound indicates. Indeed, finding a collision for short concatenating hash functions is recognized as a hard problem, and it is highly unlikely that the attack will be found on the DMMD transform.
Preimage resistance of the DMMD transform. It is simple to show that the DMMD transform is preimage resistant if $F_1$ (which corresponds to $MFF$) is preimage resistant. We follow the notation in Fig. 4.3. A preimage finding adversary against DMMD is an adversary that is given a hash value $h \in \{0, 1\}^c$, outputs a (padded) message $M \in \{(0, 1)^m\}^*$ such that $DMMD(M) = h$. Similarly, a preimage finding adversary against $F_1$ is given a hash value $h \in \{0, 1\}^c$, and outputs $(H, \hat{H}) \in \{(0, 1)^m\}^2$ such that $F_1(H, \hat{H}) = h$.

We have the following result.

**Theorem 4 (Preimage resistance of DMMD)** If there exists a preimage finding adversary against DMMD, then there exists a preimage finding adversary against $F_1$.

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

### 4.2.3 Security Properties of AURORA structure

#### Guaranteed Active S-boxes in AURORA structure

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

From the specification of AURORA, it can be seen that $MSM$ and $CPM$ employ 8-round and 17-round AURORA structure, respectively. The AURORA structure is based on an 8-bit S-box, matrices and a byte diffusion $BD$ design, and all components are byte-oriented. Thus, it is natural for evaluating the immunity against differential cryptanalysis by counting the minimum number of active S-boxes of AURORA structure using a blockcipher evaluation method [50][51][52][54].

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

Table 12 shows the obtained guaranteed numbers of active S-boxes for each round of AURORA structure. From the fact that AURORA employs an S-box whose maximum differential probability is $2^{-6}$, we can conclude that 8-round AURORA structure does not hold characteristics whose differential probability is higher than $2^{-156} < 2^{-128}$. Similarly, 17-round AURORA structure does not hold characteristics with probability higher than $2^{-336} < 2^{-256}$. We explain the immunity of AURORA against differential cryptanalysis by using the above observations.

There are several steps in recently developed differential type attacks for hash functions 1) finding a local collision and a differential path, 2) finding sufficient conditions applied to a message $M$, 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 less than 128-bit security. But if the attacker insert a difference into $MSM$, the probability of the differential that follows a specific characteristic which is useful for finding collision is low, which is less than $2^{-128}$.
Table 4.2: Guaranteed Numbers of Active S-boxes in AURORA structure.

<table>
<thead>
<tr>
<th>Round</th>
<th>( # ) of Active S-boxes</th>
<th>Round</th>
<th>( # ) of Active S-boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>14</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>15</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>16</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
<td>17</td>
<td>56</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>19</td>
<td>62</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>20</td>
<td>66</td>
</tr>
</tbody>
</table>

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

Output Truncation

As stated in Sec. 4.1, any \( m \) bits by taking a fixed subset of the AURORA function’s output bits expected to meet the desirable security requirements. On the other hand, if we see the AURORA structure carefully, it is noticed that dropping consecutive 32 bits at once from the output of the structure sometimes waste the calculation effort of 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-256/512/256M with different IVs to generate the output values for AURORA-224/384/224M, respectively. 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 \((X_{64(0)}, X_{64(1)}, X_{64(2)}, X_{64(3)}) \leftarrow X \). In this case, a truncation function should not drop any of \( X_{64(i)} \) at once, because output of an F-function at the last round in CPM only affects one of \( X_{64(i)} \), which means that the F-function is invalidated for the calculation of the output values. Therefore, the truncation functions in the AURORA family is 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 [18].

We searched for impossible differential paths by considering that the matrices satisfy the DSM
conditions (i.e. Conditions I and II described in Sec. 3.4.2). 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 modified-AURORA structure where byte diffusion function $BD$ is replaced with “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 module employs the 17-round AURORA structure, and the message scheduling modules employ the 8-round AURORA structure, it is expected that there is no impossible differential in the AURORA compression function which can allow us to distinguish the AURORA hash function from a random function.

4.3 Algorithm Analysis

This section describes a preliminary analysis of AURORA hash functions regarding collision attacks, preimage attacks, second preimage attacks, length-extension attacks, 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. 4.6).

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. 4.2.3 describes differential cryptanalysis of the AURORA structure, and shows that there is no differential characteristic in $MSM$ and $CPM$ with high probability.

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

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

In Table 4.3, notice that $\delta_i$ 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 from the 2nd round due to the heavy message scheduling functions. Considering the message scheduling functions, we have found collision for up to 3-round AURORA-224/256 with complexity less than the birthday bound. The differential
A 2-round local collision for AURORA family.

<table>
<thead>
<tr>
<th>round</th>
<th>chaining value difference</th>
<th>message value difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>△X₀, △X₁, △X₂, △X₃</td>
<td>△U₈i, △U₉i+1, △U₉i+2</td>
</tr>
<tr>
<td>i+1</td>
<td>0, 0, 0, 0</td>
<td>δ₂, δ₂, δ₂, δ₂</td>
</tr>
<tr>
<td>i+2</td>
<td>0, 0, 0, 0</td>
<td>0, 0, 0, 0</td>
</tr>
</tbody>
</table>

Note: δ₁, δ₂, δ₃, δ₄ are independent zero or non-zero arbitrary 32-bit values. At least one of δᵢ’s should be non-zero. Here the message schedule is ignored.

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

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

<table>
<thead>
<tr>
<th>round</th>
<th>chaining value difference</th>
<th>message value difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>△X₀, △X₁, △X₂, △X₃</td>
<td>△T₉₁₀, △T₉₁₁, △T₉₁₂</td>
</tr>
<tr>
<td>1</td>
<td>0, 0, 0, 0</td>
<td>x₄, 0, 0, 0</td>
</tr>
<tr>
<td>2</td>
<td>△X₀, △X₁, △X₂, △X₃</td>
<td>△T₉₁₃, △T₉₁₄, △T₉₁₅</td>
</tr>
<tr>
<td>3</td>
<td>0, 0, 0, 0</td>
<td>0, 0, 0, 0</td>
</tr>
<tr>
<td>4</td>
<td>△X₀, △X₁, △X₂, △X₃</td>
<td>△T₉₁₆, △T₉₁₇, △T₉₁₈</td>
</tr>
<tr>
<td>5</td>
<td>0, 0, 0, 0</td>
<td>0, 0, 0, 0</td>
</tr>
</tbody>
</table>

The chaining value difference △Xᵢ is the difference in the input chaining value Xᵢ of each round. For other symbols, see Fig. 4.6.

Let α be an 8-bit non-zero value, β be an 8-bit non-zero value where the least significant bit is zero, and γ = β ⊙ β. Then x₀, x₁, x₂, and x₃ are defined as follows:

- x₀: A 32-bit value whose 4th byte is β and the other three bytes are zero. (i.e. 000β)
- x₁: A 32-bit value whose 4th byte is γ and the other three bytes are zero. (i.e. 000γ)
- x₂: A 32-bit value whose 3rd byte is α and the other three bytes are zero. (i.e. 00α0)
- x₃: A 32-bit value whose 2nd byte is α and the other three bytes are zero. (i.e. 0α00)

If we set the message difference △M_L = (x₃, 0, 0, 0, x₁, 0, x₂, 0) and △M_R = (0, 0, 0, 0, x₃, 0, x₂, 0, x₃), 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., (Us|U₁₁) = (TR₁₁|TR₁₃) ⊙ ⊙ 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 x₀ and x₁ to be zero and γ, respectively. Then in the 2nd round, there are differences in three bytes which are input from message words T₁₁₅, T₁₁₃, and T₁₁₁. 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 F₁. Similarly, there are three active S-boxes in the left F₂ in the message scheduling function MS_R. Under the conditions above, there is a possibility that the output differences of F₁ and F₂ cancel. (On the other side, if there are less than five active S-boxes in F₁ and F₂ in total, the output differences of F₁ and F₂ never cancel due to the DSM condition (See Sec. 4.4.2.) When the cancellation occurs, there is a collision in the leftmost 32-bit word ΔX₀, and there is a collision in ΔX₁ at the same time. It is expected that one can find a cancellation in 32-bit output differences by trying 2¹⁶ message blocks due to birthday paradox. Therefore, it is expected that one can find a collision for 3-round (out of 17-round) AURORA with a complexity of 2¹⁶ 3-round AURORA compression function.

This attack works for 3-round AURORA-384/512 and AURORA-224M/256M.
Figure 4.6: Compression function of AURORA-256 (reduced to 4-round).
**Approach II : Application of the collision attack on Grindahl.** Another approach for finding collisions is a method used in the cryptanalysis of Grindahl [44]. Although it is very hard to find a low-weight and/or small differential path for Grindahl, Peyrin succeeded in building a truncated differential path starting from an all-difference pair of states. The points for the attack to work on Grindahl include

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

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

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

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

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

### 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 [30][11][2][1].
Approach I: Meet-in-the-middle approach. In most of the recent preimage attacks \cite{Leurent30}, 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. Therefore, we start by analysis of the compression function.

Leurent \cite{Leurent30} 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 \cite{Leurent30} directly to other hash functions. Aoki and Sasaki used the meet-in-the-middle technique in finding pseudo-preimages \cite{Aoki&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 \cite{47}.

The key idea in the meet-in-the-middle approach in \cite{Aoki&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 \cite{Aoki&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 Canni`ere and Rechberger at CRYPTO2008 \cite{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 \cite{16}. We describe the preliminary analytic results of preimage attack of AURORA using a SAT-solver. Here, we consider two variants of reduced version of AURORA for the attack.

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

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

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

4.3.3 Second Preimage Attacks

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

Compared with collision resistance, second preimage resistance has not been analyzed much, but we consider possible approaches 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., but has some limitations: one problem is that the success probability of the attack drops by fixing the message. Another problem is that it only works for a small subset of the message space.

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

Approach II : Using multi-near-collision differentials. Another approach for finding second preimages in the literature is to use multi-near-collision differentials. The idea is to compute 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, in the second preimage attacks on SMASH by Lamberger et al., and the (second) preimage attacks on GOST by Mendel et al.

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

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, 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 $k \times 2^{2^k+1} + 2^{n-k+1}$ work.

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

\footnote{A good summary of possible approaches for finding (second) preimages is written in [10].}
blocks, i.e. $k < 64$, and $384 < 512 - k$. AURORA-224M and AURORA-256M have multicollision resistance with the internal chaining value size of 512 bits, therefore, they provide second preimage resistance of about 224 and 256 bits, respectively.

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

### 4.3.4 Length-Extension Attack

Length-extension attack is the attack for hash functions. Given a hash value $h(M)$, the attacker obtains $h(M \parallel M')$ 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 [3], Lemma 5.1. In the abstract model, this property ensures that AURORA-256 looks like an ideal random oracle $H : \{0,1\}^* \rightarrow \{0,1\}^{256}$, and thus length-extension attack does not work. The same observation holds for AURORA-224. The proof for the pseudorandom oracle preservation (PRO-Pr) is based on the fact that the finalization function is used, and since we follow the same design principle in the DMMD transform, the attack is unlikely to be applicable to AURORA-384/512/224M/256M.

### 4.3.5 Multicollision Attack

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

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

### 4.3.6 Slide Attacks

Slide attacks have mostly been used for blockcipher cryptanalysis. As shown in [24], 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 $\text{MAC}(K,M) = H(K\|M)$, can be vulnerable to forgery and even to key recovery attacks.


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 $CF$s and $FF$, which behaves differently.

<table>
<thead>
<tr>
<th>AURORA-224</th>
<th>AURORA-256</th>
<th>AURORA-384</th>
<th>AURORA-512</th>
<th>AURORA-224M</th>
<th>AURORA-256M</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\min{224, 256 - k}$</td>
<td>$256 - k$</td>
<td>384</td>
<td>$512 - k$</td>
<td>224</td>
<td>256</td>
</tr>
</tbody>
</table>

**Table 4.5:** Second preimage resistance for $2^k$ block messages ($k < 64$) (bits).
from CFs. In AURORA-384/512, CFs, MFs and MFFs behave differently with different constants and different I/O. In the compression function level, randomly chosen constants avoid a periodic message schedule.

4.4 Tunable Security Parameters

There are two tunable security parameters in the AURORA hash function family. The first parameter is an iteration number of round functions in AURORA structure used in MSM and CPM used in AURORA-224/256/384/512/224M/256M. 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 round are 8 for MSM and 17 for CPM as described in the specification. The tuning is done keeping a relationship between these numbers such that \( c = 2^m + 1 \) where \( m \) and \( c \) are numbers of rounds for MSM and CPM, 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 increase 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 AURORA hash function family only supports hash sizes of 224, 256, 384, and 512 bits. By setting the initial vectors appropriately, we can also define an alternative hash function family which supports variable hash sizes for the range of from 1-bit to 512-bit. The hash functions for 1-bit to 256-bit output are obtained by modifying AURORA-256, and hash functions for 257-bit to 512-bit output are obtained by modifying AURORA-512. These hash functions are defined as follows.

- \( l \)-bit output hash functions for \( 1 \leq l \leq 256 \).
  
  **Step. 1** Let \( H_{0(256)} \leftarrow 1^l|0^{256-l} \).
  
  **Step. 2** Execute the AURORA-256 procedure for a message \( M \), then obtain \( H_m \).
  
  **Step. 3** Let \( (X_0(64), X_1(64), X_2(64), X_3(64)) \leftarrow H_m \).
  
  **Step. 4** Let \( d = \lfloor l/4 \rfloor \) and \( m = l \mod 4 \).
  
  **Step. 5** Drop the right-most \( d \)-bit for all \( X_i \) (\( 0 \leq i \leq 3 \)).
  
  **Step. 6** Additionally, drop the right-most 1-bit for \( X_i \) (\( 0 \leq i \leq m - 1 \)).
  
  **Step. 7** Output \( X_0||X_1||X_2||X_3 \) as an \( l \)-bit hash value.

- \( l \)-bit output hash functions for \( 257 \leq l \leq 512 \).
  
  **Step. 1** Let \( H_{0(512)} \leftarrow 1^l|0^{512-l} \).
  
  **Step. 2** Execute the AURORA-512 procedure for a message \( M \), then obtain \( H_m \).
  
  **Step. 3** Let \( (X_0(64), X_1(64), \ldots, X_7(64)) \leftarrow H_m \).
  
  **Step. 4** Let \( d = \lfloor l/8 \rfloor \) and \( m = l \mod 8 \).
  
  **Step. 5** Drop the right-most \( d \)-bit for all \( X_i \) (\( 0 \leq i \leq 8 \)).
  
  **Step. 6** Additionally, drop the right-most 1-bit from remaining \( X_i \) (\( 0 \leq i \leq m - 1 \)).
  
  **Step. 7** Output \( X_0||X_1||X_2||X_3||X_4||X_5||X_6||X_7 \) as an \( l \)-bit hash value.
Chapter 5

Efficient Implementation of AURORA

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

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

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

5.1 Software Implementation

This section describes the software performance results of AURORA.

5.1.1 Implementation Types

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

Notations

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

AURORA has the following four different 32-bit input/output F functions. Those notations are used to explain how to implement AURORA on 32-bit processors.
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 $(x_0^1, ..., x_7^1)$ be an input of $F^*$-function $F_0^*$ and $(y_0^1, ..., y_7^1)$ be an output of $F_0^*$. Similarly, let $(x_0^2, ..., x_7^2), (x_0^3, ..., x_7^3)$ and $(x_0^4, ..., x_7^4)$ be inputs of $F_1^*, F_2^*$ and $F_3^*$, respectively and let $(y_0^2, ..., y_7^2), (y_0^3, ..., y_7^3)$ and $(y_0^4, ..., y_7^4)$ 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.
Type-S1

Type-S1 is a straightforward implementation suitable for 32-bit processors. This implementation requires the following eight different 8-bit to 32-bit tables $T_0^0$, $T_1^1$, $T_2^2$, $T_3^3$, $T_0^1$, $T_1^2$, $T_2^3$ and $T_3^4$ [1].

$$
F_0^* = \begin{pmatrix}
    y_0^0 \\
y_1^0 \\
\vdots \\
y_{7}^0
\end{pmatrix} = \begin{pmatrix}
    M_0 & 0 \\
    0 & M_1
\end{pmatrix}
\begin{pmatrix}
    S(x_0^0) \\
    S(x_1^0)
\end{pmatrix}
$$

$$
F_1^* = \begin{pmatrix}
    y_0^1 \\
y_1^1 \\
\vdots \\
y_{7}^1
\end{pmatrix} = \begin{pmatrix}
    M_1 & 0 \\
    0 & M_0
\end{pmatrix}
\begin{pmatrix}
    S(x_0^1) \\
    S(x_1^1)
\end{pmatrix}
$$

$$
F_2^* = \begin{pmatrix}
    y_0^2 \\
y_1^2 \\
\vdots \\
y_{7}^2
\end{pmatrix} = \begin{pmatrix}
    M_2 & 0 \\
    0 & M_3
\end{pmatrix}
\begin{pmatrix}
    S(x_0^2) \\
    S(x_1^2)
\end{pmatrix}
$$

$$
F_3^* = \begin{pmatrix}
    y_0^3 \\
y_1^3 \\
\vdots \\
y_{7}^3
\end{pmatrix} = \begin{pmatrix}
    M_3 & 0 \\
    0 & M_2
\end{pmatrix}
\begin{pmatrix}
    S(x_0^3) \\
    S(x_1^3)
\end{pmatrix}
$$

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

$$
T_0^0(x) = \begin{cases}
    S(x), & \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x)
\end{cases}
$$

$$
T_1^1(x) = \begin{cases}
    S(x), & \{02\} \times S(x), \{03\} \times S(x), \{02\} \times S(x)
\end{cases}
$$

$$
T_2^2(x) = \begin{cases}
    S(x), & \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)
\end{cases}
$$

$$
T_3^3(x) = \begin{cases}
    S(x), & \{02\} \times S(x), \{02\} \times S(x), \{02\} \times S(x)
\end{cases}
$$

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

$$
\begin{pmatrix}
    T_0^0(x) \\
    T_1^1(x) \\
    T_2^2(x) \\
    T_3^3(x)
\end{pmatrix} = \begin{pmatrix}
    T_0^1(x) \\
    T_1^2(x) \\
    T_2^3(x) \\
    T_3^4(x)
\end{pmatrix}
$$

95
\begin{align*}
(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) \\
(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)
\end{align*}

The required operations for this implementation are estimated as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of table (KB):</td>
<td>8</td>
</tr>
<tr>
<td>Operations of $F_0$, $F_1$, $F_2$ and $F_3$</td>
<td></td>
</tr>
<tr>
<td># of table lookups:</td>
<td>16</td>
</tr>
<tr>
<td># of XORs:</td>
<td>12</td>
</tr>
</tbody>
</table>

**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 $T_0^0$, $T_0^1$, $T_1^0$, $T_2^0$, $T_3^0$, $T_3^1$, $T_1^1$ can be replaced as follows:

\begin{align*}
T_0^1(x) &= T_0^0(x) \gg 8 \\
T_1^0(x) &= T_0^0(x) \gg 16 \\
T_2^0(x) &= T_0^0(x) \gg 24 \\
T_3^0(x) &= T_0^0(x) \gg 8 \\
T_2^1(x) &= T_0^1(x) \gg 16 \\
T_3^1(x) &= T_0^1(x) \gg 24
\end{align*}

This implementation requires the following operations.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of table (KB):</td>
<td>2</td>
</tr>
<tr>
<td>Operations of $F_0$, $F_1$, $F_2$ and $F_3$</td>
<td></td>
</tr>
<tr>
<td># of table lookups:</td>
<td>16</td>
</tr>
<tr>
<td># of XORs:</td>
<td>12</td>
</tr>
<tr>
<td># of rotations:</td>
<td>12</td>
</tr>
</tbody>
</table>

**Type-S3**

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


\[ T_0^{\phi} (x) = (S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), 0, 0, 0, 0) \]

\[ T_1^{\phi} (x) = (\{02\} \times S(x), S(x), \{03\} \times S(x), \{02\} \times S(x), 0, 0, 0, 0) \]

\[ T_2^{\phi} (x) = (\{02\} \times S(x), \{02\} \times S(x), S(x), \{03\} \times S(x), 0, 0, 0, 0) \]

\[ T_3^{\phi} (x) = (\{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), S(x), 0, 0, 0, 0) \]

\[ T_4^{\phi} (x) = (0, 0, 0, S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x)) \]

\[ T_5^{\phi} (x) = (0, 0, 0, 0, \{06\} \times S(x), S(x), \{02\} \times S(x), \{08\} \times S(x)) \]

\[ T_6^{\phi} (x) = (0, 0, 0, 0, \{08\} \times S(x), \{06\} \times S(x), S(x), \{02\} \times S(x)) \]

\[ T_7^{\phi} (x) = (0, 0, 0, 0, \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), S(x)) \]

\[ T_8^{\phi} (x) = (0, 0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_9^{\phi} (x) = (0, 0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), S(x)) \]

\[ T_{10}^{\phi} (x) = (0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_{11}^{\phi} (x) = (0, 0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_{12}^{\phi} (x) = (S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), 0, 0, 0, 0) \]

\[ T_{13}^{\phi} (x) = (\{06\} \times S(x), S(x), \{02\} \times S(x), \{08\} \times S(x), 0, 0, 0, 0) \]

\[ T_{14}^{\phi} (x) = (\{08\} \times S(x), \{06\} \times S(x), S(x), \{02\} \times S(x), 0, 0, 0, 0) \]

\[ T_{15}^{\phi} (x) = (\{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x), S(x), 0, 0, 0, 0) \]

\[ T_{16}^{\phi} (x) = (S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), 0, 0, 0, 0) \]

\[ T_{17}^{\phi} (x) = (\{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_{18}^{\phi} (x) = (0, 0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_{19}^{\phi} (x) = (0, 0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

\[ T_{20}^{\phi} (x) = (0, 0, 0, \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x)) \]

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

\[ Y^{\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_1^{\phi}(x_1^{\phi}) \oplus T_2^{\phi}(x_2^{\phi}) \oplus T_3^{\phi}(x_3^{\phi}) \oplus T_4^{\phi}(x_4^{\phi}) \oplus T_5^{\phi}(x_5^{\phi}) \oplus T_6^{\phi}(x_6^{\phi}) \oplus T_7^{\phi}(x_7^{\phi}) \]

\[ Y'^{\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_1^{\phi}(x_1^{\phi}) \oplus T_2^{\phi}(x_2^{\phi}) \oplus T_3^{\phi}(x_3^{\phi}) \oplus T_4^{\phi}(x_4^{\phi}) \oplus T_5^{\phi}(x_5^{\phi}) \oplus T_6^{\phi}(x_6^{\phi}) \oplus T_7^{\phi}(x_7^{\phi}) \]

\[ Y^{2\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_2^{\phi}(x_2^{\phi}) \oplus T_3^{\phi}(x_3^{\phi}) \oplus T_4^{\phi}(x_4^{\phi}) \oplus T_5^{\phi}(x_5^{\phi}) \oplus T_6^{\phi}(x_6^{\phi}) \oplus T_7^{\phi}(x_7^{\phi}) \]

\[ Y^{3\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_3^{\phi}(x_3^{\phi}) \oplus T_2^{\phi}(x_2^{\phi}) \oplus T_4^{\phi}(x_4^{\phi}) \oplus T_5^{\phi}(x_5^{\phi}) \oplus T_6^{\phi}(x_6^{\phi}) \oplus T_7^{\phi}(x_7^{\phi}) \]

This implementation requires the following operations:

<table>
<thead>
<tr>
<th>Size of table (KB):</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations of ( F_1^<em>, F_2^</em>, F_3^* ) and ( F_4^* )</td>
<td>32</td>
</tr>
<tr>
<td># of table lookups</td>
<td>32</td>
</tr>
<tr>
<td># of XORs</td>
<td>28</td>
</tr>
</tbody>
</table>

**Type-S4**

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

\[ Y^{\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_1^{\phi}(x_1^{\phi}) \oplus \cdots \oplus T_7^{\phi}(x_7^{\phi}) \]

\[ Y^{1\phi} = (T_0^{1\phi}(x_0^{\phi}) \oplus T_1^{1\phi}(x_1^{\phi}) \oplus \cdots \oplus T_7^{1\phi}(x_7^{\phi})) \gg 32 \]

\[ Y^{2\phi} = T_0^{\phi}(x_0^{\phi}) \oplus T_2^{\phi}(x_2^{\phi}) \oplus \cdots \oplus T_7^{\phi}(x_7^{\phi}) \]

\[ Y^{3\phi} = (T_0^{1\phi}(x_0^{\phi}) \oplus T_3^{1\phi}(x_3^{\phi}) \oplus \cdots \oplus T_7^{1\phi}(x_7^{\phi})) \gg 32 \]

This implementation requires the following operations.
**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.

\[
T_0(x) = (S(x), \{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{03\} \times S(x), \{02\} \times S(x), \{08\} \times S(x), \{06\} \times S(x))
\]

\[
T_1(x) = (\{02\} \times S(x), S(x), \{03\} \times S(x), \{02\} \times S(x), \{06\} \times S(x), S(x), \{02\} \times S(x), \{08\} \times S(x))
\]

\[
T_2(x) = (\{02\} \times S(x), \{02\} \times S(x), S(x), \{03\} \times S(x), \{08\} \times S(x), S(x), \{02\} \times S(x), \{06\} \times S(x))
\]

\[
T_3(x) = (\{03\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{02\} \times S(x), \{08\} \times S(x), S(x), \{06\} \times S(x), S(x))
\]

\[
Y^{0'} = ((T_0(x_0^0) \oplus T_1(x_1^0) \oplus T_2(x_2^0) \oplus T_3(x_3^0)) \& \text{0xffffffff00000000} \oplus
\]
\[
((T_0(x_0^1) \oplus T_1(x_1^1) \oplus T_2(x_2^1) \oplus T_3(x_3^1)) \& \text{0x00000000ffffffff})
\]

\[
Y^{1'} = ((T_0(x_0^2) \oplus T_1(x_1^2) \oplus T_2(x_2^2) \oplus T_3(x_3^2)) \ll 32) \oplus
\]
\[
((T_0(x_0^3) \oplus T_1(x_1^3) \oplus T_2(x_2^3) \oplus T_3(x_3^3)) \gg 32)
\]

\[
Y^{2'} = ((T_0(x_0^4) \oplus T_0(x_1^4) \oplus T_1(x_2^4) \oplus T_2(x_3^4)) \& \text{0xffffffff00000000} \oplus
\]
\[
((T_1(x_0^5) \oplus T_2(x_1^5) \oplus T_3(x_2^5) \oplus T_3(x_3^5)) \& \text{0x00000000ffffffff})
\]

\[
Y^{3'} = ((T_3(x_0^6) \oplus T_0(x_1^6) \oplus T_1(x_2^6) \oplus T_2(x_3^6)) \ll 32) \oplus
\]
\[
((T_1(x_0^7) \oplus T_2(x_1^7) \oplus T_3(x_2^7) \oplus T_3(x_3^7)) \gg 32)
\]

This implementation requires the following operations.

<table>
<thead>
<tr>
<th>Size of table (KB):</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations of ( F_0^1, F_1^1, F_2^1 ) and ( F_3^1 )</td>
<td>32</td>
</tr>
<tr>
<td># of table lookups:</td>
<td>32</td>
</tr>
<tr>
<td># of XORs :</td>
<td>28</td>
</tr>
<tr>
<td># of ANDs :</td>
<td>4</td>
</tr>
<tr>
<td># of shift operations :</td>
<td>4</td>
</tr>
</tbody>
</table>

### 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.

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

Table 5.2: 8-bit Processors.

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

5.1.2 Evaluation Results

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

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

32/64-bit Processors

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

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

8-bit Processors

We present the evaluation results on performance of AURORA-224/256/384/512 on 8-bit processors at the present. The platforms used for the evaluation are shown in Table 5.2. Table 5.9 shows
the evaluation results of the compression function for AURORA-224/256 and AURORA-384/512. Tables 5.10, 5.11, 5.12, and 5.13 represent the evaluation results of AURORA-224, AURORA-256, AURORA-384 and AURORA-512, respectively.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td><strong>Platform A (Core 2 Duo (32-bit))</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-S1 (unroll)</td>
<td>1,847.4</td>
<td>188.4</td>
<td>36.6</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,860.8</td>
<td>190.2</td>
<td>36.8</td>
</tr>
<tr>
<td>Type-S2 (unroll)</td>
<td>1,788.9</td>
<td>183.2</td>
<td>35.7</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,929.4</td>
<td>195.8</td>
<td>38.1</td>
</tr>
<tr>
<td>Type-S3 (unroll)</td>
<td>3,117.2</td>
<td>317.1</td>
<td>62.4</td>
</tr>
<tr>
<td>(looped)</td>
<td>2,586.0</td>
<td>265.6</td>
<td>51.8</td>
</tr>
<tr>
<td>Type-S4 (unroll)</td>
<td>2,803.6</td>
<td>283.9</td>
<td>55.4</td>
</tr>
<tr>
<td>(looped)</td>
<td>2,625.7</td>
<td>265.4</td>
<td>51.7</td>
</tr>
<tr>
<td>Type-S5 (unroll)</td>
<td>2,686.2</td>
<td>272.6</td>
<td>52.9</td>
</tr>
<tr>
<td>(looped)</td>
<td>2,477.9</td>
<td>251.3</td>
<td>48.7</td>
</tr>
<tr>
<td><strong>Platform B (Core 2 Duo (64-bit))</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-S1 (unroll)</td>
<td>1,270.9</td>
<td>125.6</td>
<td>23.9</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,412.5</td>
<td>140.1</td>
<td>26.7</td>
</tr>
<tr>
<td>Type-S2 (unroll)</td>
<td>1,490.4</td>
<td>147.8</td>
<td>28.2</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,608.2</td>
<td>159.6</td>
<td>30.4</td>
</tr>
<tr>
<td>Type-S3 (unroll)</td>
<td>1,155.4</td>
<td>119.0</td>
<td>22.5</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,308.2</td>
<td>132.7</td>
<td>25.3</td>
</tr>
<tr>
<td>Type-S4 (unroll)</td>
<td>1,177.8</td>
<td>119.3</td>
<td>22.6</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,262.2</td>
<td>128.7</td>
<td>24.3</td>
</tr>
<tr>
<td>Type-S5 (unroll)</td>
<td>1,342.9</td>
<td>134.9</td>
<td>25.5</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,421.3</td>
<td>142.8</td>
<td>27.0</td>
</tr>
<tr>
<td><strong>Platform C (Opteron)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-S1 (unroll)</td>
<td>2,742.1</td>
<td>276.1</td>
<td>50.2</td>
</tr>
<tr>
<td>(looped)</td>
<td>2,912.1</td>
<td>292.1</td>
<td>54.0</td>
</tr>
<tr>
<td>Type-S2 (unroll)</td>
<td>2,972.8</td>
<td>299.7</td>
<td>55.3</td>
</tr>
<tr>
<td>(looped)</td>
<td>3,091.9</td>
<td>311.5</td>
<td>57.8</td>
</tr>
<tr>
<td>Type-S3 (unroll)</td>
<td>2,196.4</td>
<td>221.9</td>
<td>40.0</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,900.4</td>
<td>161.1</td>
<td>28.1</td>
</tr>
<tr>
<td>Type-S4 (unroll)</td>
<td>2,114.6</td>
<td>213.8</td>
<td>38.7</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,611.0</td>
<td>164.8</td>
<td>28.7</td>
</tr>
<tr>
<td>Type-S5 (unroll)</td>
<td>2,173.0</td>
<td>220.0</td>
<td>39.7</td>
</tr>
<tr>
<td>(looped)</td>
<td>1,709.0</td>
<td>173.3</td>
<td>30.0</td>
</tr>
<tr>
<td><strong>Platform D (Pentium 4)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-S1 (unroll)</td>
<td>4,299.5</td>
<td>436.3</td>
<td>79.5</td>
</tr>
<tr>
<td>(looped)</td>
<td>4,197.9</td>
<td>428.0</td>
<td>78.9</td>
</tr>
<tr>
<td>Type-S2 (unroll)</td>
<td>5,069.7</td>
<td>498.5</td>
<td>94.7</td>
</tr>
<tr>
<td>(looped)</td>
<td>5,093.4</td>
<td>525.4</td>
<td>100.3</td>
</tr>
<tr>
<td>Type-S3 (unroll)</td>
<td>10,748.0</td>
<td>1,082.7</td>
<td>199.8</td>
</tr>
<tr>
<td>(looped)</td>
<td>7,504.9</td>
<td>755.5</td>
<td>143.9</td>
</tr>
<tr>
<td>Type-S4 (unroll)</td>
<td>10,486.6</td>
<td>1,051.0</td>
<td>194.2</td>
</tr>
<tr>
<td>(looped)</td>
<td>7,471.2</td>
<td>762.0</td>
<td>142.0</td>
</tr>
<tr>
<td>Type-S5 (unroll)</td>
<td>10,005.3</td>
<td>1,010.8</td>
<td>188.4</td>
</tr>
<tr>
<td>(looped)</td>
<td>7,213.3</td>
<td>725.8</td>
<td>136.8</td>
</tr>
</tbody>
</table>
Table 5.4: AURORA-256 on 32/64-bit processors.

<table>
<thead>
<tr>
<th>Type-S1</th>
<th>Type-S2</th>
<th>Type-S3</th>
<th>Type-S4</th>
<th>Type-S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unroll)</td>
<td>(unroll)</td>
<td>(unroll)</td>
<td>(unroll)</td>
<td>(unroll)</td>
</tr>
<tr>
<td>(looped)</td>
<td>(looped)</td>
<td>(looped)</td>
<td>(looped)</td>
<td>(looped)</td>
</tr>
</tbody>
</table>

Platform A (Core 2 Duo (32-bit))

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Platform A</th>
<th>1 CF call</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>message size [bytes]</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Type-S1</td>
<td>1,836.3</td>
<td>185.7</td>
<td>36.3</td>
</tr>
<tr>
<td>Type-S2</td>
<td>1,770.2</td>
<td>179.9</td>
<td>35.0</td>
</tr>
<tr>
<td>Type-S3</td>
<td>3,069.0</td>
<td>311.5</td>
<td>61.6</td>
</tr>
<tr>
<td>Type-S4</td>
<td>2,787.6</td>
<td>280.9</td>
<td>55.1</td>
</tr>
<tr>
<td>Type-S5</td>
<td>2,449.7</td>
<td>267.4</td>
<td>52.4</td>
</tr>
</tbody>
</table>

Platform B (Core 2 Duo (64-bit))

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Platform B</th>
<th>1 CF call</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>message size [bytes]</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Type-S1</td>
<td>1,235.3</td>
<td>123.4</td>
<td>23.6</td>
</tr>
<tr>
<td>Type-S2</td>
<td>1,576.1</td>
<td>156.3</td>
<td>30.4</td>
</tr>
<tr>
<td>Type-S3</td>
<td>1,142.2</td>
<td>115.4</td>
<td>22.3</td>
</tr>
<tr>
<td>Type-S4</td>
<td>1,273.7</td>
<td>130.1</td>
<td>25.0</td>
</tr>
<tr>
<td>Type-S5</td>
<td>1,315.3</td>
<td>132.6</td>
<td>25.2</td>
</tr>
</tbody>
</table>

Platform C (Opteron)

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Platform C</th>
<th>1 CF call</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>message size [bytes]</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Type-S1</td>
<td>2,575.6</td>
<td>262.1</td>
<td>48.9</td>
</tr>
<tr>
<td>Type-S2</td>
<td>2,848.6</td>
<td>286.9</td>
<td>54.0</td>
</tr>
<tr>
<td>Type-S3</td>
<td>2,074.4</td>
<td>209.9</td>
<td>38.8</td>
</tr>
<tr>
<td>Type-S4</td>
<td>2,005.7</td>
<td>202.5</td>
<td>37.6</td>
</tr>
<tr>
<td>Type-S5</td>
<td>2,051.7</td>
<td>204.5</td>
<td>29.0</td>
</tr>
</tbody>
</table>

Platform D (Pentium 4)

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Platform D</th>
<th>1 CF call</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>message size [bytes]</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Type-S1</td>
<td>4,036.5</td>
<td>422.3</td>
<td>77.9</td>
</tr>
<tr>
<td>Type-S2</td>
<td>5,069.7</td>
<td>498.5</td>
<td>94.7</td>
</tr>
<tr>
<td>Type-S3</td>
<td>10,475.5</td>
<td>1,045.4</td>
<td>196.9</td>
</tr>
<tr>
<td>Type-S4</td>
<td>10,055.7</td>
<td>1,016.1</td>
<td>191.1</td>
</tr>
<tr>
<td>Type-S5</td>
<td>9,712.1</td>
<td>984.7</td>
<td>185.5</td>
</tr>
</tbody>
</table>

| message size [bytes] | 1 | 10 | 100 | 1,000 | 10,000 | - | - |
| Type-S1 | 6,992.4 | 709.0 | 134.6 | 98.9 | 96.3 | 6,210.2 | 29,580 |
Table 5.5: AURORA-384 on 32/64-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform A (Core 2 Duo (32-bit))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Type-S1 (unroll)</td>
<td>5,709.0</td>
<td>574.9</td>
<td>861.1</td>
</tr>
<tr>
<td></td>
<td>(looped)</td>
<td>6,160.2</td>
<td>628.3</td>
</tr>
<tr>
<td>Type-S2 (unroll)</td>
<td>5,724.6</td>
<td>574.7</td>
<td>85.0</td>
</tr>
<tr>
<td></td>
<td>(looped)</td>
<td>5,743.4</td>
<td>580.9</td>
</tr>
<tr>
<td>Type-S3 (unroll)</td>
<td>21,527.2</td>
<td>2,153.0</td>
<td>320.7</td>
</tr>
<tr>
<td></td>
<td>(looped)</td>
<td>8,686.9</td>
<td>875.5</td>
</tr>
<tr>
<td>Type-S4 (unroll)</td>
<td>20,457.5</td>
<td>2,048.5</td>
<td>305.3</td>
</tr>
<tr>
<td></td>
<td>(looped)</td>
<td>7,603.6</td>
<td>764.1</td>
</tr>
<tr>
<td>Type-S5 (unroll)</td>
<td>21,207.6</td>
<td>2,118.5</td>
<td>315.0</td>
</tr>
<tr>
<td></td>
<td>(looped)</td>
<td>7,270.0</td>
<td>732.0</td>
</tr>
</tbody>
</table>

|                      | Platform B (Core 2 Duo (64-bit)) |                  |                  |
|                      | 1 | 10 | 100 | 1,000 | 10,000 | - | - |
| Type-S1 (unroll)     | 3,786.4 | 378.0 | 55.4 | 31.7 | 30.7 | 1,711.8 | 149,072 |
|                      | (looped) | 4,079.6 | 407.6 | 59.7 | 34.0 | 33.2 | 1,853.4 | 66,326 |
| Type-S2 (unroll)     | 4,267.8 | 426.4 | 62.6 | 35.7 | 34.9 | 1,962.1 | 189,792 |
|                      | (looped) | 4,572.6 | 456.8 | 67.2 | 38.4 | 37.5 | 2,091.1 | 62,126 |
| Type-S3 (unroll)     | 3,455.2 | 346.4 | 50.8 | 28.2 | 27.4 | 1,521.4 | 205,626 |
|                      | (looped) | 4,002.3 | 403.3 | 59.1 | 33.1 | 32.2 | 1,774.7 | 128,490 |
| Type-S4 (unroll)     | 3,506.9 | 352.0 | 51.6 | 28.9 | 28.1 | 1,566.7 | 181,694 |
|                      | (looped) | 3,694.3 | 371.2 | 54.6 | 30.7 | 29.9 | 1,670.8 | 103,718 |
| Type-S5 (unroll)     | 3,803.9 | 382.6 | 55.9 | 31.8 | 31.0 | 1,725.2 | 170,894 |
|                      | (looped) | 4,057.3 | 404.4 | 59.3 | 33.7 | 32.9 | 1,837.9 | 91,446 |

|                      | Platform C (Opteron) |                  |                  |
|                      | 1 | 10 | 100 | 1,000 | 10,000 | - | - |
| Type-S1 (unroll)     | 7,943.1 | 796.6 | 115.7 | 65.6 | 63.9 | 3,587.9 | 57,305 |
|                      | (looped) | 7,212.3 | 723.9 | 104.7 | 59.2 | 57.7 | 3,253.4 | 21,641 |
| Type-S2 (unroll)     | 8,864.7 | 886.1 | 129.1 | 74.1 | 72.3 | 4,060.8 | 51,241 |
|                      | (looped) | 8,103.2 | 816.5 | 118.6 | 67.3 | 65.6 | 3,675.1 | 15,625 |
| Type-S3 (unroll)     | 8,427.3 | 844.4 | 122.8 | 70.1 | 68.4 | 3,844.2 | 83,609 |
|                      | (looped) | 4,214.4 | 422.6 | 59.7 | 32.6 | 31.5 | 1,775.7 | 46,169 |
| Type-S4 (unroll)     | 8,938.8 | 895.7 | 130.9 | 75.5 | 73.9 | 4,139.6 | 70,073 |
|                      | (looped) | 4,223.2 | 423.1 | 60.0 | 32.9 | 31.9 | 1,777.1 | 30,073 |
| Type-S5 (unroll)     | 7,850.4 | 787.3 | 114.6 | 65.4 | 63.8 | 3,577.0 | 60,537 |
|                      | (looped) | 4,380.0 | 439.5 | 62.5 | 34.4 | 33.3 | 1,854.6 | 21,881 |

|                      | Platform D (Pentium 4) |                  |                  |
|                      | 1 | 10 | 100 | 1,000 | 10,000 | - | - |
| Type-S1 (unroll)     | 13,832.6 | 1,410.5 | 204.5 | 106.5 | 98.3 | 5,483.4 | 59,772 |
|                      | (looped) | 13,475.2 | 1,342.5 | 200.5 | 112.2 | 107.9 | 5,811.4 | 22,092 |
| Type-S2 (unroll)     | 16,059.2 | 1,600.5 | 233.7 | 135.6 | 126.2 | 7,090.0 | 55,172 |
|                      | (looped) | 17,817.9 | 1,778.9 | 263.5 | 150.9 | 147.4 | 6,120.3 | 16,560 |
| Type-S3 (unroll)     | 29,331.6 | 2,949.2 | 433.6 | 247.9 | 241.8 | 13,570.7 | 127,828 |
|                      | (looped) | 21,295.5 | 2,144.9 | 316.4 | 181.0 | 176.5 | 9,930.2 | 55,436 |
| Type-S4 (unroll)     | 28,286.8 | 2,839.3 | 416.8 | 237.6 | 232.0 | 13,046.8 | 103,356 |
|                      | (looped) | 20,301.7 | 2,051.2 | 301.4 | 171.2 | 167.1 | 9,451.8 | 38,536 |
| Type-S5 (unroll)     | 26,879.0 | 2,703.9 | 395.7 | 225.2 | 219.7 | 12,367.1 | 89,528 |
|                      | (looped) | 20,039.9 | 2,000.5 | 295.0 | 167.4 | 162.8 | 9,348.4 | 29,580 |

103
Table 5.6: AURORA-512 on 32/64-bit processors.

<table>
<thead>
<tr>
<th>message size [bytes]</th>
<th>Hash Function</th>
<th>1 CF call [cycles]</th>
<th>code size [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[cycles/byte]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

**Platform A** (Core 2 Duo (32-bit))

| Type-S1 | (unroll) | 5,733.8 | 577.0 | 86.1 | 48.9 | 47.6 | 2,666.9 | 142,926 |
| Type-S2 | (unroll) | 5,737.9 | 573.2 | 85.4 | 48.1 | 46.9 | 2,612.9 | 179,662 |
| Type-S3 | (unroll) | 21,441.6| 2,147.6| 319.8| 186.2| 183.3| 10,343.1| 198,002 |
| Type-S4 | (unroll) | 7,587.3 | 759.8 | 113.2| 64.9 | 63.3 | 3,554.4 | 96,262 |
| Type-S5 | (unroll) | 7,233.3 | 728.5 | 108.5| 62.1 | 60.6 | 3,395.7 | 83,990 |

**Platform B** (Core 2 Duo (64-bit))

| Type-S1 | (unroll) | 3,743.1 | 372.1 | 54.9 | 31.4 | 30.7 | 1,711.8 | 149,072 |
| Type-S2 | (unroll) | 4,029.8 | 401.5 | 59.2 | 34.0 | 33.2 | 1,853.4 | 66,326 |
| Type-S3 | (unroll) | 4,210.4 | 421.0 | 61.9 | 35.7 | 34.9 | 1,962.1 | 189,792 |
| Type-S4 | (unroll) | 3,928.2 | 394.6 | 58.5 | 33.0 | 32.2 | 1,774.7 | 128,490 |
| Type-S5 | (unroll) | 3,440.3 | 346.1 | 51.0 | 28.8 | 28.1 | 1,566.7 | 81,694 |

**Platform C** (Opteron)

| Type-S1 | (unroll) | 7,766.1 | 776.1 | 113.5| 65.3 | 63.8 | 3,587.9 | 57,305 |
| Type-S2 | (unroll) | 8,639.4 | 865.4 | 127.1| 73.9 | 72.3 | 4,060.8 | 51,241 |
| Type-S3 | (unroll) | 7,861.8 | 788.1 | 115.6| 67.1 | 65.6 | 3,075.1 | 15,025 |
| Type-S4 | (unroll) | 8,133.3 | 813.6 | 119.7| 69.8 | 68.4 | 3,844.2 | 83,009 |
| Type-S5 | (unroll) | 7,636.7 | 764.4 | 112.4| 65.2 | 63.8 | 3,577.0 | 60,537 |

**Platform D** (Pentium 4)

| Type-S1 | (unroll) | 13,666.9| 1,370.1| 198.9| 108.5| 98.2 | 5,483.4 | 59,772 |
| Type-S2 | (unroll) | 15,518.0| 1,557.4| 229.4| 129.7| 128.8| 7,090.0 | 55,172 |
| Type-S3 | (unroll) | 17,203.8| 1,729.2| 259.2| 150.6| 147.2| 6,420.3 | 16,560 |
| Type-S4 | (unroll) | 29,074.0| 2,915.4| 430.1| 247.3| 241.7| 13,570.7| 127,828 |
| Type-S5 | (unroll) | 20,959.9| 2,030.2| 298.1| 171.0| 166.9| 9,351.8 | 38,536 |
| Type-S6 | (unroll) | 26,513.4| 2,662.9| 391.9| 224.8| 219.6| 12,367.1| 89,528 |

| Type-S1 | (looped) | 19,608.8| 1,978.8| 291.8| 166.4| 162.5| 9,348.4 | 29,580 |

| Type-S2 | (looped) | 19,608.8| 1,978.8| 291.8| 166.4| 162.5| 9,348.4 | 29,580 |

| Type-S3 | (looped) | 19,608.8| 1,978.8| 291.8| 166.4| 162.5| 9,348.4 | 29,580 |

| Type-S4 | (looped) | 19,608.8| 1,978.8| 291.8| 166.4| 162.5| 9,348.4 | 29,580 |

| Type-S5 | (looped) | 19,608.8| 1,978.8| 291.8| 166.4| 162.5| 9,348.4 | 29,580 |
Table 5.7: SHA-256 on 32/64-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform A (Core 2 Duo (32-bit))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,609.3</td>
<td>162.2</td>
<td>31.0</td>
</tr>
<tr>
<td></td>
<td>Platform B (Core 2 Duo (64-bit))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,376.1</td>
<td>138.6</td>
<td>26.9</td>
</tr>
<tr>
<td></td>
<td>Platform C (Opteron)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,686.0</td>
<td>169.4</td>
<td>31.9</td>
</tr>
<tr>
<td></td>
<td>Platform D (Pentium 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3,084.2</td>
<td>311.4</td>
<td>57.2</td>
</tr>
</tbody>
</table>

Table 5.8: SHA-512 on 32/64-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform A (Core 2 Duo (32-bit))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6,191.1</td>
<td>621.2</td>
<td>61.2</td>
</tr>
<tr>
<td></td>
<td>Platform B (Core 2 Duo (64-bit))</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,805.1</td>
<td>181.5</td>
<td>19.0</td>
</tr>
<tr>
<td></td>
<td>Platform C (Opteron)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2,237.0</td>
<td>224.7</td>
<td>22.7</td>
</tr>
<tr>
<td></td>
<td>Platform D (Pentium 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15,873.8</td>
<td>1,684.2</td>
<td>176.5</td>
</tr>
</tbody>
</table>
Table 5.9: Compression functions for AURORA-224/256 and AURORA-384/512 on 8-bit processors.

<table>
<thead>
<tr>
<th>CF</th>
<th>Platform</th>
<th>1 CF call [cycles/byte]</th>
<th>code size [bytes]</th>
<th>stack [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AURORA-224/256</td>
<td>Platform E</td>
<td>446,675</td>
<td>6,158</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>3,410,460</td>
<td>4,596</td>
<td>216</td>
</tr>
<tr>
<td>AURORA-384/512</td>
<td>Platform E</td>
<td>676,814</td>
<td>6,158</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>5,152,644</td>
<td>4,596</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 5.10: AURORA-224 on 8-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform E</td>
<td>451,055</td>
<td>9,147.8</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>3,428,682</td>
<td>68,803.2</td>
</tr>
</tbody>
</table>

Table 5.11: AURORA-256 on 8-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform E</td>
<td>450,601</td>
<td>9,143.3</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>3,425,578</td>
<td>68,767.1</td>
</tr>
</tbody>
</table>

Table 5.12: AURORA-384 on 8-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform E</td>
<td>1,358,852</td>
<td>155,252.8</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>10,331,178</td>
<td>155,566.0</td>
</tr>
</tbody>
</table>

Table 5.13: AURORA-512 on 8-bit processors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform E</td>
<td>1,358,698</td>
<td>155,959.3</td>
</tr>
<tr>
<td></td>
<td>Platform F</td>
<td>10,324,052</td>
<td>155,179.1</td>
</tr>
</tbody>
</table>
5.2 Hardware Implementation

This section describes the hardware optimization techniques and performance results of AURORA. Since the implementations of AURORA-224 and AURORA-384 are basically same as AURORA-256 and AURORA-512, respectively, except the initial value and truncation of final hash value, we designed and evaluated the implementations of AURORA-256 and AURORA-512 in this section.

5.2.1 Optimization Techniques of F-functions

We introduce optimization techniques of F-functions focusing on an S-box, matrices and a pipeline architecture in hardware implementation.

S-box

The 8-bit S-box of AURORA consists of three layers: affine transformation \( f \), inversion over \( \text{GF}((2^4)^2) \) and affine transformation \( g \). In Fig. 5.2 we show the schematic design of our S-box implementation. The inversion is performed in \( \text{GF}((2^4)^2) \) defined by the following polynomials:

\[
\{ \\
\text{GF}(2^4) : p(x) = x^4 + x + 1 \\
\text{GF}((2^4)^2) : q(x) = x^2 + x + \lambda \ (\lambda = \{1001\} \in \text{GF}(2^4)) \\
\}
\]

For an arbitrary element \( a_0 \beta + a_1 \) over \( \text{GF}((2^4)^2) \) where \( a_0, a_1 \in \text{GF}(2^4) \) and \( \beta \) is a root of \( q(x) \), the inversion \( b_0 \beta + b_1 = (a_0 \beta + a_1)^{-1} \) \( (b_0, b_1 \in \text{GF}(2^4)) \) is computed as follows [46]:

\[
\begin{align*}
   b_0 &= a_0 \Delta^{-1}, \\
   b_1 &= (a_0 + a_1) \Delta^{-1}, \\
   \Delta &= (a_0 + a_1)a_1 + \lambda a_0^2.
\end{align*}
\]

These arithmetics except an inversion over \( \text{GF}(2^4) \), 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.

Matrices \( M_0, M_1, M_2 \) and \( M_3 \)

The 4 \( \times \) 4 matrices \( M_0, M_1, M_2 \) and \( M_3 \) are multiplied to the outputs of S-boxes as a linear \( (4, 4) \) multipermutation over \( \text{GF}(2^8) \) which is defined by an irreducible polynomial \( x^8 + x^4 + x^3 + x^2 + 1 \). An addition of two elements in \( \text{GF}(2^8) \), 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
For an element \( a \) in \( \text{GF}(2^8) \), denoted by \( \times \), corresponds to a multiplication of polynomials modulo \( x^8 + x^4 + x^3 + x^2 + 1 \). For an element \( a \) in \( \text{GF}(2^8) \), \( \{02\} \times a, \{04\} \times a \) and \( \{08\} \times a \) require 3, 5 and 8 XOR logic gates, respectively.

The matrix \( M_0 \) can be decomposed into the following form.

\[
\begin{pmatrix}
01 & 02 & 02 & 03 \\
03 & 01 & 02 & 02 \\
02 & 03 & 01 & 02 \\
02 & 02 & 03 & 01
\end{pmatrix} = \begin{pmatrix}
01 & 00 & 00 & 01 \\
01 & 01 & 00 & 00 \\
00 & 01 & 01 & 00 \\
00 & 00 & 01 & 01
\end{pmatrix} \begin{pmatrix}
01 & 00 & 02 & 00 \\
00 & 01 & 00 & 02 \\
02 & 00 & 01 & 00 \\
00 & 02 & 00 & 01
\end{pmatrix} + \begin{pmatrix}
00 & 00 & 00 & 02 \\
02 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 02 & 00
\end{pmatrix}
\]

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 \( M_0 \) can be computed through the following equations.

\[
\begin{align*}
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{align*}
\begin{align*}
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{align*}
\begin{align*}
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{align*}
\begin{align*}
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{align*}

The total number and the maximum delay of XOR gates required for multiplication by \( M_0 \) are 112 and 4, respectively.

The matrix \( M_1 \) can be decomposed into the following.

\[
\begin{pmatrix}
01 & 06 & 08 & 02 \\
02 & 01 & 06 & 08 \\
08 & 02 & 01 & 06 \\
06 & 08 & 02 & 01
\end{pmatrix} = \begin{pmatrix}
01 & 04 & 00 & 00 \\
00 & 01 & 04 & 00 \\
00 & 00 & 01 & 02 \\
02 & 00 & 00 & 01
\end{pmatrix} \begin{pmatrix}
01 & 02 & 00 & 00 \\
00 & 01 & 02 & 00 \\
00 & 00 & 02 & 00 \\
00 & 00 & 00 & 02
\end{pmatrix}
\]

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 \( M_1 \) can be computed through the following equations.

\[
\begin{align*}
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{align*}
\begin{align*}
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{align*}
\begin{align*}
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{align*}
\begin{align*}
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{align*}

The total number and the maximum delay of XOR gates required for multiplication by \( M_1 \) are 128 and 4, respectively.

The matrices \( M_2 \) and \( M_3 \) are composed of the common row vectors to \( M_0 \) and \( M_1 \). Therefore, the multiplications by \( M_2 \) and \( M_3 \) are computed by substituting elements of an output vector of the multiplication by \( M_0 \) and \( M_1 \), respectively.

**Dividing F-functions for pipeline architecture**

In Fig. 5.3, we show the circuits of F-functions \( F_0 \) and \( F_1 \). The characters \( f, I \) and \( g \) in the figure represent the circuit of the function \( f \), the inverse function over \( \text{GF}(2^8) \) and the function \( g \) in the S-box \( S \), respectively. In Sec. 5.2.2, we apply the pipeline architecture to both Type-H3 and Type-H4 implementations of AURORA-256 and AURORA-512 in order to achieve higher throughput. By dividing the circuit \( F_0 \) into the two parts \( \alpha \) and \( \beta \) and inserting registers between \( \alpha \) and \( \beta \), 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 \( \alpha \) and \( \gamma \).

**5.2.2 Data Path Architectures**

For both AURORA-256 and AURORA-512, we designed four types of hardware implementations: Type-H1, Type-H2, Type-H3 and Type-H4 implementation. All the implementations do not include padding function; we assume that an input message is padded and divided into message blocks of 512 bits. We give the data path architecture of each implementation, where all registers represented by a box with shadow are composed of registers without enable signal.
AURORA-256 Type-H1

AURORA-256 Type-H1 implementation processes a round of the AURORA architecture both in one of the message scheduling module MSM and in the chaining value processing module CPM simultaneously in one clock cycle. It requires 8 F-function circuits and takes 18 cycles for both the compression function CF and the finalization function FF. Fig. 5.4 shows the data path architecture of AURORA-256 Type-H1 implementation. It is divided into two blocks: the message scheduling block and the chaining value processing block.

In the message scheduling block, a 512-bit message block is input in two cycles; the left 256-bit \( M_L \) is input at the 1st cycle and the right 256-bit \( M_R \) is input at the 2nd cycle. 256-bit intermediate values of \( MS_L \) (\( MSF_L \)) are stored in eight 32-bit registers \( \{ R_{00}, \ldots, R_{07} \} \) at the cycle of even order and stored in eight 32-bit registers \( \{ R_{10}, \ldots, R_{17} \} \) at the cycle of odd order. On the other hand, 256-bit intermediate values of \( MS_R \) (\( MSF_R \)) are stored in \( \{ R_{10}, \ldots, R_{17} \} \) at the cycle of even order and stored in \( \{ R_{00}, \ldots, R_{07} \} \) at the cycle of odd order. The pipeline architecture described in Sec. 5.2.1 is introduced into the message scheduling block; 32-bit registers are inserted between
\(\alpha\) and \(\beta\), and between \(\alpha\) and \(\gamma\). The architecture cannot shorten the critical path of the whole circuit because the longer paths exist in the chaining value processing block, but can reduce the rate of increase in area of the message scheduling block at high operating frequency. We note that the outputs of \(\beta\) and \(\gamma\) are byte-rotated to the left and to the right, respectively, when 256-bit intermediate values of \(MS_R\) (\(MSF_R\)) are processed.

In the chaining value processing block, the chaining value stored in eight 32-bit registers \(\{C_{00},...,C_{07}\}\) is loaded and set into eight 32-bit registers \(\{R_{20},...,R_{27}\}\) through the byte diffusion circuit \(BD\) after being XORed with the data fed from the message scheduling block and constant values \(CONC\). \(BD\) can be implemented by simple wiring of byte data without any transistors. From the 2nd cycle to the 17th cycle, the data stored in \(\{R_{20},...,R_{27}\}\) are input to the round function, and its output is re-stored into \(\{R_{20},...,R_{27}\}\) 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 \(\{C_{00},...,C_{07}\}\), and then re-stored into \(\{C_{00},...,C_{07}\}\). The 128-bit XOR gates required for updating \(\{C_{01},C_{03},C_{05},C_{07}\}\) can be merged with those for \(CONC\) by appending a 128-bit 2:1 selector.

**AURORA-256 Type-H2**

AURORA-256 Type-H2 implementation processes a round of the AURORA architecture both in one of \(MSM\) and in \(CPM\) simultaneously in two clock cycles, when the left 128-bit data are processed first. It requires 4 F-function circuits and takes 36 cycles for both \(CF\) and \(FF\). Fig.5.5 shows the data path architecture of AURORA-256 Type-H2 implementation, where the data path width is 128 bits. A 512-bit message block is input in 128-bit blocks using 4 cycles. \(PROTL_H\) and \(PROTR_H\) in the figure show the functions whose input and output are the left 128-bit of the input and output of the data rotating function \(PROTL\) and \(PROTR\), respectively. The number of F-functions and XOR gates are reduced to half compared to those in Type-H1 implementation. The pipeline architecture is introduced into the message scheduling block in order to reduce the
rate of increase in area of the message scheduling at high operating frequency.

In a 128-bit data path architecture such as Type-H2 implementation, the byte diffusion function $BD$ cannot be implemented only by simple wiring of byte data; generally it requires a 256-bit 2:1 selector. In our implementations, we utilize the 128-bit byte diffusion ($BD$) circuit, as shown in Fig. 5.6. The 128-bit $BD$ circuit consists of byte wiring, sixteen 8-bit registers and sixteen 8-bit 2:1 selectors, where selectors of 128 bits can be reduced. The 256-bit data, which are input into the 128-bit $BD$ circuit in two clock cycles, are output in the order corresponding to $BD$ by controlling selectors.

**AURORA-256 Type-H3**

AURORA-256 Type-H3 implementation processes a round of the AURORA architecture either in one of $MSM$ or in $CPM$ mutually in every one clock cycle. It requires 4 F-function circuits and takes 36 cycles for both $CF$ and $FF$. Fig. 5.7 shows the data path architecture of AURORA-256 Type-H3 implementation. Unlike AURORA-256 Type-H1 and Type-H2 implementation, the round function circuit is shared for $MSM$ and $CPM$. The round function is processed by repeating the following order:

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

111
We can shorten the critical path of the whole circuit and improve the maximum operating frequency by applying the pipeline architecture into the round function circuit.

The left 256-bit $M_L$ of a 512-bit message block is input at the 1st cycle, and then 256-bit intermediate values of $MS_L$ ($MSF_L$) are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$ or $\{R_{10}, \ldots, R_{17}\}$ or $\{R_{20}, \ldots, R_{27}\}$ by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \cdots$$

The right 256-bit $M_R$ of a 512-bit message block is input at the 3rd cycle, and then intermediate values of $MS_R$ ($MSF_R$) are stored in registers by repeating the same order as $MS_L$. On the other hand, the chaining value stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded at the 2nd cycle, and then 256-bit intermediate values of $CP$ ($CPF$) are stored in $\{R_{00}, \ldots, R_{07}\}$ or $\{R_{10}, \ldots, R_{17}\}$ by repeating the following order:

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

The input and output of F-functions must be adequately selected because either the kind or the positioning of F-functions among $MS_L$ ($MSF_L$), $MS_R$ ($MSF_R$) and $CP$ ($CPF$) is different; for intermediate values of $MS_R$ ($MSF_R$), the output of F-functions must be byte-rotated to the left or right. For intermediate values of $CP$ ($CPF$), both of the 1st and 3rd 32-bit line, and the 5th and 7th 32-bit line of the input and output of F-functions must be swapped. Note that the chaining value to be fed forward is XORed into intermediate values of $MS_R$ ($MSF_R$) through PROTL in advance at the 35th cycle, which can reduce one cycle for updating the chaining value.

**AURORA-256 Type-H4**

AURORA-256 Type-H4 implementation is hybrid of Type-H2 and Type-H3 implementation; it processes a round of the AURORA architecture either in one of $MSM$ or in $CPM$ mutually in every two clock cycles. It requires 2 F-function circuits and takes 72 cycles for both $CF$ and $FF$. Fig. 5.8 shows the data path architecture of AURORA-256 Type-H4 implementation, where the data path width is 128 bits. The round function circuit is shared for $MSM$ and $CPM$ in the same way as Type-H3 implementation. The processing order of the round function is also the same as
AURORA-256 Type-H3 implementation, but it requires two clock cycles for each round function. The pipeline architecture is introduced into the round function circuit, which can improve the maximum operating frequency.

The left 256-bit $M_L$ of a 512-bit message block is input in 128-bit blocks at the 1st and 2nd cycle, and then intermediate values of $MS_L$ ($MSF_L$) are stored in registers by repeating the following order:

$$
\{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{03}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{20}, \ldots, R_{23}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \cdots
$$

The right 256-bit $M_R$ of a 512-bit message block is input in 128-bit blocks at the 5th and 6th cycle, and then intermediate values of $MS_R$ ($MSF_R$) are stored in registers by repeating the same order as $MS_L$. On the other hand, the chaining value stored in four 32-bit registers $\{C_{00}, \ldots, C_{03}\}$ and $\{C_{10}, \ldots, C_{13}\}$ is loaded via $\{C_{10}, \ldots, C_{13}\}$ at the 3rd and 4th cycle, and then 256-bit intermediate values of $CP$ ($CPF$) are stored in registers by repeating the following order:

$$
\{R_{20}, \ldots, R_{23}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \cdots
$$

Note that the chaining value to be fed forward is XORed into intermediate values of $MS_R$ ($MSF_R$) in advance at the 69th and 70th cycle, which can reduce two cycles for updating the chaining value.

**AURORA-512 Type-H1**

AURORA-512 Type-H1 implementation processes a round of the AURORA architecture both in one of the message scheduling module $MSM$ and in the two chaining value processing modules $CPM$ simultaneously in one clock cycle. It requires 12 F-function circuits and takes 18 cycles for the compression functions $CF_s$ ($0 \leq s \leq 7$), the mixing functions $MF$ and the mixing function for finalization $MFF$. The data path architecture of AURORA-512 Type-H1 implementation can be constructed by appending another chaining value processing block to that of AURORA-256 Type-H1 implementation. In addition, two 256-bit paths from the eight 32-bit chaining value registers in both of the two chaining value processing blocks to the message scheduling block must be appended to process $MF$ and $MFF$. The two chaining value processing blocks are basically same except constant values and the F-functions circuits; one block arranges the F-function circuits of $F_1$ and $F_0$, and the other arranges those of $F_4$ and $F_2$.

**AURORA-512 Type-H2**

AURORA-512 Type-H2 implementation processes a round of the AURORA architecture both in one of $MSM$ and in two $CPM$ simultaneously in two clock cycles. It requires 6 F-function circuits and takes 36 cycles for $CF_s$, $MF$ and $MFF$. The data path architecture of AURORA-512 Type-H2 implementation can be constructed by appending another chaining value processing block to that of AURORA-256 Type-H2 implementation. In addition, two 128-bit paths from the four 32-bit chaining value registers in both of the two chaining value processing blocks to the message scheduling block must be appended to process $MF$ and $MFF$.

**AURORA-512 Type-H3**

AURORA-512 Type-H3 implementation processes a round of the AURORA architecture either in one of $MSM$ or in one of $CPM$ mutually in one clock cycle. It requires 4 F-function circuits and takes 56 cycles for $CF_s$, $MF$ and $MFF$: 54 cycle for message scheduling and chaining value processing, and 2 cycles for updating the chaining value. Fig. 5.9 shows the data path architecture of AURORA-512 Type-H3 implementation. The round function is processed by repeating the following order:

$$
MS_{L,i} \rightarrow CP_{L,i} \rightarrow CP_{R,i} \rightarrow MS_{R,i} \rightarrow CP_{L,i} \rightarrow CP_{R,i} \rightarrow \cdots
$$
for $0 \leq i \leq 9$. The pipeline architecture is introduced into the round function circuit, which can improve the maximum operating frequency.

For $CF_s$, the left 256-bit $M_L$ of a 512-bit message block is input at the 1st cycle. For $MF$ ($MFF$), the chaining value stored in eight 32-bit registers $\{C_{10}, \ldots, C_{17}\}$ is loaded at the 1st cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{L,i}$ are stored in eight 32-bit registers $\{R_{00}, \ldots, R_{07}\}$, $\{R_{10}, \ldots, R_{17}\}$, $\{R_{20}, \ldots, R_{27}\}$ or $\{R_{30}, \ldots, R_{37}\}$ by repeating the following order:

$$\{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{00}, \ldots, R_{07}\} \rightarrow \{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow $$
$$\{R_{30}, \ldots, R_{37}\} \rightarrow \cdots$$

For $CF_s$, the right 256-bit $M_R$ of a 512-bit message block is input at the 4th cycle. For $MF$ ($MFF$), the chaining value stored in eight 32-bit registers $\{C_{00}, \ldots, C_{07}\}$ is loaded via $\{C_{10}, \ldots, C_{17}\}$ at the 4th cycle as the input of $MS_{R,8}$ ($MS_{R,9}$). Intermediate values of $MS_{R,i}$ are stored in registers by repeating the same order as $MS_{L,i}$.

On the other hand, the chaining value stored in $\{C_{10}, \ldots, C_{17}\}$ is loaded at the 2nd cycle as the input of $CP_{L,i}$, and then intermediate values of $CP_{L,i}$ are stored in registers by repeating the following order:

$$\{R_{10}, \ldots, R_{17}\} \rightarrow \{R_{20}, \ldots, R_{27}\} \rightarrow \{R_{30}, \ldots, R_{37}\} \rightarrow \cdots$$

The chaining value stored in $\{C_{00}, \ldots, C_{07}\}$ is loaded via $\{C_{10}, \ldots, C_{17}\}$ at the 3rd cycle as the input of $CP_{R,i}$, and then intermediate values of $CP_{R,i}$ are stored in registers by repeating the same order as $CP_{L,i}$.

The input and output of F-functions must be adequately selected because the kind or the positioning of F-functions among $MS_{L,i}$, $MS_{R,i}$, $CP_{L,i}$ and $CP_{R,i}$ is different; for intermediate values of $MS_{R,i}$ and $CP_{R,i}$, the output of F-functions must be byte-rotated to the left or right. For intermediate values of $CP_{L,i}$ and $CP_{R,i}$, both of the 1st and 3rd 32-bit line, and the 5th and 7th 32-bit line of the input and output of F-functions must be swapped.
AURORA-512 Type-H4

AURORA-512 Type-H4 implementation processes a round of the AURORA architecture in one of MSM or in one of CPM mutually in two clock cycle. It requires 2 F-function circuits and takes 112 cycles for $CF_s$, $MF$ and $MFF$: 108 cycle for message scheduling and chaining value processing, and 4 cycles for updating the chaining value. Fig. 5.10 shows the data path architecture of AURORA-512 Type-H4 implementation, where the data path width is 128 bits. The processing order of the round function is the same as AURORA-512 Type-H3 implementation, but it requires two clock cycles for each round function.

For $CF_s$, the left 256-bit $M_L$ of a 512-bit message block is input in 128-bit blocks at the 1st and 2nd cycle. For $MF$ (MFF), the chaining value stored in four 32-bit registers $\{C_{30}, \ldots, C_{33}\}$ and $\{C_{20}, \ldots, C_{23}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 1st and 2nd cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{L,i}$ are stored in registers by repeating the following order:

\[
\{R_{00}, \ldots, R_{63}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{63}\} \rightarrow \{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{00}, \ldots, R_{63}\} \rightarrow \\
\{R_{10}, \ldots, R_{13}\} \rightarrow \{R_{20}, \ldots, R_{23}\} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \\
\{R_{50}, \ldots, R_{53}\} \rightarrow \{R_{60}, \ldots, R_{63}\} \rightarrow \ldots
\]

For $CF_s$, the right 256-bit $M_R$ of a 512-bit message block is input at the 7th and 8th cycle. For $MF$ (MFF), the chaining value stored in four 32-bit registers $\{C_{10}, \ldots, C_{13}\}$ and $\{C_{00}, \ldots, C_{03}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 7th and 8th cycle as the input of $MS_{L,8}$ ($MS_{L,9}$). Intermediate values of $MS_{R,i}$ are stored in registers by repeating the same order as $MS_{L,i}$.

On the other hand, the chaining value stored in $\{C_{30}, \ldots, C_{33}\}$ and $\{C_{20}, \ldots, C_{23}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 3rd and 4th cycle as the input of $CP_{L,i}$, and then intermediate values of $CP_{L,i}$ are stored in registers by repeating the following order:

\[
\{R_{20}, \ldots, R_{23}\} \rightarrow \{R_{30}, \ldots, R_{33}\} \rightarrow \text{128-bit BD circuit} \rightarrow \{R_{40}, \ldots, R_{43}\} \rightarrow \{R_{50}, \ldots, R_{53}\} \rightarrow \\
\{R_{60}, \ldots, R_{63}\} \rightarrow \ldots
\]

The chaining value stored in $\{C_{10}, \ldots, C_{13}\}$ and $\{C_{00}, \ldots, C_{03}\}$ is loaded via $\{C_{30}, \ldots, C_{33}\}$ at the 5th and 6th cycle as the input of $CP_{R,i}$, and then intermediate values of $CP_{R,i}$ are stored in registers by repeating the same order as $CP_{L,i}$.

Figure 5.10: Data path architecture of AURORA-512 Type-H4 implementation.
5.2.3 Evaluation Results

We show our evaluation results on hardware performance of AURORA-256 and AURORA-512 at the present. For both AURORA-256 and AURORA-512, Type-H1, Type-H2, Type-H3 and Type-H4 implementations with S-boxes based on inversion over $\text{GF}((2^4)^2)$ are evaluated. In addition, Type-H1 implementation with table-lookup S-boxes is also evaluated in order to achieve higher throughput. Control signals for all selectors and constant values are generated in a controller module which is included in each implementation.

The environment of our hardware design and evaluation is as follows.

<table>
<thead>
<tr>
<th>Language</th>
<th>Verilog-HDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design library</td>
<td>0.13 $\mu$m CMOS ASIC library</td>
</tr>
<tr>
<td>Simulator</td>
<td>VCS version 2006.06</td>
</tr>
<tr>
<td>Logic synthesis</td>
<td>Design Compiler version 2007.03-SP3</td>
</tr>
</tbody>
</table>

One gate is equivalent to a 2-way NAND and speed is evaluated under the worst-case conditions. Table 5.14 represents the evaluation results. For each implementation of AURORA-256 and AURORA-512, two types of circuits are synthesized by specifying either area or speed optimization. In the addition, we investigate the condition to maximize “Efficiency” that indicates “Throughput” per area, which we call efficiency optimization. For AURORA-256 implementations, “Throughput” is defined as follows:

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

On the other hand, for AURORA-512 implementations, “Throughput” is defined as follows:

$$\text{Throughput [Mbps]} = \frac{\text{Frequency [MHz]} \times \text{Block Size (512 [bits])}}{\text{Cycles}} \times \frac{8}{9},$$

because the mixing functions $MF$ are inserted after every 8 compression functions $CF_0, CF_1, \ldots, CF_7$.

We also show, for comparison, the best known results of hardware performance of SHA-2 using a 0.13 $\mu$m CMOS ASIC library by Satoh et al. [49]. The performance of AURORA cannot be directly compared with them because different design libraries and different logic synthesis tools were used. However, AURORA enables a variety of implementations from small-area to high-throughput implementations; for AURORA-256, the smallest area (11,111 gates) is about 3% smaller with about 2.06 times higher efficiency (196.1 Kbps/gate) than that of SHA-224/256 (11,184 gates, 95.4 Kbps/gate), and the highest throughput (10,352 Mbps) is about 4.37 times higher than that of SHA-224/256 (2,370 Mbps). For AURORA-512, the smallest area (14,613 gates) is about 37% smaller with about 30% higher efficiency (81.5 Kbps/gate) than that of SHA-384/512 (23,146 gates, 62.8 Kbps/gate), and the highest throughput (9,132 Mbps) is about 3.14 times higher than that of SHA-224/256 (2,909 Mbps).

The highest efficiency of AURORA-256 (344.3 Kbps/gate) and AURORA-512 (194.9 Kbps/gate) is about 2.23 times and 1.83 times higher than that of SHA-224/256 (154.6 Kbps/gate) and SHA-384/512 (106.6 Kbps/gate), respectively, which indicates that AURORA is highly efficient hash function family in hardware implementation.
Table 5.14: Results on Hardware Performance of AURORA-256 and AURORA-512.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AURORA-256 (0.13µm)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-H1</td>
<td>18</td>
<td>GF((2^4)^2)</td>
<td>18,883</td>
<td>194.3</td>
<td>5,528</td>
<td>292.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24,645</td>
<td>287.9</td>
<td>8,189</td>
<td>332.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20,825</td>
<td>252.1</td>
<td>7,171</td>
<td>344.3</td>
</tr>
<tr>
<td></td>
<td>Table</td>
<td></td>
<td>27,854</td>
<td>213.2</td>
<td>6,065</td>
<td>217.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35,016</td>
<td>363.9</td>
<td>10,352</td>
<td>295.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32,997</td>
<td>345.9</td>
<td>9,838</td>
<td>298.2</td>
</tr>
<tr>
<td>Type-H2</td>
<td>36</td>
<td>GF((2^4)^2)</td>
<td>13,446</td>
<td>189.2</td>
<td>2,691</td>
<td>200.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17,797</td>
<td>293.9</td>
<td>4,180</td>
<td>234.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15,523</td>
<td>266.2</td>
<td>3,786</td>
<td>243.9</td>
</tr>
<tr>
<td>Type-H3</td>
<td>36</td>
<td>GF((2^4)^2)</td>
<td>15,173</td>
<td>260.7</td>
<td>3,707</td>
<td>244.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23,490</td>
<td>464.3</td>
<td>6,603</td>
<td>281.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17,064</td>
<td>360.9</td>
<td>5,132</td>
<td>300.8</td>
</tr>
<tr>
<td>Type-H4</td>
<td>72</td>
<td>GF((2^4)^2)</td>
<td>11,111</td>
<td>306.4</td>
<td>2,179</td>
<td>196.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14,255</td>
<td>475.3</td>
<td>3,380</td>
<td>237.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12,257</td>
<td>423.6</td>
<td>3,012</td>
<td>245.7</td>
</tr>
<tr>
<td>SHA-224/256 (0.13µm)</td>
<td>-</td>
<td>-</td>
<td>11,484</td>
<td>154.1</td>
<td>1,096</td>
<td>95.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15,329</td>
<td>333.3</td>
<td>2,370</td>
<td>154.6</td>
</tr>
<tr>
<td>AURORA-512 (0.13µm)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-H1</td>
<td>18</td>
<td>GF((2^4)^2)</td>
<td>29,235</td>
<td>195.5</td>
<td>4,943</td>
<td>169.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40,219</td>
<td>285.4</td>
<td>7,217</td>
<td>179.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31,746</td>
<td>244.7</td>
<td>6,187</td>
<td>194.9</td>
</tr>
<tr>
<td></td>
<td>Table</td>
<td></td>
<td>42,691</td>
<td>213.2</td>
<td>5,391</td>
<td>126.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>56,748</td>
<td>361.2</td>
<td>9,132</td>
<td>160.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>48,377</td>
<td>317.1</td>
<td>8,018</td>
<td>165.9</td>
</tr>
<tr>
<td>Type-H2</td>
<td>36</td>
<td>GF((2^4)^2)</td>
<td>20,685</td>
<td>185.7</td>
<td>2,347</td>
<td>113.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>28,538</td>
<td>286.3</td>
<td>3,619</td>
<td>127.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22,731</td>
<td>244.7</td>
<td>3,093</td>
<td>136.1</td>
</tr>
<tr>
<td>Type-H3</td>
<td>56</td>
<td>GF((2^4)^2)</td>
<td>19,335</td>
<td>236.4</td>
<td>1,921</td>
<td>99.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25,915</td>
<td>455.8</td>
<td>3,705</td>
<td>143.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22,129</td>
<td>406.2</td>
<td>3,302</td>
<td>149.2</td>
</tr>
<tr>
<td>Type-H4</td>
<td>112</td>
<td>GF((2^4)^2)</td>
<td>14,613</td>
<td>293.1</td>
<td>1,191</td>
<td>81.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16,969</td>
<td>504.2</td>
<td>2,049</td>
<td>120.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16,670</td>
<td>496.7</td>
<td>2,018</td>
<td>121.1</td>
</tr>
<tr>
<td>SHA-384/512 (0.13µm)</td>
<td>-</td>
<td>-</td>
<td>23,146</td>
<td>125.0</td>
<td>1,455</td>
<td>62.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>27,297</td>
<td>250.0</td>
<td>2,909</td>
<td>106.6</td>
</tr>
</tbody>
</table>

For each implementation, the 1st row and the 2nd row show the results of the synthesized circuits by area and speed optimization, respectively. The 3rd row also shows the results by efficiency optimization for each implementation of AURORA-256 and AURORA-512.
Chapter 6

Applications of AURORA

6.1 Digital Signature

The digital signature standard (DSS) is specified in FIPS 186-2 [20]. In this standard, the hash function SHA-1 specified in FIPS 180-1 (FIPS 180-3) is used in many occasions including the generation of a message digest, the generation and the verification of parameters [19]. Due to that the same hash size of SHA-1 is not supported by 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 [21]. In the draft, usages of SHA-2 algorithm family are specified. Thus, our AURORA algorithm can be used as a replacement of corresponding SHA-2 algorithm which has the same hash size.

6.2 Keyed-Hash Message Authentication Code (HMAC)

In FIPS 198, the keyed hash message authentication code (HMAC) is standardized [23]. From the definition of HMAC that any hash function can be applicable in principle, any algorithm of AURORA family can be used as a base hash function for it. The output length $L$ and the block length $B$ should be selected according to the specification of a considered hash function. Table 6.1 summarizes the actual values of $L$ and $B$ for each AURORA hash algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$L$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AURORA-224</td>
<td>224</td>
<td>512</td>
</tr>
<tr>
<td>AURORA-256</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>AURORA-384</td>
<td>384</td>
<td>512</td>
</tr>
<tr>
<td>AURORA-512</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>AURORA-224M</td>
<td>224</td>
<td>512</td>
</tr>
<tr>
<td>AURORA-256M</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>
6.3 Key Establishment Schemes Using Discrete Logarithm Cryptography

The pair-wise key establishment schemes using discrete logarithm cryptography is described in NIST SP800-56A [40]. In this document, minimum bit length of the hash function output is assigned according to the selected parameter set on of FA, FB, FC, EA, EB, EC, ED, and EE. Among them FB and EB require 224-bit output, FC and EC require 256-bit output. ED and EE require 384-bit and 512-bit output, respectively. Accordingly, AURORA algorithms can be used when one of the above domain parameters is selected. To be concrete, AURORA algorithm is used as a hash function \( H \) in the concatenation key derivation function or the ASN.1 key derivation function use a hash function in the document.

6.4 Random Number Generation Using Deterministic Random Bit Generators

NIST SP800-90 specifies the recommendation for random number generation using deterministic random generators (DRBG) [41]. There are three DRBGs that use a secure hash function. HMAC$_{DRBG}$ uses the aforementioned HMAC scheme, thus AURORA algorithms can be applied by following the rule of the HMAC. Hash$_{DRBG}$ and Dual$_{EC}$DRBG employ a derivation function using a hash function called Hash$_{df}$ which call one of SHA-1 and SHA-2 algorithms. Accordingly, one of AURORA algorithms can be used as a replacement for one of SHA-2 algorithm called in Hash$_{df}$. It may be helpful to note that the seed length for Hash$_{DRBG}$ is 440-bit when using AURORA-384 and AURORA-512, on the other hand the seed length is 888-bit when using SHA-384 and SHA-512. This is due to the block length for these AURORA algorithms are 512-bit, not 1024-bit. However this is consistent with the specification because it is required that minimum entropy for seed and reseed are 192-bit and 256-bit for AURORA-384 and AURORA-512, respectively. The specified seed length 440-bit apparently exceeds these minimum required entropy.
Chapter 7

Advantages and Limitations

The hash function family AURORA has the following advantages and limitations. The 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 [38], we defined one of our design goals 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, microprocessors 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 sacrificing the well-balanced performance on multi platform implementations. But as explained above, we did not aim for the excellent performance only on specific platform.

• **Sufficient Security Arguments**
  Moreover, as for the security evaluation, we tried to adopt well-studied components to construct AURORA, otherwise newly developed components are employed if reasonable security arguments are provided for the components. For the AURORA structure, the strength against differential cryptanalysis and impossible differential cryptanalysis can be evaluated in a relatively reasonable way. For the new domain extension Double Mix Merkle-Damgaard (DMMD) transform for the longer output, the expected security proof has been provided.

• **Multicollision Resistance with Low Additional Cost**
  Furthermore, we adopted the DMMD transform to offer multicollision resistance by combining parallel compression functions and mixing functions together. As a result, we can provide AURORA-256M, which is an almost identical hash function with AURORA-512, and additional implementation cost from AURORA-256 is limited. This fact emphasizes that the AURORA hash function family has good consistency among hash functions in the family.
Acknowledgments

We would like to express our deep appreciation to Asami Mizuno, Satoshi Higano, and Eiji Fujii for their kind support for this hash design project. Thanks also to Kazuya Kamio, Tadaoki Yamamoto and Hiroyuki Abe for evaluating performance of AURORA algorithms. We would also like to thank Koichi Sakamoto for his support for analysis of AURORA.
Bibliography


123
cycle.h. available at http://www.fftw.org/cycle.h


