Sarmal: SHA-3 Proposal

Kerem VARICI

Institute of Applied Mathematics
Middle East Technical University
06531, Ankara
TURKEY
varicikerem@gmail.com

Onur ÖZEN*

Laboratory for Cryptologic Algorithms
INJ331 I&J, Station 14, EPFL
1015, Lausanne
SWITZERLAND
onur.ozen@epfl.ch

Çelebi KOCAiR

Department of Computer Engineering
Middle East Technical University
06531, Ankara
TURKEY
celebi@ceng.metu.edu.tr

*Most of the work was done when the submitter was at Middle East Technical University

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hash functions are one of the milestones of the field of cryptology that are extensively used in various applications including message integrity, message authentication, address generation and verification, digital signatures and several others each demanding corresponding security properties of the underlying hash function.

Recent breakthroughs in the design and analysis of cryptographic hash functions led to great developments in this field including a demand in a new hash standard `SHA-3`[51]. In this document, we describe a new hash function family Sarmal as a `SHA-3`[51] candidate. Starting from the mathematical preliminaries and the necessary notation throughout the document, we describe the specification, design rationale, security, implementation and performance of Sarmal Hash Family. We conclude with the acknowledgements, references and appendix.

Chapter 2 mainly deals with the necessary mathematical background and the notation used in the document which help to understand the properties of Sarmal Hash Family. Necessary mathematical background is quite familiar from the existing literature which is basic finite field and modular arithmetic. Notation, on the other hand, is fixed to be used throughout the document.

Chapter 3 is dedicated to the specification of the Sarmal Hash Family which makes it clear to understand and implement the overall hash function. This chapter is divided into two sections that cover the specification of the mode of operation and the compression function respectively. Specification of the mode of operation details how a given message is used to create the corresponding digest. Specification of the compression function describes the components of the underlying compression function used in the mode of operation. We provide the design rationale behind the specification in Chapter 4 which covers the reasons why the underlying primitives are used as components of Sarmal.

Chapter 5 consists the basic security claims about Sarmal Hash Family. Again, we make a distinction between the security of the mode of operation and the compression function of Sarmal despite of the fact that they are closely related to each other. That is, in the first part, assuming the underlying compression function has no known weaknesses, namely ideal, we provide the security claims for the mode of operation. In the second section, we give the security analysis of Sarmal's compression function against known attack scenarios. Here, we maturely assume the *blindness of a designer* and conjecture that the Sarmal compression function is secure.

In Chapter 6, implementation and performance results of Sarmal Hash Family are given. We provide performance figures on 32/64-bit processors and comment the performance of Sarmal Hash Family on 8-bit processors. Besides, a detailed explanation is provided about the optimized implementation of Sarmal Hash Family.

Chapter 2

Preliminaries

2.1 Notation

Throughout the document we use a fixed notation which is given in Table 2.1. As a convention we number the words and bytes from left to right. The specific values are shown in hexadecimal and denoted by $:_x$ and the binary representation is denoted by $(:)_2$. Index i is used to show the ith compression function evaluation.

Table 2.1: Notation

Variable	Size	Definition
$\oplus$		Exclusive OR (XOR) Operation
$\boxplus$		Addition Modulo 2^{64}
$\boxminus$		Subtraction Modulo 2^{64}
w	64-bit	Word
$H(M,s,d)$		Sarmal Hash Function
$f(h_{i-1},M_i,s,t_i)$		Compression Function of Sarmal
$G(.,.)$		Round Function
$g(.)$		Nonlinear Subround Function
$\sigma_k(M_i)$		Message Permutation
h_i	$8w$	Chaining Value
X_i	$8w$	State Value
X_i^{left}	$8w$	Left State Value
X_i^{right}	$8w$	Right State Value
$X_{i,r'}^{left}[j]$	w	jth word of the left state after r' rounds
$X_{i,r'}^{right}[j]$	w	jth word of the right state after r' rounds
M		Message to be hashed
M_i	$16w$	i^{th} Message Block

s	$4w$	Salt Value
t_i	w	Number of bits hashed up to ith f evaluation
c	$2w$	Constant Value
r		Non-negative number of rounds
$.[i]$	w	i^{th} word of given value '.'
$.[i \cdots j]$	$(i-j+1)w$	The words from i to j for given value '.'
$a_0 \| a_1 \| \cdots \| a_n$		Concatenation of the n blocks of data
$S[.]$	8×8-bit	S-box Transformation
$A_{8\times 8}$		8×8 Maximum Distance Seperable (MDS) Matrix

2.2 Mathematical Background

2.2.1 $GF(2^8)$ Arithmetic

Mathematical operations used in Sarmal are quite common in the cryptology literature. One of the basic mathematical operations in the compression function is the arithmetic operations over $GF(2^8)$. The structure of the finite field is of the form $GF(2)[x]/p(x)$ where $p(x)$ is primitive polynomial over $GF(2)$ which is given by $p(x) = x^8 + x^4 + x^3 + x^2 + 1$ [38]. Thus, the elements in $GF(2^8)$ can be represented as polynomials over $GF(2)$ whose degrees are less than 8. As an example, a byte $a = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ is mapped to the polynomial:

$$a = a_7 \cdot x^7 + a_6 \cdot x^6 + a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x^1 + a_0 \cdot x^0$$

Example:

$$\begin{aligned} 65_x &= (01100101)_2 \\ &= 0 \cdot x^7 + 1 \cdot x^6 + 1 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \\ &= 1 \cdot x^6 + 1 \cdot x^5 + 1 \cdot x^2 + 1 \cdot x^0 \end{aligned}$$

Addition in $GF(2^8)$: Addition of polynomials in $GF(2^8)$ is the bitwise XOR of the corresponding binary representations of the polynomials.

Example: Let $f(x) = x^7 + x^6 + x^2 + 1$ and $g(x) = x^4 + x^3 + x^2$ be two polynomials defined over the finite field above. Then,

$$\begin{aligned} f(x) + g(x) &= (x^7 + x^6 + x^2 + 1) + (x^4 + x^3 + x^2) \\ &= x^7 + x^6 + x^4 + x^3 + 1 \\ f(x) + g(x) &= (11000101)_2 \oplus (00011100)_2 \\ &= (11011001)_2 \end{aligned}$$

Multiplication in $GF(2^8)$: Multiplication of two bytes (or polynomials) in $GF(2^8)$ is done by the multiplication of the corresponding polynomials over the finite field described above. Two polynomials are multiplied and reduced to modulo $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

Example: Let $D8_x$ and $4A_x$ be two bytes. Then,

$$\begin{aligned}
D8_x \cdot 4A_x &= (11011000)_2 \cdot (01001010)_2 \\
&= (x^7 + x^6 + x^4 + x^3) \cdot (x^6 + x^3 + x) \\
&= x^{13} + x^{12} + x^{10} + x^9 + x^{10} + x^9 + x^7 + x^6 + x^8 + x^7 + x^5 + x^4 \\
&= x^{13} + x^{12} + x^8 + x^6 + x^5 + x^4 \\
&= x^8 \cdot (x^5 + x^4 + 1) + x^6 + x^5 + x^4 \\
&= (x^4 + x^3 + x^2 + 1)(x^5 + x^4 + 1) + x^6 + x^5 + x^4 \\
&= x^5 + x^2 + x + 1
\end{aligned}$$

Circulant Matrix An $m \times m$ matrix (in our case 8×8) which is of the form

$$C = \begin{bmatrix}
c_0 & c_1 & \cdot & \cdot & \cdot & & c_{m-2} & c_{m-1} \\
c_{m-1} & c_0 & c_1 & & & & & c_{m-2} \\
\cdot & c_{m-1} & c_0 & \cdot & & & & \cdot \\
\cdot & \cdot & \cdot & & \cdot & & & \cdot \\
\cdot & \cdot & & \cdot & & \cdot & & \cdot \\
& \cdot & & & \cdot & & \cdot & \\
c_2 & & & & & & & c_1 \\
c_1 & c_2 & \cdot & \cdot & \cdot & & c_{m-1} & c_0
\end{bmatrix}$$

called *a circulant matrix* over $GF(2^8)$ (i.e. $c_i \in GF(2^8)$). This special type of a matrix is used in the nonlinear subround function g which has significant advantages both in security and implementation.

Chapter 3

Specification

The specification of Sarmal Hash Family consists of the specification of the mode of operation and the compression function of Sarmal. In this chapter, we provide the necessary information to be able to implement and understand the description of Sarmal.

Sarmal Hash Family accepts messages M of arbitrary length (no more than $(2^{64} - 1)$-bits) as input and produces various d-bit message digests D by using Sarmal Hash Function $H(M, s, d)$:

$$H : \{0,1\}^* \times \{0,1\}^{4w} \times \Delta \rightarrow \{0,1\}^d$$

where $d \in \Delta = \{224, 256, 384, 512\}$.
Each member of Sarmal uses same structure with minor differences which is mainly due to the variable digest size d:

- Each Sarmal-d has different initial and constant values.
- Number of rounds r in compression function of f is 16 and 20 for Sarmal-224/256 and Sarmal-384/512 respectively.
- 8 different message permutations are used in Sarmal-224/256 while 10 different message permutations are used in Sarmal-384/512.
- Each Sarmal-d has different number of d-bit truncations at the end.

The operations in Sarmal are described starting from the mode of operation followed by the specification of the compression function in the following sections.

3.1 Sarmal Mode of Operation

Sarmal follows an iterative mode of operation which has been recently proposed as HAIFA [12] (HAsh Iterative FrAmework). In HAIFA, additional parameters, such as salt s and the number of bits hashed up to i^{th} iteration t_i, are added to the standard Merkle-Damgård construction [21, 46] with a different padding rule. The reason behind this is to provide randomized hashing and withstand the latest attack scenarios which have been revealed in recent years [22, 32, 33, 35]. We describe the security properties of the Sarmal mode of operation in detail in Chapter 5.

The input of the Sarmal Hash Family is a message M of arbitrary length l ($l < 2^{64} - 1$), the user supplied salt s and the digest size d. Sarmal mode of operation starts with an injective padding rule (see Section 3.1.1) to extend the length of M to a multiple of $16w$. Then, the padded message $M' = (M_1 \| \cdots \| M_n)$ is divided into $16w$-bit message blocks M_i to which the compression function f is applied iteratively until the end of message blocks. Chaining values h_i which are the output of the compression function f at the end of each iteration are of $8w$-bit and calculated exactly the same manner for all digest sizes. As described above, the only differences are the constants, initial values and the number of rounds for different digest sizes. The message digest D which is of $8w$-bit, is calculated after truncation to d bits of the last chaining value h_n. The details of the compression function are provided in Section 3.2. The overall process is described in Table 3.1.

Table 3.1: Sarmal Mode of Operation

Input:	M**:** l-bit Message Value ($l \leq 2^{64} - 1$)
	s**:** $4w$-bit Salt Value
	d**:** d-bit Digest Size
Output:	$H(M, s, d) = D$**:** Hash value of the message M
Preprocess:	
1.	Pad the message M according to the procedure in Section 3.1.1.
2.	Divide the padded message into n $16w$-bit blocks, $M' = (M_1, M_2, \cdots M_n)$.
3.	Initialize $IV = h_0$ and c using the Table 3.4
Process:	
1.	**for**$(1 \leq i \leq n)$
	{
	$h_i = f(h_{i-1}, M_i, s, t_i)$
	}
Output Generation:	
1. Sarmal-224**:**	$H(M, s, d) =$ right most $224-$bit of $h_n[4 \cdots 7]$
2. Sarmal-256**:**	$H(M, s, d) = h_n[0 \cdots 3]$
3. Sarmal-384**:**	$H(M, s, d) = h_n[0 \cdots 5]$
4. Sarmal-512**:**	$H(M, s, d) = h_n$

3.1.1 Padding

Padding is necessary for all iterative mode of operations as the underlying compression functions are defined by fixed sized input and outputs. In Sarmal, we use the same padding rule for all digest sizes except for the step where the digest size d is added. It is an additional update to the standard Merkle-Damgård strengthening which is specified in the proposal of HAIFA [12].

As the compression function f of Sarmal accepts message blocks M_i of length $16w$ bits, the aim is to pad the message to a multiple of $16w$ bits without any security loss. We use exactly the same padding rule given in [12] which is specified in Table 3.2. It basically appends one bit to the end of the message and additional zero bits until the length of the message is congruent to $16w - w - l$ modulo $16w$. Finally the length of the message and the digest sizes are encoded in w and l bits respectively. The details are given in Table 3.2.

Table 3.2: Padding

Input:	M: l-bit Message Value
Output:	A multiple of $16w$-bit Padded Message.
Process:	
1.	Check the length of the message M. If it is congruent to 950 modulo $16w$, pass to step 4.
2.	Add a single bit '1' to end of the message. Check the length of the new message. If it is congruent to 950 modulo 1024, pass to step 4.
3.	Add 0-bits following the bit 1 until the length of the message is congruent to 950 modulo 1024.
4.	Pad the hash size d as a 10-bit string. (0011100000, 0100000000, 0110000000, 1000000000 are the 10-bit strings which are used for Sarmal-224/256/384/512, respectively.)
5.	Pad the message length l in 64-bits.
Output Generation:	
1.	$M' = (M_1, M_2, \cdots M_n)$

3.2 Sarmal Compression Function

3.2.1 High Level Description of f

Compression function $f(h_{i-1}, M_i, s, t_i)$ of Sarmal, at ith step, takes the previous chaining value h_{i-1} of $8w$-bit, message block M_i of $16w$-bit, user supplied salt s of $4w$-bit and the number of bits hashed t_i up to step

i of w-bit as inputs at each step and produces $8w$-bit output h_i. It is defined as follows:

$$f : \{0,1\}^{8w} \times \{0,1\}^{16w} \times \{0,1\}^{4w} \times \{0,1\}^{w} \longrightarrow \{0,1\}^{8w}.$$

Compression function makes use of two parallel parts operating independently each consisting of same nonlinear round function G and a Davies-Meyer form feedforward at the end. The security properties and the design rationale behind f are provided in Chapters 4 and 5 respectively. The general scheme of compression function of Sarmal is visualized in Figure 3.1 and the details are given in Table 3.3.

Table 3.3: Compression function of ith Step of Sarmal

Input:	M_i**:** $16w$-bit Message Block	
	s**:** $8w$-bit Salt Value	
	t_i**:** w-bit Number of bits hashed up to ith step	
	h_{i-1}**:** $8w$-bit Previous Chaining Value	
Output:	h_i**:** $8w$-bit Following Chaining Value	
Preprocess:		
1.	Obtain σ and c from Table 3.9 and Table 3.4 resp.	
Process:		
1.		$X_0^{left} = h_{i-1}[0\cdots 3] \parallel s[0\cdots 1] \parallel c[0] \parallel t_i$
2.		$X_0^{right} = h_{i-1}[4\cdots 7] \parallel s[2\cdots 3] \parallel c[1] \parallel t_i$
3.		**for**$(1 \leq j \leq r)$
		$\{$
a)		$k = \lfloor \frac{j-1}{4} \rfloor$
b)		$\ell \equiv (4j-1)\ mod\ 16$
c)		$X_j^{left} = G(X_{j-1}^{left}, \sigma_k(M_i)[(\ell-3)\cdots\ell])$
d)		$X_j^{right} = G(X_{j-1}^{right}, \sigma_{k+(r/4)}(M_i)[(\ell-3)\cdots\ell])$
		$\}$
Output Generation:		
1.		$h_i = (X_r^{left} \oplus X_r^{right}) \oplus h_{i-1}$

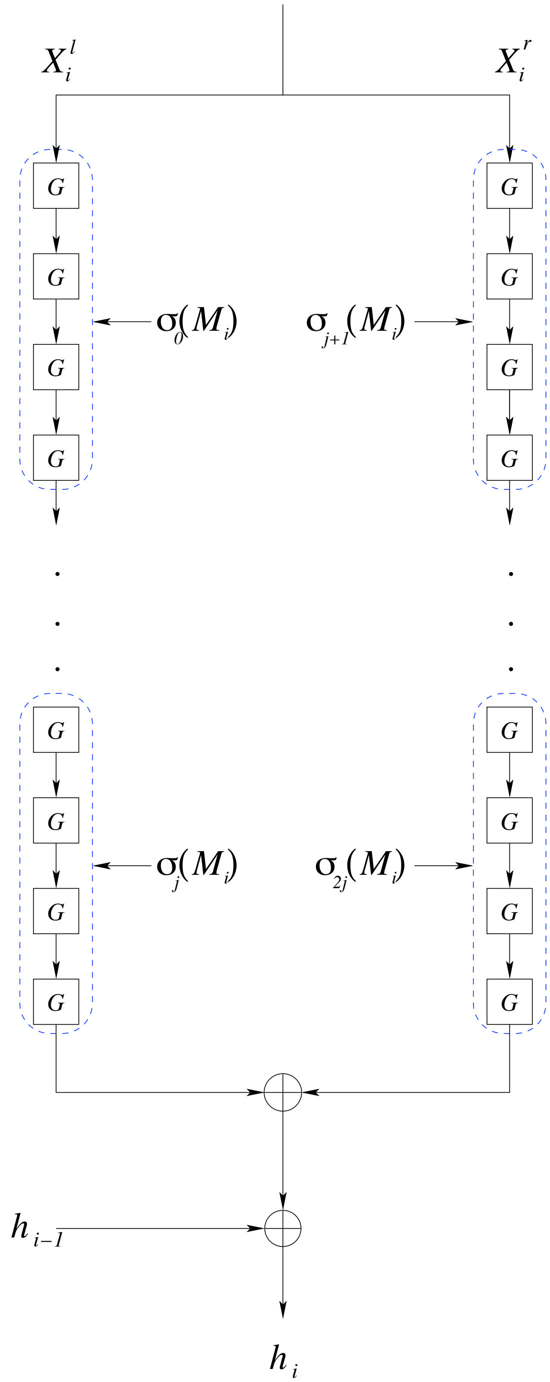

Figure 3.1: Compression function f of Sarmal

3.2.2 Initial Values and Constants

Different $8w$-bit initial values h_0 and $2w$-bit constants c are required for the evaluation of f which are given in Tables 3.4 and 3.5. The values are different for various digest sizes and obtained from fractional part of the square root of 3, golden ratio, square root of 5 and π for the Sarmal-224, Sarmal-256, Sarmal-384 and Sarmal-512 respectively.

Table 3.4: Initial Values of Sarmal

Initial Values of Sarmal-224	
$h_0[0] = BB67AE8584CAA73B_x$	$h_0[4] = 490BCFD95EF15DBD_x$
$h_0[1] = 25742D7078B83B89_x$	$h_0[5] = A9930AAE12228F87_x$
$h_0[2] = 25D834CC53DA4798_x$	$h_0[6] = CC4CF24DA3A1EC68_x$
$h_0[3] = C720A6486E45A6E2_x$	$h_0[7] = D0CD33A01AD9A383_x$
Constants of Sarmal-224	
$c[0] = B9E122E6138C3AE6_x$	$c[1] = DE5EDE3BD42DB730_x$
Initial Values of Sarmal-256	
$h_0[0] = 9E3779B97F4A7C15_x$	$h_0[4] = 2767F0B153D27B7F_x$
$h_0[1] = F39CC0605CEDC834_x$	$h_0[5] = 0347045B5BF1827F_x$
$h_0[2] = 1082276BF3A27251_x$	$h_0[6] = 01886F0928403002_x$
$h_0[3] = F86C6A11D0C18E95_x$	$h_0[7] = C1D64BA40F335E36_x$
Constants of Sarmal-256	
$c[0] = F06AD7AE9717877E_x$	$c[1] = 85839D6EFFBD7DC6_x$

Table 3.5: Cont. Initial Values of Sarmal

Initial Values of Sarmal-384	
$h_0[0] = 3C6EF372FE94F82B_x$	$h_0[4] = 4ECFE162A7A4F6FE_x$
$h_0[1] = E73980C0B9DB9068_x$	$h_0[5] = 068E08B6B7E304FE_x$
$h_0[2] = 21044ED7E744E4A3_x$	$h_0[6] = 0310DE1250806005_x$
$h_0[3] = F0D8D423A1831D2A_x$	$h_0[7] = 83AC97481E66BC6D_x$
Constants of Sarmal-384	
$c[0] = E0D5AF5D2E2F0EFD_x$	$c[1] = 0B073ADDFF7AFB8C_x$
Initial Values of Sarmal-512	
$h_0[0] = 243F6A8885A308D3_x$	$h_0[4] = 452821E638D01377_x$
$h_0[1] = 13198A2E03707344_x$	$h_0[5] = BE5466CF34E90C6C_x$
$h_0[2] = A4093822299F31D0_x$	$h_0[6] = C0AC29B7C97C50DD_x$
$h_0[3] = 082EFA98EC4E6C89_x$	$h_0[7] = 3F84D5B5B5470917_x$
Constants of Sarmal-512	
$c[0] = 9216D5D98979FB1B_x$	$c[1] = D1310BA698DFB5AC_x$

3.2.3 G Function

G is the nonlinear round function of f which is a special Generalized Unbalanced Feistel Network (GUFN) with 8-branches of w-bit aords each. Contrary to the standard Generalized Unbalanced Networks, Sarmal uses 2 different branches to update 6 remaining ones. An AES [20](or Whirlpool[5])-like nonlinear subround function g is used together with the basic arithmetic operations like XOR, addition and subtraction modulo 2^{64}. At each G evaluation, $4w$-bit of permuted message is mixed with the input data and $4G$ evaluations use whole $16w$-bit of message block M_i. Round function can be defined as follows:

$$G : \{0,1\}^{8w} \times \{0,1\}^{4w} \rightarrow \{0,1\}^{8w}$$

The number of G evaluations are same for parallel left and right parts. However, it changes for different digest sizes (16 and 20 for Sarmal-224/256 and Sarmal-384/512 respectively). The security properties and the design rationale behind G are provided in Chapters 4 and 5 respectively. The general view of G is given in Figure 3.2 and the operations are described in Table 3.6.

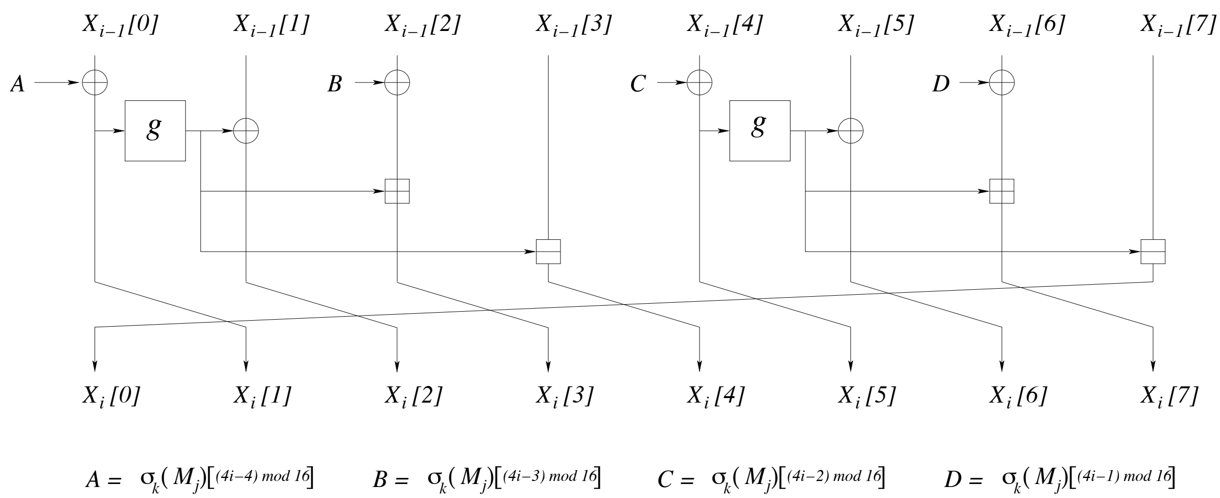

Figure 3.2: G Function

3.2.4 g Function

The nonlinear subround function g is a component of G which is defined on w-bit words:

$$g : \{0,1\}^{w} \rightarrow \{0,1\}^{w}$$

It is an AES [20](or Whirlpool[5])-like Substitution-Permutation Network (SPN) which makes use of 8 parallel 8×8-bit S-box followed by a permutation layer which is defined on $GF(2^8)$ and similar to the one in Whirlpool. Function g is described in the Table 3.7 and in visualized Figure 3.3.

Table 3.6: Description of G at r'th Round

Input:	$8w$-bit State Value $X^{r'-1}$
	$4w$-bit Permuted Message $\sigma_k(M_j)[(i-3)\cdots i]$
Output:	$8w$-bit Updated State Value $X^{r'}$
PreProcess:	
1.	$A = \sigma_k(M_j)[(4i-4)\ mod\ 16]$
2.	$B = \sigma_k(M_j)[(4i-3)\ mod\ 16]$
3.	$C = \sigma_k(M_j)[(4i-2)\ mod\ 16]$
4.	$D = \sigma_k(M_j)[(4i-1)\ mod\ 16]$
Process:	
1.	$X_i[0] = X_{i-1}[7] \boxminus g(X_{i-1}[4] \oplus C)$
2.	$X_i[1] = X_{i-1}[0] \oplus A$
3.	$X_i[2] = X_{i-1}[1] \oplus g(X_{i-1}[0] \oplus A)$
4.	$X_i[3] = (X_{i-1}[2] \oplus B) \boxplus g(X_{i-1}[0] \oplus A)$
5.	$X_i[4] = X_{i-1}[3] \boxminus g(X_{i-1}[0] \oplus A)$
6.	$X_i[5] = X_{i-1}[4] \oplus C$
7.	$X_i[6] = X_{i-1}[5] \oplus g(X_{i-1}[4] \oplus C)$
8.	$X_i[7] = (X_{i-1}[6] \oplus D] \boxplus g(X_{i-1}[4] \oplus C)$
Output Generation:	
1.	$X^{r'} = X_i[0] \parallel X_i[1] \parallel \cdots \parallel X_i[7]$

Table 3.7: Nonlinear Function g at Round i

Input:	w-bit Input Value I
Output:	w-bit Output Value $g(I)$
Process:	
1.	$I = I[0] \parallel I[1] \parallel \cdots \parallel I[7]$
2.	$I = S(I[0]) \parallel S(I[1]) \parallel \cdots \parallel S(I[7])$
Output Generation:	
1.	$g(I) = A_{8\times 8} \cdot I_{8\times 1}$ where A is given in Section 3.2.6

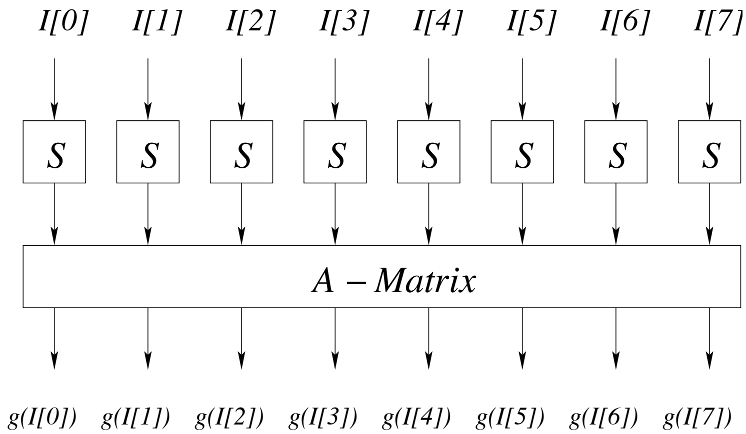

Figure 3.3: g Function

3.2.5 S-box

Sarmal g function makes use of an 8×8-bit S-box whose design is inspired from the S-boxes of CLEFIA [60] and Whirlpool [5] where several 4×4 S-boxes are combined to generate a bigger 8×8-bit S-box. In this subsection we only provide the construction method and the specification of the smaller S boxes in Figure 3.4 and in Table 3.8 respectively. Exact values and the details about the S-box are provided in Appendix A and Section 4.2.3.

Table 3.8: S-boxes of Sarmal

	0	1	2	3	4	5	6	7	8	9	A_x	B_x	C_x	D_x	E_x	F_x
S_0	E_x	A_x	4_x	7_x	C_x	9_x	F_x	0_x	B_x	D_x	5_x	1_x	6_x	3_x	2_x	8_x
S_1	2_x	E_x	8_x	1_x	F_x	D_x	0_x	5_x	6_x	3_x	4_x	7_x	A_x	9_x	B_x	C_x
S_2	6_x	5_x	C_x	E_x	9_x	7_x	B_x	A_x	4_x	8_x	3_x	D_x	0_x	F_x	2_x	1_x
S_3	4_x	B_x	D_x	6_x	E_x	C_x	0_x	2_x	3_x	5_x	1_x	8_x	7_x	A_x	F_x	9_x

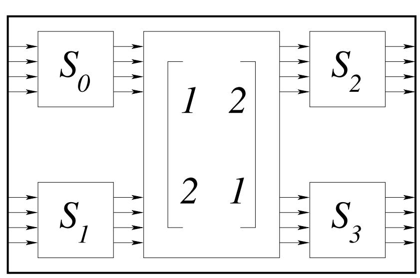

Figure 3.4: S-box of Sarmal

3.2.6 MDS Matrix

The nonlinear subround function g makes use of a permutation which is similar to the one in Whirlpool[5]. The circulant matrix A used in g-function is a $[16, 8, 9]$ MDS code on $GF(2^8)$ which refers to the name MDS Matrix. The matrix $A_{8\times8}$ given below.

$$A = \begin{bmatrix} 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x \\ 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x \\ 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x \\ 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x \\ 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x \\ 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x \end{bmatrix}.$$

There are several advantages of using such a permutation based on a circulant matrix. The main advantage is due to the implementation in both 32 and 64-bit architectures. Secondly, it is highly diffusive providing nice security features.

Let w-bit input value I be the concatenation of 8-bytes in the form $I = (I[7], I[6], \cdots, I[0])$ and similarly w-bit output value O be $O = (O[7], O[6], \cdots, O[0])$. Then the permutation is defined as a matrix multiplication $O = A \cdot I$ over GF(2^8):

$$\begin{bmatrix} O[0] \\ O[1] \\ O[2] \\ O[3] \\ O[4] \\ O[5] \\ O[6] \\ O[7] \end{bmatrix} = \begin{bmatrix} 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x \\ 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x \\ 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x \\ 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x \\ 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x \\ 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x \end{bmatrix} \cdot \begin{bmatrix} I[0] \\ I[1] \\ I[2] \\ I[3] \\ I[4] \\ I[5] \\ I[6] \\ I[7] \end{bmatrix}$$

The security and the implementation properties of the multi-permutation are provided in Section 4.2.4 in detail. The addition and multiplication over GF(2^8) are performed according to operations described in Section 2.2.

3.2.7 Message Permutation

The compression function of Sarmal uses $16w$-bit message block M_i each iteration. The message block M_i is first divided into sixteen 64-bit words, then 16 words are permuted by several permutations $\sigma_k(M_i)$. One execution of the round function G uses 4 permuted message words leading to a full mixing in $4G$ invocations at each left and right parts.

Since the full message block M_i is used in four consecutive rounds and we have $16\times2 = 32$ ($20\times2 = 40$) rounds for Sarmal-224/256 (Sarmal-384/512), 8-permutations (10-permutations) are needed for the overall compression function f. There are several design choices for the permutations used for each member of Sarmal which are given in Chapter 4 in detail. Here, we provide the necessary permutations in Table 3.9.

Table 3.9: Message Permutations of Sarmal

Sarmal-224/256																
Left Part																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(M_j)[.]$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(M_j)[.]$	1	14	15	10	12	2	7	4	13	8	3	9	11	5	0	6
$\sigma_2(M_j)[.]$	11	4	10	7	14	9	13	1	6	5	8	2	3	15	12	0
$\sigma_3(M_j)[.]$	8	2	0	5	10	3	14	13	12	7	1	15	9	4	6	11
Right Part																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_4(M_j)[.]$	2	8	5	7	11	1	12	4	6	14	15	10	0	13	9	3
$\sigma_5(M_j)[.]$	13	14	2	1	10	12	11	7	5	3	9	15	8	4	0	6
$\sigma_6(M_j)[.]$	3	13	4	0	5	6	2	10	9	8	7	11	12	15	1	14
$\sigma_7(M_j)[.]$	6	3	11	14	4	0	5	8	7	13	2	12	10	1	15	9
Sarmal-384/512																
Left Part																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(M_j)[.]$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(M_j)[.]$	1	14	15	10	12	2	7	4	13	8	3	9	11	5	0	6
$\sigma_2(M_j)[.]$	11	4	10	7	14	9	13	1	6	5	8	2	3	15	12	0
$\sigma_3(M_j)[.]$	8	2	0	5	10	3	14	13	12	7	1	15	9	4	6	11
$\sigma_4(M_j)[.]$	13	10	3	2	8	11	1	5	9	12	0	4	15	6	7	14
Right Part																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_5(M_j)[.]$	2	8	5	7	11	1	12	4	6	14	15	10	0	13	9	3
$\sigma_6(M_j)[.]$	13	14	2	1	10	12	11	7	5	3	9	15	8	4	0	6
$\sigma_7(M_j)[.]$	3	13	4	0	5	6	2	10	9	8	7	11	12	15	1	14
$\sigma_8(M_j)[.]$	6	3	11	14	4	0	5	8	7	13	2	12	10	1	15	9
$\sigma_9(M_j)[.]$	15	7	9	12	3	13	10	0	4	6	1	14	2	5	8	11

3.2.8 s and t Values

The salt value s is a user defined constant string of $4w$-bit which is used to extend the $8w$-bit chaining value to $16w$ together with the round constants and t. The t_i value, on the other hand, is w-bit counter that represents the number of bits hashed up to ith compression function evaluation. Starting from *zero* string it is incrementally updated at each compression function evaluation.

Chapter 4

Design Rationale

The design rationale behind the design of Sarmal Hash Family basically tries to solve the main problem in designing cryptologic algorithms: The trade-off between security, speed and implementation cost. These problems are dealt with seperately, but in a close relation with the mode of operation and the compression function of Sarmal.

Security, being the main concern in cryptographic hash functions, can not be reduced to solve a mathematically hard problem for Sarmal. Instead, we choose the components of Sarmal to be not provably secure but fast and efficient in multiple platforms. One of the reasons behind this is that we can not provide fast and efficient implementations for such provably secure schemes. Obviously, the efficiency is not the only issue. As the recent breakthroughs in cryptanalysis of hash functions lead to the design of SHA-3[51], we propose Sarmal being resistant to the recent attack scenarios.

Speed, as one of the primary concerns, is crucially important since a significantly slower design than `SHA-2`[49] does not improve the existing properties of `SHA-2`. On the other hand, a more secure and faster scheme can lead to significant improvements. In Sarmal, we choose fast components for both hardware and software which satisfy and provide necessary security requirements both for mode of operation and the compression function of Sarmal.

Implementation cost has become fundamentally important especially in hardware due to the emerging technologies in extremely constrained environments. As the use of cryptographic hash functions show great progress in various applications which require equally constrained environments, we choose the components of Sarmal to be able to be compatible in several platforms.

The design rationale of the components of the mode of operation and compression function of Sarmal are detailed in the following sections in terms of these three building blocks. We refrain from repeating the specification of the components as they are detailed in the previous chapter.

4.1 Sarmal Mode of Operation

Despite of the fact that there have been significant breaktroughs in cryptanalysis of iterative mode of operations Sarmal assumes an iterative mode of operation that has been recently proposed as HAIFA [12]. Having been analyzed in detail in recent years is one of the reasons to choose HAIFA as a mode of operation for Sarmal as it provides concrete security claims. The detailed security properties of Sarmal are given in Chapter 5.

Besides, among the existing constructions, HAIFA is one of the most practical mode of operation in terms of supporting salts, variable digest size and flexible implementation. In Sarmal, we use only one fixed compression function with different variables to define several digest sizes. Moreover, we just need to deal with the blocks of messages rather than keeping full message that reduces the memory requirements significantly. The only disadvantage is the parallelizability in mode of operation as it resumes iteratively. Nevertheless, we provide parallelizability in the evaluation of compression function. Still, as its compression function permits, Sarmal can also be used in different mode of operations both iteratively and parallelly. Yet, we choose not to make a flexibility in mode of operation and decide to use HAIFA as a standard mode of operation for Sarmal.

Summary of design features of Sarmal in mode of operation can be listed as follows.

1. Sarmal mode of operation has been analyzed extensively and designed to practically resist all existing attacks.

2. Theoretical reduction proofs for collision and preimage resistances are possible. For the second preimage resistance, we follow the recent research results for HAIFA mode of operation and conjecture Sarmal to be second preimage resistant.

3. It is possible to reduce the immunity against recent generic attacks to the iterative mode of operations by using the properties of HAIFA and the compression function.

4. Sarmal mode of operation supports salts and randomized hashing.

5. Flexibility in several digest sizes is possible by truncation at the end. Thus, only one construction is sufficient to design several hash outputs (It is not limited only to the supported hash sizes).

6. The memory requirement is tolerable as it only requires the blocks of messages rather than the whole message to be hashed.

4.2 Sarmal Compression Function

Sarmal compresssion function f has been designed to satisfy three basic properties for a cryptographic algorithm. We use very well known components to provide security, speed and low implementation cost. Besides, we design one compression function f to support variable digest sizes which provides a lot of flexibility in implementation.

The design choices for the compression function of Sarmal are closely related with the ones for the mode of operation. As detailed in previous section the main design criteria, from security point of view, is to resist all known attack models in a practical sense. Therefore, the first step while designing f to choose the number of bits in the chaining values. As Sarmal has to support variable digest sizes ($224, 256, 384$ and 512 bits), $16w$-bit chaining value would be sufficient to resist all known attacks both theoretically and practically. However, it has a lot of practical implications and we believe $8w$-bit chaining value is necessary and sufficient as described in detail in Chapter 5. Even if the compression function operates on 2 parallel blocks of $8w$-bit each, we use this property to resist the attacks to the compression function itself. Moreover, we choose to digest $16w$-bit of messages at a time so as to increase the speed and the efficiency of the algorithm. Besides, it is suitable for HMAC. The only drawback is the increasing memory, but it is tolerable by the increasing amount of memory spaces with the help of emerging technology.

As described in Section 3.2, f is composed of two parts operating on parallel which is the main property of f. The choice for this to satisfy parallelizability in implementation and provide security at the same time. The reason for parallelizability is obvious in the sense that *left* and *right* parts in Sarmal operate totally independent of each other until the end of f and it provides reasonable amount of speed. The reason for security, on the other hand, is the evolution of the recent attack models to the well known cryptographic hash functions. Starting from the attacks of Wang *et.al* [61, 62, 63, 64], the attack models cannot easily deal with two different parallel blocks at the same time. The only attacks to that kind are the attack on FORK [39, 43] which uses 4 parallel blocks and the attack on RIPEMD-128 [61] where the former uses weak round functions together with less number of rounds and the latter does not make use of different message permutations.

The details of the components of the compression function f will be given in the following subsections. We summarize the basic design criteria for f:

1. The flexibility in the design of f leads to be able to define all modes of Sarmal depending on the digest size.

2. It is possible to provide practical and theoretical security with $8w$-bit of chaining value.

3. At each f evaluation, it is possible to digest $16w$-bit of messages which incerases the efficiency of Sarmal.

4. It is highly paralellizable in the sense that the whole compression function f is composed of two parallel independent $8w$-bit of blocks.

5. It is difficult to control 2 parallel blocks at the same time which makes it difficult to attack f.

6. The components of f are well known and analyzed which makes it easier to analyze its security and to implement it efficiently.

4.2.1 G Function

The compression function f of Sarmal makes use of successive application of a nonlinear function G. As described in Section 3.2.3, the function G follows a GUFN of 8 branches where 2 of which are used to update remaining 6 branches. Our model is quite different from the standard GUFN model which has been used in several designs including block ciphers and hash functions [1, 30, 31, 65]. The main reason why we choose this structure is quite obvious that the number of g executions per G computations, that is the main cost of implementation, is quite low which leads to a more compact and less hardware-demanding design. In order to be able to update $16w$-bit of data at a time more securely, we choose to use less demanding components in G.

Another issue here is to increase the efficiency in both 64 and 32-bit architectures at the same time. One solution is to choose w-bit words at each branch which is also our main design criteria as Sarmal is aimed to be a future design. Nevertheless, on 32-bit architectures, Sarmal is not as efficient as on 64-bit architectures since the operations used in Sarmal are w-bit oriented. Still, it is highly efficient on 32-bit architectures. Besides, to increse the speed, nonlinear g function can be processed parallelly at the same time to update the data and different arithmetic operations are used to differentiate the update of the branches. We summarize the basic design criteria for G:

1. It is less hardware demanding comparing to the nonlinear round functions which update $16w$-bit at a time. Even if the number of G invocations have to be increased, it is tolerable.

2. w-bit of branches are used to be able to increase the efficiency especially in 64-bit architectures. Still it is not slow on 32-bit architectures.

3. Different arithmetic operations are used to update 6 branches by using 2 branches. Therefore, each branch is updated at each G invocation.

4.2.2 g Function

In the round function G, nonlinear function g plays a crucial role in the security of Sarmal compression function. It is an AES [20](or Whirlpool[5])-like w-bit bijection that satisfies certain cryptographic properties. The reason behind the choice of g is mainly due to the extensive work done on that kind of nonlinear functions. The security evaluations of g are well established and known such that we can provide concrete results about the cryptographic properties of g. Also, the sound research done on g-like functions allows us to find fast, elegant and low-cost implementations. We summarize the basic design criteria for g:

1. It is possible to provide security claims, especially for differential kind of attacks.

2. Fast, secure and low-cost implementations for each architecture are possible which is due to sound work done on this kind of functions.

4.2.3 S-Box

S-Box of Sarmal is mainly inspired from the S-boxes of CLEFIA [60] and Whirlpool [5] which use smaller S-boxes to generate a bigger one. The obvious reason for this is to reduce the hardware requirements

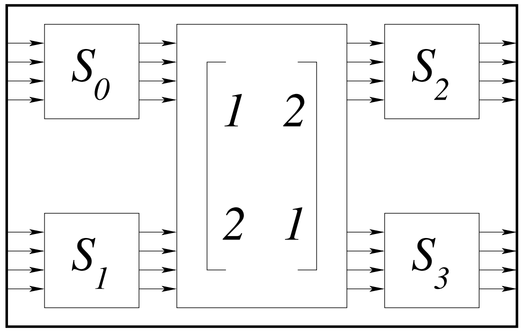

Figure 4.1: S-Box of Sarmal

of 8×8-bit S-boxes as it reduces the required memory dramatically. We follow the same fashion as done in [60] and generate Sarmal's S-box from 4 different 4×4-bit S-boxes connected by a permutation over GF(2^4) defined by the primitive polynomial $p(x) = x^4 + x + 1$. 4×4-bit S-boxes are selected randomly and combined with a manner described in Figure 4.1.

From the security point of view, it is not as good as the optimal 8×8-bit S-box. However it satisfies several cryptographic properties which are provided in Table 4.1. We summarize the design criteria for the S-box of Sarmal as follows.

1. It is 8 times less hardware demanding comparing to the optimal 8×8-bit S-boxes.

2. It is possible to satisfy basic cryptographic properties.

Table 4.1: Properties of S-box

Probability of Maximum Difference Value	$2^{-4.68}$
Probability of Maximum Linear Value	$2^{-4.38}$
Maximum Degree of Boolean Functions	6
Minimum Nonlinearity	100

4.2.4 MDS Matrix

In the design of Sarmal, an MDS matrix is used to diffuse the incoming data in the g-function. It is based on a $[16, 8, 9]$ MDS code which helps us to evaluate the security of Sarmal against differential type of attacks. Being also circulant, it enables us to use various definitions for our MDS matrix. The following definition provides us nice security results about Sarmal g function

Definition 4.2.1 (Branch Number[19]) *Let G be a linear transformation operating on bytes and let $W(.)$ be the byte weight of an input value (i.e. counts the non-zero bytes of the given value). Then, the branch number of G is defined as $min_{a \neq 0}\{W(a) + W(G(a)\}$.*

The branch number of Sarmal permutation is 9, which guarantees minimum number of active bytes that will be used to evaluate the total number of active S-boxes in differential attacks. In the following, we summarize the basic criteria behind the choice of this permutation.

1. The MDS matrix guaranties at least 9 active bytes in the input and output that enables us to evaluate the security.

2. It is possible to implement the matrix efficiently in 8, 32, 64 bit platforms with the aid of matrix properties.

4.2.5 Message Permutation

In Sarmal, we choose not to make an additional operation on the original message block and use the message as is in the compression function f. There are several reasons for this. The trivial answer is to decrease the cost of this computation. As we do not make any modifications on the original message, it is efficient for any implementation. Besides, message block can be stored externally and reached from the external memory.

The selection of message permutation is one of the crucial parts of Sarmal compression function f. Message permutations can be written as a 4×16 (or 5×16) matrix for each (left and right) part of the compression function (Table 3.9) where the entry in i^{th} row and j^{th} column is denoted by $\alpha_{i,j}$. There are several restrictions while choosing the message permutations which are described below.

In the G-function of Sarmal, half of the message words are not used just before the g-function and this can lead to a self-cancelation under certain circumstances depending on the message values. This situation is depicted in Figure 4.2. In the figure, given four message values $(\alpha_{i,9}, \alpha_{i,11}, \alpha_{i,13}, \alpha_{i,15})$ in the i^{th} permutation, their positions in the next permutation can cause cancellations if they are chosen appropriately as $(\alpha_{i+1,2}, \alpha_{i,0}, \alpha_{i,6}, \alpha_{i,4})$ respectively. Here, our aim is to force the attacker to increase the number message words to be modified so as to find local collisions.

Secondly, we do not want to allow the similar case for the chosen two message pairs. If one of the message pairs $(\alpha_{i,1}, \alpha_{i,10})$ or $(\alpha_{i,3}, \alpha_{i,8})$ and their iterated versions up to 4^{th} round are taken identical, then they cancel each other due to the structure of Sarmal. We construct our message permutation by taking these conditions also into account. Here, the aim is again the same in the sense of the first case.

So, we can summarize the design criteria for the message permutation of Sarmal as follows.

1. Simple message permutations are used to spend less time in message expansion part.

2. Message permutations are chosen to increase the number of message words to be modified to find local collisions.

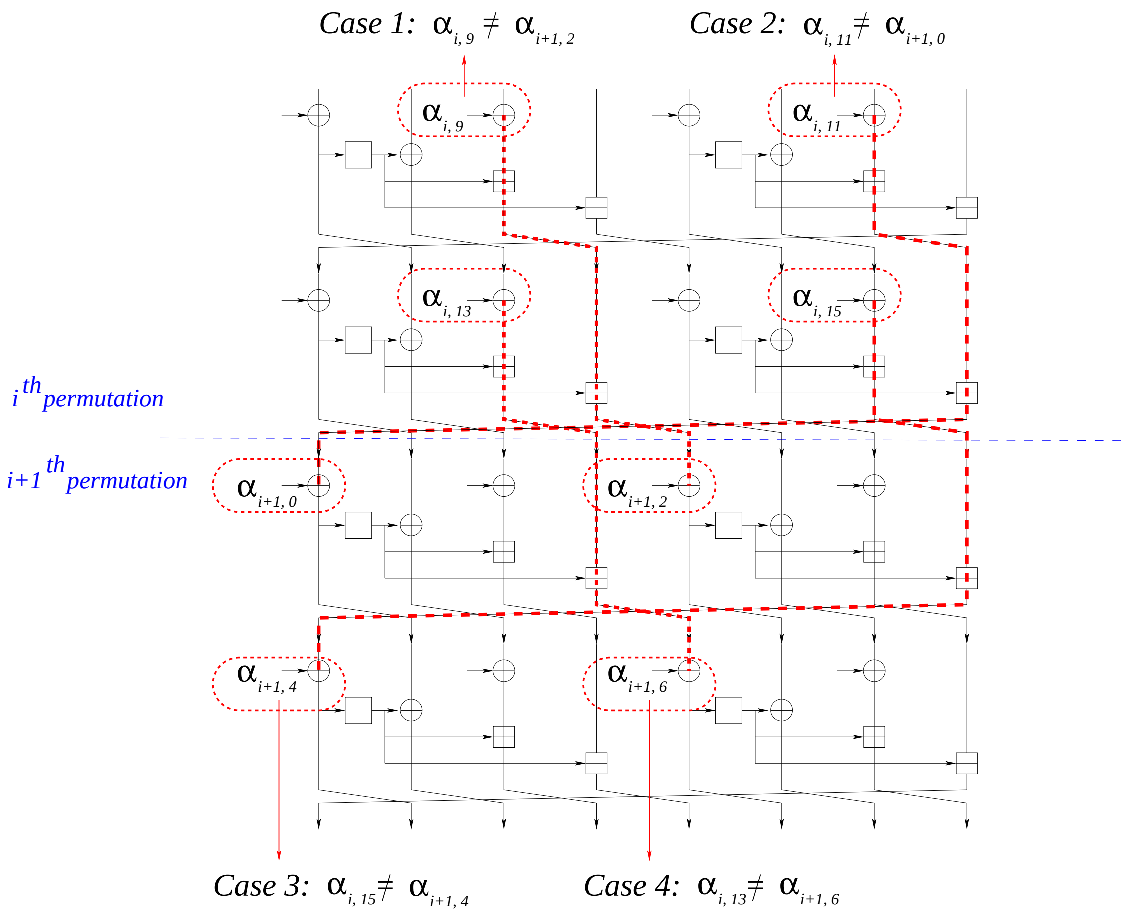

Figure 4.2: Conditions on Message Permutation

4.2.6 s and t Values

User supplied salt value s and the counter value t are mainly used to improve the strength of the Sarmal against generic attacks on iterative modes of operations. They are fundamental components of HAIFA mode of operation and satisfy certain security properties against known generic attacks. These properties are considered mainly in Chapter 5.

We decide to place s and t directly in the state so as to make inversion or meet-in-the-middle attacks harder. Similar methods have also been used in [7, 8]. Namely, instead of imposing these values in the rounds we apply, in particular s, directly in *left* and *right* parts of the state in order to prevent partial recovery especially when s is used as key.

Number of bits for s and t values are chosen accordingly with the number of necessary bits. That is, we are bound to choose w-bit value for t as the length l of the message M can be at most $2^{64} - 1$. The case for s is quite different in that the only requirement is when it is used as *key* especially for MAC. In HMAC [25], the size of the key, s, shall be equal to or greater than $d/2$. As we choose s to be $4w$ bits, it is sufficient for all digest sizes.

Chapter 5

Security

Aforementioned applications of cryptographic hash functions require three basic security properties which are given below.

1. Collision resistance: For an adversary, it should be *hard* to find two distinct messages M and M' such that $H(M) = H(M')$.

2. Preimage resistance: For an adversary, given the target hash value D, it should be *hard* to find a preimage M such that $H(M) = D$.

3. Second-preimage resistance: For an adversary, given a message M, it should be hard to find another different message M' such that $H(M) = H(M')$.

From these definitions, it is clear that finding a second preimage is equivalent to finding a collision for the entire hash function H. In terms of security requirements today, one would expect

- Collision resistance of about $d/2$ bits which is due to the birthday paradox and equivalent to $O(2^{d/2})$ queries for an adversary.

- Preimage resistance of about d bits which is equivalent to $O(2^d)$ queries for an adversary.

- Second-preimage resistance of $d-l$ bits for any message shorter than 2^l bits which is equivalent to $O(2^{d-l})$ queries for an adversary.

The security proofs contain a close interaction between the security of the mode of operation and the compression function. Namely, in order to show the security of mode of operation, one assumes compression function's being secure. So, in the first part of this chapter we assume that our compression function f does not have any weaknesses. In the second part, we show the resistance of f to the recent and the possible attacks which will provide a sound idea about the overall security of Sarmal. The final section expresses the ideas of the submitters about the expected security of Sarmal.

5.1 Security of the Mode of Operation of Sarmal

As described, the security evaluation of Sarmal is divided into two parts regarding the security of mode of operation and compression function. Security of the mode of operation of Sarmal heavily depends on the security of HAIFA [12]. We add further details by using the internal properties of Sarmal to provide concrete results. We present the collision resistance, preimage resistance, second-preimage resistance, pseudorandomness of Sarmal together with the resistance against recent generic attacks to the iterative modes of operations.

5.1.1 Collision Resistance

The reduction proof of the collision resistance of the mode of operation of Sarmal is provided in HAIFA [12, 9] which is very similar to the reduction proof of collision resistance of Merkle-Damgård construction with minor differences. As done in [12, 9], we assume that the attacker has full control over all parameters which is the strongest definition of a collision resistance. The result follows from the fact that if an attacker can find two arbitrary but finite length messages $M, M' \in \{0,1\}^*$ and $s, s' \in \{0,1\}^{4w}$ such that $H(M,d,s) = H(M',d,s')$, then he can construct a collision in f such that $f(h,m,s,t) = f(h',m',s',t')$ or in f_d which is the last iteration of f together with d-bit truncation. Therefore, we can conclude if the underlying compression function is collision resistant our hash function family Sarmal is collision resistant.

5.1.2 Preimage Resistance

For the preimage resistance of the mode of operation of Sarmal, one can use several works [2, 12, 9] that discuss the preimage resistance of HAIFA. However, the reduction proof of preimage resistance changes according to the definition of the preimage finding advantages of the adversaries which have been formalized in [58] as *Pre, aPre* and *ePre* that stand for preimage, always and everwhere preimage resistances, repectively. These definitions play respective roles depending on the applications of the underlying cryptographic hash function. Based on [2], HAIFA mode of operation satisfies *ePre* where there are also counter-examples for *Pre* and *aPre*. However, in [9] it is simply accepted that HAIFA mode of operation is preimage resistant.

5.1.3 Second-Preimage Resistance

Second-preimage resistance is stronger assumption than the collision resistance, as one can produce collisions if it is possible to create second preimages. Unfortunately, we cannot make a reduction proof for the second preimage resistance for Sarmal mode of operation. Also, the work [2, 12, 9] support this claim by assuming a $d/2$-bit of security for the second preimage resistance of HAIFA.

However, a sketch proof has been provided in [12, 9] recently about the second preimage resistance of HAIFA mode of operation if the underlying compression function is ideal. Nevertheless, we conjecture the second preimage resistance of mode of operation of Sarmal as $d - l$ bits for any message shorter than 2^l bits which is equivalent to $O(2^{d-l})$ queries for an adversary.

5.1.4 Pseudorandomness

Pseudorandom oracle preservation and pseudorandom function preservation were investigated in several papers [27, 28, 40, 41, 42]. However, there is no known work for HAIFA mode of operation about its pseudorandomness and unpredictability. As a `SHA-3` candidate, Sarmal should support $2^{d/2}$ level of security as MAC. We propose Sarmal to be used as *keyed* in place of salt value s which provides $4w$ bits of security which is sufficient for all digest sizes of Sarmal. Besides, as it is a form of Merkle-Damgård iterative construction, it can be used in place of existing MACs [6].

5.1.5 Resistance Against Generic Attacks to the Iterative Hash Functions

Security notions discussed in the previous parts do not deal with some practical attacks which have emerged recently such as multicollision [32], fixed-points [22], expandable message [22, 35], long message second preimage [35], herding [33] and Nostradamus [33] attacks. There is no established theoretical background to resist these type of attacks which are generic to all iterative mode of operations. This section considers the security of Sarmal against these attacks.

Resistance Against Multicollision Attacks

Multicollisions (r-collisions) are defined to simply to be the generalization of collisions (2-collisions). This time the attacker tries to find r-tuple of messages which give the same hash value rather than only one collision. This attack was first described by Joux [32] for standard Merkle-Damgård construction and used for attacking concatenated schemes.

The strength of this attack stems from the fact that it can be applied to any iterative schemes. The hardness of the applicability of the attack heavily relies on the collision resistance of the underlying compression function. Still, it is possible to apply this attack in the worst case, that is in the birthday bound. Assuming this is the case, in order to find 2^t-collisions, the attacker needs to call $O(t2^{d/2})$ queries, where d is the number of bits in the chaining variables.

The resistance of Sarmal against multicollision attacks should be investigated in several cases depending on the digest size. First of all, as noted in [12], if the attacker does not have control over the salts, it is impossible to precompute multicollisions and apply the attacks. This works for all digest sizes and we can deduce this property thanks to the mode of operation of Sarmal. If the attacker has control over the salts, the attacker needs $O(t2^{4w})$ queries to apply the attack for t-collision.

An obvious way to resist this attack is to enlarge the chaining value. However, it has some practical and performance implications. A $16w$-bit of chaining variable would be sufficient to resist this attack for all digest sizes of Sarmal, but we choose to make a trade-off. As Sarmal supports chaining value of $8w$ bits, clearly Sarmal-224 and Sarmal-256 can resist multicollisions. Sarmal-384, on the other hand, can resist t-multicollisions for many of the t-values and for increasing values of t, the complexity of the attack gets closer to $O(2^{8w})$. For Sarmal-512, the resistance against multicollisions is at the same level for standard Merkle-Damgård construction under the assumption that the attacker has full control over the salt. Nevertheless, the applicability of the attack is still questionable as it does not seem to be reasonable to apply this attack for $d = 512$ assuming the underlying compression function is collision resistant.

Fixed-Points and Dean's Attack

An expandable message is a kind of multicollision that consists of colliding messages before the last compression function evaluation of different lengths. In [22], Dean showed an efficient way of finding expandable messages by using fixed points of the compression function. This attack entirely depends on the simplicity of finding fixed points which is the case for Davies-Meyer mode of operation. Many hash functions including SHA family [48, 49, 50], MD4 [55], MD5 [56], Tiger [1] and RIPEMD [23] use Davies-Meyer mode of operation.

In Sarmal, we also use a version of Davies-Meyer construction which was detailed in Chapter 3. In order to show the resistance of Sarmal against Dean's attack, we first show the resistance against finding fixed points. Next, we show the infeasibility of iterating fixed points by using the properties of the mode of operation of Sarmal.

As Sarmal uses Davies-Meyer for the compression function f, it is still possible to find fixed points of f, but it is not that simple. As usual, we start with assuming that the attacker does not have control over s. Since the attacker does not have control over the s, once he chooses X^l and X^r at the end of r rounds as *zero*-string, he can revert the whole compression function by taking a random message. Now, the probability of obtaining the actual s, t and c is 2^{-8w} as we assume that the attacker does not have control over s which makes the attack infeasible. If the attacker has control over s, the probability increases to 2^{-4w}. Nonetheless, to apply the attack the attacker has to generate 2^{4w} random fixed points that is again infeasible. Besides, so as to generate messages by using fixed points Dean [22] makes an extensive use of the iteration of same compression function in standard Merkle-Damgård construction. However, even if the attacker can find the fixed points very easily, the mode of operation of Sarmal does not allow to iterate fixed points as t differentiates each compression function.

Resistance Against Expandable Messages and Long-Message Second-Preimage Attack

In [35], Kelsey and Schneier proposed a method to find multicollisions of different lengths and used this idea to create expandable messages. The outcome of this work transforms the attack of Dean to the case where the fixed points cannot easily be found. Besides, a solution to the inapplicability of long-message second-preimage attack to Merkle-Damgård strengthened constructions is provided.

As for in multicollisions and Dean's attack, we consider the security of Sarmal against expandable messages and long-message second-preimage attack by considering the cases where the attacker has/does not have control over s. When the attacker does not have control over s, the attack becomes infeasible as the attacker cannot create expandable messages without knowing s. For the case when the attacker can control s, we need to consider the computational complexity of the attack for several versions of Sarmal. Again, we choose to make a trade-off between the resistance against this attack and the performance of Sarmal by choosing the chaining value as $8w$ bits.

As the attack basically depends on finding collisions in the compression function, the main work factor comes from $O(2^{4w})$ number of queries that the attacker has to make. Obviously, Sarmal-224 and Sarmal-256 can resist this attack. For Sarmal-384, the theoretical applicability of this attack depends on the length of the message where the computation effort is still close to $O(2^{6w})$ queries. For Sarmal-512, the resistance against this attack is at the same level for standard Merkle-Damgård construction under the assumption that the attacker has full control over the salt. Nevertheless, the applicability of the attack is still questionable as it does not seem to be reasonable to apply this attack for $d = 512$ assuming the underlying compression function is collision resistant.

Resistance Against Herding Attack

The herding attack [33] allows an attacker to commit to the hash of a message that is not fully known with an additional cost of a large precomputation. The attacker starts producing a special search structure which contains many intermediate hash values which is called a *diamond structure*. In this structure, an attacker can produce a message leading to the same final hash D from any intermediate value. After determining a prefix P, the attacker starts searching for a single-block which would yield an intermediate value in diamond structure when combined with P. Finally, the attacker is able to produce message blocks from the diamond structure to link this intermediate hash value. At the end of this process, the attacker first committed to a hash D, then decided what message she will provide which hashes to H and which begins with the prefix P.

The overall complexity of this attack is dominated by constructing the diamond structure and searching for an intermediate value in diamond structure to match. While constructing diamond structure the attacker has to find collisions repeatedly which is not as effective as multicollisions. As Sarmal uses $8w$-bit of chaining value, it makes this precomputation phase quite infeasible, in particular for Sarmal-224 and Sarmal-256. Further, the attacker has to know s to make this precomputation. If the attacker does not have access to s, $4w$-bit s value is sufficient to resist this attack regardless of the digest size. While searching for an intermediate value in diamond structure to match, it is again overcome by choosing $8w$-bit of chaining value as it decreases the probability of the success of the attack. Assuming the attacker has full control over all variables, for Sarmal-512, it is theoretically possible to apply the attack. Nevertheless, we believe it is much more efficient to make a choice in favor of the performance.

5.2 Security of the Compression Function of Sarmal

5.2.1 Differential Properties of Compression Function of Sarmal

Last term attacks, which are basically differential in nature [10, 11, 15, 47, 59, 61, 62, 63, 64], mainly focus on the collision resistance of the cryptographic hash functions. Therefore, differential properties of Sarmal have an important role while giving the security against similar attacks. We analyze the compression function in Section 5.2.2 and 5.2.4 against differential kind of attacks. Our results mainly reveal the following properties.

- The structure of Sarmal compression function reduces the applicability of recent attacks in which a difference is defined and controlled in the rest of the hash function. In Sarmal, controlling the introduced difference is not easy due to the independence of left and right parts.
- Due to the fast diffusion properties of *G*-function, it is difficult to handle the propagation of differences.

5.2.2 Collision Resistance

While experiencing the resistance of Sarmal against several collision attack models, we introduce differences from various parts (from message block or state or both at the same time) of Sarmal and search for best differential paths. When a difference propagates through the structure of f, it passes the non-linear layer(s) with some cost (an output difference is obtained from non-linear layer with some probability). Therefore, the best path for an introduced difference becomes the one that passes through minimum number of S-boxes. In oder to find out these paths, an algorithm is developed which calculates the Active S-box Number (ASN). We mainly make use of byte-oriented structure of g function to calculate ASN. The details of the algorithm are given as follows:

1. S-box of Sarmal takes 1-byte of input and produces an output of 1-byte. Thus, we choose byte-wise notation and each w-bit word is considered as 8-byte.
2. A byte is called *active* unless it has a zero difference and if a byte is active it cannot have inactive output. Thus, activity of a w-bit word is ranges from 0 to 8.
3. g-function accepts 8-byte inputs and its output is again 8-byte. It has an MDS matrix in its structure which guarantees at least nine active bytes for the input and output differences.
4. Addition and subtraction operations modulo 2^{64} are difficult operations to analyze due to the carry and borrow bits. So, they are converted to XOR operation to ease the calculation. Since both addition and subtraction operations are non-linear, the original design's result is not worse than the modified version's result, and actually it is expected to see more active S-boxes in Sarmal on average. Still, the addition and subtraction modulo 2^{64} may have undesirable effects on the propagation of differences. However, this will have an additional cost.

In the following, Tables (5.1 and 5.2) provide obtained ASN for 12 and 16 rounds of Sarmal respectively. Since the maximum value in the XOR Table of Sarmal's S-box is $2^{-4.68}$, these numbers show theoretical minimum number of rounds required for Sarmal.

Table 5.1: Active S-box Number for 12 round Sarmal

Part	ASN	Number of Active Bytes of States								Active Messages and Active Byte Number	
		$X[7]$	$X[6]$	$X[5]$	$X[4]$	$X[3]$	$X[2]$	$X[1]$	$X[0]$		
Left	3	0	0	0	1	0	0	0	1	$M[0] = 1$	$M[2] = 1$
Right	29	0	0	1	0	8	8	8	0	$M[0] = 1$	$M[2] = 1$
Total	32										
Left	4	0	0	1	0	8	8	8	1	$M[12] = 1$	$M[15] = 1$
Right	31	0	0	0	0	1	1	0	0	$M[12] = 1$	$M[15] = 1$
Total	35										
Left	6	0	0	0	2	0	0	0	2	$M[0] = 2$	$M[2] = 2$
Right	30	0	0	2	0	0	0	0	2	$M[0] = 2$	$M[2] = 2$
Total	36										
Left	8	0	0	2	0	7	7	7	2	$M[12] = 2$	$M[15] = 2$
Right	33	6	0	4	3	0	1	4	0	$M[12] = 2$	$M[15] = 2$
Total	41										
Left	9	6	0	3	3	0	0	0	0	$M[6] = 3$	$M[10] = 6$
Right	35	0	0	2	0	0	0	7	0	$M[6] = 3$	$M[10] = 6$
Total	44										
Left	9	0	0	0	3	0	0	0	3	$M[0] = 3$	$M[2] = 3$
Right	33	3	0	0	0	0	6	3	3	$M[0] = 3$	$M[2] = 3$
Total	42										
Left	32	0	0	7	0	0	8	7	0	$M[1] = 7$	$M[8] = 1$
Right	3	0	0	8	0	0	2	8	0	$M[1] = 7$	$M[8] = 1$
Total	35										
Left	39	1	0	6	0	0	0	2	0	$M[1] = 5$	$M[8] = 2$
Right	6	0	0	7	0	0	4	7	0	$M[1] = 5$	$M[8] = 2$
Total	45										

Table 5.2: Active S-box Number for 16 round Sarmal

Part	ASN	Number of Active Bytes of States								Active Messages and Active Byte Number	
		$X[7]$	$X[6]$	$X[5]$	$X[4]$	$X[3]$	$X[2]$	$X[1]$	$X[0]$		
Left	27	0	0	3	0	0	3	0	0	$M[1] = 3$	$M[9] = 6$
Right	45	3	0	6	0	3	0	3	3	$M[1] = 3$	$M[9] = 6$
Total	72										
Left	30	6	0	3	3	3	0	0	0	$M[10] = 6$	$M[15] = 0$
Right	42	6	0	0	3	3	3	6	0	$M[10] = 6$	$M[15] = 0$
Total	72										
Left	30	6	0	3	3	3	0	0	0	$M[10] = 6$	$M[14] = 0$
Right	42	6	0	0	3	3	3	6	0	$M[10] = 6$	$M[14] = 0$
Total	72										
Left	30	6	0	3	3	3	0	0	0	$M[10] = 6$	$M[15] = 0$
Right	42	6	0	0	3	3	3	6	0	$M[10] = 6$	$M[15] = 0$
Total	72										
Left	30	6	0	3	3	3	0	0	0	$M[10] = 6$	$M[12] = 0$
Right	42	6	0	0	3	3	3	6	0	$M[10] = 6$	$M[12] = 0$
Total	72										

5.2.3 The Attacks to the Similar Constructions

As a compression function, many designs follow some similarities with Sarmal. Firstly, it makes use of GUFN of 8 branches which is quite common in many designs including block ciphers and hash functions. In the round function, Sarmal uses well-established arithmetic operations together with AES-like Substitution-Permutation structure. Therefore, it makes sense to recapitulate recent developments in the analysis of hash functions whose compression functions use such structures.

Assuming $GUFN$ as a basic building block, Sarmal is similar to FORK [30] which also uses independent parallel blocks at the same time. Recent attacks [17, 39, 43] can break FORK faster than generic birthday attack by using clever differential paths. The main feature of the attack against FORK is the so called *micro-collisions* in the round transformation which can also be defined for Sarmal.

A micro-collision is defined to be the propagation of zero difference in round r' in one of the branches $X^{r'}[1], X^{r'}[2], X^{r'}[3]$ (or $X^{r'}[5], X^{r'}[6], X^{r'}[7]$) while having a nonzero difference in $X^{r'}[0]$(or $X^{r'}[4]$). Simultaneous micro-collisions occur if more than one branch has zero difference, which is the main weakness of FORK. The attacker can obtain micro-collisions for FORK as it uses modular addition and XOR at the same time in each branch which allows attacker to cancel additive and XOR differences in the same branch. The case for Sarmal can be considered as a special case of the differential properties provided in previous section where we assume to cancel differences in branches even if it is not the case. Nevertheless, the attack model in FORK can not be applied directly to Sarmal as the round function is stronger than FORK's round function which decreases the probability of the attack model in FORK.

As another example, Sarmal is similar to Tiger [1] in that its round function uses modular addition, subtraction and XOR at the same time. Also, the round update is quite similar which uses one branch to update the others. There are serious attacks [34, 44, 45, 53] to Tiger which can be used to find reduced-round collisions/pseudocollisions, nonrandomness and full pseudo-near collision. These attacks are differential in their nature and make use of mainly the weknesses in the round function and the message expansion of Tiger.

The message update in Sarmal is weaker than Tiger as in the latter a nonlinear message expansion is used while the former uses the message words as are. However, the attack model in Tiger allows attacker to control one of the left or right parts of Sarmal, not both at the same time. Also, the iterative message modification becomes difficult by the help of two parallel blocks in Sarmal. Besides, the round function of Sarmal uses whole data in one branch to update three other branches which is not the case in Tiger.

Moreover, Grindahl[36] and Whirlpool[5] use AES-like round functions in their compression functions where the former has been attacked recently [54] and the latter is still secure. The attack on Grindahl seems inapplicable as the sponge construction allows attacker to control message words which is not the case for Sarmal. Finally, the strengthened versions of RIPEMD[24] is similar to Sarmal as its compression function consist of two parallel blocks. There is no serious threat to that version of RIPEMD.

5.2.4 Possible Attack Scenarios

Recent attacks on hash functions mostly require to find a local collision. In Sarmal, we look for the possibilities of finding local collisions and conclude that one needs two or five message differences to obtain a local collision. We investigate these results in several cases.

Case-I The first possible case to find a collision is illustrated in Figure 5.1. As seen from the figure, if the difference in the message words are chosen accordingly a local collision can be found for four rounds in one of the left or right parts. We use $\alpha_{i,j}$ to denote the i^{th} row and j^{th} column value of the Table 3.9 and $\backsim$ shows choosing the difference Δ in corresponding entries accordingly. Possible cases are drawn in Figure 5.1 and given in Table 5.3.

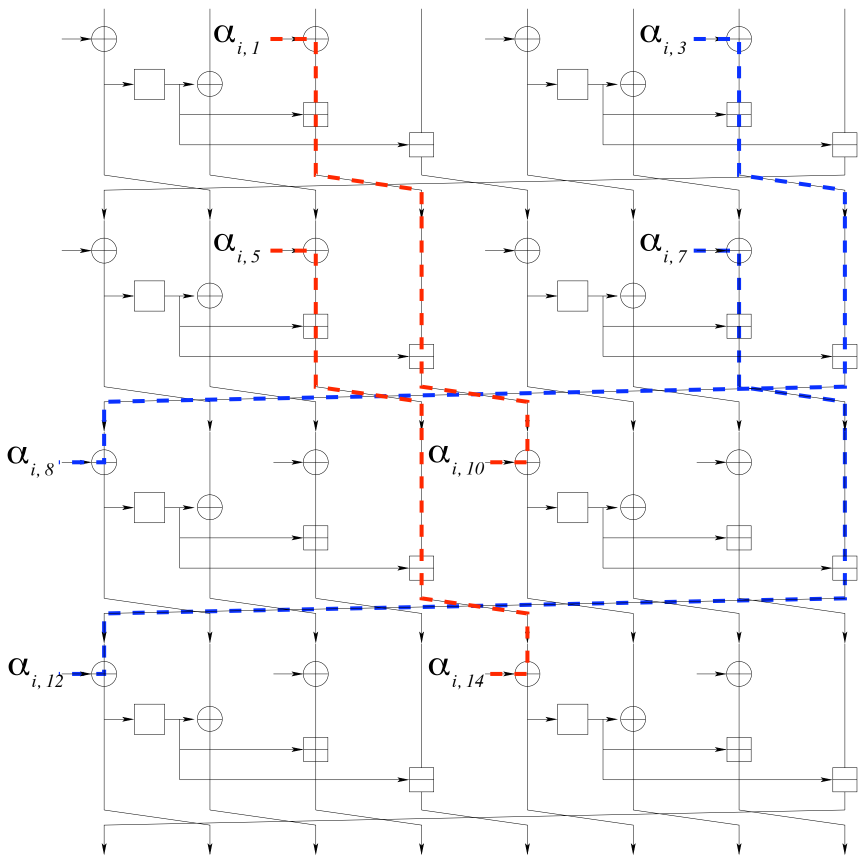

Figure 5.1: Local Collision (Case I)

As shown in Figure 5.1 and Table 5.3 it is possible to find a local collision for four rounds by modifying 2 messages only. However, we cannot control this behaviour for the other part. Namely, message differences propagate independently. We investigate this local collision to find the best possible path for finding collisions. The only drawback wpuld be to find similar behaviour in the other branch. The results are shown in Table 5.4.

For this attack type, results show that after 8^{th} round Sarmal-224/256 and after 12^{th} round Sarmal-

Table 5.3: Conditions for Local Collision (Case I)

Required Message Equivalency (General Form)
$\Delta\alpha_{i,1} \backsim \Delta\alpha_{i,10}$
$\Delta\alpha_{i,5} \backsim \Delta\alpha_{i,14}$
$\Delta\alpha_{i,3} \backsim \Delta\alpha_{i,8}$
$\Delta\alpha_{i,7} \backsim \Delta\alpha_{i,12}$

Table 5.4: Results for Local Collision (Case I)

	ASN of Left Part	ASN of Right Part	Total		ASN of Left Part	ASN of Right Part	Total
Round 1:	0	0	0	Round 1:	0	0	0
Round 2:	0	0	0	Round 2:	3	0	3
Round 3:	0	8	8	Round 3:	9	0	9
Round 4:	0	9	9	Round 4:	18	0	18
Round 5:	0	18	18	Round 5:	21	0	21
Round 6:	0	24	24	Round 6:	24	0	24
Round 7:	0	34	34	Round 7:	27	0	27
Round 8:	1	41	42	Round 8:	30	0	30
Round 9:	2	48	50	Round 9:	30	3	33
Round 10:	3	58	61	Round 10:	36	3	39
Round 11:	19	62	81	Round 11:	48	9	57
Round 12:	19	76	95	Round 12:	51	15	66
Round 13:	24	79	103	Round 13:	57	18	75
Round 14:	28	92	120	Round 14:	57	24	81
Round 15:	33	98	131	Round 15:	63	27	90
Round 16:	38	104	142	Round 16:	69	27	96

384/512 one reaches the birthday attack bound. It can be concluded that message differences should be handled simultaneously in each branch to find a collision.

Case-II In Sarmal, another way of obtaining local collisions for four rounds in one part requires five message differences. In contrast to the first case, this attack model works probabilistically. The total number of way of finding local collisions with this method is 16 where two of which are provided in Figure 5.2. Other cases are similar and provided in Table 5.5.

We follow the same strategy as in the first case and investigate the possible message differences which ought to be satisfied probabilistically. All cases are defined in Table 5.5 and it can be deduced from Table 5.5 that all five message difference groups are different for the left and right parts. Thus, local collisions can only be obtained in one part of Sarmal and message differences diffuse in the other one.

Following that manner, we find the minimum active S-box numbers for each possible local collision scenarios and provide the best possible attack scenario in Table 5.6 . For this attack type, results show that after 7^{th} round Sarmal-224/256 and after 10^{th} round Sarmal-384/512, one reaches the birthday attack bound. It can be concluded that message differences should be handled simultaneously in each part to find a collision.

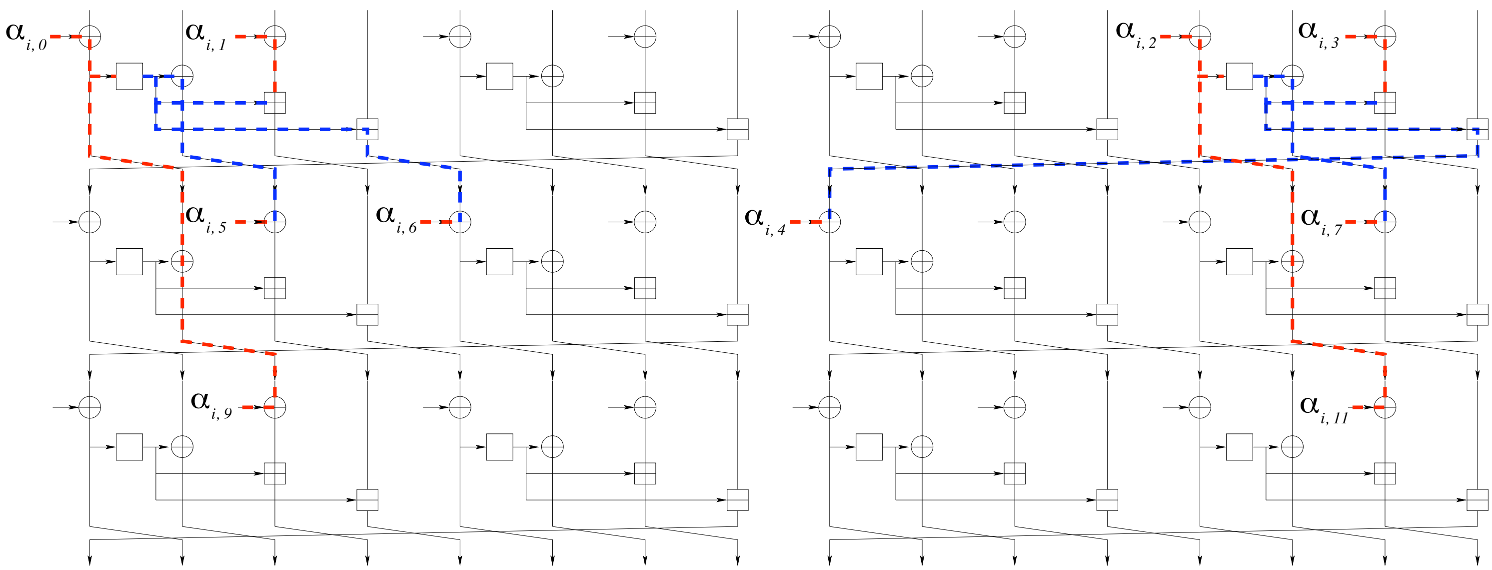

Figure 5.2: Local Collisions (Case II)

5.2.5 Preimage and Second-Preimage Attacks

Recent preimage and the second-preimage attacks have not cought too much attention and success comparing to the effective collision attacks. One of the reasons is obviously the difficulty of these attacks comparing to collision search. However, the works [3, 4, 16, 37] provide serious threats to existing cryptographic hash functions. We try to measure the resistance of Sarmal against preimage and the second-preimage attacks by showing the resistance against these last term attack scenarios.

Table 5.5: Conditions for Local Collision (Case II)

Required Message Values (General Form)						Required Message Values (General Form)					
1	$\Delta\alpha_{i,0}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,5}$	$\Delta\alpha_{i,1}$	$\Delta\alpha_{i,6}$	9	$\Delta\alpha_{i,4}$	$\Delta\alpha_{i,13}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,5}$	$\Delta\alpha_{i,10}$
2	$\Delta\alpha_{i,0}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,5}$	$\Delta\alpha_{i,10}$	$\Delta\alpha_{i,6}$	10	$\Delta\alpha_{i,4}$	$\Delta\alpha_{i,13}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,14}$	$\Delta\alpha_{i,10}$
3	$\Delta\alpha_{i,0}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,14}$	$\Delta\alpha_{i,1}$	$\Delta\alpha_{i,6}$	11	$\Delta\alpha_{i,4}$	$\Delta\alpha_{i,13}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,5}$	$\Delta\alpha_{i,1}$
4	$\Delta\alpha_{i,0}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,14}$	$\Delta\alpha_{i,10}$	$\Delta\alpha_{i,6}$	12	$\Delta\alpha_{i,4}$	$\Delta\alpha_{i,13}$	$\Delta\alpha_{i,9}$	$\Delta\alpha_{i,14}$	$\Delta\alpha_{i,1}$
5	$\Delta\alpha_{i,2}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,7}$	$\Delta\alpha_{i,3}$	$\Delta\alpha_{i,4}$	13	$\Delta\alpha_{i,6}$	$\Delta\alpha_{i,15}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,7}$	$\Delta\alpha_{i,8}$
6	$\Delta\alpha_{i,2}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,7}$	$\Delta\alpha_{i,8}$	$\Delta\alpha_{i,4}$	14	$\Delta\alpha_{i,6}$	$\Delta\alpha_{i,15}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,12}$	$\Delta\alpha_{i,8}$
7	$\Delta\alpha_{i,2}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,12}$	$\Delta\alpha_{i,3}$	$\Delta\alpha_{i,4}$	15	$\Delta\alpha_{i,6}$	$\Delta\alpha_{i,15}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,7}$	$\Delta\alpha_{i,3}$
8	$\Delta\alpha_{i,2}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,12}$	$\Delta\alpha_{i,8}$	$\Delta\alpha_{i,4}$	16	$\Delta\alpha_{i,6}$	$\Delta\alpha_{i,15}$	$\Delta\alpha_{i,11}$	$\Delta\alpha_{i,12}$	$\Delta\alpha_{i,3}$

Table 5.6: Results for Local Collision (Case II)

	ASN of Left Part	ASN of Right Part	Total		ASN of Left Part	ASN of Right Part	Total
Round 1:	0	0	0	Round 1:	0	0	0
Round 2:	0	0	0	Round 2:	0	0	0
Round 3:	0	0	0	Round 3:	6	0	6
Round 4:	0	1	1	Round 4:	9	0	9
Round 5:	0	8	8	Round 5:	15	0	15
Round 6:	0	13	13	Round 6:	18	3	21
Round 7:	16	18	34	Round 7:	24	6	30
Round 8:	18	24	42	Round 8:	36	9	45
Round 9:	20	32	52	Round 9:	39	15	54
Round 10:	25	34	59	Round 10:	39	15	54
Round 11:	33	45	78	Round 11:	42	21	63
Round 12:	34	50	84	Round 12:	48	24	72
Round 13:	38	56	94	Round 13:	51	30	81
Round 14:	48	59	107	Round 14:	57	30	87
Round 15:	58	71	129	Round 15:	66	39	105
Round 16:	64	74	138	Round 16:	69	45	114

The main characteristic of the works [3, 4, 16, 37] is to make use of the weaknesses of the underlying compression functions. First of all, we choose to make use of two independent parts in Sarmal compression function to resist that sort of atacks as there is no real threat to the hash functions that use that kind of structure. In Sarmal, two independent *left* and *right* parts make it diffucult to control both parts at the same time. Secondly, the works [3, 4, 16, 37] share the efficiency of the so called *meet-in-the-middle* attack. The case for Sarmal against *meet-in-the-middle* attack is quite similar for one of parts since Sarmal uses $GUFN$ as the main structure. Nevertheless, the large state size and two independent parts make *meet-in-the-middle* attack inapplicable. In the mean time, the user supplied salt s decreases the applicability of these attacks as the attacker has to control the salt at the same time.

5.3 Expected Strength

In this chapter, we try to give concrete security results for Sarmal Hash Family. Firstly, we provide some results about the resistance of Sarmal mode of operation by reducing the problem to HAIFA mode of operation. Besides, the close relation between the chaining values and the compression function is provided. We conclude that Sarmal mode of operation is at least practically secure for all generic attacks to the iterative mode of operations known so far.

For the compression function of Sarmal, we provide basic differential properties which include some bounds and the minimum required number of rounds. More precisely, our results show that Sarmal compression function is theoretically secure up to 12 and 16 rounds for Sarmal-224/256 and Sarmal-384/512 respectively which leads us to choose number of rounds 16 and 20 for these versions. These results are derived subject to some attack models and valid for attacks that are differential in nature. Still, it is possible to increase the number of rounds used in G function by adding extra permutations on message blocks. This will definitely increase the safety margin of Sarmal. The only drawback is the performance. We leave this issue for the later stages of the competition period.

For the preimage and second-preimage resistance of Sarmal, we conjecture that the compression function is secure against these attack models. This is mainly due to larger state size comparing to message digest and the independent *left* and *right* parts in the compression function. We do not expect any weaknesses of Sarmal against these attacks.

Chapter 6

Implementation and Performance

In this chapter, the implementation methods of Sarmal for various platforms are discussed and the required number of operations for each one is estimated. The performance of Sarmal is also tested in different enviroments and the results are presented.

6.1 Implementation

6.1.1 Optimization Techniques

8-bit Optimization

The following implementation method can be used for implementing matrix multiplication on 8-bit processors to reduce the required RAM amount.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x \\ 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x \\ 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x \\ 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x \\ 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x \\ 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

In order to perform the operation efficiently in 8-bit processor, it can be rearranged as follows:

$$\begin{bmatrix} a_0 \oplus & a_3 \oplus & a_5 \oplus & a_6 \oplus & a_7 \oplus & x \cdot \{a_1 \oplus & a_4 \oplus & x \cdot \{a_1 \oplus & a_4 \oplus & a_6 \oplus & x \cdot \{a_2 \oplus & a_3 \oplus & a_5\}\}\} \\ a_0 \oplus & a_1 \oplus & a_4 \oplus & a_6 \oplus & a_7 \oplus & x \cdot \{a_2 \oplus & a_5 \oplus & x \cdot \{a_2 \oplus & a_5 \oplus & a_7 \oplus & x \cdot \{a_3 \oplus & a_4 \oplus & a_6\}\}\} \\ a_0 \oplus & a_1 \oplus & a_2 \oplus & a_5 \oplus & a_7 \oplus & x \cdot \{a_3 \oplus & a_6 \oplus & x \cdot \{a_0 \oplus & a_3 \oplus & a_6 \oplus & x \cdot \{a_4 \oplus & a_5 \oplus & a_7\}\}\} \\ a_0 \oplus & a_1 \oplus & a_2 \oplus & a_3 \oplus & a_6 \oplus & x \cdot \{a_4 \oplus & a_7 \oplus & x \cdot \{a_1 \oplus & a_4 \oplus & a_7 \oplus & x \cdot \{a_0 \oplus & a_5 \oplus & a_6\}\}\} \\ a_1 \oplus & a_2 \oplus & a_3 \oplus & a_4 \oplus & a_7 \oplus & x \cdot \{a_0 \oplus & a_5 \oplus & x \cdot \{a_0 \oplus & a_2 \oplus & a_5 \oplus & x \cdot \{a_1 \oplus & a_6 \oplus & a_7\}\}\} \\ a_0 \oplus & a_2 \oplus & a_3 \oplus & a_4 \oplus & a_5 \oplus & x \cdot \{a_1 \oplus & a_6 \oplus & x \cdot \{a_1 \oplus & a_3 \oplus & a_6 \oplus & x \cdot \{a_0 \oplus & a_2 \oplus & a_7\}\}\} \\ a_1 \oplus & a_3 \oplus & a_4 \oplus & a_5 \oplus & a_6 \oplus & x \cdot \{a_2 \oplus & a_7 \oplus & x \cdot \{a_2 \oplus & a_4 \oplus & a_7 \oplus & x \cdot \{a_0 \oplus & a_1 \oplus & a_3\}\}\} \\ a_2 \oplus & a_4 \oplus & a_5 \oplus & a_6 \oplus & a_7 \oplus & x \cdot \{a_0 \oplus & a_3 \oplus & x \cdot \{a_0 \oplus & a_3 \oplus & a_5 \oplus & x \cdot \{a_1 \oplus & a_2 \oplus & a_4\}\}\} \end{bmatrix}$$

In the above expression, $x.\{.\}$ denotes the multiplication with x over $GF(2^8)$ and it is stored in a lookup table with the size of 256-byte. The evaluation of this expression requires 96 XORs and 24 table lookups. If the first four substitutions are made, the number of XORs are reduced to 72. Proceeding with the next four substitutions reduces this number further to 68. Finally, when all substitutions given below are made, the whole expression can be evaluated via 64 XORs where the number of table lookups remains unchanged. The final form of the expression is the following:

a_8	=	a_1	$\oplus$	a_4	[1.1]
a_9	=	a_0	$\oplus$	a_5	[1.2]
a_{10}	=	a_3	$\oplus$	a_6	[1.3]
a_{11}	=	a_2	$\oplus$	a_7	[1.4]
a_{12}	=	a_2	$\oplus$	a_5	[2.1]
a_{13}	=	a_4	$\oplus$	a_7	[2.2]
a_{14}	=	a_1	$\oplus$	a_6	[2.3]
a_{15}	=	a_0	$\oplus$	a_3	[2.4]
a_{16}	=	a_0	$\oplus$	a_{10}	[3.1]
a_{17}	=	a_2	$\oplus$	a_9	[3.2]
a_{18}	=	a_4	$\oplus$	a_{11}	[3.3]
a_{19}	=	a_6	$\oplus$	a_8	[3.4]

$$\begin{bmatrix}
a_9 \oplus a_{10} \oplus a_7 \oplus x\cdot\{a_8 \oplus x\cdot\{a_{19} \oplus x\cdot\{a_{12} \oplus a_3\}\}\} \\
a_0 \oplus a_{19} \oplus a_7 \oplus x\cdot\{a_{12} \oplus x\cdot\{a_{11} \oplus a_5 \oplus x\cdot\{a_{10} \oplus a_4\}\}\} \\
a_9 \oplus a_{11} \oplus a_1 \oplus x\cdot\{a_{10} \oplus x\cdot\{a_{16} \oplus x\cdot\{a_{13} \oplus a_5\}\}\} \\
a_{16} \oplus a_1 \oplus a_2 \oplus x\cdot\{a_{13} \oplus x\cdot\{a_8 \oplus a_7 \oplus x\cdot\{a_9 \oplus a_6\}\}\} \\
a_8 \oplus a_{11} \oplus a_3 \oplus x\cdot\{a_9 \oplus x\cdot\{a_{17} \oplus x\cdot\{a_{14} \oplus a_7\}\}\} \\
a_{17} \oplus a_3 \oplus a_4 \oplus x\cdot\{a_{14} \oplus x\cdot\{a_1 \oplus a_{10} \oplus x\cdot\{a_0 \oplus a_{11}\}\}\} \\
a_8 \oplus a_{10} \oplus a_5 \oplus x\cdot\{a_{11} \oplus x\cdot\{a_{18} \oplus x\cdot\{a_{15} \oplus a_1\}\}\} \\
a_{18} \oplus a_5 \oplus a_6 \oplus x\cdot\{a_{15} \oplus x\cdot\{a_9 \oplus a_3 \oplus x\cdot\{a_8 \oplus a_2\}\}\}
\end{bmatrix}$$

Table 6.1: MDS Matrix of Sarmal in 8-bit

Required Memory (Byte):	256
# of table lookups:	24
# of 8-bit XOR ($\oplus$):	64

Table 6.2: S-box in 8-bit

Required Memory (Byte):	10
# of table lookups:	6
# of 8-bit XOR ($\oplus$):	2

32-bit Optimization

Let $I_0||I_1$ be the input value for the g-function in the 32-bit implementation of Sarmal (Both I_0 and I_1 are 32-bit values) and $O_0||O_1$ be the output value (Similarly, both O_0 and O_1 are 32-bit values). The g-function can be defined through the following matrix multiplication:

$$\begin{bmatrix} O_0 \\ O_1 \end{bmatrix} = \begin{bmatrix} A_0 & A_1 \\ A_1 & A_0 \end{bmatrix} \cdot \begin{bmatrix} I_0 \\ I_1 \end{bmatrix}$$

$$\begin{bmatrix} O_0[0] \\ O_0[1] \\ O_0[2] \\ O_0[3] \\ O_1[0] \\ O_1[1] \\ O_1[2] \\ O_1[3] \end{bmatrix} = \begin{bmatrix} 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x \\ 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x \\ 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x \\ 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x \\ 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x \\ 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x \\ 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x \\ 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x \end{bmatrix} \cdot \begin{bmatrix} S(I_0[0]) \\ S(I_0[1]) \\ S(I_0[2]) \\ S(I_0[3]) \\ S(I_1[0]) \\ S(I_1[1]) \\ S(I_1[2]) \\ S(I_1[3]) \end{bmatrix}$$

The expanded form of the above expression is given below. The results of the operations in the shaded area of the matrix are stored in the lookup table LT_0. Similarly, the remaining blocks are stored in the tables LT_i where $i = 1, 2, \cdots, 7$, which are also presented below.

$$\begin{bmatrix}
01_x \cdot S(I_0[0]) \oplus & 06_x \cdot S(I_0[1]) \oplus & 08_x \cdot S(I_0[2]) \oplus & 09_x \cdot S(I_0[3]) \oplus & 06_x \cdot S(I_1[0]) \oplus & 09_x \cdot S(I_1[1]) \oplus & 05_x \cdot S(I_1[2]) \oplus & 01_x \cdot S(I_1[3]) \\
01_x \cdot S(I_0[0]) \oplus & 01_x \cdot S(I_0[1]) \oplus & 06_x \cdot S(I_0[2]) \oplus & 08_x \cdot S(I_0[3]) \oplus & 09_x \cdot S(I_1[0]) \oplus & 06_x \cdot S(I_1[1]) \oplus & 09_x \cdot S(I_1[2]) \oplus & 05_x \cdot S(I_1[3]) \\
05_x \cdot S(I_0[0]) \oplus & 01_x \cdot S(I_0[1]) \oplus & 01_x \cdot S(I_0[2]) \oplus & 06_x \cdot S(I_0[3]) \oplus & 08_x \cdot S(I_1[0]) \oplus & 09_x \cdot S(I_1[1]) \oplus & 06_x \cdot S(I_1[2]) \oplus & 09_x \cdot S(I_1[3]) \\
09_x \cdot S(I_0[0]) \oplus & 05_x \cdot S(I_0[1]) \oplus & 01_x \cdot S(I_0[2]) \oplus & 01_x \cdot S(I_0[3]) \oplus & 06_x \cdot S(I_1[0]) \oplus & 08_x \cdot S(I_1[1]) \oplus & 09_x \cdot S(I_1[2]) \oplus & 06_x \cdot S(I_1[3]) \\
06_x \cdot S(I_0[0]) \oplus & 09_x \cdot S(I_0[1]) \oplus & 05_x \cdot S(I_0[2]) \oplus & 01_x \cdot S(I_0[3]) \oplus & 01_x \cdot S(I_1[0]) \oplus & 06_x \cdot S(I_1[1]) \oplus & 08_x \cdot S(I_1[2]) \oplus & 09_x \cdot S(I_1[3]) \\
09_x \cdot S(I_0[0]) \oplus & 06_x \cdot S(I_0[1]) \oplus & 09_x \cdot S(I_0[2]) \oplus & 05_x \cdot S(I_0[3]) \oplus & 01_x \cdot S(I_1[0]) \oplus & 01_x \cdot S(I_1[1]) \oplus & 06_x \cdot S(I_1[2]) \oplus & 08_x \cdot S(I_1[3]) \\
08_x \cdot S(I_0[0]) \oplus & 09_x \cdot S(I_0[1]) \oplus & 06_x \cdot S(I_0[2]) \oplus & 09_x \cdot S(I_0[3]) \oplus & 05_x \cdot S(I_1[0]) \oplus & 01_x \cdot S(I_1[1]) \oplus & 01_x \cdot S(I_1[2]) \oplus & 06_x \cdot S(I_1[3]) \\
06_x \cdot S(I_0[0]) \oplus & 08_x \cdot S(I_0[1]) \oplus & 09_x \cdot S(I_0[2]) \oplus & 06_x \cdot S(I_0[3]) \oplus & 09_x \cdot S(I_1[0]) \oplus & 05_x \cdot S(I_1[1]) \oplus & 01_x \cdot S(I_1[2]) \oplus & 01_x \cdot S(I_1[3])
\end{bmatrix}$$

$$\begin{array}{lcllll}
LT_0(x) & = & 01_x \cdot S(x) \,\| & 01_x \cdot S(x) \,\| & 05_x \cdot S(x) \,\| & 09_x \cdot S(x) \\
LT_1(x) & = & 06_x \cdot S(x) \,\| & 01_x \cdot S(x) \,\| & 01_x \cdot S(x) \,\| & 05_x \cdot S(x) \\
LT_2(x) & = & 08_x \cdot S(x) \,\| & 06_x \cdot S(x) \,\| & 01_x \cdot S(x) \,\| & 01_x \cdot S(x) \\
LT_3(x) & = & 09_x \cdot S(x) \,\| & 08_x \cdot S(x) \,\| & 06_x \cdot S(x) \,\| & 01_x \cdot S(x) \\
LT_4(x) & = & 06_x \cdot S(x) \,\| & 09_x \cdot S(x) \,\| & 08_x \cdot S(x) \,\| & 06_x \cdot S(x) \\
LT_5(x) & = & 09_x \cdot S(x) \,\| & 06_x \cdot S(x) \,\| & 09_x \cdot S(x) \,\| & 08_x \cdot S(x) \\
LT_6(x) & = & 05_x \cdot S(x) \,\| & 09_x \cdot S(x) \,\| & 06_x \cdot S(x) \,\| & 09_x \cdot S(x) \\
LT_7(x) & = & 01_x \cdot S(x) \,\| & 05_x \cdot S(x) \,\| & 09_x \cdot S(x) \,\| & 06_x \cdot S(x)
\end{array}$$

Once the lookup tables are obtained, the output of the g-function $(O_0 \| O_1)$ can be calculated in the following way:

$$\begin{array}{lcl}
O_0 & = & LT_0(I_0[0]) \oplus LT_1(I_0[1]) \oplus LT_2(I_0[2]) \oplus LT_3(I_0[3]) \oplus LT_4(I_1[0]) \oplus LT_5(I_1[1]) \oplus LT_6(I_1[2]) \oplus LT_7(I_1[3]) \\
O_1 & = & LT_4(I_1[0]) \oplus LT_5(I_1[1]) \oplus LT_6(I_1[2]) \oplus LT_7(I_1[3]) \oplus LT_0(I_0[0]) \oplus LT_1(I_0[1]) \oplus LT_2(I_0[2]) \oplus LT_3(I_0[3])
\end{array}$$

64-bit Optimization

Let $I = I[0 \cdots 7]$ be the input value for g-function and $O = O[0 \cdots 7]$ be the output value. The g-function can be defined through the following matrix multiplication, whose expanded form is also presented below:

Table 6.3: G-function Operations in 32-bit

Required Memory (KB):	8
# of table lookups:	32
# of XOR (⊕):	40
# of Addition (⊞):	6
# of Subtraction (⊟):	6

Table 6.4: Number of Operations Used in Sarmal

Sarmal-224/256	
Required Memory (KB):	8
# of table lookups:	1024
# of XOR (⊕):	1312
# of Addition (⊞):	192
# of Subtraction (⊟):	192
Sarmal-384/512	
Required Memory (KB):	8
# of table lookups:	1280
# of XOR (⊕):	1632
# of Addition (⊞):	240
# of Subtraction (⊟):	240

$$
\begin{bmatrix} O[0] \\ O[1] \\ O[2] \\ O[3] \\ O[4] \\ O[5] \\ O[6] \\ O[7] \end{bmatrix}
=
\begin{bmatrix}
01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x \\
01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x & 05_x \\
05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x & 09_x \\
09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x & 06_x \\
06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x & 09_x \\
09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x & 08_x \\
08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x & 06_x \\
06_x & 08_x & 09_x & 06_x & 09_x & 05_x & 01_x & 01_x
\end{bmatrix}
\cdot
\begin{bmatrix} S(I[0]) \\ S(I[1]) \\ S(I[2]) \\ S(I[3]) \\ S(I[4]) \\ S(I[5]) \\ S(I[6]) \\ S(I[7]) \end{bmatrix}
$$

$$
\begin{bmatrix}
01_x \cdot S(I[0]) \oplus & 06_x \cdot S(I[1]) \oplus & 08_x \cdot S(I[2]) \oplus & 09_x \cdot S(I[3]) \oplus & 06_x \cdot S(I[4]) \oplus & 09_x \cdot S(I[5]) \oplus & 05_x \cdot S(I[6]) \oplus & 01_x \cdot S(I[7]) \\
01_x \cdot S(I[0]) \oplus & 01_x \cdot S(I[1]) \oplus & 06_x \cdot S(I[2]) \oplus & 08_x \cdot S(I[3]) \oplus & 09_x \cdot S(I[4]) \oplus & 06_x \cdot S(I[5]) \oplus & 09_x \cdot S(I[6]) \oplus & 05_x \cdot S(I[7]) \\
05_x \cdot S(I[0]) \oplus & 01_x \cdot S(I[1]) \oplus & 01_x \cdot S(I[2]) \oplus & 06_x \cdot S(I[3]) \oplus & 08_x \cdot S(I[4]) \oplus & 09_x \cdot S(I[5]) \oplus & 06_x \cdot S(I[6]) \oplus & 09_x \cdot S(I[7]) \\
09_x \cdot S(I[0]) \oplus & 05_x \cdot S(I[1]) \oplus & 01_x \cdot S(I[2]) \oplus & 01_x \cdot S(I[3]) \oplus & 06_x \cdot S(I[4]) \oplus & 08_x \cdot S(I[5]) \oplus & 09_x \cdot S(I[6]) \oplus & 06_x \cdot S(I[7]) \\
06_x \cdot S(I[0]) \oplus & 09_x \cdot S(I[1]) \oplus & 05_x \cdot S(I[2]) \oplus & 01_x \cdot S(I[3]) \oplus & 01_x \cdot S(I[4]) \oplus & 06_x \cdot S(I[5]) \oplus & 08_x \cdot S(I[6]) \oplus & 09_x \cdot S(I[7]) \\
09_x \cdot S(I[0]) \oplus & 06_x \cdot S(I[1]) \oplus & 09_x \cdot S(I[2]) \oplus & 05_x \cdot S(I[3]) \oplus & 01_x \cdot S(I[4]) \oplus & 01_x \cdot S(I[5]) \oplus & 06_x \cdot S(I[6]) \oplus & 08_x \cdot S(I[7]) \\
08_x \cdot S(I[0]) \oplus & 09_x \cdot S(I[1]) \oplus & 06_x \cdot S(I[2]) \oplus & 09_x \cdot S(I[3]) \oplus & 05_x \cdot S(I[4]) \oplus & 01_x \cdot S(I[5]) \oplus & 01_x \cdot S(I[6]) \oplus & 06_x \cdot S(I[7]) \\
06_x \cdot S(I[0]) \oplus & 08_x \cdot S(I[1]) \oplus & 09_x \cdot S(I[2]) \oplus & 06_x \cdot S(I[3]) \oplus & 09_x \cdot S(I[4]) \oplus & 05_x \cdot S(I[5]) \oplus & 01_x \cdot S(I[6]) \oplus & 01_x \cdot S(I[7])
\end{bmatrix}
$$

$$
\begin{array}{lcllllllll}
LT_0(x) & = & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \\
LT_1(x) & = & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \\
LT_2(x) & = & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \\
LT_3(x) & = & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \\
LT_4(x) & = & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \\
LT_5(x) & = & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \\
LT_6(x) & = & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x) \parallel & 01_x \cdot S(x) \\
LT_7(x) & = & 01_x \cdot S(x) \parallel & 05_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 09_x \cdot S(x) \parallel & 08_x \cdot S(x) \parallel & 06_x \cdot S(x) \parallel & 01_x \cdot S(x)
\end{array}
$$

The results of the operations in the columns are saved in eight lookup tables, namely LT_i, where $i = 0, 1, \cdots, 7$. Utilizing the lookup tables, the output value O is calculated as follows:

$$O = LT_0(I[0]) \oplus LT_1(I[1]) \oplus LT_2(I[2]) \oplus LT_3(I[3]) \oplus LT_4(I[4]) \oplus LT_5(I[5]) \oplus LT_6(I[6]) \oplus LT_7(I[7])$$

Table 6.5: *G*-function Operations

Required Memory (KB):	16
# of table lookups:	16
# of XOR (⊕):	20
# of Addition (⊞):	2
# of Subtraction (⊟):	2

Table 6.6: Number of Operations Used in Sarmal

Sarmal-224/256	
Required Memory (KB):	16
# of table lookups:	512
# of XOR ($\oplus$):	656
# of Addition ($\boxplus$):	64
# of Subtraction ($\boxminus$):	64
Sarmal-384/512	
Required Memory (KB):	16
# of table lookups:	640
# of XOR ($\oplus$):	800
# of Addition ($\boxplus$):	80
# of Subtraction ($\boxminus$):	80

6.2 Performance

We provide the sofware performance of Sarmal on different platforms whose details are given in Table 6.7 case by case. The software performance is measured in Table 6.8 at each architecture depending on the data size. Namely, starting from hashing 1 byte of message we increase the message size up to 10^5 bytes. The preformance is given by cycles per byte in Table 6.8.

Table 6.7: Implementation Platforms

Properties	Case I	Case II	Case III
Processor	Core 2 Duo	Core 2 Duo	Core 2 Duo
CPU Frequency	2.0 GHz	1.6 GHz	2.0 GHz
FSB / L2 Cache	800 MHz/ 4-MB	800 MHz / 4-MB	800 MHz / 4-MB
RAM	2-GB DDR2 667 MHz	2-GB DDR2 667 MHz	2-GB DDR2 667 MHz
Operating System	Windows Vista 32-bit	Mac OS X 10.5.5	Ubuntu 8.04.1 64-bit
Compiler	Visual Studio 2005	GNU C Compiler (GCC) v4.0.1	GNU C Compiler (GCC) v4.2.4
Properties	**Case IV**	**Case V**	**Case VI**
Processor	Core 2 Duo	Core 2 Duo	AMD Athlon(tm)64 X2
CPU Frequency	1.8 GHz	1.8 GHz	2.4 GHz
FSB / L2 Cache	800 MHz / 2-MB	800 MHz / 2-MB	2000 MHz / 1-MB
RAM	1-GB DDR2 667 MHz	1-GB DDR2 667 MHz	2-GB DDR2 333 MHz
Operating System	Windows Vista 64-bit	Ubuntu 8.04.1 32-bit	Ubuntu 8.04.1 64-bit
Compiler	Visual Studio 2005	GNU C Compiler (GCC) v4.2.4	GNU C Compiler (GCC) v4.2.4

Table 6.8: Software Performance of Sarmal

Case I						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	2640	263	25.70	19.08	18.68	19.18
Sarmal-256	2670	267	26.00	19.08	18.67	19.20
Sarmal-384	3150	315	31.00	23.13	22.66	23.33
Sarmal-512	3160	317	31.10	23.17	22.67	23.33
Case II						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	9496	949.60	94.40	58.34	56.41	63.59
Sarmal-256	9568	955.20	94.96	58.42	56.30	56.16
Sarmal-384	13552	1353.60	134.64	92.26	90.70	89.87
Sarmal-512	15968	1348.80	130.08	92.43	91.23	89.96
Case III						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	1580	157	14.00	10.23	10.00	10.05
Sarmal-256	1580	156	14.00	10.26	10.05	10.04
Sarmal-384	1930	192	17.40	12.96	12.71	12.67
Sarmal-512	1930	192	17.40	12.96	12.68	12.66
Case IV						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	1386	139.50	13.14	9.68	9.50	9.43
Sarmal-256	1386	138.60	12.96	9.62	9.44	9.38
Sarmal-384	1602	162.90	15.30	11.36	11.16	11.07
Sarmal-512	1593	161.10	15.39	11.18	10.98	10.90
Case V						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	5850	584	57.51	37.85	36.50	36.03
Sarmal-256	5625	567	55.62	37.82	36.44	36.02
Sarmal-384	10989	1114.20	109.71	84.20	83.56	83.09
Sarmal-512	11133	1094.40	109.44	84.49	83.78	79.21
Case VI						
Data Length(bytes)	1	10	100	1 000	10 000	100 000
Sarmal-224	2223	220.10	19.50	14.20	13.89	13.84
Sarmal-256	2207	218.10	19.32	14.16	13.86	13.83
Sarmal-384	2721	269.10	24.42	18.18	17.83	17.76
Sarmal-512	2715	268.80	24.37	18.20	17.83	17.74

6.3 Remarks

The suitability of Sarmal to be used for ubiquitious devices (including Voice Satellite applications) which have constrained environments can be given depending on the processor on which Sarmal is implemented. As Sarmal can be implemented efficiently in software on 8/32/64-bit processors with sufficient parallelism, it is well suited for that kind of sensitive applications. The only limitations and the drawbacks of Sarmal on 8/32-bit processors are w-bit oriented structure of the compression function. However, the main workload is to implement the subround function g which is highly suitable for all kind of processors. The remaining operations, although they are defined on w-bit, are simple and easy to handle for all kind of processors as they consist XOR, modular addition and subtraction.

We did not perform any hardware implementation for Sarmal. An upper bound for the area estimates can be given according to the number of operations given in this chapter. The memory requirements can be given as 616-bytes for all digest sizes and 376-bytes for a specific digest size. These values are given excluding the code size. We expect to implement Sarmal in different architectures in the later stages of the competition. However, we expect that Sarmal fits at most 1KB which is tolerable for many devices.

Chapter 7

Acknowledgements

Firstly, we would like to thank to The Scientific and Technological Research Council of Turkey to support this work financially. We also thank Çağdaş Çalık, Meltem Sönmez Turan, Deniz Toz, Cihangir Tezcan, Barış Ege and Elif Vural for proof-reading this version of the documentation, providing many suggestions for improvement and fruitful discussions. Special thanks to Ali Doğanaksoy for always being there for us. Moreover, we thank all people at Middle East Technical University for making it all possible.

Bibliography

[1] Ross J. Anderson and Eli Biham. TIGER: A Fast New Hash Function. In Gollmann [29], pages 89–97.

[2] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-Property-Preserving Iterated Hashing: ROX. In *ASIACRYPT*, pages 130–146, 2007.

[3] Kazumaro Aoki and Yu Sasaki. Preimage Attacks on One-Block MD4 and Full-Round MD5. In *Selected Areas in Cryptography,to appear*, 2008.

[4] Jean-Philippe Aumasson, Willi Meier, and Florian Mendel. Preimage Attacks on 3-pass HAVAL and Step-Reduced MD5. In *Selected Areas in Cryptography,to appear*, 2008.

[5] Paulo S. L. M. Barreto and Vincent Rijmen. The Whirlpool Hashing Function. *First open NESSIE Workshop*, 2000.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[7] D.J Bernstein. ChaCha, A Variant of Salsa20. In *SASC 2008 – The State of the Art of Stream Ciphers. ECRYPT (2008), http://cr.yp.to/rumba20.html*, 2008.

[8] D.J Bernstein. Salsa20. In *Technical Report 2005/025, eSTREAM, ECRYPT Stream Cipher Project (2005), http://cr.yp.to snuffle.html*, 2008.

[9] Eli Biham, Charles Bouillaguet, Orr Dunkelman, Pierre-Alain Fouque, and Sebastian Zimmer. Re-Visiting HAIFA and Why You Should Visit,too. In *Hash Functions in Cryptology: Theory and Practice*, 2008.

[10] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Franklin [26], pages 290–305.

[11] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In Cramer [18], pages 36–57.

[12] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. *Cryptology ePrint Archive, Report 2007/278*, 2007.

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

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

[15] Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[16] Christophe De Cannière and Christian Rechberger. Preimages for Reduced SHA-0 and SHA-1. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 179–202. Springer, 2008.

[17] Scott Contini, Krystian Matusiewicz, and Josef Pieprzyk. Extending FORK-256 Attack to the Full Hash Function. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS*, volume 4861 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 2007.

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

[19] J Daemen and V Rijmen. The Block Cipher Rijndael. *Smart Card Research and Applications, Proceedings*, 1820:277–284, 2000.

[20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[21] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [14], pages 416–427.

[22] Richard D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[23] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In Gollmann [29], pages 71–82.

[24] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. *Fast Software Encryption*, pages 71–82, 1996.

[25] FIPS. *The Keyed-Hash Message Authentication Code (HMAC)*. pub-NIST, pub-NIST:adr, March 2002.

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

[27] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the Cryptographic Applications of Random Functions. In *CRYPTO*, pages 276–288, 1984.

[28] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. *J. ACM*, 33(4):792–807, 1986.

[29] Dieter Gollmann, editor. *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*. Springer, 1996.

[30] Deukjo Hong, Donghoon Chang, Jaechul Sung, Sangjin Lee, Seokhie Hong, Jaesang Lee, Dukjae Moon, and Sungtaek Chee. A New Dedicated 256-Bit Hash Function: FORK-256. In Robshaw [57], pages 195–209.

[31] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jaesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. Hight: A new block cipher suitable for low-resource device. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2006.

[32] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin [26], pages 306–316.

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

[34] John Kelsey and Stefan Lucks. Collisions and Near-Collisions for Reduced-Round Tiger. In Robshaw [57], pages 111–125.

[35] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Cramer [18], pages 474–490.

[36] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Biryukov [13], pages 39–57.

[37] Gaëtan Leurent. MD4 is Not One-Way. In Nyberg [52], pages 412–428.

[38] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, New York, NY, USA, 1997.

[39] Krystian Matusiewicz, Thomas Peyrin, Olivier Billet, Scott Contini, and Josef Pieprzyk. Cryptanalysis of FORK-256. In Biryukov [13], pages 19–38.

[40] Ueli M. Maurer. Indistinguishability of Random Systems. In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 110–132. Springer, 2002.

[41] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

[42] Ueli M. Maurer and Johan Sjödin. Single-Key AIL-MACs from Any FIL-MAC. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 472–484. Springer, 2005.

[43] Florian Mendel, Joseph Lano, and Bart Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2007.

[44] Florian Mendel, Bart Preneel, Vincent Rijmen, Hirotaka Yoshida, and Dai Watanabe. Update on Tiger. In Rana Barua and Tanja Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2006.

[45] Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In *ASIACRYPT*, pages 536–550, 2007.

[46] Ralph C. Merkle. One way hash functions and des. In Brassard [14], pages 428–446.

[47] Ivica Nikolic and Alex Biryukov. Collisions for Step-Reduced SHA-256. In Nyberg [52], pages 1–15.

[48] NIST. Secure Hash Standard. In *Federal Information Processing Standard, FIPS-180*, April 1995.

[49] NIST. FIPS 180-2 Secure Hash Standard. In *http://csrc.nist.gov/publications/fips/fips180-2 fips180-2withchangenotice.pdf*, August 2002.

[50] NIST. Secure Hash Standard. In *Federal Information Processing Standard, FIPS-180*, May 1993.

[51] NIST. Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. In *http://csrc.nist.gov/groups/ST/hash/index.html*, November 2007.

[52] Kaisa Nyberg, editor. *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*. Springer, 2008.

[53] Onur Özen and Kerem Varıcı. On the Security of the Encryption Mode of Tiger. In *Information Security and Cryptology, Ankara*, December 2007.

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

[55] Ronald L. Rivest. The MD4 Message Digest Algorithm. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990.

[56] Ronald L. Rivest. The MD5 message-digest Algorithm, 1992.

[57] Matthew J. B. Robshaw, editor. *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*. Springer, 2006.

[58] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[59] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New Message Difference for MD4. In Biryukov [13], pages 329–348.

[60] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In Biryukov [13], pages 181–195.

[61] XY Wang, XJ Lai, DG Feng, H Chen, and XY Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. *Advances In Cryptology - Eurocrypt 2005,Proceedings*, 3494:1–18, 2005.

[62] XY Wang, YL Yin, and HB Yu. Finding Collisions in the Full SHA-1. *Advances In Cryptology - Crypto 2005, Proceedings*, 3621:17–36, 2005.

[63] XY Wang and HB Yu. How to Break MD5 and Other Hash Functions. *Advances In Cryptology - Eurocrypt 2005,Proceedings*, 3494:19–35, 2005.

[64] XY Wang, HB Yu, and YL Yin. Efficient Collision Search Attacks on SHA-0. *Advances In Cryptology - Crypto 2005, Proceedings*, 3621:1–16, 2005.

[65] Hirotaka Yoshida, Dai Watanabe, Katsuyuki Okeya, Jun Kitahara, Hongjun Wu, Özgül Küçük, and Bart Preneel. Mame: A compression function with reduced hardware requirements. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2007.

Appendix A

S-box of Sarmal

Table A.1: S-box

	00_x	01_x	02_x	03_x	04_x	05_x	06_x	07_x	08_x	09_x	$0A_x$	$0B_x$	$0C_x$	$0D_x$	$0E_x$	$0F_x$
00_x	$3A_x$	$5B_x$	$F2_x$	$0F_x$	$E4_x$	AD_x	29_x	91_x	$C5_x$	47_x	$B8_x$	63_x	$8C_x$	10_x	DE_x	76_x
10_x	$2C_x$	75_x	89_x	40_x	$A3_x$	$E1_x$	32_x	$6D_x$	BB_x	$0E_x$	$C6_x$	94_x	FA_x	DF_x	17_x	58_x
20_x	61_x	$D0_x$	$A4_x$	$B5_x$	82_x	FC_x	93_x	$2A_x$	$4F_x$	$C8_x$	07_x	39_x	ED_x	$7B_x$	56_x	$1E_x$
30_x	$E7_x$	44_x	90_x	79_x	$3B_x$	26_x	AF_x	$F8_x$	$D3_x$	$5A_x$	11_x	85_x	$6E_x$	$B2_x$	CC_x	$0D_x$
40_x	45_x	EC_x	16_x	21_x	$5E_x$	70_x	08_x	BF_x	$6A_x$	33_x	99_x	$C7_x$	DB_x	FD_x	84_x	$A2_x$
50_x	$F6_x$	$B9_x$	35_x	$D4_x$	$9F_x$	67_x	$8B_x$	EE_x	72_x	$1D_x$	$5C_x$	$A0_x$	28_x	43_x	01_x	CA_x
60_x	$D9_x$	66_x	$0C_x$	$F7_x$	CD_x	$B4_x$	$1A_x$	73_x	$E8_x$	$8F_x$	$A5_x$	51_x	42_x	$2E_x$	30_x	$9B_x$
70_x	$9D_x$	$1F_x$	$E3_x$	CB_x	$F9_x$	$8A_x$	64_x	$3C_x$	00_x	$B6_x$	$4E_x$	22_x	$A1_x$	55_x	78_x	$D7_x$
80_x	12_x	98_x	$4A_x$	$8E_x$	$B1_x$	$C3_x$	DC_x	54_x	$A6_x$	$F0_x$	EB_x	$7D_x$	09_x	37_x	$2F_x$	65_x
90_x	88_x	$C2_x$	$2B_x$	13_x	60_x	$9E_x$	$F5_x$	$A7_x$	59_x	$D1_x$	$7A_x$	EF_x	36_x	04_x	$4D_x$	BC_x
$A0_x$	53_x	$3E_x$	BD_x	$A8_x$	$4C_x$	02_x	71_x	19_x	87_x	$E5_x$	FF_x	DA_x	$C4_x$	96_x	$6B_x$	20_x
$B0_x$	74_x	27_x	$C1_x$	$E6_x$	$0A_x$	49_x	$5D_x$	$D2_x$	FE_x	AB_x	80_x	$1C_x$	$B3_x$	68_x	95_x	$3F_x$
$C0_x$	CF_x	$8D_x$	$7E_x$	$9A_x$	$D6_x$	$1B_x$	$B7_x$	05_x	31_x	69_x	23_x	48_x	50_x	AC_x	$E2_x$	$F4_x$
$D0_x$	AE_x	03_x	$6F_x$	52_x	25_x	38_x	$E0_x$	86_x	14_x	$7C_x$	DD_x	FB_x	97_x	$C9_x$	BA_x	41_x
$E0_x$	$B0_x$	$F1_x$	57_x	$6C_x$	18_x	$D5_x$	CE_x	$4B_x$	$2D_x$	92_x	34_x	06_x	$7F_x$	EA_x	$A9_x$	83_x
$F0_x$	$0B_x$	AA_x	$D8_x$	$3D_x$	77_x	$5F_x$	46_x	$C0_x$	$9C_x$	24_x	62_x	BE_x	15_x	81_x	$F3_x$	$E9_x$