Submission to the SHA-3 Competition: The CHI Family of Cryptographic Hash Algorithms

Design Team: Cryptographic Hash Initiative, Qualcomm International

Lead Designers: Phil Hawkes, Cameron McDonald
phawkes@qualcomm.com, cameronm@qualcomm.com
Level 3, 230 Victoria Rd, Gladesville, NSW 2111, Australia

January 13, 2009

Abstract

This document specifies the CHI family of cryptographic hash algorithms for the SHA-3 hash function competition being sponsored by NIST. The CHI family improves on the SHA-2 family by utilizing a stronger bit-wise non-linear function and ensuring better diffusion. This reduces the number of steps required to make the hash function secure and reduces the number of addition operations required. The addition operations are the largest and most power-hungry operations of the SHA-2 family, so the CHI family offers significant reduction in both energy usage and (in the case of hardware implementations) speed /area metrics. The CHI family targets 64-bit architecture, but is efficiently implemented on smaller architectures.

1 Introduction

This document introduces four cryptographic hash algorithms collectively known as the CHI algorithms. The algorithms are denoted CHI-224, CHI-256, CHI-384 and CHI-512 according to the length of the output hash. The algorithms for CHI-224 and CHI-384 are almost identical to CHI-256 and CHI-512 respectively, differing only in the values of some constants. The basic parameters of the CHI algorithms are shown in Table 1.

Features of the CHI algorithms Regarding padding and parsing messages into message blocks for submitting to the compression function, the CHI algorithm processing is identical to FIPS 180-2 [56]. The compression function is then applied iteratively to the message blocks, as with the FIPS 180-2 algorithms. In order to provide resistance to length-extension attacks, the compression function for the final message block is slightly different from the compression function for the earlier blocks.

The main distinguishing feature of the CHI compression function is the *MAP*, a complex bit-sliced function with four inputs and three outputs. Research indicated that such a *MAP* function is one of the most efficient methods of maximizing non-linear effects when followed by addition operations. Since the *MAP* function is bit-sliced, the *MAP* is well suited to implementation on any platform. The remaining operations are common to most processors, making the CHI algorithms efficient across many platforms.

Implementing the CHI algorithms The primary goal of the designers has been to have good overall performance on all software platforms. For CHI-224 and CHI-256, this means that the algorithms may not be as fast as their SHA-2 counterparts on 32-bit platforms, in order to improve the efficiency on 64-bit processors. CHI-384 and CHI-512, on the other hand, are faster than their SHA-2 counterparts on all platforms. In hardware, all CHI algorithms are significantly more efficient that their SHA-2 counterparts.

Algorithm	Output Size	Internal State Size	Message Block Size	Maximum Message Length
CHI-224	224	256	512	$2^{64} - 1$
CHI-256	256	256	512	$2^{64} - 1$
CHI-384	384	512	1024	$2^{128} - 1$
CHI-512	512	512	1024	$2^{128} - 1$

Table 1: Basic parameters of the CHI algorithms. All sizes are given in bits.

The Cryptographic Hash Initiative (CHI) Project The CHI Project was instigated by Greg Rose, head of the Qualcomm Product Security Initiative. The CHI project was coordinated by Phil Hawkes. Phil Hawkes and Cameron McDonald led the design effort. Brian Rosenberg and Lu Xiao were major contributors to the design and design review, with notable contributions from Steve Millendorf, Craig Northway, David Jacobson, Lu Xiao and Yafei Yang. Cameron McDonald coordinated the software implementation, with assistance from Craig Brown, Craig Northway and Jessica Purser. Bijan Ansari implemented the hardware simulation. Arun Balakrishnan, Alexander Gantman, John Jozwiak, Yinian Mao, Michael Paddon, Anand Palanigounder, Aram Perez and Miriam Wiggers de Vries completed the CHI project team.

1.1 Organization of this Document

This document is organized in the following way. The remainder of the introduction contains definitions that are used by all parts of this document. The CHI algorithms are specified in Part I. The design rationale for the CHI algorithms is provided in Part II. A discussion on the cryptanalysis of CHI algorithms follows in Part III. Part IV discusses implementation characteristics of the CHI algorithms. Part V clarifies how this submission fulfills the submission requirements, and explains why we believe the CHI algorithms rate well with respect to the evaluation criteria. Appendix A shows an example of the intermediate values for CHI-224, CHI-256, CHI-384 and CHI-512. The changes history for this document is provided in Appendix B.

1.2 Definitions

1.2.1 Glossary of Terms and Acronyms

Tables 2 and 3 contain a glossary of the terms used in this document. Table 4 contains a list of the acronyms used in this document.

Bias	A measure of the distance of a given distribution from a uniform distribution.
Bit	A binary digit having a value of 0 or 1.
Block	A group of bits of specified length.
Byte	A group of eight bits.
Compression Function	A function that produces an output hash value by combining an input hash value and a message block.
Compression Feedback Function	A function that alters the input hash value prior to combining with the output of the step functions.
Constant	A value that is independent of the message being processed.
Differential Cryptanalysis	A form of analysis based on tracing the propagation of differences through an algorithm.
Linear Approximation	An XOR sum of bits from the inputs and outputs of an operation.
Linear Cryptanalysis	A form of analysis based on biased linear approximations to non-linear components of an algorithm.
Message Block	The padded message is partitioned into message blocks. The size of the message block depends on the algorithm.
Message Expansion	The process of expanding a message block into a sequence of step inputs.
Microcontrollers	Low-end processors. These are typically 8-bit processors and 16-bit, although some 32-bit processors are now considered microcontrollers.

Table 2: Glossary of terms beginning with A-M used in this document

Nabla differential characteristic	A description of the location of bit differences and the sign of those bit differences.
Padding	The process of adding bits to the input message in order to form a multiple of the required block length.
Parsing	The process of partitioning the padded message into message blocks for processing by the compression function.
SHA-1	SHA-1 algorithm specified in [56].
SHA-2	The set of SHA-224, SHA-256, SHA-384 and SHA-512 algorithms specified in [56].
SHA-3 predecessors	The MD5, SHA-1 and SHA-2 algorithms.
Sign	(of a bit difference): a positive difference "+" indicates that the bit values changes from 0 to 1, while a negative difference "−" indicates that the bit values changes from 1 to 0.
Step	One application of the step function.
Step Constant	A value that is independent of the message but which changes from one step to the next.
Step Input	The sequence of word64s output from the message expansion, that are used in the step function .
Step Input Package	The set of step inputs and step constants used in a single step.
Step Function	A set of operations used to update the working variables based on the step input package.
Underlying Block Cipher	The CHI algorithms use a compression function in which most of the processing takes place in an underlying block cipher.
Word32	A group of 32 bits (4 bytes).
Word64	A group of 64 bits (8 bytes).
Working Variables	A set of variables whose values are updated by the step function.
XOR differential	A description of the location of bit differences.

Table 3: Glossary of terms beginning with N-Z used in this document

AES	Advanced Encryption Standard
ALU	Arithmetic Logical Unit
ANF	Algebraic Normal Form
ASIC	Application Specific Integrated Circuit
CHI	Cryptographic Hash Initiative (both the name of the development team and the name of the submission algorithm)
DES	Data Encryption Standard
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
HMAC	Keyed-Hash Message Authentication Code
LAB	Linear Approximation Bias
LSB	Least Significant Bit
MAC	Message Authentication Code
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
NDP	Nabla Differential Probability
PRF	Pseudo-Random Function
SHA	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data (a type of co-processor)
XDP	XOR-Differential Probability

Table 4: List of acronyms used in this document

1.2.2 Symbols

The uses of the "=" symbol in this document are defined in Table 5. Symbols that do not denote transformations on bit-strings are defined in Table 6. Symbols denoting transformations on bit-strings used in this document are defined in Table 7.

Symbol	Meaning
=	The expressions on both sides of the symbol can be considered equivalent.
:=	The variable on the left of the symbol is defined to be the resulting value of the expression to the right.
==	Returns TRUE if the evaluation of both sides are identical, and returns FALSE otherwise.

Table 5: Uses of the "=" symbol in this document.

Symbol	Meaning
$x[i]$	The i-th least significant bit of the bit string x
$\|x\|$	The number of non-zero bits in the bit string x (Hamming weight)
$Pr(\text{Event } \epsilon_1 \| \text{Event } \epsilon_2)$	The probability of Event ϵ_1 occurring given that ϵ_2 has occurred

Table 6: Other symbols in this document.

Symbol	Meaning	Operand Bit Length
$\wedge$	Bitwise AND operation	Any
$\vee$	Bitwise OR ("inclusive-OR") operation	Any
$\oplus$	Bitwise XOR ("exclusive-OR") operation	Any
$\neg$	Bitwise complement operation	Any
$\boxplus$	Addition modulo 2^{64}	64
$\ll_{64}$	Left-shift operation, where $x \ll_{64} n$ is obtained by discarding the left-most n bits of the word64 x and then padding the result with n zeroes on the right	64
$\gg_{64}$	Right-shift operation, where $x \gg_{64} n$ is obtained by discarding the rightmost n bits of the word64 x and then padding the result with n zeroes on the left	64
$\ll_{32}$	Left-shift operation, where $x \ll_{32} n$ is obtained by discarding the left-most n bits of the word32 x and then padding the result with n zeroes on the right	32
$\gg_{32}$	Right-shift operation, where $x \gg_{32} n$ is obtained by discarding the rightmost n bits of the word32 x and then padding the result with n zeroes on the left	32
$\|\|$	Concatenation	Any

Table 7: Symbols denoting operations in this document.

Contents

Part I
Specification of the CHI Algorithms

2 DEFINITIONS

2.1 Algorithm Parameters

The following parameters are used in this specification.

A, B, C, D, E, F	Working variables (word64s) used in all CHI algorithms for the computation of the hash values, $H^{(i)}$
G, P, Q	Additional working variables (word64s) used in CHI-384 and CHI-512 only
$preR$,$preS$,$preT$, $preU$, V_0, V_1, R, S, T, U, X, Y, Z, XX, YY, ZZ, AA, DD, $newA, newD$	Temporary word64 variables used in the hash computation in all CHI algorithms
$newG, GG$	Additional temporary word64 variables used in CHI-384 and CHI-512 only
$H^{(i)}$	The i-th hash value. $H^{(0)}$ is the initial hash value; $H^{(N)}$ is the final hash value and is used to determine the message digest.
$H_j^{(i)}$	The j-th word64 of the i-th hash value, where $H_0^{(i)}$ is the left-most word64 of hash value $H^{(i)}$.
K_t	Constant values.
κ	The number of zeroes appended when padding the message.
`len`	Length of the message, M, in bits.
L	A bit string representation of `len`.
M	Message to be hashed.
$M(i)$	Message block i, with a size of 512 bits (for CHI-224 and CHI-256) or 1024 bits (for CHI-384 and CHI-512).
$M_j^{(i)}$	The j-th word64 of the i-th message block, where $M_0^{(i)}$ is the left-most word64 of message block $M^{(i)}$.
n	Number of bits to be rotated or shifted when a word64 is operated upon.
N	Number of blocks in the padded message.
s	The chosen size of the hash output.
W_t	The t-th word64 of the step inputs.

3 NOTATION AND CONVENTIONS

3.1 Bit Strings and Integers

The following terminology related to bit strings and integers will be used.

1. A hex digit is an element of the set {`0, 1,..., 9, a,..., f`}. A hex digit is the representation of a 4-bit string. For example, the hex digit "`7`" represents the 4-bit string "0111", and the hex digit "`a`" represents the 4-bit string "1010". Note that the `typewriter` font has been used to maintain identical spacing for hex digit representations.

2. When the integer n is a multiple of 4, an n-bit string may be represented as a sequence of hex digits. In this document, hex digit representations can be identified by the symbols "`0x`" preceding the representation. To convert a bit string to hex digits, the identifier "`0x`" is written and then each 4-bit string is converted to its hex digit equivalent, as described in (1) above. For example, the word64 with binary representation

   $$1010000100000011111111000100011$$
   $$0011001011101111001100000011010$$

   can be expressed in hex digits as `0xa103fe2332ef301a`.

   Throughout this specification, the "big-endian" convention is used when expressing word64s, so that within each word64, the most significant bit is stored in the left-most bit position.

3. An integer may be represented as a word64 or pair of word64s. A representation of the message length, `len` , in bits, is required for the padding techniques of Section 5.1.

   An integer between 0 and $(2^{64} - 1)$ *inclusive* may be represented as a word64. The least significant four bits of the integer are represented by the right-most hex digit of the word64 representation. For example, the integer $291 = 2^8 + 2^5 + 2^1 + 2^0 = 256 + 32 + 2 + 1$ is represented by the hex word64 `0x0000000000000123`.

   If γ is an integer, $0 \leq \gamma < 2^{128}$, then $\gamma = 2^{64}\alpha + \beta$, where $0 \leq \alpha < 2^{64}$ and $0 \leq \beta < 2^{64}$. Since α and β can be represented as word64s x and y, respectively, the integer γ can be represented as the pair of words (x, y). This property is used for CHI-384 and CHI-512.

4. A word64 can be considered as two word32s, and the following notation is used.

   - The upper word32 (most significant word32) of a word64 is indicated by following the variable's symbol with a "u" (for example, Xu indicates the most significant word32 of the word64 X).
   - The lower word32 (least significant word32) of a word64 is indicated by following the variable's symbol with a "l" (for example, Xl indicates the least significant word32 of the word64 X).

   For example, in CHI-224 and CHI-256 the word64 A is $A = Au \| Al$.

5. For the CHI algorithms, the number of bits in a *message block*, m, depends on the algorithm:

   a) For CHI-224 and CHI-256, each message block has 512 bits, which are represented as a sequence of eight word64s.

b) For CHI-384 and CHI-512, each message block has 1024 bits, which are represented as a sequence of sixteen word64s.

3.2 Operations on Word32s

The following operation is applied to word32s in the CHI-224 and CHI-256 hash algorithms.

1. The rotate right (circular right shift) operation $ROTR32^n(x)$, where x is a word32 and n is an integer with $0 \leq n < 32$, is defined by

$$ROTR32^n(x) := (x \gg_{32} n) \vee (x \ll_{32} 32 - n).$$

Thus, $ROTR32^n(x)$ is equivalent to a circular shift (rotation) of x by n positions to the right in a block of 32-bits.

3.3 Operations on Word64s

The following operations are applied to word64s in all four CHI algorithms.

1. Addition modulo 2^{64}. The operation "$x \boxplus y$" is defined as follows. The word64s x and y represent integers α and β, where $0 \leq \alpha < 2^{64}$ and $0 \leq \beta < 2^{64}$. For positive integers δ and ϵ, let $\delta \mod \epsilon$ be the remainder upon dividing δ by ϵ. Compute

$$\gamma := (\alpha + \beta) \mod 2^{64}.$$

Then $0 \leq \gamma < 2^{64}$. Convert the integer γ to a word64, z, and define $z := x \boxplus y$.

2. The right shift operation $SHR64^n(x)$, where x is a word64 and n is an integer with $0 \leq n < 64$, is defined by

$$SHR64^n(x) := x \gg_{64} n.$$

3. The rotate right (circular right shift) operation $ROTR64^n(x)$, where x is a word64 and n is an integer with $0 \leq n < 64$, is defined by

$$ROTR64^n(x) := (x \gg_{64} n) \vee (x \ll_{64} 64 - n).$$

Thus, $ROTR64^n(x)$ is equivalent to a circular shift (rotation) of x by n positions to the right in a block of 64-bits.

4. The operation $SWAP8(x)$ returns a word64 formed by reversing the bytes in (a copy of) the word64 x, without altering the order of the bits in each byte.

5. The operation $SWAP32(x) = xl \| xu$ returns a word64 formed by reversing the word32s in (a copy of) the word64 x, without altering the order of the bits in each word32. Note that $SWAP32(x)$ is identical to $ROTR64^{32}(x)$.

6. The function $MSB_{32}(x)$ outputs only the 32 most significant bits (leftmost bits) of the word64 x.

The following operation is applied to word64s in the CHI-224 and CHI-256 algorithms.

1. The rotate right operation $DROTR32^{nu,nl}(x)$, where $x = xu \| xl$ is a word64 and nu, nl are integers with $0 \leq nu < 32$, $0 \leq nl < 32$, is defined by

$$\begin{aligned} DROTR32^{nu,nl}(x) &:= ((xu \gg_{32} nu) \vee (xu \ll_{32} (32 - nu))) \; \| \\ &\qquad ((xl \gg_{32} nl) \vee (xl \ll_{32} (32 - nl))) \\ &= ROTR32^{nu}(xu) \| ROTR32^{nl}(xl). \end{aligned}$$

   Thus, $DROTR32^{nu,nl}(x)$ is equivalent to circular shifts (rotation) of the two word32s xu and xl by nu and nl positions to the right (respectively) in sub-blocks of 32-bits.

4 FUNCTIONS AND CONSTANTS

4.1 The *MAP* Function

The *MAP* Function acts on four input word64s (R, S, T, U), and generates three output word64s: the first output word64 is computed by MAP_0, the second by MAP_1 and the third by MAP_2. The functions MAP_i can be implemented as described in the following formula. In this description, multiplication implicitly corresponds to the bit-wise AND operation.

$$\begin{aligned} MAP_0(R,S,T,U) &:= (R\neg STU) \vee (\neg S\neg TU) \vee (\neg R\neg TU) \vee (\neg RST) \vee (R\neg T\neg U); \\ MAP_1(R,S,T,U) &:= (RST\neg U) \vee (R\neg S\neg T\neg U) \vee (\neg R\neg ST) \vee (S\neg TU) \vee (\neg RU); \\ MAP_2(R,S,T,U) &:= (\neg R\neg STU) \vee (\neg RST\neg U) \vee (S\neg TU) \vee (R\neg T\neg U) \vee (RSU) \vee (R\neg S\neg U); \end{aligned}$$

Note: This representation is the "sum of products" representation obtained by using a Karnaugh map and is provided for reference purposes. There are methods of computing the *MAP* outputs that are more efficient than simply implementing the *MAP* as described above.

Note that the *MAP* function acts bits-wise. At a given bit position i, the three output bits $MAP_0[i], MAP_1[i], MAP_2[i]$ at that position are a function of the bits in the same positions of the four inputs $R[i]$, $S[i]$, $T[i]$, $U[i]$. The values of $MAP_0[i], MAP_1[i], MAP_2[i]$ as functions of $R[i]$, $S[i]$, $T[i]$, $U[i]$ are given in Table 8.

4.2 Message Expansion Functions

The message expansion components use two logical functions, where each function operates on a word64 input and the result is a new word64. For CHI-224 and CHI-256, these logical functions are:

$$\begin{aligned} \mu_0^{\{256\}}(x) &:= ROTR64^{36}(x) \oplus ROTR64^{18}(x) \oplus SHR64^{1}(x); \\ \mu_1^{\{256\}}(x) &:= ROTR64^{59}(x) \oplus ROTR64^{37}(x) \oplus SHR64^{10}(x). \end{aligned}$$

For CHI-384 and CHI-512, these logical functions are:

$$\begin{aligned} \mu_0^{\{512\}}(x) &:= ROTR64^{36}(x) \oplus ROTR64^{18}(x) \oplus SHR64^{1}(x); \\ \mu_1^{\{512\}}(x) &:= ROTR64^{60}(x) \oplus ROTR64^{30}(x) \oplus SHR64^{3}(x). \end{aligned}$$

	R	S	T	U	MAP_0	MAP_1	MAP_2	$MAP_0 \| MAP_1 \| MAP_2$
0	0	0	0	0	1	0	0	4
1	0	0	0	1	1	1	0	6
2	0	0	1	0	0	1	0	2
3	0	0	1	1	0	1	1	3
4	0	1	0	0	0	0	0	0
5	0	1	0	1	1	1	1	7
6	0	1	1	0	1	0	1	5
7	0	1	1	1	1	1	0	6
8	1	0	0	0	1	1	1	7
9	1	0	0	1	0	0	0	0
10	1	0	1	0	0	0	1	1
11	1	0	1	1	1	0	0	4
12	1	1	0	0	1	0	1	5
13	1	1	0	1	0	1	1	3
14	1	1	1	0	0	1	0	2
15	1	1	1	1	0	0	1	1

Table 8: The truth tables for the values of $MAP_0[i]$, $MAP_1[i]$, $MAP_2[i]$ as functions of $R[i]$, $S[i]$, $T[i]$, $U[i]$. The final column is the evaluation of the three functions with $MAP_0[i]$ in the most significant bit, $MAP_1[i]$ in the middle bit and $MAP_2[i]$ in the least significant bit.

4.3 Input Mixing Functions

The *DATA-INPUT* phase of the step function components use two logical functions, where each function operates on a word64 input, and the result is a new word64. For CHI-224 and CHI-256, these logical functions are:

$$\begin{aligned}\theta_0^{\{256\}}(x) &:= DROTR32^{21,21}(x) \oplus DROTR32^{26,26}(x) \oplus DROTR32^{30,30}(x);\\ \theta_1^{\{256\}}(x) &:= DROTR32^{14,14}(x) \oplus DROTR32^{24,24}(x) \oplus DROTR32^{31,31}(x).\end{aligned}$$

Note that the two functions for CHI-224 and CHI-256 operate on the upper word32 and lower word32 independently, and can be written as

$$\begin{aligned}\theta_0^{\{256\}}(x) &:= \theta_0^{\{256\}h}(xu) \| \theta_0^{\{256\}h}(xl);\\ \theta_1^{\{256\}}(x) &:= \theta_1^{\{256\}h}(xu) \| \theta_1^{\{256\}h}(xl);\end{aligned}$$

where $\theta_0^{\{256\}h}(y)$ and $\theta_1^{\{256\}h}(y)$ operate on a word32 to produce a word32 result. These functions are:

$$\begin{aligned}\theta_0^{\{256\}h}(y) &:= ROTR32^{21}(y) \oplus ROTR32^{26}(y) \oplus ROTR32^{30}(y);\\ \theta_1^{\{256\}h}(y) &:= ROTR32^{14}(y) \oplus ROTR32^{24}(y) \oplus ROTR32^{31}(y).\end{aligned}$$

The "h" in the superscript is intended to indicate that the operation is applied to half of a word64.

For CHI-384 and CHI-512, these logical functions are:

$$\begin{aligned}\theta_0^{\{512\}}(x) &:= ROTR64^{5}(x) \oplus ROTR64^{6}(x) \oplus ROTR64^{43}(x);\\ \theta_1^{\{512\}}(x) &:= ROTR64^{20}(x) \oplus ROTR64^{30}(x) \oplus ROTR64^{49}(x).\end{aligned}$$

4.4 Step Constants

The CHI algorithms use a common set of word64 step constants K_0 ,$K_1, \ldots, K_{79}$. The first set of sixteen values represent the first 1024 bits of the fractional part of the square root of 13, partitioned into word64 values. Similarly the following four sets of sixteen values represent the first 1024 bits of the fractional part of the square root of 17, 19, 23 and 29 (respectively), which have then been partitioned into word64 values. In hex, these eighty constant word64s are (from left to right):

```
0x9b05688c2b3e6c1f, 0xdbd99e6ff3c90bdc, 0x4dbc64712a5bb168, 0x767e27c3cf76c8e7,
0x21ee9ac5ef4c823a, 0xb36ccfc1204a9bd8, 0x754c8a7fb36bd941, 0xcf20868f04a825e9,
0xabb379e0a8838bb0, 0x12d8b70a5959e391, 0xb59168fa9f69e181, 0x15c7d4918739f18f,
0x4b547673ea8d68e0, 0x3ced7326fcd0ef81, 0x09f1d77309998460, 0x1af3937415e91f32,
0x1f83d9abfb41bd6b, 0x23c4654c2a217583, 0x2842012131573f2a, 0xa59916aca3991fca,
0xec1b7c06fd19d256, 0x1ec785bcdc8baf26, 0x69ca4e0ff2e6bdd8, 0xca2575ee6c950d0e,
0x5bcf66d2fb3d99f6, 0x9d6d08c7bbca18f3, 0xeef64039f2175e83, 0x00ed5aebaa2ab6e0,
0x5040712fc29ad308, 0x6dafe433438d2c43, 0xbd7faa3f06c71f15, 0x03b5aa8ce9b6a4dd,
0x5be0cd19137e2179, 0x867f5e3b72221265, 0x43b6cbe0d67f4a20, 0xdb99d767cb0e4933,
0xdc450dbc469248bd, 0xfe1e5e4876100d6f, 0xb799d29ea1733137, 0x16ea7abcf92053c4,
0xbe3ece968dba92ac, 0x18538f84d82c318b, 0x38d79f4e9c8a18c0, 0xa8bbc28f1271f1f7,
0x2796e71067d2c8cc, 0xde1bf2334edb3ff6, 0xb094d782a857f9c1, 0x639b484b0c1daed1,
0xcbbb9d5dc1059ed8, 0xe7730eaff25e24a3, 0xf367f2fc266a0373, 0xfe7a4d34486d08ae,
0xd41670a136851f32, 0x663914b66b4b3c23, 0x1b9e3d7740a60887, 0x63c11d86d446cb1c,
0xd167d2469049d628, 0xdddbb606b9a49e38, 0x557f1c8bee68a7f7, 0xf99dc58b50f924bd,
0x205acc9f653512a5, 0x67c66344e4bab193, 0x18026e467960d0c8, 0xa2f5d84daeca8980,
0x629a292a367cd507, 0x98e67012d90cbb6d, 0xeed758d1d18c7e35, 0x031c02e4437dc71e,
0x79b63d6482198eb7, 0x936a9d7e8c9e4b33, 0xb30ca682c3e6c65d, 0xcc442382ba4262fa,
0x51179ba5a1d37ff6, 0x7202bde7a98eea51, 0x2b9f65d1df9c610f, 0xf56b742b0af1ce83,
0xf9989d199b75848b, 0xd142f19d8b46d578, 0x7a7580514d75ea33, 0xb74f9690808e704d
```

Note that the CHI-224 and CHI-256 algorithms require only the first forty of these constants.

5 PREPROCESSING

Preprocessing shall take place before hash computation begins. This preprocessing consists of three steps: padding the message, M (Section 5.1), parsing the padded message into message blocks (Section 5.2), and setting the initial hash value, $H^{(0)}$ (Section 5.3).

5.1 Padding the Message

The message, M, shall be padded before hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 512 (for CHI-224 and CHI-256) or 1024 bits (for CHI-384 and CHI-512).

5.1.1 CHI-224 and CHI-256

Suppose that the length of the message, M, is `len` bits. Append the bit "1" to the end of the message, followed by κ zero bits, where κ is the smallest, non-negative solution to the equation

$\texttt{len} + 1 + \kappa + 64 \equiv 0 \bmod 512$. Then append the 64-bit block L that is equal to the number $\texttt{len}$ expressed using a binary representation. For example, the (8-bit ASCII) message "abc" has length $\texttt{len} = 8 \times 3 = 24$, so the message is padded with a one bit, then $\kappa = 448 - (24 + 1) = 423$ zero bits, and then the message length, to become the 512-bit padded message

$$\underbrace{01100001}_{\text{“a”}}\ \underbrace{01100010}_{\text{“b”}}\ \underbrace{01100011}_{\text{“c”}}\ 1\ \overbrace{00\ldots00}^{\kappa=423}\ \overbrace{00\ldots0\ \underbrace{11000}_{\texttt{len}=24}}^{64}.$$

The length of the padded message should now be a multiple of 512 bits.

5.1.2 CHI-384 and CHI-512

Suppose that the length of the message, M, is $\texttt{len}$ bits. Append the bit "1" to the end of the message, followed by κ zero bits, where κ is the smallest, non-negative solution to the equation $\texttt{len} + 1 + \kappa + 128 \equiv 0 \bmod 1024$. Then append the 128-bit block L that is equal to the number $\texttt{len}$ expressed using a binary representation. For example, the (8-bit ASCII) message "abc" has length $\texttt{len} = 8 \times 3 = 24$, so the message is padded with a one bit, then $\kappa = (1024 - 128) - (24 + 1) = 871$ zero bits, and then the message length, to become the 1024-bit padded message

$$\underbrace{01100001}_{\text{“a”}}\ \underbrace{01100010}_{\text{“b”}}\ \underbrace{01100011}_{\text{“c”}}\ 1\ \overbrace{00\ldots00}^{\kappa=871}\ \overbrace{00\ldots0\ \underbrace{11000}_{\texttt{len}=24}}^{128}.$$

The length of the padded message should now be a multiple of 1024 bits.

5.2 Parsing the Padded Message

After a message has been padded, it must be parsed into N m_{size}-bit blocks $M^{(1)}$, $M^{(2)}, \ldots, M^{(N)}$ before the hash computation can begin. The blocks $M^{(1)}$, $M^{(2)}, \ldots, M^{(N)}$ are called *message blocks.*

5.2.1 CHI-224 and CHI-256

For CHI-224 and CHI-256, the padded message is parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}, \ldots,$ $M^{(N)}$. Since the 512 bits of the input block may be expressed as eight word64s, the first 64 bits of message block i are denoted $M_0^{(i)}$, the next 64 bits are $M_1^{(i)}$, and so on up to $M_7^{(i)}$.

5.2.2 CHI-384 and CHI-512

For CHI-384 and CHI-512, the padded message is parsed into N 1024-bit blocks, $M^{(1)}$, $M^{(2)}, \ldots,$ $M^{(N)}$. Since the 1024 bits of the input block may be expressed as sixteen word64s, the first 64 bits of message block i are denoted $M_0^{(i)}$, the next 64 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

5.3 Setting the Initial Hash Value ($H^{(0)}$)

Before hash computation begins for each of the secure hash algorithms, the initial hash value, $H^{(0)}$, must be set.

5.3.1 CHI-224

For CHI-224, the initial hash value, $H^{(0)}$, shall consist of the following six word64s, in hex:

$$\begin{aligned} H_0^{(0)} &:= \texttt{0xa54ff53a5f1d36f1}; \\ H_1^{(0)} &:= \texttt{0xcea7e61fc37a20d5}; \\ H_2^{(0)} &:= \texttt{0x4a77fe7b78415dfc}; \\ H_3^{(0)} &:= \texttt{0x8e34a6fe8e2df92a}; \\ H_4^{(0)} &:= \texttt{0x4e5b408c9c97d4d8}; \\ H_5^{(0)} &:= \texttt{0x24a05eee29922401}. \end{aligned}$$

These values represent the first 384 bits of the fractional part of the square root of seven, partitioned into word64 values. Note that these values agree with the first six word64s of the initial hash vale for CHI-384.

5.3.2 CHI-256

For CHI-256, the initial hash value, $H^{(0)}$, shall consist of the following six word64s, in hex:

$$\begin{aligned} H_0^{(0)} &:= \texttt{0x510e527fade682d1}; \\ H_1^{(0)} &:= \texttt{0xde49e330e42b4cbb}; \\ H_2^{(0)} &:= \texttt{0x29ba5a455316e0c6}; \\ H_3^{(0)} &:= \texttt{0x5507cd18e9e51e69}; \\ H_4^{(0)} &:= \texttt{0x4f9b11c81009a030}; \\ H_5^{(0)} &:= \texttt{0xe3d3775f155385c6}. \end{aligned}$$

These values represent the first 384 bits of the fractional part of the square root of eleven, partitioned into word64 values. Note that these values agree with the first six word64s of the initial hash vale for CHI-512.

5.3.3 CHI-384

For CHI-384, the initial hash value, $H^{(0)}$, shall consist of the following nine word64s, in hex:

$$
\begin{aligned}
H_0^{(0)} &:= \texttt{0xa54ff53a5f1d36f1};\\
H_1^{(0)} &:= \texttt{0xcea7e61fc37a20d5};\\
H_2^{(0)} &:= \texttt{0x4a77fe7b78415dfc};\\
H_3^{(0)} &:= \texttt{0x8e34a6fe8e2df92a};\\
H_4^{(0)} &:= \texttt{0x4e5b408c9c97d4d8};\\
H_5^{(0)} &:= \texttt{0x24a05eee29922401};\\
H_6^{(0)} &:= \texttt{0x5a8176cffc7c2224};\\
H_7^{(0)} &:= \texttt{0xc3edebda29bec4c8};\\
H_8^{(0)} &:= \texttt{0x8a074c0f4d999610}.
\end{aligned}
$$

These values represent the first 576 bits of the fractional part of the square root of seven, partitioned into word64 values.

5.3.4 CHI-512

For CHI-512, the initial hash value, $H^{(0)}$, shall consist of the following nine word64s, in hex:

$$
\begin{aligned}
H_0^{(0)} &:= \texttt{0x510e527fade682d1};\\
H_1^{(0)} &:= \texttt{0xde49e330e42b4cbb};\\
H_2^{(0)} &:= \texttt{0x29ba5a455316e0c6};\\
H_3^{(0)} &:= \texttt{0x5507cd18e9e51e69};\\
H_4^{(0)} &:= \texttt{0x4f9b11c81009a030};\\
H_5^{(0)} &:= \texttt{0xe3d3775f155385c6};\\
H_6^{(0)} &:= \texttt{0x489221632788fb30};\\
H_7^{(0)} &:= \texttt{0x41921db8feeb38c2};\\
H_8^{(0)} &:= \texttt{0x9af94a7c48bbd5b6}.
\end{aligned}
$$

These values represent the first 576 bits of the fractional part of the square root of eleven, partitioned into word64 values.

6 SUMMARY DESCRIPTION (Informative Only)

The padding and parsing have been described in Sections 5.1 and 5.2. The output of the padding and parsing is a sequence of message blocks $M^{(1)}$, $M^{(2)}, \ldots, M^{(N)}$. The message blocks are processed in order, producing a sequence of hash values $H^{(1)}$, $H^{(2)}, \ldots, H^{(N)}$. The message digest is formed from a specified subset of bits from the final hash value $H^{(N)}$.

When processing the i-th message block, the inputs to the compression function are the current hash value $H^{(i-1)}$, the message block $M^{(i)}$, and an indication if this message block is the final message block (that is, if $i == N$). The compression function computes the next hash value $H^{(i)}$ as a function of the inputs.

6.1 Summary of the Compression Function

The compression function comprises a message expansion, a set of step constants $\{K_t\}$, a set of working variables and a step function.

The message expansion is initialized using the message block $M^{(i)}$. The message expansion generates a sequence of word64s $\{W_t\}$ that are the step inputs.

CHI-224 and CHI-256 use a set of six working variables, while CHI-384 and CHI-512 use a set of nine working variables. The working variables are first initialized to the input hash value $H^{(i-1)}$. When processing the last message block, the working variables are rotated by one bit, so as to make processing the last message block distinct from other message blocks. The step function is applied multiple times to the set of working variables under the influence of the step inputs and step constants. The t-th iteration of the step function uses step inputs W_{2t}, W_{2t+1} and step constants K_{2t}, K_{2t+1}.

The output hash value $H^{(i)}$ is obtained by rotating the word64s of the input hash value $H^{(i-1)}$, and XORing the result with the final value of the working variables after applying the appropriate number of steps.

6.2 Summary of the Message Expansion Components

The message expansion uses a linear feedback shift register (LFSR) to generate a sequence of step inputs $\{W_t\}$. The state of the LFSR, called the message state, is initialized with the message block $M^{(i)}$. The message state is then updated iteratively using operations that are linear with respect to the XOR operation. When updating the LFSR in CHI-224 and CHI-256, a new word64 is generated by applying the linear functions $\mu_0^{\{256\}}$ and $\mu_1^{\{256\}}$ to two word64s of the message state, and XORing the output with another word64 of the state. When updating the LFSR in CHI-384 and CHI-512, a new word64 is generated by applying the linear functions $\mu_0^{\{512\}}$ and $\mu_1^{\{512\}}$ to two word64s of the message state, and XORing the output with two other word64s of the state. The last word64 is discarded, the remaining word64s of the message state are shifted one position, and the new word64 is inserted. The t-th step input W_t is the word64 in the message state that is discarded when updating the message state for the t-th time. The $\mu_0^{\{256\}}$, $\mu_1^{\{256\}}$, $\mu_0^{\{512\}}$ and $\mu_1^{\{512\}}$ functions consist of rotations in 64-bit blocks, right shift operations and XOR operations.

6.3 Summary of the Step Function Components

The step functions of the CHI algorithms share many similarities. The main similarity is that the algorithms all follow a sequence of phases: *PRE-MIXING*, *DATA-INPUT*, *MAP*, *POST-MIXING*, and *FEEDBACK*. All of the algorithms also utilize the *MAP* function described in Section 4.1. The following sub-sections provide a little more insight to the step function.

6.3.1 Summary of the CHI-224 and CHI-256 Step Function Components

For the CHI-224 and CHI-256 algorithms, the step function state consists of six word64s denoted (A, B, C, D, E, F), also known as the *working variables*. In CHI-224 and CHI-256, the step function updates two sub-blocks each step. The word64s most recently updated are A, D, the next most recently updated are B, E, and finally C, F. The step function of CHI-224 and CHI-256 follows a sequence of five phases:

1. **PRE-MIXING**: A linear mixing function is applied to the values of word64s A, B, D, E, resulting in four pre-mixed word64s $preR, preS, preT, preU$. This expansion involves performing the $SWAP32$ operation, rotations in 32-bit blocks and bit-wise XOR operations.

2. **DATA-INPUT**: The step inputs W_{2t}, W_{2t+1} are XORed with step constants K_{2t}, K_{2t+1} to form V_0, V_1. The values V_0 and V_1 are XORed with two pre-mixed words to form two MAP inputs. The remaining MAP inputs are obtained by applying the input mixing functions $\theta_0^{\{256\}}$ and $\theta_1^{\{256\}}$ to V_0 and V_1 respectively, and XORing with another two premixed words. The functions involve rotations in 32-bit blocks and bit-wise XOR operations.

3. **NONLINEARITY**: A nonlinear mapping is applied in bit-slice manner to the four MAP input word64s R, S, T, U, resulting in three MAP output words X, Y, Z. Each bit of X, Y and Z is a nonlinear mapping of the corresponding bits of the four MAP inputs.

4. **POST-MIXING**: The three MAP output word64s X, Y, Z are combined to form two feedback word64s AA, DD. The post-mixing uses the $SWAP8$ operation, the $SWAP32$ operation, rotations in 32-bit blocks, rotations in 64-bit blocks, and addition modulo 2^{64}.

5. **FEEDBACK**: The value of $newA$ is the XOR of AA and F, and the value of $newD$ is the XOR of DD and C. The word64s C and F are discarded, then:

   - word64 E is copied to word64 F; word64 D is copied to word64 E; the new word64 $newD$ is copied to word64 D;
   - word64 B is copied to word64 C; word64 A is copied to word64 B; and the new word64 $newA$ is copied to word64 A.

6.3.2 Summary of the CHI-384 and CHI-512 Step Function Components

For the CHI-384 and CHI-512 algorithms, the step function state consists of nine word64s denoted $(A, B, C, D, E, F, G, P, Q)$. As for CHI-224 and CHI-256, the word64s in the step function state are also known as the *working variables*. In CHI-384 and CHI-512 the step function updates three word64s each step. The word64s most recently updated are A, D, G, the next most recently updated are B, E, P, and finally C, F, Q. The step function of CHI-384 and CHI-512 follows a sequence of five phases:

1. **PRE-MIXING**: A linear mixing function is applied to the values of word64s A, B, D, E, G, P, resulting in four pre-mixed word64s $preR$, $preS$, $preT$, $preU$. This expansion involves performing rotations in 64-bit blocks and bit-wise XOR operations.

2. **DATA-INPUT**: The step inputs W_{2t}, W_{2t+1} are XORed with step constants K_{2t}, K_{2t+1} to form V_0, V_1. The values V_0 and V_1 are XORed with two pre-mixed words to form two MAP

inputs. The remaining two MAP inputs are obtained by applying the input mixing functions $\theta_0^{\{512\}}$ and $\theta_1^{\{512\}}$ to V_0 and V_1 respectively, and XORing with another two premixed words. The functions involve rotations in 64-bit blocks and bit-wise XOR operations.

3. **NONLINEARITY**: A nonlinear mapping is applied in bit-slice manner to the four MAP input word64s R, S, T, U, resulting in three MAP output words X, Y, Z. Each bit of the MAP output words is a nonlinear mapping of the corresponding bits in the corresponding word64s of the four MAP inputs.

4. **POST-MIXING**: The three MAP output word64s X, Y, Z are combined to form three feedback word64s AA, DD, GG. The post-mixing uses rotations in 64-bit blocks, the $SWAP8$ operation and addition modulo 2^{64}.

5. **FEEDBACK**: The value of $newA$ is the XOR of AA and Q, the value of $newD$ is the XOR of DD and C, and the value of $newG$ is the XOR of GG and F. The word64s C, F and Q are discarded, then:

   - word64 P is copied to word64 Q; word64 G is copied to word64 P; the new word64 $newG$ is copied to word64 G;
   - word64 E is copied to word64 F; word64 D is copied to word64 E; the new word64 $newD$ is copied to word64 D;
   - word64 B is copied to word64 C; word64 A is copied to word64 B; and the new word64 $newA$ is copied to word64 A.

7 HASH ALGORITHM DESCRIPTIONS

7.1 Description of CHI-256

CHI-256 may be used to hash a message, M, having a length of `len` bits, where $0 \leq \texttt{len} \leq 2^{64} - 1$.

7.1.1 CHI-256 Preprocessing

1. Pad the message M according to Section 5.1.1.

2. Parse the message into N 512-bit message blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, according to Section 5.2.1.

3. Set the initial hash value $H^{(0)}$, as specified in Section 5.3.2.

7.1.2 CHI-256 Hash Computation

After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed in order as follows.

For $i := 1$ to N, perform the following four steps.

1. Prepare the step inputs:

$$W_t := \begin{cases} M_t^{(i)}, & 0 \le t \le 7; \\ W_{t-8} \oplus \mu_0^{\{256\}}(W_{t-7}) \oplus \mu_1^{\{256\}}(W_{t-2}), & 8 \le t \le 39. \end{cases}$$

   Note: functions $\mu_0^{\{256\}}(\cdot)$ and $\mu_1^{\{256\}}(\cdot)$ are specified in Section 4.2.

2. Initialize the six state variables, A, B, C, D, E, F, with the $(i-1)$-st hash value:

$$\begin{aligned} A &:= H_0^{(i-1)}; \\ B &:= H_1^{(i-1)}; \\ C &:= H_2^{(i-1)}; \\ D &:= H_3^{(i-1)}; \\ E &:= H_4^{(i-1)}; \\ F &:= H_5^{(i-1)}. \end{aligned}$$

   If processing the last message block (that is, if $i == N$), then apply:

$$\begin{aligned} A &:= ROTR64^1(A); \\ B &:= ROTR64^1(B); \\ C &:= ROTR64^1(C); \\ D &:= ROTR64^1(D); \\ E &:= ROTR64^1(E); \\ F &:= ROTR64^1(F). \end{aligned}$$

3. For $t := 0$ to 19:

{

$$
\begin{aligned}
preR &:= DROTR32^{8,8}(B) \oplus DROTR32^{5,1}(SWAP32(D));\\
preS &:= A \oplus DROTR32^{18,17}(SWAP32(D));\\
preT &:= DROTR32^{7,26}(SWAP32(A)) \oplus DROTR32^{14,22}(D);\\
preU &:= DROTR32^{17,12}(A) \oplus DROTR32^{2,23}(SWAP32(E));\\
V_0 &:= W_{2t} \oplus K_{2t};\\
V_1 &:= W_{2t+1} \oplus K_{2t+1};\\
R &:= preR \oplus V_0;\\
S &:= preS \oplus V_1;\\
T &:= preT \oplus \theta_0^{\{256\}}(V_0);\\
U &:= preU \oplus \theta_1^{\{256\}}(V_1);\\
X &:= MAP_{t \pmod 3}(R,S,T,U);\\
Y &:= MAP_{1+t \pmod 3}(R,S,T,U);\\
Z &:= MAP_{2+t \pmod 3}(R,S,T,U);\\
XX &:= X;\\
YY &:= SWAP8(DROTR32^{5,5}(Y));\\
ZZ &:= SWAP32(Z);\\
AA &:= ROTR64^{16}(XX \boxplus YY);\\
DD &:= ROTR64^{48}(YY \boxplus ZZ);\\
newA &:= AA \oplus F;\\
newD &:= DD \oplus C;\\
F := E; \quad E := D; &\quad D := newD;\\
C := B; \quad B := A; &\quad A := newA;
\end{aligned}
$$

}

4. Compute the i-th intermediate hash value $H^{(i)}$:

$$
\begin{aligned}
H_0^{(i)} &:= ROTR64^1(H_0^{(i-1)}) \oplus A;\\
H_1^{(i)} &:= ROTR64^1(H_1^{(i-1)}) \oplus B;\\
H_2^{(i)} &:= ROTR64^1(H_2^{(i-1)}) \oplus C;\\
H_3^{(i)} &:= ROTR64^1(H_3^{(i-1)}) \oplus D;\\
H_4^{(i)} &:= ROTR64^1(H_4^{(i-1)}) \oplus E;\\
H_5^{(i)} &:= ROTR64^1(H_5^{(i-1)}) \oplus F.
\end{aligned}
$$

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is obtained by concatenating

$$H_0^{(N)} \| H_1^{(N)} \| H_3^{(N)} \| H_4^{(N)}.$$

Note that the word64s $H_2^{(N)}$ and $H_5^{(N)}$ are not included in the message digest.

7.1.3 Alternate description of CHI-256 Hash Computation

The step function of CHI-256 can also be described using operations on word32s. The word32 *MAP* functions used below are simply the word64 version of *MAP* restricted to 32-bits.

$$
\begin{aligned}
& \text{//Compute upper half of feedback} \\
preRu &:= ROTR32^{8}(Bu) \oplus ROTR32^{5}(Dl); \\
preSu &:= Au \oplus ROTR32^{18}(Dl); \\
preTu &:= ROTR32^{7}(Al) \oplus ROTR32^{14}(Du); \\
preUu &:= ROTR32^{17}(Au) \oplus ROTR32^{3}(El); \\
Vu_0 &:= Wu_{2t} \oplus Ku_{2t}; \\
Vu_1 &:= Wu_{2t+1} \oplus Ku_{2t+1}; \\
Ru &:= preRu \oplus Vu_0; \\
Su &:= preSu \oplus Vu_1; \\
Tu &:= preTu \oplus \theta_0^{\{256\}h}(Vu_0); \\
Uu &:= preUu \oplus \theta_1^{\{256\}h}(Vu_1); \\
Xu &:= MAP_{t \pmod 3}(Ru, Su, Tu, Uu); \\
Yu &:= MAP_{1+t \pmod 3}(Ru, Su, Tu, Uu); \\
Zu &:= MAP_{2+t \pmod 3}(Ru, Su, Tu, Uu); \\
XXu &:= Xu; \\
YYl &:= SWAP8(ROTR32^{5}(Yu)); \\
ZZl &:= Zu;
\end{aligned}
$$

$$
\begin{aligned}
& \text{//Compute lower half of feedback} \\
Rl &:= ROTR32^{8}(Bl) \oplus ROTR32^{1}(Du); \\
Sl &:= Al \oplus ROTR32^{17}(Du); \\
Tl &:= ROTR32^{26}(Au) \oplus ROTR32^{22}(Dl); \\
Ul &:= ROTR32^{12}(Al) \oplus ROTR32^{23}(Eu); \\
Vl_0 &:= Wl_{2t} \oplus Kl_{2t}; \\
Vl_1 &:= Wl_{2t+1} \oplus Kl_{2t+1}; \\
Rl &:= preRl \oplus Vl_0; \\
Sl &:= preSl \oplus Vl_1; \\
Tl &:= preTl \oplus \theta_0^{\{256\}h}(Vl_0); \\
Ul &:= preUl \oplus \theta_1^{\{256\}h}(Vl_1); \\
Xl &:= MAP_{t \pmod 3}(Rl, Sl, Tl, Ul); \\
Yl &:= MAP_{1+t \pmod 3}(Rl, Sl, Tl, Ul); \\
Zl &:= MAP_{2+t \pmod 3}(Rl, Sl, Tl, Ul); \\
XXl &:= Xl; \\
YYu &:= SWAP8(ROTR32^{5}(Yl)); \\
ZZu &:= Zl;
\end{aligned}
$$

$$
\begin{aligned}
& \text{//Concatenate to 64-bit words: add and rotate in 64-bit blocks} \\
XX &:= XXu \| XXl; \\
YY &:= YYu \| YYl; \\
ZZ &:= ZZu \| ZZl; \\
AA &:= ROTR64^{16}(XX \boxplus YY); \\
DD &:= ROTR64^{48}(YY \boxplus ZZ); \\
& \text{//Feedback into upper half} \\
newAu &:= AAu \oplus Fu; \\
newDu &:= DDu \oplus Cu; \\
Fu := Eu; \quad Eu := Du; \quad & Du := newDu; Cu := Bu; Bu := Au; Au := newAu; \\
& \text{//Feedback into lower half} \\
newAl &:= AAl \oplus Fl; \\
newDl &:= DDl \oplus Cl; \\
Fl := El; \quad El := Dl; \quad & Dl := newDl; Cl := Bl; Bl := Al; Al := newAl;
\end{aligned}
$$

This description provides some indication of how the designers imagine CHI-256 would be implemented on 32-bit processor.

7.2 Description of CHI-224

CHI-224 may be used to hash a message, M, having a length of `len` bits, where $0 \leq \texttt{len} \leq 2^{64} - 1$. The function is defined in the exact same manner as CHI-256 (Section 7.1), with the following

exceptions:

1. For CHI-224, the initial hash value $H^{(0)}$ is set to the values specified in Section 5.3.1.
2. The 224-bit message digest is obtained by concatenating

$$H_0^{(N)} \| H_1^{(N)} \| H_3^{(N)} \| MSB_{32}(H_4^{(N)}).$$

7.3 Description of CHI-512

CHI-512 may be used to hash a message, M, having a length of `len` bits, where $0 \leq \texttt{len} \leq 2^{128} - 1$.

7.3.1 CHI-512 Preprocessing

1. Pad the message M, according to Section 5.1.2.
2. Parse the message into N 1024-bit message blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, according to 5.2.2.
3. Set the initial hash value $H^{(0)}$, as specified in Section 5.3.4.

7.3.2 CHI-512 Hash Computation

After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed in order as follows.

For $i := 1$ to N, perform the following four steps.

1. Prepare the step inputs:

$$W_t := \begin{cases} M_t^{(i)}, & 0 \leq t \leq 15; \\ W_{t-16} \oplus \mu_0^{\{512\}}(W_{t-15}) \oplus W_{t-7} \oplus \mu_1^{\{512\}}(W_{t-2}), & 16 \leq t \leq 79. \end{cases}$$

   Note: The functions $\mu_0^{\{512\}}(\cdot)$ and $\mu_1^{\{512\}}(\cdot)$ are specified in Section 4.2.

2. Initialize the nine state variables, $A, B, C, D, E, F, G, P, Q$, with the $(i-1)$-st hash value:

$$\begin{aligned} A &:= H_0^{(i-1)}; \\ B &:= H_1^{(i-1)}; \\ C &:= H_2^{(i-1)}; \\ D &:= H_3^{(i-1)}; \\ E &:= H_4^{(i-1)}; \\ F &:= H_5^{(i-1)}. \\ G &:= H_6^{(i-1)}; \\ P &:= H_7^{(i-1)}; \\ Q &:= H_8^{(i-1)}. \end{aligned}$$

If processing the last message block (that is, if $i == N$), then apply:

$$\begin{aligned}
A &:= ROTR64^1(A);\\
B &:= ROTR64^1(B);\\
C &:= ROTR64^1(C);\\
D &:= ROTR64^1(D);\\
E &:= ROTR64^1(E);\\
F &:= ROTR64^1(F);\\
G &:= ROTR64^1(G);\\
P &:= ROTR64^1(P);\\
Q &:= ROTR64^1(Q).
\end{aligned}$$

3. For $t := 0$ to 39:
   {

$$
\begin{aligned}
preR &:= ROTR64^{11}(B) \oplus ROTR64^{8}(D) \oplus ROTR64^{13}(G);\\
preS &:= A \oplus ROTR64^{21}(D) \oplus ROTR64^{29}(G);\\
preT &:= ROTR64^{11}(A) \oplus ROTR64^{38}(D) \oplus P;\\
preU &:= ROTR64^{26}(A) \oplus ROTR64^{40}(E) \oplus ROTR64^{50}(G);\\
V_0 &:= W_{2t} \oplus K_{2t};\\
V_1 &:= W_{2t+1} \oplus K_{2t+1};\\
R &:= preR \oplus V_0;\\
S &:= preS \oplus V_1;\\
T &:= preT \oplus \theta_0^{\{512\}}(V_0);\\
U &:= preU \oplus \theta_1^{\{512\}}(V_1);\\
X &:= MAP_{t \pmod 3}(R,S,T,U);\\
Y &:= MAP_{1+t \pmod 3}(R,S,T,U);\\
Z &:= MAP_{2+t \pmod 3}(R,S,T,U);\\
XX &:= X;\\
YY &:= SWAP8(ROTR64^{31}(Y));\\
ZZ &:= SWAP32(Z);\\
AA &:= ROTR64^{16}(XX \boxplus YY);\\
DD &:= ROTR64^{48}(YY \boxplus ZZ);\\
GG &:= ZZ \boxplus (XX << 1);\\
newA &:= AA \oplus Q;\\
newD &:= DD \oplus C;\\
newG &:= GG \oplus F;\\
Q := P;\quad P := G;\quad G &:= newG;\\
F := E;\quad E := D;\quad D &:= newD;\\
C := B;\quad B := A;\quad A &:= newA;
\end{aligned}
$$

   }

4. Compute the i-th intermediate hash value $H^{(i)}$:

$$\begin{aligned}
H_0^{(i)} &:= ROTR64^1(H_0^{(i-1)}) \oplus A; \\
H_1^{(i)} &:= ROTR64^1(H_1^{(i-1)}) \oplus B; \\
H_2^{(i)} &:= ROTR64^1(H_2^{(i-1)}) \oplus C; \\
H_3^{(i)} &:= ROTR64^1(H_3^{(i-1)}) \oplus D; \\
H_4^{(i)} &:= ROTR64^1(H_4^{(i-1)}) \oplus E; \\
H_5^{(i)} &:= ROTR64^1(H_5^{(i-1)}) \oplus F; \\
H_6^{(i)} &:= ROTR64^1(H_6^{(i-1)}) \oplus G; \\
H_7^{(i)} &:= ROTR64^1(H_7^{(i-1)}) \oplus P; \\
H_8^{(i)} &:= ROTR64^1(H_8^{(i-1)}) \oplus Q.
\end{aligned}$$

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 512-bit message digest of the message, M, is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}.$$

Note that the word64 $H_8^{(N)}$ is not included in the message digest.

7.4 Description of CHI-384

CHI-384 may be used to hash a message, M, having a length of `len` bits, where $0 \leq \texttt{len} \leq 2^{128} - 1$. The function is defined in the exact same manner as CHI-512 (Section 7.3), with the following exceptions:

1. For CHI-384, the initial hash value $H^{(0)}$ is set to the values specified in Section 5.3.3.
2. The 384-bit message digest is obtained by truncating the final hash value, $H^{(N)}$, to its leftmost 384 bits:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)}.$$

Part II
Design Rationale

8 Introduction to the CHI Algorithms' Design Rationale

This part of the documentation explains the rationale behind the decisions the CHI team made when designing the CHI algorithms. The design rationale sections are organized as follows. The remainder of Section 8 describes the levels of magnification with which we examine the CHI algorithms: Input/Output Formatting; META structure; MACRO structure; and MICRO structure. Section 9 discusses design considerations: factors that need to be considered before beginning to design the algorithm. This is followed by a discussion of the design philosophies underpinning the CHI algorithm design in Section 11. Section 12 discusses the rationale behind the designs of the Input/Output Formatting and the META structure. The design rationale for the MACRO structure and MICRO structure are discussed in Section 13 and Section 14 respectively.

8.1 Four views of the CHI Algorithms

The structure of the CHI algorithms can be viewed or described with varying levels of magnification. The levels we consider are (in the order of increasing magnification): Input/Output Formatting (I/O Formatting); META structure; MACRO structure; and MICRO structure.

Input/Output (I/O) Formatting At this level, we consider how the input message is partitioned into message blocks for processing by the compression function; we also consider how the output hash states of the compression functions are mapped to input hash states of other compression functions, so that many message blocks can be compressed to a small output.

META structure At this level, we are looking at the overall structure of the compression function. We consider how the input hash state and message block are used to form the plaintext input and key input of an underlying block cipher; we also consider how the output hash state is formed from the input state, the message block and the output (ciphertext) of the underlying block cipher.

MACRO structure At this level, we are looking at the overall structure of the underlying block cipher. We consider how sub-blocks of the plaintext input and the key input interact to compute the output (ciphertext), and consider what kind of operations are used in these interactions. We consider the MACRO structure of the step function component and the message expansion component.

MICRO structure At this level, we are looking at the details of the underlying block cipher, and any remaining details of the compression function and Input/Output formatting. At this level we consider desired properties and requirements for the operations used.

9 Design Considerations

"NIST expects SHA-3 to have a security strength that is at least as good as the hash algorithms currently specified in FIPS 180-2, and that this security strength will be achieved with significantly improved efficiency." [57].

To achieve this goal, a design must take into account (a) those factors that influence how strong of the security must be, and (b) those factors that influence how the algorithm will be implemented. We call these the *design considerations*. The design considerations fall into a few categories:

Usage considerations Section 9.1: examining how the hash function be used.

Functionality criteria and design characteristics Section 9.2: NIST's evaluation criteria [57, Section 4], can be partitioned into *functionality criteria* (goals for the algorithm to achieve) and *design characteristics* (properties that are likely to result in fulfilling the criteria). We examine the SHA-3 call for submissions [57, Section 4] and the report [51] on the development of the Advanced Encryption Standard (AES) [54] to obtain a useful profile of the functionality criteria and design characteristics of a winning submission.

Security considerations Section 9.3.

Priorities for various Implementation Classes Section 9.4. We look at four classes of implementation, and consider (a) the priorities for what is desired of a hash function (eg. speed, software size, hardware size, energy efficiency), (b) what the typical platform (processor or hardware) looks like for this class, and (c) the priorities for how the hash function will be used (as discussed in Section 9.1).

Section 9.5 discusses the implications of these considerations, with Section 9.6 providing a summary.

In the following discussion, we examine how requirements and evaluation criteria have changed since the time when the SHA-2 algorithms were designed. In these discussions, "yesteryear" refers to the time of designing the SHA-2 algorithms.

9.1 Usage Considerations

In this section, we briefly look at the various ways in which a hash function can be used. NIST expects the SHA-3 algorithm to be suitable for: FIPS 186-2 [53], Digital Signature Standard; FIPS 198 [55], The Keyed-Hash Message Authentication Code (HMAC); SP 800-56A [58], Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography; and SP 800-90 [59], Recommendation for Random Number Generation Using Deterministic Random Bit Generators (DRBGs).

There are other uses of hash functions, including authentication, key derivation, entropy distillation, fingerprint/checksums and multi-cast key generation.

Of all these applications, digital signatures, authentication and integrity protection seem to be slightly higher in priority than the other applications. However, the other application are still considered high priority applications in many scenarios. In view of this, the CHI team decided not to design a submission with a particular application of hash functions in mind. All applications are treated as equally important.

9.2 Functionality Criteria and Design Characteristics

Early in the CHI project, the CHI team performed some background research in order to clarify what our team would use as requirements and evaluation criteria. This process incorporated the NIST SHA-3 requirements and evaluation criteria [57], a search of public literature, and an analysis of the report [51] on the development of the Advanced Encryption Standard (AES) [54].

The goals of our analysis of the report on the development of the AES [51] was to obtain a general picture of the properties of a winning submission. The analysis has two parts. The first part of the analysis considers how the final five competitors rate against a set of functionality criteria, and the second part of the analysis considers how the final five competitors rate against a set of

design characteristics. The CHI team gave ratings of Low, Medium, Medium-High and High. The outcome of the analysis of the criteria is shown in Table 9. The analysis shows that the winer (Rijndael) never got a low rating, and received a high rating for all functionality criteria excepting FPGA implementations and (interestingly) security margin.

Functionality Criteria			Algorithm				
			MARS	RC6	Rijndael	Serpent	Twofish
Security	Security Margin		High	Med	Med	High	High
	Amount of Analysis		Med	High	High	High	Low
	Side Channel Resistance		Low	High	High	Med	Med
SW Speed	Overall		Med	High	High	Low	Med
	Constrained Environments		Med	Med	High	Low	Med
HW Efficiency	FPGA		Low	Med	High	Med-High	High
	ASIC		Low	Low	High	High	Med
Small Size	Software		Low	Low	High	Med	Low
	Hardware	FPGA	Med	High	Med	Low	High
		ASIC	Low	Med	High	High	High

Table 9: Ratings for the final five competitors against the set of functionality criteria. The first column indicates a grouping of the criteria, the next columns indicate a specific criterion. The final five columns provide the ratings for each competitor, with "High" being the best rating and "Low" being the worst rating. In this table, "Med" is the abbreviation for "Medium", and "Med-High" is an abbreviation for "Medium-High"

Both the SHA-3 call for submissions [57] and AES report [51] indicated a preference for the algorithms with the following design characteristics:

Tunability There should be a tunable parameter such as the number of rounds or steps [57, Sections 2.B.1 and 4.C.i.a], [51, Section 3.9.1].

Implementation Tradeoffs This can mean a couple of things. One aspect of this characteristic is that the algorithm performs well for a variety of message sizes. Another aspect of this characteristic is the possibility of "... optimizing ... elements for particular environments" [51, Section 3.9.2], [57, Section 4.C].

Parallelism This is a measure of the degree to which operations in the algorithm can be performed concurrently. This affects the ability to do fast computation on medium and high-end implementations [51, Section 3.10], [57, Section 4.C.i.c].

Low Critical Path The path from input to output requiring the largest number of cycles, accounting for potential parallelism. This is a measure of the effect of parallelizing.

Simplicity The algorithm should be easy to remember and understand. [51, Section 3.2.4], [57, Section 4.C.ii].

Familiarity The cryptographic community is familiar and comfortable with the way the algorithm works. This is an aspect of simplicity.

Ease of Analysis The strength of the algorithm should be easy to determine. This is another aspect of simplicity.

Familiarity and ease of analysis were an explicit objectives of the Serpent designers: "... we limited ourselves to well understood mechanisms, so that we could rely on the existing experience of ... cryptanalysis" [17].

The outcome of the analysis of the design characteristics is shown in Table 10. The analysis shows that winner (Rijndael) rated highly for all these design characteristics.

Design Characteristics	Algorithm				
	MARS	RC6	Rijndael	Serpent	Twofish
Tunability	Med	High	High	Med	Med
Implementation Tradeoffs	Med	Med	High	High	High
Parallelism	Low	Low	High	Low	Med
Low Critical Path	Med	Med	High	Low	Med
Simplicity	Low	Med	High	High	Low
Familiarity	Low	Med	High	High	Low
Ease of Analysis	Low	Low	High	High	Low

Table 10: Ratings for final five competitors against the set of design characteristics. The first column indicates the design characteristic. The final five columns provide the ratings for each competitor, with "High" being the best rating and "Low" being the worst rating. In this table, "Med" is the abbreviation for "Medium", and "Med-High" is an abbreviation for "Medium-High"

The final outcome of the analysis was that, in order to win the competition, our submission must rate highly against almost all of the functional criteria, and rate highly against all of the design characteristics.

9.3 Security Considerations

The discussion on security considerations begins by looking at the state of the art in hash function cryptologic research and side channel attacks. The security requirements in the NIST SHA-3 Call for Submissions [57] are then briefly discussed, followed by a discussion of the CHI team's cryptographic background.

9.3.1 Cryptologic Considerations

Cryptologic research into hash functions have seen some advances in recent years.

- Cryptographic algorithms are more widespread, and found in a wide range of devices. Many user devices are protecting valuable data such as financial information.
- Side channel attacks (see Section 9.3.2) are considered a realistic threat. In particular, cache-based attacks [18] have caused designers to shy away from lookup-table operations.
- Differential-based collision attacks appeared in the public arena. Wang *et al*'s attacks on MD5 [67, 69] and SHA-1 [68, 68] provided new tools for analyzing hash functions. The application to other hash functions has proven the power of this tool. These attacks will be discussed in more detail in Sections 14.1 and 17.

- Randomized hash functions are gaining popularity.
- There have been new attacks on the Merkle-Dåmgard construction, exploiting properties that arise when an attacker can exceed the birthday bound. These attacks include multi-collision attacks [35], Long-message attacks [39] and Herding attacks [37].

However, the field of hash-function cryptanalysis is still relatively immature in the public arena. While the introduction of Wang *et al* attacks have helped analyze hash functions, there are few other tools at the cryptanalysts disposal. Furthermore, collision attacks apply only to the use of hash functions to obtain a key-less message digest. There has been little advance in the analysis of other uses of hash functions such as for message integrity or for key generation (for example, using HMAC).

9.3.2 Side-Channel Attacks

Side channel attacks find secret information in the hash computation by analyzing physical properties that depend on the secret information. When processing a message block, the secret information could be either data already compressed into the input hash state, or data in the message block. Most devices have at least one application of hash functions that involves using secret information, so the impact of side-channel attacks always needs to be considered.

Basic Timing Attacks. Some timing attacks are very coarse grained, and the attacker can infer information by examining only the total time of a hash computation. Other timing attacks are very fine grained, and the attacker infers information by examining a portion of the hash computation such as one step or even a single operation. These attacks typically exploit data-dependent conditional statements or instructions that can take a variable amount of time (such as some integer multiplication implementations). Consequently, the CHI team decided to avoid data-dependent conditional statements or instructions that can take a variable amount of time.

Cache-Timing Attacks. These attacks are relatively new, and greatly feared. Cache-Timing attacks can be used when implementations retrieve information from large look-up tables using an index containing secret information. The cache-timing attack provides the attacker with information about the index. The attacks only apply when the attacker has the ability to run a program on the processor that attempts to over-write a portion of the look-up table while the table resides in cache. Since processors tend to allocate cache one line at a time, these attacks only apply when a table requires more than one line of cache. Consequently, the CHI team decided to avoid large look-up tables for which the index may depend on the data.

Power Analysis Attacks. Power analysis attacks look at the fluctuations in the power supply for the processor or dedicated hardware while the hash is being computed. The power fluctuations are caused by the changes in transistors during the processing. Often the fluctuations can only be detected by observing a large number of computations. This field of research is a arms race: as new techniques are developed for power analysis; new technologies are being developed to make implementations resistant to power analysis; and new techniques are developed to circumvent the new technology. It seems if the attacker can monitor power usage for a device, then it is very difficult to protect that device against power analysis attacks, . The best defence is to prevent the attacker having the ability to track the power usage.

At the algorithm level, there are some basic countermeasures that can make power analysis more difficult, but few countermeasures can make an algorithm completely resistant to power analysis. The best countermeasure is to ensure that there are no conditional computations that depend on values that might be considered secret (such as hash state values or the message).

9.3.3 NIST's Security Requirements and Evaluation Criteria

The security requirements provided by NIST did not introduce many new requirements over previously typically required for hash functions. The SHA-3 algorithm is expected to be suitable for a set of NIST-standardized applications: we have discussed these applications in Section 9.1. Other requirements of note include the addition of requirements for randomized hash functions, and requirements addressing multi-collision attacks. NIST also mentioned some criteria that would be considered in judging candidate algorithms: "... the quality of the security arguments/proofs, the clarity of the documentation of the algorithm, the quality of the analysis on the algorithm performed by the submitters, the simplicity of the algorithm, and the confidence of NIST and the cryptographic community in the algorithms long-term security ... " [57].

The simplicity of the algorithm is a very important factor. If an algorithm's description is too complex to understand, or if the algorithm is to complex to analyse, then the design will receive little attention from cryptanalysts. A lack of cryptanalysis reduces the cryptologic community's faith in an algorithm. The CHI team has tried to keep the design in a form that will be familiar (and thus simpler) to analyse.

9.3.4 The CHI Team's Cryptographic Background

The CHI team has many years combined experience in the design and analysis of stream ciphers and (to a lesser extent) block ciphers. In additional to this experience, the design leads (Cameron McDonald and Phil Hawkes) have spent significant time using the approach of Wang *et al* in analyzing SHA-1 and SHA-2. The CHI team decided on a conservative approach to their design that would leverage this experience.

Rather than creating a radical design that required completely new analysis techniques, the team targeted a design that could be examined using a similar technique to SHA-1 and SHA-2, but which would be different enough to "... a possibly successful attack on the SHA-2 hash functions is unlikely to be applicable to SHA-3" [57].

9.4 Priorities of Various Implementation Classes

In analyzing the implementation considerations, we found it useful to consider various classes of implementation.

High-End Processors : Processors in laptops, desktop and servers. Typical examples are AMD [1] and Intel[2] 64-bit multi-core processors.

Mid-Range Processors : Processors in mobile phones and other similarly sized devices. Typical examples are ARM[3] processors.

[1] AMD is a trademark of Advanced Micro Devices, Inc. [1]
[2] Intel is a trademark of Intel Corporation [7]
[3] ARM is z trademark of ARM Limited [2]

Low-End Processors : Used in a variety of devices. Typical examples are 8-bit microcontrollers.

Hardware implementations : Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs). Typical examples are cryptographic accelerators / co-processors.

Each of these classes has its own set of characteristics. We consider (a) the priorities for what is desired of a hash function (eg. speed, software size, hardware size, energy efficiency), (b) what the typical platform (processor or hardware) looks like for this class, and (c) priorities for how the hash function will be used (as discussed in Section 9.1).

9.4.1 Priorities of High-End Processors

Implementation Priorities. In this class of implementation, speed is the highest priority, although energy efficiency is also important for laptop processors.

Typical Platforms. The dominant software architectures are 64-bit processors; it is unlikely that the register size will increase past 64-bits in the foreseeable future. These platforms tend to have multi-core processors, with the number of cores currently doubling about every 3 years. Each core typically has access to 16 registers. These processors typically include Single Instruction Multiple Data (SIMD) co-processors. The SIMD co-processors typically have 128-bit registers. In the future it seems likely that High-End processors may include either 256-bit SIMD co-processors or dual 128-bit SIMD co-processors.

Priorities for hash function usage All of the uses in Section 9.1 are a priority to this class of implementations. All hash sizes are likely to be a priority for this class of implementations.

Susceptibility to Side Channel Attacks. Many of the fastest processors tend to be tightly controlled, with small likelihood of malicious software being run on the processor. Consequently, cache-timing attacks and basic timing attacks (unless very coarse grained) are unlikely to be a concern. This tight control also limits the opportunity for power analysis. However, the remainder of this class is dominated by desktops, laptops and servers running all manner of applications, so cache-timing attacks are a big concern. Power analysis attacks are also a concern.

9.4.2 Priorities of Mid-Range Processors

Implementation Priorities. In this class of implementation, energy efficiency and speed are the highest priorities. These devices are required to support a wide variety of applications.

Typical Platforms. The dominant software architectures are high performance, energy-efficient 32-bit processors. It is unclear if (in the near future) there will be 64-bit processors in this part of the market. Multi-core processors are likely to become commonplace in this market. ARM processor cores typically have access to 16 registers, while 32-bit x86 processor cores only have access to 8 registers. SIMD co-processors are likely to become commonplace in this market. Initially these co-processor would have only 64-bit registers, but the future may see dual SIMD co-processors or even 128-bit and 256-bit SIMD co-processors. Cryptographic accelerators (see Section 9.4.4) are common.

Priorities for hash function usage Not all the uses in Section 9.1 are currently a priority to this class of implementations, but all these uses are likely to quickly become a priority. Smaller hash sizes (256-bits and less) are likely to be the priority for this class of implementations.

Susceptibility to Side Channel Attacks. There is slightly more control over the applications run on these processors (compared to high end processors) so cache-timing attacks are slightly more difficult. Many of these devices are battery powered, making it difficult to do power analysis while the device is in the user's possession. Other devices are susceptible to power analysis. Of particular concern: the size of the device then makes it easier to steal a device for analysis. It seems wise to assume that these devices could be susceptible to all forms of side-channel attacks.

9.4.3 Priorities of Low-End Processors

Implementation Priorities. The priority for hash function implementations in this class are: small software program size, small RAM requirements, and (in many cases) energy efficiency.

Typical Platforms.. In the 70's and 80's, this class was dominated by 8-bit microcontrollers. These 8-bit microcontrollers are still wide-spread in use, with a variety of new processors being introduced in recent years. However, there is a trend towards larger word sizes. A variety of 16-bit and 32-bit microcontrollers are now available. A survey of some microcontrollers can be found in Section 25.1.

An interesting change is that the ARM7 and ARM9 processors[4], mid-range processors of yesteryear, are now affordable enough to be in this price range, as are some 32-bit x86 processors. In 5 years, ARM11 processors may well be in this price range. Trends in the architectures of the smaller microcontrollers include increasing number of registers, and pipelined architectures. Most processors in this range have a single execution unit, although some DSP microcontrollers have 2 concurrent execution units. Some of these processors have hardware cryptographic accelerators (see Section 9.4.4). Processors in this class often have limited Read-Only Memory (ROM) for storing the software, and limited RAM for storing variables during processing. The amount of RAM and ROM available continues to increase.

Priorities for Hash Function Usage Authentication, integrity protection, key derivation are most likely to be the higher priority uses of hash functions discussed in Section 9.1. It is interesting that all these applications require a key or some other secret data. However as more powerful processors enter this class, and the number of applications increases, other uses are likely to increase in priority. Smaller hash sizes (256-bits and less) are almost certain to be the priority for this class of implementations.

Susceptibility to Side Channel Attacks. In most cases, it is not possible for users to load applications other than the legitimate program loaded by the manufacturer or distributor. This makes the devices resistant to cache-attacks. However, the size of the device then makes it easier to steal a device for power analysis. It seems wise to assume that these devices could be susceptible to most side-channel attacks, but cache-timing attacks are somewhat less of a priority.

9.4.4 Priorities of Hardware implementations

Implementation Priorities. There are three main classes of hardware implementation: High-speed hardware implementations built purely for speed; cryptographic accelerators for mobile devices, designed for energy efficiency and size; and cryptographic accelerators for dedicated authentication devices.

[4]ARM7, ARM9 and ARM11 are trademarks of ARM Limited [2]

Typical Platforms. The two main platforms are Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). FPGAs have not gained as much popularity as imagined a few years ago. ASICs, however, have remained popular. Many user-level devices have cryptographic accelerators available to the processor; energy efficiency is often a priority for such co-processors since these devices tend to be mobile.

Priorities for hash function usage Regarding the types of use of hash functions discussed in Section 9.1: all these uses are a priority to high-speed hardware implementations; not all these uses are currently a priority to cryptographic accelerators for mobile devices, but all these uses are likely to quickly become a priority; while authentication, integrity protection, key derivation are most likely to be a priority for small authentication devices. All hash sizes are almost certain to be the priority for high-speed hardware implementations. Smaller hash sizes (256-bits and less) are likely to be the priority for the cryptographic accelerators for mobile devices and dedicated authentication devices.

Susceptibility to Side Channel Attacks. Cache-attacks do not impact this class of implementation. High-speed hardware implementations and cryptographic accelerators for mobile devices can be susceptible to most other forms of side channel attack. Some dedicated authentication devices are battery powered and tamper resistant, and most side channel attacks are less feasible on these devices. On the other hand other dedicated authentication deices are externally powered and power analysis can be easily carried out; however most devices account for this and incorporate some addition countermeasures against power analysis.

9.4.5 Summary of Priorities of Various Implementation Classes

Some interesting points of the analysis include:

- *Implementation Priorities.* For high-end processors and many hardware implementations, speed is the priority. For mid-range processors and some high end processors (such as for laptops), both speed and energy efficiency are a priority. For low-end processors, energy efficiency and memory usage are the priorities. It is interesting to note that energy efficiency has become a priority.

- *Typical Platforms.* High-end processors are now dominated by 64-bit processors, instead of 32-bit x86 processors. Mid-range processors are high performance 32-bit processors. Low-range processors may have 8, 16 or 32-bit processors, but typically have limited memory resources. Processers tend to have more registers and more memory. SIMD co-processors and multi-core processors have become common in high-end processors, and will become common in mid-range processors. ASICs dominate the hardware class.

- *Priorities for hash function usage.* For high-end processors, mid-range processors and hardware implementations, all uses of hash functions in Section 9.1 are high priorities. For Low-range processors, hash functions uses are those typically used with secret material. High-end processors and hardware implementations expect high speed for all hash size, but mid-range and low-end processors expect high speed for only the smaller hash sizes (256-bits and less).

- *Susceptibility to Attacks.* For implementations requiring high speed (such as high-speed routers using hash functions in either software or hardware implementations), the ability

to run software on the device is typically tightly controlled. Physical access to the device is also typically tightly controlled. Attackers therefore have limited ability to run side-channel attacks such as cache-timing attacks or power analysis. We could conclude that implementations with the highest speed requirements do not need to account for side channel attacks.

Cache timing attacks are a danger to devices for which users can load applications (such as the remaining high-end processors and many mid-range processors). Mobile devices (typically using mid-range implementations, low-range implementations and/or cryptographic hardware accelerators) seem more susceptible to side-channel attacks of one form or the other (either cache-timing attacks or power analysis attacks). Implementations of SHA-3 for these device should account for side channel attacks. The exception are tamper-proof devices, which should be resistant to most side channel attacks.

This distinction was considered when choosing the operations for the CHI algorithms. This is discussed in detail in Section 10.8.

There will be exceptions to these observations, but they provide a useful model.

9.5 Design Considerations: Discussion

Conservative vs Radical Design. The discussion Section 9.2 indicated the functionality criteria and design characteristics that underpin a good submission. The submission needs to perform well against the list of the design characteristics (Tunability, Implementation Tradeoffs, Parallelism, Low Critical Path, Simplicity, Familiarity and Ease of Analysis).

Register Size. The high-end processors are the software class for which speed was highest priority. This implementation class is dominated by 64-bit architectures, so it makes sense to target 64-bit processors. However, it is difficult to match the speed of SHA-224 and SHA-256 on 32-bit processors when using operations targeting 64-bit architectures like rotations in 64-bit blocks. For CHI-224 and CHI-256, the CHI team decide to compromise, with some such operations favoring 32-bit processors and other such operations favoring 64-bit processors. In this way, neither 32-bit processors nor 64-bit processors would be disadvantaged significantly. The remaining operations suit all register sizes equally well.

Number of Registers. Most processor have access to more general purpose registers than in yesteryear. This means that algorithms can allow more temporary variables, without incurring additional cost of copying data to and from registers.

Memory. Most processor have access to more RAM and ROM that in yesteryear. However, many implementation still have limited RAM and ROM. The CHI team has attempted to limit the required amounts of RAM and ROM.

SIMD Co-processors. The CHI team considered using SIMD operations. Due to the latency of SIMD operations, these operations do not allow fast interactions between variables. Hence, it is difficult to leverage an advantage from SIMD operations during the step function, where many interactions are required. The message expansion, which is typically simpler and requires fewer interactions, might be able to exploit SIMD operations. In view of this, CHI team kept SIMD operations in mind when designing the message expansion; but SIMD operations had no significant impact on the design of the step function.

Parallel Message-Block Processing. There are some devices, such as a high-end routers, that will need to be quick at hashing large amounts of data. These devices will tend to be multi-core processors, possibly with cryptographic accelerators implemented in hardware. Some hash algorithms can process a single large message quickly by processing portions of the message in parallel. Multi-core processors and cryptographic accelerator hardware with multiple implementations can take advantage of such a design. However, the messages processed by such devices tend to be smaller packets, so these devices need hash algorithms that can process many such messages quickly. In these cases, multi-core processors and cryptographic accelerator hardware can typically be employed more efficiently by processing multiple messages in parallel, rather than using multi-threading to speed up the processing of a single message. In view of this, the CHI team decided not to include parallel message-block processing or other structures that could exploit multi-thread capabilities of multi-core processors or multiple hardware implementations when processing a single message.

Circuitry Re-Use. In hardware implementations, gate count and area should be minimized. For hardware implementations supporting all hash output sizes, it is permissible (and advantageous) for a large percentage of the circuitry to be common to all the algorithms. This will greatly reduce gate count and area for such implementations. At the algorithm level, this is achieved by maximizing similarities between the algorithms.

9.6 Design Considerations: Summary

- The design should not target a particular application of hash functions.
- To be a good submission, the algorithm must rate highly against all the design characteristics in section 9.2.
- Large look-up tables should be avoided in implementations that need to resist cache-timing attacks.
- The algorithms should favor 64-bit processors, although CHI-224 and CHI-256 need to make some concessions for 32-bit processors to remain competitive with SHA-224 and SHA-256.
- The CHI algorithms may allow more temporary variables than the SHA-2 algorithms.
- Speed and then energy efficiency are the two highest priorities of a hash function.
- The CHI algorithms should limit the required amounts of RAM and ROM, so as not to disadvantage low-end processors.
- The step function is not intended to exploit SIMD operations. However, SIMD operations were kept in mind when designing the message expansion.
- The design is not intended to exploit multi-core processors or multiple hardware implementations when processing a single message.
- Maximize similarities between the algorithms to allow circuitry re-use for hardware implementations supporting all hash output sizes, thus reducing gate count and area.
- Suitability to ASICs should be a higher priority than suitability to FPGA implementations.

10 Choice of Operations

In this section, we attempt to explain the rationale behind our decision to choose some operations and reject other operations. We discuss the security advantages and weaknesses of using the operations. One weakness that interested us is whether the operations preserve bit adjacency of information. That is: does the information in adjacent bits of the input only affect adjacent bits in the outputs. We also discuss the relative cost in terms of software clocks or hardware gate count and area. The cost is given in relative terms of "almost free", "cheap", "modest", and "expensive".

10.1 Confusion and Diffusion

Claude Shannon [64] defined confusion and diffusion as two properties required for frustrating a statistical analysis of a cipher. More recently, with cryptographic algorithms being designed from smaller components, these names are also used to denote properties of components of cryptographic algorithms. Components that provide confusion make the relationship between the inputs and outputs of that component as "complex and as involved as possible"[72]. Components that provide diffusion dissipate the information from inputs throughout the outputs of that component .

Typical designs have multiple iterations of a round, where each round contains a group of confusion-oriented components and a group of diffusion-oriented components. The role of the diffusion-oriented components is to take the confusion effects from the confusion-oriented components, and dissipate this effect amongst the inputs of the next round of confusion-oriented components. After several iterations, this structure can result in complex relationships between the input to the first round and the outputs of the last round.

The CHI algorithms are built using this philosophy: some operations have been chosen for their ability to provide confusion, and other operations have been chosen for their ability to provide diffusion.

10.2 Bit-wise Logic Operations

All processors support the bit-wise logic operations of AND, OR, NOT and XOR; and these operations can be implemented very cheaply in hardware. Combinations of these operations have been used effectively in cryptography to produce bit-sliced functions: MD5 [63], SHA-1, the SHA-2 algorithms, Serpent [17] and PRESENT [24] use these functions. The functions in Serpent and PRESENT were the original inspiration for the bit-wise logic function *MAP*. Such functions could provide complex interactions between the inputs, while being efficient for all implementation classes.

Summary:

Security advantage: : These operations can be combined to provide complex interactions (confusion).

Security Weaknesses: There is no interaction between distinct bit positions. Preserves bit adjacency of information.

Cost: Very cheap on all processors; and very cheap in hardware. Energy efficient.

Conclusion: Logic operations may be used liberally.

10.3 Addition

This operation is defined in Section 3.3, point 1.

Combinations of addition operations and logic operations can provide moderately complex interactions, which is why these operations have been used in the previous hash function designs. However, additions do not diffuse information to less significant bit positions. Additions do diffuse information to more significant bit positions, but the diffusion can be slow. The propagation of information to more significant bit positions has other implications. For example, the LSB of the output can only depend on the LSB of this inputs: the LSB of the output is the XOR sum of the LSB of the inputs. There are also undesirable differential properties related to the MSB, but is difficult to explain these effects without a suitable introduction to differential attacks, so the discussion of these effects is postponed to Section 17.

Integer addition is fast on all processors. Most processors include an "add with carry" operation in which allows addition of values of size greater than the register size. This means that addition of 64-bit values (as defined in Section 3.1, point 4) can be implemented on processors with 8-bit, 16-bit or 32-bit registers. The number of instructions to add 64-bit values on a smaller register is simply 64 divided by the register size. This is identical to the number of instructions required to perform 64-bit logic operations on smaller registers. The only disadvantage of 64-bit addition is that hardware implementations tend to be either large (use a large number of gates) or slow (the latency is high). The best compromise is to implement the addition as two sequential 32-bit addition operations (with a carry from the first addition operation).

An alternative is to replace the 64-bit addition by two parallel 32-bit addition operations. This would incur no addition penalty on 8-bit, 16-bit and 32-bit processors, and would benefit 32-bit processors with multiple execution units. This would also benefit hardware implementations, for which there exist small, fast circuits. The disadvantage of parallel 32-bit addition operations is that current 64-bit processors would require two instructions to perform the operations (one instruction for each addition).

The CHI team debated the merits of parallel 32-bit addition and 64-bit addition at length. It seems that either option would be fine. In the end, 64-bit addition operations were chosen.

Summary:

Security advantage: These operations provide moderately complex interactions between two or more word64s.

Security Weaknesses: Information is never diffused to less significant bit positions. Most of the time, information is only diffused to nearby (more significant) bit positions. The LSB has undesirable linear properties. The MSB has undesirable differential properties. Consecutive addition operations can combine to cancel out or form a single addition operation. Preserves bit adjacency of information.

Cost: Very cheap on all processors; and modest in hardware. Moderately energy efficient.

Conclusion: Addition operation may be used sparingly, so as to limit the hardware cost.

10.4 Rotation Operations $\boldsymbol{ROTR}32^n$ and $\boldsymbol{ROTR}64^n$

The $ROTR32^n$ and $ROTR64^n$ operations are defined in Section 3.2 and Section 3.3 point 3 respectively. These operations are used to move bit information from one bit position to another bit

position, thus providing diffusion.

The main security weakness of the $ROTR32^n$ operation is that consecutive $ROTR32^n$ operations can combine to cancel out or form a single $ROTR32^n$ operation. The $ROTR64^n$ operations shares the same property. Thus, including more rotation operations does not necessarily increase the diffusion. Rotations also preserves bit adjacency of information.

The cost of rotation operations depends on the register size and the size of the block which is being rotated. Processors with register size of 8-bits and 16-bits require a combination of shift operations and bit-wise OR operations to perform a $ROTR32^n$. This makes the $ROTR32^n$ operation of medium cost on such processors. Most 32-bit processors include an instruction for rotating a 32-bit block any number of bit positions, making this operation cheap on 32-bit processors. Some ARM processors[5] can combine a addition or logic operations with a 32-bit rotate operation, allowing the rotation to be incorporated into other parts of the hash computation. Current 64-bit processors do not allow provide double $ROTR32^n$ instructions, so 64-bit processors can only rotate 32-bits per instruction.

Most 64-bit processors include an instruction for rotating a 64-bit block any number of bit positions, making the $ROTR64^n$ operation cheap on 64-bit processors. Processors with smaller register size require a combination of shift operations and bit-wise OR operations to perform a $ROTR64^n$. Some ARM processors can combine a addition or logic operations with a shift operation, and this may allow the $ROTR64^n$ operation to be performed faster.

On 64-bit processors, using two parallel $ROTR32^n$ operations is less efficient than the $ROTR64^n$, since the former requires two instructions and the latter requires only one. This makes the $ROTR32^n$ operations less preferable on 64-bit processors. For similar reasons the $ROTR64^n$ operations are less preferable on 32-bit processors. Either operation degrades the performance on a large set of processors. Since 64-bit processors have the highest priority for speed, the CHI team choose to use the $ROTR64^n$ operations where possible. The CHI team allowed some parallel $ROTR32^n$ operations in CHI-224 and CHI-256, as otherwise the performance on 32-bit machines was not competitive with SHA-224 and SHA-256.

Since rotation operations only involve moving bits, both of these operations are almost free in hardware, requiring only wiring connections to be placed appropriately.

Summary:

Security advantage: Diffusion

Security weaknesses: Consecutive $ROTR32^n$ operations can combine to cancel out or form a single $ROTR32^n$ operation. The same applies for $ROTR64^n$ operations. Preserves bit adjacency of information.

Cost of $ROTR32^n$ operation: Cheap on most 32-bit processors (especially some ARM processors); modest on most other processors; and almost free in hardware. Energy efficient.

Cost of $ROTR64^n$ operation: Cheap on 64-bit processors; modest on most other processors (cheaper on some ARM processors); and almost free in hardware. Energy efficient.

Conclusion: The $ROTR64^n$ operation may be used sparingly. The $ROTR32^n$ operation may be used sparingly in CHI-224 and CHI-256.

[5] ARM is trademarks of its owner [2]

10.5 Shift Operation $SHR64^n$

The shift operation $SHR64^n$ is defined in Section 3.3, point 2. These operations are used to move bit information from one bit position to another bit position, thus providing diffusion. The operations share many characteristics with rotation operations $ROTR64^n$ discussed in Section 10.4 (so the discussion is not copied here). However, shift operations differ in that they discard information from the input and cannot commute with rotation operations.

The summary is the same as for $ROTR64^n$, so it isn't shown here.

10.6 *SWAP*8 and *SWAP*32

The $SWAP8$ and $SWAP32$ operations are defined in Section 3.3, point 4 and 5 respectively. The $SWAP32$ operation helps diffuse bits like a rotation operation, but does not contribute anything additional to the security. The $SWAP8$ operation also helps diffuse bits, but it serves another purpose in the CHI algorithms. The main security advantage of the $SWAP8$ operation is that it destroys the adjacency of some bits. This is important, since all other operations preserve the adjacency of almost all bits.

Both of the $SWAP8$ and $SWAP32$ operation, share the weakness of being self-inverses:

$$SWAP8(SWAP8(x)) = x;\ SWAP32(SWAP32(x)) = x.$$

The two operations also commute:

$$SWAP8(SWAP32(x)) = SWAP32(SWAP8(x)).$$

The $SWAP32$ operation can be absorbed into preceding or following rotation operations.

The $SWAP8$ operations can be implemented on 8-bit processors using appropriate addressing, which is almost free. Most other processors, have a dedicated instruction that can perform the $SWAP8$ operations cheaply. Some 32-bit processors do not have a dedicated instruction, and a $SWAP8$ operation can be expensive.

The $SWAP32$ operations can be implemented on 8-bit, 16-bit and 32-bit processors using appropriate addressing, which is almost free. The $SWAP32$ operations can be implemented cheaply on 64-bit processors.

Since $SWAP8$ and $SWAP32$ operations only involve moving bits, both of these operations are almost free in hardware, requiring only wiring connections to be placed appropriately.

Summary:

Security advantage: Diffusion. The $SWAP8$ operation primarily destroys the adjacency of some bits.

Security weaknesses: The operations are self-inverses and commute with each other. The $SWAP32$ operation can be absorbed into $ROTR64^n$.

Cost of *SWAP*8 operation: Almost free on 8-bit processors; cheap on most other processors; and almost free in hardware.

Cost of *SWAP*32 operation: Almost free on 8-bit, 16-bit, and 32-bit processors; cheap on 64-bit processors; and almost free in hardware. Energy efficient.

Conclusion: $SWAP8$ should be used sparingly, while $SWAP32$ operations may be used liberally.

10.7 Integer Multiplication

Integer multiplication operations can be quite fast on modern large processors, but are slow on smaller processors. Integer multiplication operations are very costly in hardware, both in terms of the number of gates and the latency.

Integer multiplication operations can provide complex interactions between bits positions of multiple inputs, however it is unclear if the complexity justifies the cost. Integer multiplication operations are also more likely to allow timing attacks.

Summary:

Security advantage: Complex interactions between bits positions of multiple inputs.

Security Weaknesses: More likely to allow timing attacks.

Cost: Incurs medium to high cost in software; and is very expensive in hardware. Not energy efficient.

Conclusion: The CHI team chose not to use integer multiplication operations.

10.8 Linear Look-up Tables

At one point the CHI team considered using complex linear functions that could be implemented using look-up tables. However, we need to consider that if the look-up tables are sufficiently large, then the implementation becomes susceptible to cache-timing attacks. The following discussion shows how a linear function could avoid this danger (note that this discussion assumes that the function is linear with respect to the XOR operation, but the same principles can be applied to a function that is linear with respect to modular addition operation).

Consider a linear function f with an input x of 8-bits. This could be naively implemented by having a 256-entry table `LinearTable[]`, with the function implemented as

$$f(x) \;:=\; \texttt{LinearTable}[x].$$

Due to the large number of entries in the table, this table would typically be susceptible to cache-timing attacks.

Suppose the input is be partition into two parts of four bits each x' and x''. Construct two tables `LinearTableUp[]` and `LinearTableDown[]` defined by

$$\begin{aligned} \texttt{LinearTableUp}[x'] \;&:=\; \texttt{LinearTable}[x' \| 0]; \\ \texttt{LinearTableDown}[x''] \;&:=\; \texttt{LinearTable}[0 \| x'']. \end{aligned}$$

These two tables contain only 16 entries each. Since the function is linear, the function f can be implemented as

$$f(x' \| x'') \;:=\; \texttt{LinearTableUp}[x'] \oplus \texttt{LinearTableDown}[x''].$$

This implementation uses tables with only 16 entries each. Depending on the size of the output, these tables may be small enough to resist cache-timing attacks. If these tables are not small enough, these 2 tables can be subdivided into a total of four tables or even eight tables. Eventually, the tables will be small enough to resist cache-timing attacks.

The disadvantage of using smaller tables is that the implementation becomes slower. However, we noted in Section 9.4.5 that high-speed implementers do not always need to resist cache-timing attacks. High-speed implementations could use a single look-up table (and remain fast), while the remaining implementations (that do need to resist cache-timing attacks) can use multiple look-up tables (for a slight penalty in speed).

This proposition seemed attractive at first, but there is an additional danger. Software implementers may not always be able to correctly identify the threat model and choose appropriately between the implementation options. In these circumstances, the software implementer may choose the faster (and thus insecure) implementation. This seems a high risk, so the CHI team decided not to use such functions.

Summary:

Security advantage: Complex interactions between bits.

Security Weaknesses: Fast implementations are susceptible to cache timing attacks.

Cost: Incurs medium to high cost in software; and is cheap in hardware.

Conclusion: The CHI team chose not to use linear functions based on table-lookups.

10.9 Summary of the CHI Team's Choice of Operations

The CHI team chose to use the following operations:

- 64-bit logic operations: these may be used liberally;
- 64-bit addition: these should be used sparingly;
- 64-bit shift operations: these should be used sparingly;
- Rotation in 64-bit blocks: these should be used sparingly;
- Rotation in 32-bit blocks: permitted in CHI-224 and CHI-256, but these should be used sparingly;
- $SWAP8$: these should be used sparingly; and
- $SWAP32$: these may be used liberally.

This choice of operations is not very radical: we imagine that most teams designing SHA-3 submission chose a similar set of operations.

11 Design Philosophy

Section 9 discussed the background upon which the CHI team chose their design philosophy, which is discussed the current section. We begin by examining the apparent design philosophy for the SHA-2 Algorithms, along with a discussion of the positive and negative implications of those algorithms. We then explain the CHI design philosophy.

11.1 Inferred Design Philosophy for SHA-2 Algorithms

Other SHA-3 submission designers will no doubt agree that the process of designing a hash function revealed much about the strengths of the SHA-2 algorithms.

The SHA-2 algorithms appear to use the following "general" design philosophy:

- Use the standard Merkle-Dåmgard construction [50, 28].
- Use the standard Davies-Meyer construction [46].
- Let the step function have simple functions, with the following implications
  - Small amount of non-linearity per step.
  - Each bit difference induces conditions on only a few bits.
  - Simpler functions use few temporary variables.
- Have many steps to build up the effect of simple functions.
- Focus on software implementations.
- Use the combination of bit-sliced non-linear functions, addition (which has carry effects for diffusion to nearby bit positions), with rotation for diffusing to distant bit positions and
- Focus on diffusing information from the variable you generated.

This is an admirable approach, and SHA-2 uses this philosophy to achieve an excellent balance of speed versus security.

11.2 Positive Aspects of the SHA-2 Algorithms

There are some aspects of the SHA-2 security that we consider to be a positive

I/O Formatting There are no known weaknesses on the padding and parsing of messages.

Davies-Meyer Construction. These constructions for compression functions have been analyzed for some time. Aside from the ease of finding fixed points (discussed below), the research suggests that the Davies-Meyer Construction is one of the best.

Combining Addition Operations and Bit-Slice Functions When using addition to combine the output(s) of the bit-sliced function, there are some interesting side effects. It is difficult to explain these effects without a suitable introduction to differential attacks, so the discussion of these effects is postponed to Section 14.

Bit-wise Functions When finding a differential path through the first several rounds of the SHA-1 and SHA-2 algorithms, all bit positions of the inputs to the MAJ and CH functions act independently (that is, what happens at one bit position does not affect other bit positions). The MAJ and CH functions can have between zero and three bit differences in the inputs at a particular bit position. When there are zero bit differences input to the bit-sliced function, then no conditions are imposed on the working variable bits. When there is one bit difference input to the bit-sliced function, then there can be conditions imposed on the remaining two

working variable bits. Such conditions either assign values to the working variable bits or impose a relationship between the two working variable bits. This relationship can either be of the form "the two working variables bits must be equal" or of the form "the two working variables bits must be complements of each other".

When generating a differential path, it is relatively easy to keep track of such relationships and ensure that the relationships are satisfied. If the relationships involve more working variable bits (for example, three working variable bits), then it becomes much more difficult to keep track of such relationships and ensure that the relationships are satisfied. This would go against the principle of simplicity discussed in Section 9.3.3.

Diffusion through Multiple Rotations The message expansion, and in particular the σ_0 and σ_1 provide very effective diffusion.

11.3 Negative Aspects of the SHA-2 Algorithms

There are some aspects of the SHA-2 security that we consider to be negative:

Length Extension Attacks It is well known that the Merkle-Damgård construction is susceptible to length extension attacks.

Attacks above the birthday bound Recent research has revealed attacks on the Merkle-Damgård construction, exploiting properties that arise when an attacker can exceed the birthday bound. These attacks include as the multi-collision attack [35], Long-message attacks [39] and Herding attacks [37].

Overly complex Message Expansion The message expansion combines addition operations, rotation and XOR operations. It is very difficult to predict the propagation of bit differences through this message expansion, and few researchers have presented any results on the message expansion. We feel that the SHA-2 message expansion is overly complex, contradicting the principle of simplicity discussed in Section 9.3.3. A linear message expansion, using functions similar to the σ_0 and σ_1 functions, could provide adequate diffusion, while remaining simple enough to be be analyzed.

Overuse of the Addition Operation The SHA-2 algorithms use the addition operation extensively. The addition operations add significant cost in two areas: addition operations are a significant contribution to the latency and area of hardware implementations; addition operations are a significant contribution to the energy usage of both software and hardware implementations.

Fixed points The Davies-Meyer Construction allows easy computation of fixed points: pairs of a hash input $H^{(i-1)}$ and message block $M^{(i)}$ such that the output of the compression function is the same as the input. The easy computation of fixed points is noted in [60, 37]. We feel that this should be addressed.

SHA-224 and SHA-256 cannot exploit 64-bit processors When SHA-224 and SHA-256 are performed on 64-bit processors, the most of the operations use only 32 bits of the register at a time.

Limited Circuitry Re-Use In hardware implementations, combined implementations of the SHA-256 and SHA-512 algorithms have limited ability to re-use circuitry, resulting in larger combined implementations.

11.4 Design Philosophy for the CHI Algorithms

To be a good submission, the algorithm must rate highly against all the design characteristics in section 9.2. The design philosophy for the CHI algorithms sets out how we hope to achieve that goal.

1. Use familiar constructions, with minimal modifications:
   (a) Use the Merkle-Damgård construction, (*used for SHA-2*) but make minimal modifications to prevent length extension attacks. *Similar philosophy as applied for SHA-2.*
   (b) Use the Davies-Meyer construction, (*also used for SHA-2*) but make minimal modifications to prevent to prevent easy computation of fixed points. *Similar philosophy as applied for SHA-2.*
   (c) Construct the underlying block cipher using message expansion, a set of step constants and a step function.
   (d) The message expansion should balance simplicity and achieving good diffusion.
   (e) Base the step function on well known constructions such as a Feistel Network [30] or Substitution-Permutation (SP) Network [47, Section 7.4].
   (f) The components making up the message expansion and step function should be well analyzed in the literature.
2. Use a combination of addition (which has carry effects for micro-diffusion), with rotation for macro-diffusion and bit-sliced non-linear functions. *Same philosophy as for SHA-2.*
3. Focus on diffusing information from the variable you just generated. *Same philosophy as for SHA-2.*
4. Minimize the number of addition operations, by using XOR in the place of addition operations wherever possible. A side effect is that the message expansion is linear and easier to analyze.
5. If diffusion within a word64 is required, then use functions similar to the σ_0 and σ_1 functions. *Same philosophy as for SHA-2.*
6. Create the step functions around moderately-complex bit-sliced functions that optimize the ratio of "non-linear effects" to implementation cost.
   (a) Higher non-linearity per step.
   (b) In each step, each bit difference induces conditions on many bits.
   (c) The bit-sliced function can be implemented very efficiently in hardware and software.
   (d) Accept the penalty that these complex functions use more temporary variables, since modern processors have more registers

7. Use two compression functions: one used by CHI-224 and CHI-256, and one used by CHI-384 and CHI-512. This would simplify implementations, and allow cryptanalysis to be more focused.

8. Use a larger state than required (particularly for CHI-224 and CHI-256) so as to prevent the effect of attacks when the attacker can exceed the birthday bound.

9. Maximize similarities between the algorithms to allow circuitry re-use in hardware implementations supporting all hash output sizes

The next few sections provide details of how this design philosophy translated into design rationale for the Input/Ouptut formatting and META structure (Section 12), the MACRO structure (Section 13) and the MICRO structure (Section 14).

12 The Input/Output Formatting and META Structure

12.1 Design Rationale for the Input/Output Formatting

There are no known weaknesses in the padding and parsing used in SHA-1 and the SHA-2 algorithms [56], so the CHI algorithm maintains this method of padding and parsing. These blocks are then processed according to a modification of the Merkle-Dåmgard Construction, which is a slight variation on the approach of SHA-1 and the SHA-2 algorithms.

The Merkle-Dåmgard Construction ([50, 28] and [47, Section 9.3.2]) initializes a hash state $H^{(0)}$, and then computes the sequence of hash values $H^{(1)}$, $H^{(2)}, \ldots, H^{(N)}$ as

$$H^{(i)} \;=\; \mathit{Compress}(H^{(i-1)}, M^{(i)})$$

where $\mathit{Compress}(\cdot,\cdot)$ denotes an appropriate compression function. The Merkle-Dåmgard construction has been around for many years, and is well analyzed [25, 29]. The construction is used in MD5, SHA-1 and the SHA-2 algorithms.

From an implementation perspective, the Merkle-Dåmgard construction only allows the computation to be performed as a sequence of $\mathit{Compress}(\cdot,\cdot)$ operations, which prevents multiple $\mathit{Compress}(\cdot,\cdot)$ operations being computed in parallel. There are other constructions that do permit parallelism at this level [49]. However, the CHI team concluded (see Section 9.5) that the CHI algorithm would not attempt to exploit parallelism at this level, and so the Merkle-Dåmgard construction was considered.

From a security perspective, the main weakness with the Merkle-Dåmgard Construction is the well-known existence of length-extension attacks. The CHI algorithms address this weakness by processing the final message block in a manner distinct from that used for the other message blocks. The easiest way to make the processing distinct was to provide some modification of the working variables after they have been initialized with the with the $(i-1)$-st hash value. If the compression function is suitably secure, then any simple modification should suffice. The CHI team chose to modify the working variables by applying a rotation by a single bit, as described in Section 7.1.2 step 2 and Section 7.1.2 Step 2. We call the resulting construction a *modified Merkle-Dåmgard construction.*

The message length fields were chosen to be 64-bits long for CHI-224 and CHI-256, and 128-bits long for CHI-384 and CHI-512, simply because the message length fields are these lengths for the SHA-2 family.

12.1.1 Choosing the Message Block Size

The message block sizes chosen by the CHI team are the same as for the SHA-2 algorithms. We believe that this message block should be at least twice the hash size, in order for the output distribution to be indistinguishable from uniform using less than $2^{s/2}$ complexity, where s is the block size. We can't produce a concrete argument for this rule of "thumb", but we can certainly provide a motivation.

Consider a hash function with message block size m and hash block size s. The frequency distribution of the outputs (assuming a fixed initial hash value) follows a Poisson distribution with $\lambda = 2^{m-s}$. If $m == s$, then $\lambda = 1$ and the Poisson distribution is quite obviously not uniform: a fraction of $e^{-1} = 0.37$ of the outputs are not possible. If $m == 2s$, then $\lambda = 2^s$ and the Poisson distribution is very close to uniform. An attacker would need to detect a standard deviation that is only $\frac{1}{2^{s/2}}$ times the mean value.

12.1.2 Choosing the Hash State Size

When choosing the size of the hash state, various factors came into play:

- If the hash state is too small, then multi-collision-style attacks become a threat;
- If the hash state is too small, then there are fewer working variables to manipulate (256-bit state corresponds to only four word64s: it is difficult to construct an algorithm using only four values).
- If the hash state is too large, then too many steps are required to diffuse information through the working variables.

Multi-collision-style attacks fall somewhere between collisions and second-pre-image attacks. These attacks include multi-collisions attack [35], second preimage attacks on long-messages [38, 39] and herding attacks [37]. The attack complexity is more than for a collision attack and less than for a second-pre-image attack, while accomplishing something more than achieved by a collision attack and less than achieved a second-pre-image attack. For hash functions using the Merkle-Damgård construction with hash state of s bits, the complexity of these attacks is at least $2^{s/2}$.

CHI-224 and CHI-256 SHA-224 and SHA-256 use a 256-bit state and the complexity of multi-collision-style attacks is around 2^{128}. This complexity is (relatively) low, and it would be good to increase this bound significantly: at least to 2^{192}. To ensure that the complexity of multi-collision-style attacks is at least 2^{192}, the hash state of CHI-224 and CHI-256 was chosen to be 384 bits. Thus, CHI-224 and CHI-256 provide significantly additional protection against these attacks when compared to their SHA-2 counterparts. This size state corresponds to six word64s, for which quite complex interactions can be specified.

CHI-384 and CHI-512 SHA-384 and SHA-512 use a 512-bit state and the complexity of multi-collision-style attacks is around 2^{256}. The complexity of these attacks (2^{256}) is so large that we do not consider these attacks to be a significant threat for many years. Consequently, the CHI team decided that it was acceptable for CHI-384 and CHI-512 to have state size close to that of SHA-384 and SHA-512.

At the MACRO structure level (see Section 13.1), CHI-224 and CHI-256 can be thought of as two interleaved, unbalanced Feistel networks [30], each consisting of three word64s. To maximize

the similarities between the algorithms, it was useful to design CHI-384 and CHI-512 as three interleaved, unbalanced Feistel networks , each consisting of three word64s. As a result, CHI-384 and CHI-512 consist of nine word64s, a total of 576 bits. Thus, the complexity of multi-collision-style attacks against CHI-384 and CHI-512 is at least 2^{288}, which we consider acceptable.

12.2 Design Rationale for the META Structure

Recall that the META structure is the overall structure of the compression function.

The CHI design team decided that CHI would use a compression function based on an underlying block cipher. The META structure of such compression functions has attracted some research both in the past [60] and more recently [23, 42, 43, 22, 33]. Encryption (using the underlying block cipher) of an input x using key K to form output y will be denoted $y := Enc_K(x)$, and decryption of y using key K to form x again denoted $x = Dec_K(y)$.

In these analyses, the Davies-Meyer [46] construction has proven to be one of the most secure constructions. The Davies-Meyer construction forms the next hash input as

$$H^{(i)} \;=\; \mathit{Compress}(H^{(i-1)}, M^{(i)}) \;:=\; E_M^{(i)}(H^{(i-1)}) \oplus H^{(i-1)}. \tag{1}$$

The Davies-Meyer construction is used by MD5, SHA-1 and SHA-2 algorithms.

The primary weakness of the Davies-Meyer construction is the ability to easily compute *fixed points*. A fixed point is a pair of values (VH, VM), such that if $H^{(i-1)} := VH$ and $M^{(i)} := VM$, then $H^{(i)} == VH$ is true. That is:

$$H^{(i)} = \mathit{Compress}(H^{(i-1)}, M^{(i)}) = \mathit{Compress}(VH, VM) = VH.$$

Fixed points pose weaknesses: if the attacker can reach the the hash state VH, then the attacker can form a message of any length with hash values remaining equal to VH, simply by providing multiple message blocks equal to VM. There are generic methods [38, 39] for finding fixed points of any compression function in time $2^{n/2}$. In the case of the Davies-Meyer construction, large sets of fixed points can be computed trivially. Note that if $H^{(i-1)} = H^{(i)} = VH$, then the fixed point must satisfy:

$$\begin{aligned} VH \;&==\; E_{VM}(VH) \oplus VH; \\ \Rightarrow\; E_{VM}(VH) \;&==\; VH \oplus VH \;==\; 0. \end{aligned}$$

We now have a simple condition that the fixed point must satisfy: $E_{VM}(VH) == 0$ or equivalently $VH == Dec_{VM}(0)$. There are no restrictions on the values VH and VM other than this condition. For any choice of VM, an we can compute $Dec_{VM}(0)$ and assign $VH := Dec_{VM}(0)$, and the pair (VH, VM) is a fixed point. The computation required only one decryption using the block cipher.

The CHI team desired to prevent this weakness. The approach taken was to apply some function (let us call if *foo*()) to the hash input prior to XOR the input with the output of the block cipher to form the next hash value:

$$= \mathit{Compress}(H^{(i-1)}, M^{(i)}) \;:=\; E_M^{(i)}(H^{(i-1)}) \oplus \mathit{foo}(H^{(i-1)}). \tag{2}$$

We claim that modifying the chaining variable when we feed it back make fixed points difficult to find. Note that this method does not completely prevent fixed points. To find a fixed point of this

construction, the fixed point must satisfy:

$$\begin{aligned} VH &== E_{VM}(VH) \oplus foo(VH); \\ \Rightarrow \quad E_{VM}(VH) &== VH \oplus foo(VH). \end{aligned}$$

To explain our choice of function for *foo*(), we compare a few examples. We restrict the choice of *foo*() to linear functions.

- If $foo(x)$ is the identity function (that is $foo(x) := x$), then the range of $y = (VH \oplus foo(VH))$ contains only the value 0. Once we choose the value of y, all VH satisfy $(VH \oplus foo(VH)) == y$. That is, the requirement $(VH \oplus foo(VH)) == y$ imposes no conditions.
- If $foo(x)$ applies $SWAP32(x)$ to each word64 of the hash state, then the range of $y = (VH \oplus foo(VH))$ contains only the values of y for which the upper and lower word32s are identical.
  - For small CHI, this is a set of $2^{384/2=192}$ values for y. Once we choose a value of y, there are there are $2^{384-192=192}$ values of VH such that $(VH \oplus foo(VH)) == y$. That is, the requirement $(VH \oplus foo(VH)) == y$ imposes additional conditions on VH.
  - For big CHI, this is a set of $2^{576/2=288}$ values for y. Once we choose a value of y, there are there are $2^{576-288} = 2^{288}$ values of VH such that $(VH \oplus foo(VH)) == y$. That is, the requirement $(VH \oplus foo(VH)) == y$ imposes additional conditions on VH.
- If $foo(x)$ applies $ROTR64^n(x)$ to each word64 of the hash state (n odd), then the range of $y = (VH \oplus foo(VH))$ contains the values of y for which each word64s has even weight.
  - For small CHI, this is a set of $2^{384-6=378}$ values for y. Once we choose a value of y, there are $2^{384-378} = 2^6$ values of VH such that $(VH \oplus foo(VH)) == y$. Again, the requirement $(VH \oplus foo(VH)) == y$ imposes additional conditions on VH.
  - For small CHI, this is a set of $2^{576-6=570}$ values for y. Once we choose a value of y, there are $2^{576-570=6}$ values of VH such that $(VH \oplus foo(VH)) == y$. Again, the requirement $(VH \oplus foo(VH)) == y$ imposes additional conditions on VH.

Consider the first case where *foo*() is the identify function (used in the standard Davies-Meyer construction). There is only one value for $y = (VH \oplus foo(VH))$ to choose from. We choose that value of y, and when we evaluate $VH = Dec_{VM}(y)$ for a choice of VM, then we are guaranteed that $(VH \oplus foo(VH)) == y$.

Now consider the second case where *foo*() applies $SWAP32(x)$ to each word64 of the hash state. In the case of small CHI, there are 2^{192} values for $y = (VH \oplus foo(VH))$ to choose from. We choose one such value of y, and we evaluate $VH = Dec_{VM}(y)$ for a choice of VM. Assuming the resulting value of VH is uniformly distributed, the probability that $(VH \oplus foo(VH)) == y$ is only 2^{-192}. It is highly unlikely that VH satisfies this additional condition. We could try many values of VM until we compute a value of VH that satisfies the additional condition. However, since the decryption is implementing a secure block cipher, finding such a VM for a particular choice of y would have complexity of around 2^{192}. The attack complexity is about the same as for a generic attack trying to find fixed points on a black-box compression function.

The reason for this large complexity is that the attacker finds themselves in a chicken-and-egg problem: the attacker needs to predict y (which contains information about VH) prior to trying

to decrypt using VM as the key. Once the attacker selects y, the attacker has requirements on the value VH that results when computing $VH = Dec_{VM}(y)$, namely, the attacker can only use values VH such that $(VH \oplus foo(VH)) == y$. The same situation applies for big CHI.

Finally, consider the third case where $foo()$ applies $ROTR64^n(x)$ to each word64 of the hash state and n is odd. In the case of small CHI, there are 2^{378} values for $y = (VH \oplus foo(VH))$ to choose from. The attacker choose one such value of y, and then evaluates $VH = Dec_{VM}(y)$ for a choice of VM. Assuming the resulting value of VH is uniformly distributed, the probability that $(VH \oplus foo(VH)) == y$ is only 2^{-378}. It is highly unlikely that VH satisfies this additional condition. The attacker could try many values of VM until she computes a value of VH that satisfies the additional condition. However, since the decryption is implementing a secure block cipher, finding such a VM for a particular choice of y would have complexity of around 2^{378}. This complexity far exceeds than the complexity of generic attack trying to find fixed points on a black-box compression function. The same situation applies for big CHI.

This final case provides the greatest resistance against finding fixed points. The CHI team chose the easiest choice for the rotation amounts: 1.

13 Design Rationale for the MACRO Structure

Recall that the MACRO structure is the overall structure of the block cipher used in the compression function. The MACRO structure of the Step function and its phases are discussed first, followed by discussion of the Message Expansion.

13.1 Design Rationale for Step Function MACRO Structure

The CHI design team investigated two main approaches to constructing the step function of the block cipher.

The first approach used a Serpent-like construction [17] using a combination of addition operations, XOR-linear diffusion, and bit-sliced invertible 4-bit s-boxes. The team found that invertible s-boxes did not provide much non-linearity, and many steps/rounds would be required. The large number of steps required for a reasonable security margin would make the resulting hash function too slow to be competitive.

After this, the design team focused on a shift-register approach as used in the MD4 [62] and FIPS 180-2 [56] algorithms. A shift-register approach allowed the use of a non-invertible bit-sliced S-box (the *MAP* function), which can provide significant security advantages (such as higher non-linearity). From an implementation perspective, bit-sliced *MAP* functions are good because they scale well to any processor size and they are very efficient in hardware. These security advantages can be multiplied by applying diffusion before and after the *MAP* function. In this sense, the structure of the CHI algorithms is similar to that used in the Data Encryption Standard (DES) [52].

The step function has five phases (see Section 6.3). Each phase has a specific purpose.

- The purpose of the *PRE-MIXING* phase is to provide diffusion by ensuring that any single bit difference occurring in a working variable results in multiple bit differences being input to the *MAP* function. The *PRE-MIXING* phase can best achieve its purpose by XORing modified copies of the working variables to forming the *MAP* inputs. The modification was restricted to moving bits within the word64, so as to incur no additional cost in hardware. This is discussed further in Section 13.3.

- The purpose of the *DATA-INPUT* phase is to feed the step inputs and step constants into the computation, while minimizing the probability of a difference in the step inputs being canceled in the *MAP* phase. This is discussed and analyzed in more detail in Section 13.4.

- The purpose of the *MAP* phase is to provide most of the non-linearity in the step function. In the first few steps, this complex mapping ensures that predicting difference propagation imposes conditions on a large number of state bits. In the remaining steps, this complex mapping ensures that predicting difference propagation has low success probability. This is discussed further in Section 13.2.

- The purpose of the *POST-MIXING* phase is to provide additional diffusion (by moving bits of X, Y and Z to new positions) and a small amount of non-linearity in-between bit positions through the use of modular addition. The *POST-MIXING* phase can best achieve its purpose by combining modified copies of the *MAP* outputs using the addition operation. The outputs of the addition were further modified to prevent exploitation of the linear and differential properties of the MSB and LSB of the addition outputs. This is discussed further in Section 13.3.

- The purpose of the *FEEDBACK* phase is to combine the computed feedback values back into the working variables This is discussed and analyzed in more detail in Section 14.4.

The internal workings follow a series of phases similar to those used in the DES.

1. First, there is a diffusion phase: in DES, this corresponds to the expansion E; while in the CHI algorithms, this corresponds to the *PRE-MIXING* phase.

2. Then the message (key material in the case of DES) is combined with the output of the first diffusion phase. In DES, this corresponds to XORing the key with the output of the expansion E. In the CHI algorithms, this corresponds to the *DATA-INPUT* phase.

3. This is followed by the application of a fixed, nonlinear mapping: in DES, this corresponds to applying the S-boxes, while in the CHI algorithms, this corresponds to the *MAP* phase.

4. Another layer of diffusion is applied to the output of the nonlinear mapping: in DES, this corresponds to applying the permutation P, while in the CHI algorithms, this corresponds to the *POST-MIXING* phase.

13.2 Design Rationale for the *MAP* Phase

The dominating factors influencing the MACRO structure are the number of inputs and outputs. The security advantages typically increase with the number of inputs and outputs, but the implementation cost also increases with the number of inputs and outputs. In order to select a suitable number of inputs and outputs, the CHI team chose some applicable measures of the security advantage, and then computed these measures for the varying options for the number of inputs and outputs. The CHI team also estimated the implementation cost for a *MAP* with these numbers of inputs and outputs: the cost was evaluated in terms of software operations, hardware latency and number of hardware gates. In these calculations, it was also assumed that each output would be combined with other values using an addition operation. Finally, the measures for the security

advantage were divided by the various values for the cost, providing a scaled or normalized advantage. These sets of values were compared for the various number of inputs and outputs. The comparison indicated that a *MAP* with four inputs and three outputs was an optimal choice with respect to the measures of the security advantage divided by the implementation cost.

The two most basic requirements of the *MAP* phase are as follows:

***MAP* Requirement 1** . The output distribution is balanced: in this case, this means that for each 3-bit output value y there are two 4-bit inputs x for which $MAP(x) = y$.

***MAP* Requirement 2** . If two inputs differ in only one bit, then the corresponding outputs must differ in at least one bit.

***MAP* Requirement 3** . If two inputs differ in two bits, then the corresponding outputs are identical with probability at most $\frac{1}{8} = 2^{-3}$.

***MAP* Requirement 4** . If two inputs differ in three bits, then the corresponding outputs are identical with probability at most $\frac{1}{4} = 2^{-2}$.

Further design rationale for the choice of *MAP* are discussed in Section 14.1.

13.3 Design Rationale for the *PRE-MIXING* Phase

The CHI team decided to form the *MAP* inputs by XORing sets of the modified copies of the working variables. The *PRE-MIXING* phase for CHI-224 and CHI-256 differs from the *PRE-MIXING* phase for CHI-384 and CHI-512 for a couple of reasons. First, CHI-224 and CHI-256 use fewer working variables than CHI-384 and CHI-512, so this necessitates distinct *PRE-MIXING* phases. Secondly, CHI-384 and CHI-512 (which have a larger state size) require more diffusion than CHI-224 and CHI-256. Thirdly, as discussed in Section 9.4.5, CHI-224 and CHI-256 need to be fast on 32-bit processors, while for CHI-384 and CHI-512 speed on 32-bit platforms is a lower priority.

13.3.1 Design Rationale for the *PRE-MIXING* Phase for CHI-224 and CHI-256

For CHI-224 and CHI-256, the CHI team decided to use eight modified copies of the working variables, with each MAP input formed by XORing pairs of the modified copies. Three copies of A and D are used, and single modified copies of B and E are used. This choice maximizes the diffusion from the most recently-updated working variables (A and D), in accordance with the CHI design philosophy "Focus on diffusing information from the variable just generated" (Section 11.4), which is a philosophy also employed by the SHA-2 design.

Initially, the *ROTL*64 operations were considered for performing the modification, but the resulting estimate for the software speed on 32-bit processors was significantly slower than SHA-224 and SHA-256. In order to improve the software performance, some copies were modified by applying *ROTL*32 rotations to the upper and lower word32s in place, while the remaining copies were modified by first applying *SWAP*32 (to swap the upper and lower word32s), and then applying *ROTL*32 rotations to the resulting upper and lower word32s. This was estimated to increase the speed on 32-bit processor by a factor of 15% to 30%, while decreasing the speed on a 64-bit processor by around 5%.

Using 32-bit rotations in the premixing has the additional advantage of allowing implementations to split the *PRE-MIXING* and *MAP* phases into independent sequences of operations on

word32s. These sequences of operations are described in Section 7.1.3. An implementation can focus on each sequence separately, and this reduces the amount of temporary variable space required, making the implementation more efficient.

Further design rationale for the choice of rotations in the *PRE-MIXING* phase for CHI-224 and CHI-256 are discussed in Section 14.3.2.

13.3.2 Design Rationale for the *PRE-MIXING* Phase for CHI-384 and CHI-512

For CHI-384 and CHI-512, the CHI team decided to use twelve modified copies of the working variables, with each MAP input formed by XORing three of the modified copies. Three copies of A, D and G are used, and single modified copies of B, E and P are used. As for CHI-224 and CHI-256, this choice maximizes the diffusion from the most recently-updated working variables (A, D and G), in accordance with the CHI design philosophy "Focus on diffusing information from the variable just generated" (Section 11.4).

The CHI team considered using the $ROTR32$ and $ROTR64$ operations for performing the modification. When comparing $ROTR64$ operations to $ROTR32$ operations, $ROTR$ operations were estimated to decrease speed by 20% on 32-bit processors, while increasing speed by 35% on 64-bit platforms. As discussed in Section 9.4.5, the speed of CHI-384 and CHI-512 is a high priority for 64-bit platforms, and a lower priority for 32-bit platforms. Consequently, the CHI team concluded that the increase in speed on 64-bit processors was more important than a loss of speed on 32-bit processors, and $ROTR64$ operations were used.

Further design rationale for the choice of rotations in the *PRE-MIXING* phase for CHI-384 and CHI-512 are discussed in Section 14.3.4.

13.4 Design Rationale for the *DATA-INPUT* Phase

The *DATA-INPUT* phase combines the step inputs and step constants with the four outputs from the *PRE-MIXING* phase. Considerations include:

- Placement of the *DATA-INPUT* Phase within the step function.
- Number of step function inputs and step constants.
- Expanding the step inputs and step constants prior to combining with the four outputs from the *PRE-MIXING* phase.

To simplify the implementation, we assume that the number of step inputs and step constants is equal, and the step inputs and step constants are XORed prior to any other operations.

Also, to make some of the explanation simpler, we will use the term *step input package* to refer to the set of step inputs and step constants used in a single step.

13.4.1 Regarding the Placement of the *DATA-INPUT* Phase

In the MD5, SHA-1 and SHA-2 algorithms, the equivalent of the *DATA-INPUT* phase uses the step inputs at the end of the step, after the bit-sliced functions have been applied. For the CHI algorithms, the step inputs are used at the beginning of the step (in the *DATA-INPUT* phase), before the bit-sliced *MAP*. There is a good reason for the placement in the CHI algorithms.

The former approach (using step inputs at the end of the step) does not fully exploit the ability to input message data into the the state. The first step of the former approach applies all of the computation to the initial constants; that computation is wasted. Additionally, the effect of the last step of this approach is entirely linear in the last step input.

In the CHI algorithms, all of the computation in the first step (with the exception of the *PRE-MIXING* phase) applies to message-dependent variables, so the computation is not wasted. Furthermore, the effect of the last step of the CHI algorithms is now a complex non-linear function in the last step input.

The one negative aspect of this approach is a result of using a *MAP* with more inputs than outputs. For such a *MAP*, there must be pairs of *MAP* inputs that result in identical outputs. The danger is that a difference in the step inputs might not introduce differences in the working variables. We have designed the *DATA-INPUT* to reduce the probability of two distinct step input packages resulting in identical outputs from the *MAP*.

13.4.2 Number of the Step Inputs and Step Constants

If the number of step inputs is equal to the number of *MAP* inputs (that is, there are four step inputs) then the bit differences input to the *MAP* inputs can be completely controlled by a suitable choice of bit differences in the step inputs. There are a couple of reason why this is an issue:

First, *MAP* Requirement 4 indicates that there is a choice of three *MAP* inputs for which bit differences *MAP* inputs (at a common bit position) cancel with high probability: $\frac{1}{4} = 2^{-2}$. Consequently, an attacker can choose pairs of step input packages that result in those bit differences, and thus cancel with high probability, and then fail to introduce differences in the working variable.

Secondly, if differences are introduced to the working variables, then bit differences in the step inputs can be chosen to cancel with the bit differences output by the *PRE-MIXING* phase.

These weaknesses can be avoided by having fewer step inputs, and expanding these step inputs to the four word64s to be combined with the outputs of the *PRE-MIXING*. A choice of two inputs per step seems a good option. Note that Section 13.6 motivates CHI-224 and CHI-256 using 40 step inputs and CHI-384 and CHI-512 using 80 step inputs. Using two inputs per step would suggest 20 steps for CHI-224 and CHI-256, and 40 steps for CHI-384 and CHI-512. This fits in with our impression that, after inputting the message, 16 steps of CHI-224 and CHI-256 and 32 steps of CHI-384 and CHI-512 provide adequate security 21.

13.4.3 Expanding the Step Inputs and Step Constants

The output of the expanding can be viewed as an encoding of the step package. We desire a linear expansion, since this would be more efficient. A linear expansion also means that the XOR difference between the expansion of two step packages must be codewords of the step package. This XOR difference shows the position of bit differences, so we would prefer the minimum weight (that is the number of non-zero bits) of a non-zero codeword to be reasonably high.

A simple approach could be applied, with the four output corresponding to copies of the inputs or rotated copies of the inputs. This provides a minimum weight of only 2, and can be exploited as follows. Suppose that the rotation of the first and second step inputs is w_0 and w_1 respectively. Generate a pair of step packages with the first step input having bit differences in positions k and $k + w_1$ and second step input having bit differences in positions k and $k + w_0$. The rotation of the first step input will have bit differences in positions $k + w_0$ and $k + w_0 + w_1$. The rotation of

the first step input will have bit differences in positions $k + w_1$ and $k + w_0 + w_1$. The four copies of the set inputs will now have (between them) two bit differences in positions k, $k + w_0$, $k + w_1$ and $k + w_0 + w_1$. *MAP* requirement 3 (Section 13.2) indicates that, at a single bit position, the probability of two bit differences resulting in no difference in the output is $\frac{1}{8} = 2^{-3}$ in some cases. The probability of having no difference in the *MAP* output (and thus injecting no new differences in to the working variables) is $(2^{-3})^4 = 2^{-12}$. We considered this probability to be unacceptably high.

At the other end of the spectrum, Reed-Solomon coding [61] could be applied to obtain higher minimum weight. However, Reed-Solomon coding is reasonably expensive in software, so this option was not pursued.

We settled on input mixing functions θ_0 and θ_1 (similar to the Σ_0 and Σ_1 functions in SHA-2) that mix the words independently. Each θ function is the XOR of three rotations of an input word.

13.5 Design Rationale for the *POST-MIXING* Phase

The first thing to note about the *POST-MIXING* is that if one of the *MAP* outputs (MAP_0, MAP_1 or MAP_2) will likely have a property that favors attacks slightly more than the other *MAP* outputs. If the *POST-MIXING* always applies the same processes to the same ordering of *MAP* outputs, then this slight weakness might compound to form a serious weakness. Of course, we hope that the *MAP* functions don't have any weaknesses that can be exploited in this manner, but the history of cryptography indicates that there is always some weakness that the designers did not foresee. In an effort to prevent this, the CHI team decided that, from one step to the next, the algorithm should change the order in which the map outputs MAP_0, MAP_1, MAP_2 are assigned to the word64s X,Y, Z. That is, the*POST-MIXING* uses a step-dependent multiplexor. In a naive implementation, the step-dependent multiplexor incurs only a small cost in software and hardware implementations. In unrolled software implementations or multi-step hardware implementations, the step-dependent multiplexor is almost free.

13.5.1 Design Rationale for the *POST-MIXING* Phase for CHI-224 and CHI-256

One of the CHI design philosophies (Section 11.4) is to minimize the number of addition operations. The algorithms modify the word64s X,Y, Z to result in word64s XX, YY and ZZ respectively. The feedback values AA and DD are formed as:

$$\begin{aligned} AA &:= ROTR64^{r1}(XX \boxplus YY); \\ DD &:= ROTR64^{r2}(YY \boxplus ZZ). \end{aligned}$$

The rotations are included to prevent properties of the LSB or MSB being exploited.

Choice of Operations to modify X, Y, Z. Applying the *SWAP*8 operations just before the addition operations seemed the most effective use of an *SWAP*8 operations, since the *SWAP*8 operation would destroy some adjacency between bits on the byte boundaries, and the addition operation would start diffusing information between previously non-adjacent bits. We chose to modify YY using the *SWAP*8 operation, so that the destroyed some adjacency would be carried to both feedback values AA and DD. The remaining modifying operations were allowed to be rotations in 32-bit blocks (cheap on 32-bit processors and modest on other processors), and the *SWAP*32 operation (cheap on 64-bit processors and free on other processors).

Requirements on the Operations Used to modify X, Y, Z. To ensure good diffuse, some requirements are placed on the operations used to modify the word64s X, Y, Z to form XX, YY, ZZ. We model the addition operations as the XOR operation, so that, for example, the value of $AA[i]$ depends on the values of $XX[i]$ and $YY[i]$. Note that the values of X, Y, Z at bit position i depend on the input bits of R, S, T, U at bit position i. Furthermore, the value of $AA[i]$ depends on the values of $XX[i]$ and $YY[i]$, which in turn depend on the values of $X[i_x]$ and $Y[i_y]$ at some bit positions i_x and i_y respectively (the relationship between i and i_x, i_y depends on the operations used to modify X and Y). If we allow $i_x = i_y$ then then value of $AA[i]$ depends on only the four bits $R[i_x], S[i_x], T[i_x], U[i_x]$, while if we enforce $i_x \neq i_y$ then then value of $AA[i]$ depends on eight bit of R, S, T, U. The latter case obviously provides better diffusing, so we put a requirement on the operations modifying X, Y to form XX, YY, requiring that to that for all i, $XX[i]$ and $YY[i]$ depend on $X[i_x]$ and $Y[i_y]$ with $i_x \neq i_y$. When we analyze $DD[i]$ in the same way, we get the requirement that to that for all i, $YY[i]$ and $ZZ[i]$ depend on $Y[i_y]$ and $Z[i_z]$ with $i_y \neq i_z$.

As above, for any i, j, assume $XX[i]$,$YY[i]$ and $ZZ[i]$ depend on the bits $X[i_x], Y[i_y], Z[i_z]$ respectively, and $XX[j]$,$YY[j]$ and $ZZ[j]$ depend on the bits $X[j_x], Y[j_y], Z[j_z]$ respectively. To ensure even better diffusion, we would like to ensure that every pairs of bits of AA and DD depends on as many bits as possible, those pairs being of the form: $AA[i]$ and $AA[j]$; $DD[i]$ and $DD[j]$; and $AA[i]$ and $DD[j]$.

- Examining $AA[i]$ and $AA[j]$ results in the requirement: for all $i \neq j$ the set $\{i_x, i_y\}$ and $\{j_x, j_y\}$ are not identical.
- Examining $DD[i]$ and $DD[j]$ results in the requirement: for all $i \neq j$ the set $\{i_y, i_z\}$ and $\{j_y, j_z\}$ are not identical.
- Examining $AA[i]$ and $DD[j]$ results in the requirement: for all $i \neq j$ the set $\{i_x, i_y\}$ and $\{j_y, j_z\}$ are not identical.

The cheapest choice of operations to satisfy these requirements were:

$$\begin{aligned} XX &:= XX; \\ YY &:= SWAP8(DROTR32^{ryu,ryl}(Y); \\ ZZ &:= SWAP32(Z); \end{aligned}$$

where ryu, ryl cannot be a multiple of 8. The choice of rotation amounts $ryu, ryl, r1, r2$ is addressed in Section 14.3.2.

13.5.2 Design Rationale for the *POST-MIXING* Phase for CHI-384 and CHI-512

The *POST-MIXING* Phase for CHI-224 and CHI-256 differs from *POST-MIXING* Phase for CHI-384 and CHI-512, primarily because CHI-384 and CHI-512 require an additional feedback value. The algorithms modify the word64s X,Y, Z to result in word64s XX, YY and ZZ respectively. The feedback values AA, DD, GG are formed as:

$$\begin{aligned} AA &:= ROTR64^{r1}(XX \boxplus YY); \\ DD &:= ROTR64^{r2}(YY \boxplus ZZ); \\ GG &:= ROTR64^{r3}(ZZ \boxplus (XX \times \psi)); \end{aligned}$$

where the *psi* is a constant and $\times$ denotes integer multiplication. To determine which values of ψ are suitable, we re-write the above expression in terms of matrix multiplication modulo 2^{64}:

$$\begin{pmatrix} ROTR64^{-r1}(AA) \\ ROTR64^{-r2}(DD) \\ ROTR64^{-r3}(GG) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ \psi & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} XX \\ YY \\ ZZ \end{pmatrix} = PM \cdot \begin{pmatrix} XX \\ YY \\ ZZ \end{pmatrix};$$

$$\text{where } PM := \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ \psi & 0 & 1 \end{pmatrix}.$$

It is important that every value of (AA, DD, GG) be a possible output of this operation. Since the number of bits in (XX, YY, ZZ) is the same as the number of bits in (AA, DD, GG), every value of (AA, DD, GG) is a possible output if and only if PM has odd determinant. The determinant of PM is $(\psi + 1)$, which implies that ψ must be even. Since multiplication with $\psi = 2$ corresponds to a simple left shift by one bit, $\psi = 2$ was chosen as the constant. In the description in Section 7.3.2, the multiplication with $\psi = 2$ is implemented by as $(XX \ll_{64} 1)$.

In accordance with our design philosophy to "Maximize similarities between the algorithms to allow circuitry re-use in hardware implementations supporting all hash output sizes" (Section 11.4), we tried to make the *POST-MIXING* Phase for CHI-384 and CHI-512 agree with a large amount of the *POST-MIXING* phase of the CHI-224 and CHI-256. Consequently, the modification from X, Y, XZ to XX, YY, ZZ for CHI-224 and CHI-256 has been modified only slightly for use in CHI-384 and CHI-512. An analysis of the bit pairs, similar to the analysis for CHI-224 and CHI-256, reveals that the following modifications provide optimal diffusion:

$$\begin{aligned} XX &:= XX; \\ YY &:= SWAP8(ROTR64^{ry}(Y); \\ ZZ &:= SWAP32(Z); \end{aligned}$$

provided ry is not a multiple of 8. Note that the $DROTR32^{ryu,ryl}(Y)$ of CHI-224 and CHI-256 is changed to be $ROTR64^{ry}(Y)$ for CHI-384 and CHI-512.

The choice of rotation amounts $ry, r1, r2, r3$ is addressed in Section 14.3.4.

13.6 Design Rationale for Message Expansion MACRO Structure

The MD5, SHA-224 and SHA-256 algorithms expand the message block by a factor of four times, while SHA-1, SHA-384 and SHA-512 algorithms expand the message block by a factor of five times. To keep on the conservative side, we required the CHI message expansion to expand the message block by a factor of five times. This means that CHI-224 and CHI-256 generate 40 step inputs, while CHI-384 and CHI-512 generate 80 step inputs.

The SHA-2 algorithms have received relatively little attention, largely due to the complex message expansion. The SHA-2 message expansion involves rotations, XOR operations and addition operations. The involvement of the addition operations prevents an analysis from predicting how changes in the message block will result in changes in the step inputs. It is not known if there are pairs of message blocks for which the resulting sequences of step inputs have relatively few differences.

The SHA-1 algorithm, on the other hand, has a message expansion that is linear with respect to the XOR operation. This allows cryptanalysts to predict how bit changes in the message block

result in bit changes in the step inputs. Unfortunately, the SHA-1 message expansion does not diffuse information quickly. The effect is that pairs of message blocks exist for which the resulting sequences of step inputs have relatively few differences.

The CHI team has attempted to glean the best aspects of the SHA-1 and SHA-2 message expansions. Similar to SHA-1, the CHI message expansions are linear with respect to XOR, so that cryptanalysts can predict how bit changes in the message block result in bit changes in the step inputs. However, the CHI message expansions provides better diffusion than the SHA-1 algorithm, by using linear functions μ_0 and μ_1 similar to the σ_0 and σ_1 functions in the SHA-2 algorithms:

$$\begin{aligned} \text{For CHI-224 and CHI-256: } W_t &:= W_{t-8} \oplus \mu_0^{\{256\}}(W_{t-7}) \oplus \mu_1^{\{256\}}(W_{t-2}) \\ \text{For CHI-384 and CHI-512: } W_t &:= W_{t-16} \oplus \mu_0^{\{512\}}(W_{t-15}) \oplus W_{t-7} \oplus \mu_1^{\{512\}}(W_{t-2}) \end{aligned}$$

where the linear functions μ_0 and μ_1 are formed by XORing two rotations of the input word with a right shift of the input word. Such a message expansion can prevent the existence of pairs of message blocks for which the resulting sequences of step inputs have relatively few differences. New names have been given to the μ_0 and μ_1 functions used in CHI so as to prevent confusion with the functions used in the SHA-2 family.

The use of the shift operations, rather than a third rotation operation, needs explaining. If the shift operations is replaced by a third rotation operation, then for all inputs x, and any rotation amount n if $x^* = ROTR64^n(x)$, then (using $\mu_0^{\{256\}}$ as an example):

$$\mu_0^{\{256\}}(x^*) = ROTR64^n(\mu_0(x)).$$

Thus, if given two message blocks M, M^* such that for some n, $M_j^* = ROTR64^n(M_j)$ for all j, then the corresponding sequence of step inputs will satisfy $W_t^* = ROTR64^n(W_t)$. This property is undesirable, and is prevented using the shift operation rather than a third rotation operation.

14 Design Criteria for MICRO structure

Most of the interesting MICRO structure occurs in the step function. We begin this section by looking at the desired MICRO structure properties of each phase of the step function, and then investigate the message expansion.

14.1 Desired Properties of the MAP function

The purpose of the *MAP* phase is to provide most of the non-linearity and confusion in the step function. Recall that the *MAP* function has four input word64s and outputs 3 word64s. The function is bit-sliced, so at any given bit position, the 3 output bits at that bit position are a function only of the four input bits in the same bit position. To analyze the *MAP* function, it is only necessary to analyze that function with 4 input bits and 3 output bits.

We divide the properties into two sets. One set of properties relates to how differences in the input to *MAP* affect the differences between the outputs of *MAP*: we call these *differential properties*. The non-differential properties are discussed first.

14.1.1 Desired Non-Differential Properties of the MAP function

MAP requirement 1 (Section 13.2) imposes the condition that the output distribution is balanced: in this case, this means that for each 3-bit output value y there are two inputs x for which $MAP(x) = y$. This property is easily tested, and it is easy to find such functions.

Each output bit can be expressed in terms of the input bits using the algebraic normal form (ANF) [71]. The degree of the ANF of each output bits should be as high as possible. A minimum degree of three was required.

***MAP* Requirement 5** . For each output bit, the algebraic normal form must have degree at least three.

Algebraic attacks use low-degree algebraic expressions combining inputs and output. For functions with four inputs and any number of outputs, it is well known [26] that there must be algebraic expressions of degree two. However, it is possible to prevent the existence of algebraic expressions of degree one (algebraic expressions of degree one are called *linear*).

***MAP* Requirement 6** . There are no algebraic expressions of degree one between the inputs and outputs of the *MAP* function.

14.1.2 Desired Differential Properties of the MAP function

The next few properties that were required related to how differences in the input to *MAP* affect the differences between the outputs of *MAP*. This field of research was known to the designers of DES [52], but the technique was kept secret until recent years. Biham and Shamir [19], discovered this technique some years later, and when applied to most block ciphers, the technique either broke the algorithm, or revealed weaknesses. Differential analysis made some small appearances in research into hash functions, but the technique did not receive significant publicity until Wang *et al* [67, 69, 70, 68] improved the attack to the point of finding collisions of MD5 and showing how to find collisions of SHA-1 for complexity less than the expected bound. Biham et al [20, 21] used a similar approach in attacking SHA-0.

Differential attacks exploit situations where an input difference results in an output difference with high probability. The notion of difference can be chosen to suit the algorithm being analyzed. For algorithms that predominantly use the XOR operation, difference defined by XOR operations are often most suitable. The *XOR-difference* between two values x', x'' of identical bit length is denoted $\Delta_{\oplus}(x', x'')$, and is defined as

$$\Delta_{\oplus}(x', x'') \quad := \quad x'' \oplus x'.$$

XOR differences are defined in this way because if $z := x \oplus y$, then the following property can be used:

$$\Delta_{\oplus}(z', z'') \quad = \quad \Delta_{\oplus}(x', x'') \oplus \Delta_{\oplus}(y', y'').$$

For a function $f : GF(2^a) \rightarrow GF(2^b)$, an input XOR-difference $\alpha \in GF(2^a)$, and output XOR difference $\beta \in GF(2^b)$ the XOR-differential probability (*XDP*)for the function f with differences α and β is defined to be

$$\begin{aligned} XDP_f(\alpha, \beta) &\stackrel{\text{def}}{=} Pr\left(f(x') \oplus f(x'') == \beta | x' \oplus x'' == \alpha\right) \\ &= \frac{|\{x', x'' \text{ such that } \Delta_{\oplus}(x', x'') == \alpha \text{ and } \Delta_{\oplus}(f(x'), f(x'')) == \beta\}|}{2^a}. \end{aligned}$$

The XDP Table for f is the $2^a \times 2^b$ table for which entry (α, β) contains $XDP_f(\alpha, \beta)$.

Suppose we are hashing two messages M' and M''. For all the working variables and temporary variables we will place one prime (E.g. R') to the value of that variable when computing M' and place two primes (E.g. R'') to the value of that variable when computing M'' Let x_k denote the 4-bit value obtained by concatenating $R[k]$, $S[k]$, $T[k]$ and $U[k]$, and define $f(x_k)$ be the function with 4-bit inputs and for which the 3-bit output value is obtained by concatenating the corresponding outputs of MAP_0, MAP_1, MAP_2.

The number of bit differences in X, Y, Z will depend on the number of bit differences resulting in each bit position. If $f(x'_k) == f(x''_k)$, then there are no bit differences output at that bit position. If $f(x'_k) == f(x''_k)$ at all 64 bit positions, then $(X', Y', Z') == (X'', Y'', Z'')$. This then means that $(AA', DD') == (AA'', DD'')$ in the case of small CHI, and $(AA', DD', GG') == (AA'', DD'', GG'')$ in the case of big CHI. All these cases correspond to having $\Delta_\oplus == 0$, highlighting that we should minimize the chances of resulting in $\Delta_\oplus == 0$. Note that if $\Delta_\oplus(x'_k, x''_k) = \alpha = 0$, then $x'_k == x''_k$ and thus $f(x'_k) == f(x''_k)$: hence the first row of the table contains a probability 1 in entry $(0, 0)$, and probability zero everywhere else. This cannot be avoided. (Note, from here on, the subscript of k is implicit).

Since we are using a function $GF(2^4) \rightarrow GF(2^3)$, there must be pairs of inputs x', x'' with $x' \neq x''$ such that $f(x') = f(x'')$ (that is, for which $\Delta_\oplus(f(x'), f(x'')) == 0$) The corresponding entries $(\alpha, 0)$ are in the first column of the XDP table. We want to minimize the chances of resulting in $\Delta_\oplus(f(x'), f(x'')) == 0$.

The situations in which differential attacks are most potent are where there are only a few bit differences present in the algorithm state. This tends to result in functions having input pairs with single bit differences. For this reason, the MAP function of the CHI algorithms have been chosen with particular emphasis on the properties when input pairs have single-bit differences.

Firstly, would like to prevent $\Delta_\oplus(f(x'), f(x'')) == 0$ in the case where x' and x'' differ in a single bit (that is, when $|\Delta_\oplus(x', x'')| = 1$. This is property is already stated in MAP Requirement 2 (Section 13.2).

We tried looking for functions with the property that if $|\Delta_\oplus(x', x'')| = 2$, the $\Delta_\oplus(f(x'), f(x'')) \neq 0$. However, the only such functions for which this property hold are also functions in which the outputs are a linear (degree one) expression of the inputs. Consequently, we abandoned this requirement.

The next property is a bit difficult to explain in full detail, but it can be summarized thus. When two pairs $f(x'), f(x'')$ differ in some bits, then this means that X', Y', Z' will differ from X'', Y'', Z'' in some bits. Recall that the modifications XX, YY, ZZ of X, Y, Z are combined using the modular addition operation. Suppose we have an addition of two values (e.g.$c := a \boxplus b$), and we input a', a'' and b', b'' have say λ-bit differences shared between them. For reasons that we don't discuss here, it is generally true that the attacker can predict the $\Delta_\oplus(c', c'')$ with probability at best $2^{-\lambda}$. In other words, the larger $|\Delta_\oplus(a', a'')|$ and $|\Delta_\oplus(b', b'')|$ are, the more difficult it becomes for the attacker to predict $\Delta_\oplus(c', c'')$. In the same way, the larger the number of bit differences in XX, YY, ZZ, the more difficult it becomes for an attacker to predict the bit differences in AA, DD (and GG in the case of big CHI).

***MAP* Requirement 7** Good XOR-differential properties for single-bit input differences. If two inputs to f have XOR-difference α with $|\alpha| = 1$, and β_1, β_2 and β_3 are output XOR differences

with $|\beta_1| = 1$, $|\beta_2| = 2$ and $|\beta_3| = 3$, then we require

$$XDP_f(\alpha, \beta_1) \ \leq \ \frac{1}{2} \cdot XDP_f(\alpha, \beta_2) \ \leq \ \frac{1}{4} \cdot XDP_f(\alpha, \beta_3).$$

For differences of more than one bit, we wanted to bound the probabilities.

***MAP* Requirement 8** Unless otherwise bounded, for non-zero α, $XDP_f(\alpha, \beta) \leq \frac{6}{16}$.

Nabla Differentials The next set of properties relate to nabla differences, which are used in techniques like that of Wang *et al* [69]. The motivation for nabla differentials is a bit complex: feel free to continue reading if you have some experience in cryptanalysis, but otherwise you may like to skip to the next section.

For two values a', a'' that a single bit may take, the nabla difference $\nabla(a', a'')$ between the two bits can has one of three values:

$$\nabla(a', a'') \ := \ \begin{cases} \texttt{+} & \text{when } a' == 0,\ a'' == 1; \\ \texttt{-} & \text{when } a' == 1,\ a'' == 0; \\ \texttt{*} & \text{when } a' == a''. \end{cases}$$

Note that $\Delta_{\oplus}(a', a'') = 1$ if and only if $\nabla(a', a'') = \texttt{+}$ or $\nabla(a', a'') = \texttt{-}$; while $\Delta_{\oplus}(a', a'') = 0$ if and only if $\nabla(a', a'') = \texttt{*}$.

Nabla differences are useful when looking at interactions between bit-slice operations (such as XOR and the *MAP*). The reason will be explained in more detail in Section 17.

A *nabla-differential* between the input $x \in GF(2^a)$ and the output $f(x) \in GF(2^b)$ of a function is an expression of the form

$$\Delta_{\oplus}(x', x'') == \alpha, \ \text{AND } \Delta_{\oplus}(f(x'), f(x'')) == \beta$$

where $\alpha \in \{+, -, \cdot\}^a$, and $\beta \in \{+, -, \cdot\}^a$ are known as *nablas*.

The attack is trying to find conditions on the internal variables of the step function that will result in certain XOR differences in the working variables after a certain number of steps. Nabla differentials give the attacker more control over the differences in the internal variables. The attacker wants to control the nabla differentials for as many steps as possible. The attacker can only control the nabla differentials by controling the conditions on the internal variables of the step function. The attacker is only able to control the conditions on the internal variables for as many steps as the attacker is able control both the conditions on the internal variables and the step inputs: that is, for as many steps as the step inputs are independent of the working variables. The step function uses two word64s per step. The message block for CHI-224 and CHI-256 contain 8 word64s. After $8/2 = 4$ steps of CHI-224 and CHI-256, step inputs are no longer independent of the working variables. Similarly, after $16/2 = 8$ steps of CHI-384 and CHI-512, the step inputs are no longer independent of the working variables.

The attacker is only able to control the conditions on the internal variables for 4 steps (CHI-224 and CHI-256) and or 8 step (CHI-384 and CHI-512). After these 4 (or 8) steps, the attacker predicts the propagation probabilistically. Since the step function is dominated by XOR operations, XOR-differentials are the best way to follow the propagation of differences.

In looking for a differential path, the first task is to find an XOR-differential from the 4-th (or 8-th) step to the last step, such that this XOR differential holds with high probability p. The

second task is to find a nabla-differential and a set of conditions that will result in the correct XOR-differences after the 4-th (or 8-th) step.

The attacker will then search through the set of message blocks that provide the correct nabla differential in order to get the correct XOR-differences after the 4-th (or 8-th) step. The attacker hopes that one of these message blocks will result in the correct XOR differences in the output of the last step. For each message block that provides the correct nabla differential, the probability of getting the correct XOR differences in the output of the last step is p. If the set of message blocks is independent, then the approximately $p/2$ message blocks are tested before obtaining the correct XOR difference in the output of the last step.

However, it is possible to generate sets of related message blocks so that if one message block comes close to getting the correct XOR difference, then the other message blocks in that set are more likely to have the correct XOR difference in the last step. Such techniques exploit neutral bits [21], tunnels [40] and auxiliary differentials [36]. These techniques all rely on the nabla differential imposing few conditions on the internal variables and step inputs in the first 4 (or 8) steps. If the nabla differential imposes many conditions on the internal variables and step inputs in the first 4 (or 8) steps, then these techniques may not apply. Furthermore, if the nabla differential imposes sufficient conditions, it may not be possible to get the desired XOR-difference after the 4-th (or 8-th) step, so the attacker must use a lower-probability XOR differential.

This provides the motivation behind our final requirement on the *MAP* function. We desired that the *MAP* function would impose the maximum possible set of conditions on the step input and working variables.

Consider the *MAP* inputs and *MAP* outputs at a chosen bit position. If there are no bit differences in the *MAP* inputs (that is, the nabla difference is `****`), then there must be no bit differences in the *MAP* outputs: and no conditions on the step inputs and working variables are imposed. If there is a `+` or `-` nabla difference in the *MAP* inputs, then the *MAP* outputs have the possibility of being any of `+,-,*`. The maximum conditions are imposed if, given nabla differential, there is at most one pair of inputs satisfying that nabla differential We particularly want this property to hold when there is only a single nabla difference in the inputs.

***MAP* Requirement 9** For all nabla differentials for *MAP* with a single bit difference, there is at most one pair of inputs satisfying the differential.

14.1.3 Searching for a MAP function

There are only $2^{16} = 65536$ functions with 4 input bits and 1 output bit. There are $(2^{16})^3 = 2^{48}$ functions with 4 input bits and 3 output bits, which is beyond exhaustive testing using a naïve search. However, by appropriately filtering the set of functions from 4 input bits and 1 output bit using the requirements, we were able to do an exhaustive search to find the *MAP* function specified in Section 4.1. To obtain the representation shown in Section 4.1, a Karnaugh MAP application [41] was used. The CHI team members then did manual optimization to obtain the expressions used in the implementations.

14.2 Desired Properties of the *DATA-INPUT* phase

The purpose of the *DATA-INPUT* phase is to combine the step inputs and the step constants into the computation. There are two options under consideration in the *DATA-INPUT* phase:

- *Expanding the step inputs and step constants into four word64s.*
- *Operation for combining the result in to the computation.* This is considered in Section 14.2.2.

14.2.1 Expanding the Step Inputs and Step Constants

The initial set of requirements on the rotations for θ_0 and θ_1 was that no two bits of AA, DD (and GG for CHI-384 and CHI-512) may depend on the same pair of bits from the step inputs. This should ensure good diffusion. These requirements interact with the choice of operations in he *POST-MIXING*. However, there were no set of rotations that satisfied this set of requirements. We changed our goal to minimizing the number of cases in which two bits of AA, DD (and GG for CHI-384 and CHI-512) depend on the same pair of bits from the step inputs. The θ_0 and θ_1 functions were choices of the solution set that had an approximately balanced number of even and odd rotations.

14.2.2 Choice of Combining Operation in the *DATA-INPUT* phase

Operations are required for combining step inputs and the step constants with the temporary values R, S, T, U. The CHI team considered both the XOR operation and addition operation in deciding on a suitable operation for the *DATA-INPUT* . The following points were considered.

- Addition operations might provide a small security advantage (when compared to using XOR operations), but it did not seem that addition operations would provide a significant security advantage.
- Both addition and XOR operations would have the same cost in terms of software efficiency.
- Addition operations would contribute a small increase in hardware latency in comparison to using XOR operations.
- Addition operations would result in a significant increase in hardware gatecount in comparison to using XOR operations.

In all these points, except the last, there was no significant difference between the XOR and addition operations. However, the last point indicated that the addition operation would be more costly in hardware. This suggests that the XOR operation is the better option. Consequently, the CHI team decided to use the XOR operation.

14.3 The *PRE-MIXING* and *POST-MIXING* Phases

The main purpose of the *PRE-MIXING* and *POST-MIXING* phases is to provide diffusion.

Sections 13.3.1 and 13.3.2 explain why the *PRE-MIXING* forms the *MAP* inputs by XORing modified copies of the working variables. In the case of CHI-224 and CHI-256, these modifications uses *SWAP*32 and rotations are in blocks of 32-bits, while in the case of CHI-384 and CHI-512, these modifications are rotations are in blocks of 64-bits.

Sections 13.5.1 and 13.5.2 have already discussed requirements to achieve optimal diffusion within the *POST-MIXING* phase. For the *POST-MIXING* phase, only the rotation amounts $ryu, ryl, r1, r2$ (for CHI-224 and CHI-256) and the rotation amounts $ry, r1, r2, r3$ (for CHI-384 and CHI-512) remain undetermined after that analysis.

14.3.1 The *PRE-MIXING* Phase of CHI-224 and CHI-256

The modifications uses $SWAP32$ and rotations are in blocks of 32-bits, as discussed in Section 13.3.1. Three copies of the A and D working variables are used and single copies of B and E are used.

We selected half of the copies to have the $SWAP32$ applied. For each copy of the working variables, we allowed the upper and lower word32s to be rotated by independent amounts. Two of the rotation amounts applied to copies of A were specified to be zero, which mean that no modification would be applied in that case. This left $7 \times 2 = 14$ rotation amounts to specify.

For CHI-224 and CHI-256, the value in working variable A goes through the following *life-cycle*. In the fist step, three copies of the value are input to the *PRE-MIXING* phase using the modifications specified for A. In the next step, one copy of the value is input to the *PRE-MIXING* phase using the modifications specified for B. In the next step, the value is not input to the *PRE-MIXING* phase, but the value is XORed with DD which is an output from the *POST-MIXING* phase. In the next step, three copies of the value are input to the *PRE-MIXING* phase using the modifications specified for D. In the next step, one copy of the value is input to the *PRE-MIXING* phase using the modifications specified for E. In the next step, the value is not input to the *PRE-MIXING* phase, but the value is XORed with AA which is an output from the *POST-MIXING* phase. In the next step, the value starts the life-cycle again. A value in any of the other working variables goes through the same life-cycle, although the value starts at a different place in the life-cycle.

To maximize diffusion, we required no two bits of the working variables should interact in the MAP phase more than once within those bits' lifecycle. We applied this requirement to the*PRE-MIXING* Phase, and this results in 68 conditions on the 14 unspecified rotation amounts in the *PRE-MIXING* Phase. The set of requirements is too complex to list here. There were many combinations that satisfied this set of requirements, so the next step was to consider the interaction of the *PRE-MIXING* and *POST-MIXING* phases of CHI-224 and CHI-256

14.3.2 Interaction of the *PRE-MIXING* and *POST-MIXING* Phases of CHI-224 and CHI-256

In this section we model addition as bit-wise XOR.

The value of bit $Y[i]$ (which bears some relation to the values of $X[i]$ and $Z[i]$) gets added to other bits of X and rotated to form a bit of AA, and also gets added to a bits of Z to form a bit DD. The bit of AA is XORed with a bit of F to for a bit of the new value for A. Similarly, the bit of DD is XORed with a bit of C to for a bit of the new value for D. The bits of A and D then go through their life-cycle of 6 steps. During that life-cycle, that original bit of information $Y[i]$ ends up being combined with other bits that can be traced back to that A and D: being combined either during the *PRE-MIXING*, or in the *MAP* function (which combines bits in the same bit position of the *MAP* inputs). Those other bits of A and D can be traced back to X, Y and Z that were computed during the same step as the original bit of information $Y[i]$. If two bits of (for example) X and Y end up being combined together during the *PRE-MIXING*, or in the *MAP* function, then we say that those bits *loosely interact*.

Suppose that, during the lifecycle two of those bits from A and D were to loosely interact with each other. This would mean that the diffuse is not very successful, since both the bits would depend directly on $Y[i]$. To prevent this, we have the requirement that, for the length of a life-cycle, pairs of bits of A and D bits that depend directly on $Y[i]$ should not loosely interact with each

other.

We would like to apply even stricter requirements, since $X[i], Y[i]$ and $Z[i]$ are all computed from the same four input bits of R, S, T, U and are therefore related. Consequently, we have the requirement that, for the length of a life-cycle, pairs of bits of A and D that depend directly on $X[i]$ or $Y[i]$ or $Z[i]$ should not loosely interact with each other.

We have 14 rotations in the *PRE-MIXING* left to specify, and the 4 rotation amounts ryu, ryl, $r1$, $r2$ in the *POST-MIXING* left to specify. For simplicity, we assumed that $ryu = ryl$. The requirements on the*PRE-MIXING* Phase resulted in 68 conditions on the 14 rotations in the *PRE-MIXING*. The requirement above resulted in another large set of conditions. Of particular note: applying rotation and *SWAP*8 to Y when forming YY resulted in a large number of conditions since *SWAP*8 looks like a set of rotations, where each byte has been rotated to another part of the word64 by either 8, 24, 40 or 56 bit positions.

The number of conditions was reduced significantly if it was assumed that $(r1 - r2)$ is a multiple of 16. To further ensure good diffusion, we enforced a minimum distance of 4 positions between rotations of within each word32 of A. We also enforced this minimum distance between rotations of within each word32 of D. This set of requirements left many options for the rotations applied to B and E: to make the rotations as efficient as possible on 8-bit processors, we limited the rotations for B and E to multiples of eight.

This set of requirements resulted in millions of options for the combination of ryu, ryl and the 14 rotations in the *PRE-MIXING*; from which we selected the combination seen in Section 7.1.2.

The *POST-MIXING* already had the *SWAP*8 and *SWAP*32 kept bits localized within each word32 half of the word64 values, so we choose rotation values of $r1 = 16$ and $r2 = 48$ to move information back across the word32 boundaries. Keeping these values as multiples of 16 makes the operations almost free on 8-bit and 16-bit processors.

14.3.3 The *PRE-MIXING* Phase of CHI-384 and CHI-512

The modifications uses *SWAP*32 and rotations are in blocks of 32-bits, as discussed in Section 13.3.2. Three copies of the A, D and G working variables are used and single copies of B, E and P are used. One of rotation amounts applied to copies of A were specified to be zero, which mean that no modification would be applied in that case. This left 11 rotation amounts to specify.

For CHI-384 and CHI-512, the value in working variable goes through the following life-cycle of 9 steps, similar to the 6-step life-cycle of CHI-224 and CHI-256. To maximize diffusion, we again required no two bits of the working variables should interact in the MAP phase more than once within those bits' life-cycle. We applied this requirement to the*PRE-MIXING* Phase, and this results in 75 conditions on the 11 unspecified rotation amounts in the *PRE-MIXING* Phase. The set of requirements is too complex to list here. There were many combinations that satisfied this set of requirements, so the next step was to consider the interaction of the *PRE-MIXING* and *POST-MIXING* phases of CHI-384 and CHI-512.

14.3.4 Interaction of the *PRE-MIXING* and *POST-MIXING* Phases of CHI-384 and CHI-512

The requirements for the interaction of these phases are constructed in the same manner as for CHI-224 and CHI-256. The differences are: CHI-384 and CHI-512 uses rotations in 64-bit blocks, CHI-384 and CHI-512 have 9 working variables (rather than 6) and CHI-384 and CHI-512 have an

extra feedback. Once again, this generated a large set of conditions, and this set of conditions was reduced significantly if it was assumed that both $(r1 - r2)$ and $(r1 - r3)$ are multiples of 16.

To further ensure good diffusion, we enforced a minimum distance of 11 positions between rotations of within each word32 of A (similarly for D and G). We found that it was possible to have the rotation for P to be zero, which mean that no modification would be applied in that case. This set of requirements resulted in 241128 options for the combination of ry and the 11 rotations in the *PRE-MIXING*; from which we selected the final combination (see Section 7.1.2).

Finally, CHI-384 and CHI-512 kept the rotation values of $r1 = 16$ and $r2 = 48$ from CHI-224 and CHI-256 (applied to for the outputs AA and DD), and added $r3 = 0$ (applied to the output GG), which incurs no additional cost.

14.4 Desired Properties of the *FEEDBACK* phase

The purpose of the *FEEDBACK* is to combine the output of the *POST-MIXING* with the appropriate state variables. The only option under consideration in the *FEEDBACK* phase is the choice of operation for combining the output of the *POST-MIXING* with the step inputs, step constants and state variables. The CHI team considered both the XOR operation and addition operation in decide on a suitable operation. The influencing factors were identical to those considered in the *DATA-INPUT* phase (Section 14.2.2). As with the *DATA-INPUT* phase, the CHI team decided to use the XOR operation.

14.5 Design Rationale for Message Expansion MICRO Structure

The main requirements on the message expansion are: good diffusion of bit information; and the prevention of small cycles.

Diffusion. Good diffusion of bit information can be achieved by requiring that the rotation amounts be a large distance apart, so that information is being spread as widely as possible.

Preventing Small Cycles. Since the message expansion feedback function is linear, the process of updating the state can be viewed as the $GF(2)$ multiplication of a fixed matrix ME with a vector representing the message state. The cycle length of the message state achieves the maximum of $(2^m - 1)$ when the characteristic polynomial of the matrix ME is a primitive polynomial.

To allow circuitry re-use, we further required that $\mu_0^{\{256\}}$ and $\mu_0^{\{512\}}$ be identical.

We searched for combinations of rotation and shift amounts satisfying these constraints, and used the first combination that we found. This resulted in the functions specified in Section 4.2.

The characteristic polynomial for the message expansion used in CHI-224 and CHI-256 is:

$$x^{512} + x^{492} + x^{486} + x^{482} + x^{479} + x^{478} + x^{475} + x^{473} + x^{472} + x^{471} + x^{467} + x^{465} + x^{464} + x^{462} + x^{460} + x^{459} + x^{453} + x^{450} + x^{447} + x^{442} + x^{440} + x^{429} + x^{428} + x^{425} + x^{422} + x^{421} + x^{420} + x^{419} + x^{418} + x^{415} + x^{414} + x^{413} + x^{410} + x^{409} + x^{406} + x^{405} + x^{401} + x^{397} + x^{396} + x^{395} + x^{393} + x^{391} + x^{389} + x^{388} + x^{386} + x^{383} + x^{381} + x^{380} + x^{379} + x^{378} + x^{376} + x^{375} + x^{374} + x^{370} + x^{360} + x^{356} + x^{353} + x^{352} + x^{351} + x^{350} + x^{344} + x^{342} + x^{340} + x^{337} + x^{332} + x^{328} + x^{327} + x^{324} + x^{323} + x^{322} + x^{320} + x^{319} + x^{318} + x^{317} + x^{315} + x^{314} + x^{312} + x^{311} + x^{310} + x^{309} + x^{305} + x^{303} + x^{302} + x^{300} + x^{299} + x^{298} + x^{296} + x^{295} + x^{294} + x^{291} + x^{288} + x^{286} + x^{285} + x^{284} + x^{282} + x^{280} + x^{278} + x^{277} + x^{276} + x^{273} + x^{272} + x^{267} + x^{264} + x^{262} + x^{260} + x^{258} + x^{256} + x^{254} + x^{253} + x^{250} + x^{246} + x^{245} + x^{243} + x^{242} + x^{241} + x^{239} + x^{237} + x^{235} + x^{233} + x^{231} + x^{225} + x^{224} + x^{223} + x^{221} + x^{220} + x^{218} + x^{217} + x^{216} + x^{213} + x^{212} + x^{209} + x^{205} + x^{203} + x^{201} + x^{198} + x^{197} + x^{194} + x^{193} + x^{188} + x^{183} + x^{182} + x^{181} + x^{179} + x^{178} + x^{176} + x^{172} + x^{170} + x^{168} + x^{165} + x^{164} + x^{163} + x^{162} + x^{159} + x^{157} + x^{151} + x^{149} + x^{148} + x^{145} + x^{144} + x^{141} + x^{140} + x^{139} + x^{137} + x^{136} + x^{135} + x^{133} + x^{132} + x^{131} + x^{130} + x^{128} + x^{126} + x^{124} + x^{123} + x^{122} + x^{121} + x^{120} + x^{119} + x^{117} + x^{113} + x^{111} + x^{110} + x^{108} + x^{103} + x^{102} + x^{101} + x^{100} + x^{98} + x^{97} + x^{96} + x^{95} + x^{93} + x^{92} + x^{91} + x^{90} + x^{88} + x^{87} + x^{85} + x^{84} + x^{82} + x^{81} + x^{80} + x^{78} + x^{77} + x^{76} + x^{74} + x^{73} + x^{71} + x^{67} + x^{65} + x^{63} + x^{59} + x^{58} + x^{55} + x^{51} + x^{49} + x^{48} + x^{47} + x^{43} + x^{41} + x^{40} + x^{39} + x^{38} + x^{36} + x^{35} + x^{34} + x^{32} + x^{31} + x^{29} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{21} + x^{17} + x^{11} + 1.$$

The characteristic polynomial for the message expansion used in CHI-384 and CHI-512 is:

$$x^{1024} + x^{988} + x^{987} + x^{971} + x^{966} + x^{960} + x^{959} + x^{958} + x^{955} + x^{954} + x^{953} + x^{952} + x^{947} + x^{945} + x^{942} + x^{940} + x^{939} + x^{937} + x^{936} + x^{934} + x^{931} + x^{928} + x^{927} + x^{923} + x^{920} + x^{918} + x^{917} + x^{916} + x^{913} + x^{910} + x^{907} + x^{901} + x^{898} + x^{896} + x^{895} + x^{894} + x^{893} + x^{890} + x^{889} + x^{888} + x^{885} + x^{884} + x^{883} + x^{882} + x^{881} + x^{879} + x^{878} + x^{877} + x^{875} + x^{874} + x^{873} + x^{872} + x^{870} + x^{868} + x^{867} + x^{866} + x^{863} + x^{860} + x^{859} + x^{858} + x^{857} + x^{855} + x^{853} + x^{851} + x^{850} + x^{848} + x^{846} + x^{843} + x^{841} + x^{838} + x^{837} + x^{836} + x^{833} + x^{831} + x^{829} + x^{826} + x^{824} + x^{823} + x^{822} + x^{819} + x^{817} + x^{816} + x^{814} + x^{810} + x^{809} + x^{804} + x^{803} + x^{801} + x^{797} + x^{796} + x^{794} + x^{791} + x^{790} + x^{788} + x^{787} + x^{786} + x^{782} + x^{781} + x^{780} + x^{777} + x^{776} + x^{775} + x^{774} + x^{773} + x^{772} + x^{771} + x^{770} + x^{768} + x^{765} + x^{761} + x^{760} + x^{756} + x^{755} + x^{754} + x^{751} + x^{750} + x^{749} + x^{746} + x^{741} + x^{733} + x^{730} + x^{729} + x^{728} + x^{727} + x^{726} + x^{724} + x^{723} + x^{722} + x^{720} + x^{719} + x^{718} + x^{715} + x^{713} + x^{710} + x^{709} + x^{706} + x^{705} + x^{703} + x^{702} + x^{697} + x^{696} + x^{694} + x^{693} + x^{691} + x^{690} + x^{688} + x^{687} + x^{684} + x^{683} + x^{682} + x^{681} + x^{680} + x^{675} + x^{674} + x^{673} + x^{672} + x^{671} + x^{668} + x^{666} + x^{665} + x^{663} + x^{660} + x^{653} + x^{652} + x^{651} + x^{650} + x^{648} + x^{647} + x^{641} + x^{640} + x^{638} + x^{637} + x^{636} + x^{633} + x^{632} + x^{628} + x^{625} + x^{624} + x^{623} + x^{622} + x^{621} + x^{618} + x^{616} + x^{615} + x^{614} + x^{613} + x^{612} + x^{609} + x^{608} + x^{607} + x^{605} + x^{602} + x^{601} + x^{600} + x^{599} + x^{598} + x^{593} + x^{592} + x^{591} + x^{590} + x^{588} + x^{587} + x^{586} + x^{585} + x^{582} + x^{581} + x^{580} + x^{578} + x^{577} + x^{576} + x^{575} + x^{574} + x^{573} + x^{572} + x^{571} + x^{569} + x^{568} + x^{567} + x^{566} + x^{565} + x^{564} + x^{563} + x^{559} + x^{558} + x^{553} + x^{552} + x^{546} + x^{544} + x^{543} + x^{539} + x^{537} + x^{536} + x^{532} + x^{531} + x^{527} + x^{526} + x^{524} + x^{523} + x^{522} + x^{521} + x^{519} + x^{517} + x^{515} + x^{511} + x^{509} + x^{508} + x^{505} + x^{504} + x^{503} + x^{502} + x^{501} + x^{498} + x^{497} + x^{496} + x^{494} + x^{493} + x^{492} + x^{490} + x^{488} + x^{485} + x^{484} + x^{483} + x^{482} + x^{478} + x^{477} + x^{476} + x^{472} + x^{471} + x^{470} + x^{468} + x^{467} + x^{465} + x^{462} + x^{460} + x^{458} + x^{454} + x^{453} + x^{450} + x^{449} + x^{448} + x^{447} + x^{438} + x^{436} + x^{434} + x^{428} + x^{423} + x^{420} + x^{419} + x^{418} + x^{417} + x^{416} + x^{415} + x^{410} + x^{408} + x^{407} + x^{406} + x^{405} + x^{404} + x^{402} + x^{400} + x^{398} + x^{395} + x^{394} + x^{393} + x^{392} + x^{389} + x^{387} + x^{386} + x^{385} + x^{384} + x^{383} + x^{381} + x^{380} + x^{379} + x^{378} + x^{376} + x^{371} + x^{370} + x^{369} + x^{368} + x^{367} + x^{366} + x^{364} + x^{363} + x^{362} + x^{361} + x^{360} + x^{359} + x^{357} + x^{355} + x^{354} + x^{352} + x^{350} + x^{348} + x^{346} + x^{345} + x^{343} + x^{340} + x^{339} + x^{337} + x^{333} + x^{330} + x^{328} + x^{326} + x^{324} + x^{321} + x^{319} + x^{318} + x^{317} + x^{315} + x^{314} + x^{313} + x^{312} + x^{311} + x^{310} + x^{307} + x^{306} + x^{304} + x^{303} + x^{302} + x^{299} + x^{298} + x^{296} + x^{293} + x^{290} + x^{288} + x^{287} + x^{285} + x^{284} + x^{280} + x^{278} + x^{276} + x^{275} + x^{272} + x^{270} + x^{269} + x^{268} + x^{264} + x^{262} + x^{260} + x^{259} + x^{258} + x^{257} + x^{255} + x^{254} + x^{251} + x^{249} + x^{248} + x^{245} + x^{239} + x^{238} + x^{237} + x^{236} + x^{235} + x^{234} + x^{233} + x^{227} + x^{225} + x^{224} + x^{222} + x^{219} + x^{217} + x^{214} + x^{211} + x^{210} + x^{207} + x^{206} + x^{202} + x^{200} + x^{199} + x^{195} + x^{194} + x^{193} + x^{192} + x^{191} + x^{190} + x^{188} + x^{187} + x^{185} + x^{181} + x^{179} + x^{176} + x^{175} + x^{174} + x^{173} + x^{170} + x^{169} + x^{167} + x^{165} + x^{162} + x^{160} + x^{157} + x^{155} + x^{154} + x^{153} + x^{152} + x^{146} + x^{145} + x^{144} + x^{140} + x^{139} + x^{138} + x^{135} + x^{133} + x^{131} + x^{130} + x^{128} + x^{126} + x^{123} + x^{122} + x^{121} + x^{118} + x^{117} + x^{115} + x^{113} + x^{111} + x^{108} + x^{104} + x^{102} + x^{101} + x^{100} + x^{99} + x^{98} + x^{96} + x^{95} + x^{92} + x^{83} + x^{79} + x^{77} + x^{75} + x^{73} + x^{67} + x^{61} + x^{59} + x^{47} + x^{45} + x^{37} + x^{29} + 1.$$

15 Future Tweaks

This section mentions some further ideas that we considered, but we are still undecided as to whether these ideas was worthwhile.

15.1 Step Constants

A large portion of an ASIC implementation is currently used to store the step constants. We plan to consider having a function for generating the step constants. The function can be slow in software, since most software implementations can easily store the constants. However, it wold be good for the function to be fast in software.

15.2 Making the Hash Output Value Dependent on Intended Length

We began considering some ideas for making the hash output value dependent on the intended length. These ides included:

- *Prepending Intended Digest Length to Message.* This could make padding un-necessarily complicated.
- *Making the Compression Function Dependent on Intended length.* For example, the compression function could add a length-dependent constant to the input hash prior to applying the step functions. There was some concern that this might introduce undesirable scenarios for differential attacks (the difference could be injected via the length-dependent constant).
- *Making the Step Function Dependent on Intended length.* For example, the step constants could change according to the intended length of the output.

It was not clear if this property is sufficiently useful to warrant the cost. We will continue evaluating options.

15.3 Position Dependency

If the operation of the compression function was made dependent on the index of the message block, then this would prevent attacks using fixed points (since a fixed point would only apply to a given message block index). Our algorithm already includes countermeasures against fixed point attacks, so it is not clear if there is an advantage to making the compression function dependent on the index of the message block. We need to be careful that the introduced countermeasure does not enable new attacks.

The techniques that we considered for achieving this property included:

- *Increase the size of the message state, and include the message block index into the message state.* This approach seems secure. However, this approach would increase the number of steps required in the step function, having a significant impact on the efficiency.
- *XOR the message index with the input hash value.* This seems efficient, but requires further analysis.

Once again, it was not clear if this property is sufficiently useful to warrant the cost. We will continue evaluating options.

15.4 A Variation when including Secret Data

When incorporating secret information that is not under the control of an attacker, then it might make sense to forget the θ_0 and θ_2 functions used to expand the two step inputs, and instead use four step inputs per step. This would require the message expansion to produce twice as many outputs, but may result in better security for PRF-like applications.

Part III
Cryptanalysis

16 Introduction to Cryptanalysis of the CHI algorithms

This part of the documentation discusses those aspects of the cryptanalysis of the CHI algorithms that are are not discussed in the Design Rationale (Part II). The padding and parsing need no cryptanalysis, since CHI follows standard approach to padding and parsing. The modified Merkle-Dåmgard construction and the modified Davies-Meyer construction are analyzed sufficiently in Section 12.1 and Section 12.2, so the Input/Output Formatting and META Structure are not discussed further in Part III.

The components of the step function are analyzed in Section 17. Differential paths through multiple steps of the CHI algorithms are discussed in 18. The relationship between XOR differential paths and second preimage attacks is discussed in 19 The message expansion is analyzed in Section 20 to

17 Analysis of the Step Function Components

17.1 Definitions

Some of these definitions have been introduced elsewhere, but it is useful to clarify the definitions here.

17.1.1 XOR Differentials

An *XOR-differential* between the input $x \in GF(2^a)$ and the output $f(x) \in GF(2^b)$ of a function is an expression of the form

$$x' \oplus x'' == \alpha \rightarrow f(x') \oplus f(x'') == \beta,$$

where $\alpha \in GF(2^a)$, and $\beta \in GF(2^b)$ are known as *XOR differences*.

For a function $f : GF(2^a) \rightarrow GF(2^b)$, an input XOR-difference $\alpha \in GF(2^a)$, and output XOR difference $\beta \in GF(2^b)$ the XOR-differential probability (*XDP*)for the function f with differences α and β is defined to be

$$\begin{aligned} XDP_f(\alpha, \beta) &\stackrel{\text{def}}{=} Pr\left(f(x') \oplus f(x'') == \beta | x' \oplus x'' == \alpha\right) \\ &= \frac{|\{x', x'' \text{ such that } \Delta_{\oplus}(x', x'') == \alpha \text{ and } \Delta_{\oplus}(f(x'), f(x'')) == \beta\}|}{2^a}. \end{aligned}$$

The XDP Table for f is the $2^a \times 2^b$ table for which entry (α, β) contains $XDP_f(\alpha, \beta)$. If two XOR-differentials are independent, then the XDP of the combination of the XOR-differentials is the product of the XDPs of the individual XOR-differentials.

17.1.2 Nabla Differentials

Nabla differences, which are used in analyses like that of Wang *et al* [69]. For two values a', a'' that a single bit may take, the nabla difference $\nabla(a', a'')$ between the two bits can has one of three values:

$$\nabla(a', a'') := \begin{cases} \texttt{+} & \text{when } a' == 0,\ a'' == 1; \\ \texttt{-} & \text{when } a' == 1,\ a'' == 0; \\ \texttt{*} & \text{when } a' == a''. \end{cases}$$

Note that $\Delta_{\oplus}(a', a'') = 1$ if and only if $\nabla(a', a'') =$ `+` or $\nabla(a', a'') =$ `-`; while $\Delta_{\oplus}(a', a'') = 0$ if and only if $\nabla(a', a'') =$ `*`

An *Nabla-differential* between the input $x \in GF(2^a)$ and the output $f(x) \in GF(2^b)$ of a function is an expression of the form

$$\nabla(x', x'') == \alpha \rightarrow \nabla(f(x'), f(x'')) == \beta$$

where $\alpha \in \{+, -, \cdot\}^a$, and $\beta \in \{+, -, \cdot\}^a$ are known as *nablas*.

17.1.3 Linear Approximations

A *linear approximation* between the input $x \in GF(2^a)$ and the output $f(x) \in GF(2^b)$ of a function is an expression of the form

$$\alpha \cdot x + \beta \cdot f(x) = \text{constant},$$

the addition implicitly denotes reduction modulo 2, and $\alpha \in GF(2^a)$, and $\beta \in GF(2^b)$ are known as *masks*, since they select bits to be included in the equation.

For a function $f : GF(2^a) \rightarrow GF(2^b)$, an input mask $\alpha \in GF(2^a)$, and output mask $\beta \in GF(2^b)$ the Linear Approximation Bias (LAB)for the function f with masks α and β is defined to be

$$\begin{aligned} XDP_f(\alpha, \beta) &\stackrel{\text{def}}{=} 2 \cdot Pr\left(\alpha \cdot x + \beta \cdot f(x) == 0\right) - 1 \\ &= 2 \cdot \frac{|\{x', x'' \text{ such that } \alpha \cdot x + \beta \cdot f(x) == 0\}|}{2^a} - 1\,. \end{aligned}$$

Note that the LAB (or simply "bias") is a value between -1 and +1. The LAB Table for f is the $2^a \times 2^b$ table for which entry (α, β) contains $LAB_f(\alpha, \beta)$.

If two linear approximations are independent, then the bias has been so defined that the bias of the combination of the linear approximations is the product of the biases of the individual linear approximations. This follows form the so-called "Piling-Up Lemma" of Matsui [45]. the

17.2 Analysis of the *MAP*

17.2.1 XOR-Differentials for the *MAP* Function

The XOR Differential Probability (XDP) table for the MAP function is shown in Table 11. Note that the probabilities satisfy the requirements in Sections 13.2 and 14.1.

The only XOR Differential that causes us some concern is the entry (`9`,`4`) which indicates a 2-bit input difference that results in a 1-bit output difference with probability $\frac{6}{16}$. The other XOR Differentials with probability $\frac{6}{16}$ correspond to XOR differentials

- (`7`,`2`): 3-bit input difference, 1-bit output difference.
- (`A`,`5`): 2-bit input difference, 2-bit output difference.
- (`5`,`6`): 2-bit input difference, 2-bit output difference.
- (`8`,`7`): 1-bit input difference, 3-bit output difference.

		Output Differences							
		0	1	2	3	4	5	6	7
Input Difference	0	0	2	2	4	0	2	2	4
	1	0	2	2	4	0	2	2	4
	2	0	2	2	0	2	4	4	2
	3	0	4	2	2	4	0	2	2
	4	0	2	2	4	2	4	0	2
	5	2	0	0	2	4	2	6	0
	6	2	4	2	0	4	2	0	2
	7	0	2	6	4	0	2	2	0
	8	0	0	0	4	2	2	2	6
	9	0	2	4	2	6	0	2	0
	A	2	0	4	2	0	6	2	0
	B	2	2	2	2	2	2	2	2
	C	2	2	4	0	2	2	0	4
	D	4	4	0	4	0	0	0	4
	E	2	2	2	2	0	0	4	4
	F	0	4	0	0	4	4	4	0

Table 11: The XOR Differential Probability (XDP) table for the *MAP* function used in the CHI algorithms. The probabilities have been multiplied by 16 to make the table more readable.

17.2.2 Nabla-Differentials for the *MAP* Function

There are only eight differentials for which there are two pairs satisfying the differential These eight combinations are shown in Table 12. None of these differentials have single bit input difference, so *MAP* Requirement 9 (Section 14.1.2) is satisfied. For all other combinations of input/output nabla differences, the differential cannot occur, or there is only one pair satisfying the differential. There is a large set of other nabla differentials that hold for one input pair, so we do not show the set here.

Input Nabla	Output Nabla	Input Pairs
++	**+	(0000,0110), (1001,1111)
+-	+**	(0011,0100), (1010,1100)
-+	-**	(0101,0010), (1100,1010)
--	**-	(0110,0000), (1111,1001)
+**+	-**	(0000,1001), (0110,1111)
+**-	*-*	(0011,1010), (0101,1100)
-**+	*+*	(1010,0011), (1100,0101)
-**-	+**	(1001,0000), (1111,0110)

Table 12: The Nabla Differentials of the *MAP* function for which there are two pairs satisfying the differential. The pairs of inputs satisfying the differential are shown in the rightmost column

17.2.3 Linear Approximation of the *MAP* Function

The Linear Approximation Bias (LAB) table for the *MAP* function is shown in Table 13. The only probability here that causes us some concern is the entry (9,4) which indicates a 2-bit input difference that results in a 1-bit output difference with probability $\frac{6}{16}$.

		Output Mask							
		0	1	2	3	4	5	6	7
Input Mask	0	-1.00	0	0	0	0	0	0	0
	1	0	0	-0.25	+0.25	0	-0.50	+0.25	+0.25
	2	0	0	0	-0.50	+0.25	-0.25	-0.25	-0.25
	3	0	0	-0.25	-0.25	+0.25	+0.25	0	0
	4	0	-0.25	0	-0.25	0	+0.25	0	+0.25
	5	0	+0.25	+0.25	0	0	+0.25	-0.25	+0.50
	6	0	-0.25	0	+0.25	+0.25	0	-0.25	0
	7	0	+0.25	+0.25	-0.50	+0.25	0	+0.50	+0.25
	8	0	-0.25	+0.25	0	+0.25	0	0	-0.25
	9	0	-0.25	-0.50	-0.25	-0.25	0	+0.25	0
	A	0	-0.25	-0.25	0	0	+0.25	-0.25	+0.50
	B	0	-0.25	0	-0.25	-0.50	+0.25	0	-0.25
	C	0	0	+0.25	+0.25	-0.25	+0.25	+0.50	0
	D	0	-0.50	0	0	+0.25	-0.25	+0.25	+0.25
	E	0	0	-0.25	+0.25	+0.50	+0.50	+0.25	-0.25
	F	0	+0.50	-0.50	0	0	0	0	0

Table 13: The Linear Approximation Bias (LAB) table for the *MAP* function used in the CHI algorithms

The following linear approximations to the *MAP* have maximum bias ± 0.5.

- 1-in, 2-out: (1,5), (2,3);
- 2-in, 1-out: (9,2);
- 2-in, 2-out: (C,6);
- 3-in, 1-out: (B,4), (D,2), (E,4);
- 2-in, 3-out: (5,7), (A,7);
- 3-in, 2-out: (7,3), (7,6), (E,5);
- 4-in, 1-out: (F,1), (F,2).

The linear approximations with low weight are probably of most concern. There are no 1-in, 1-out linear approximations with high bias.

17.3 Analysis of Addition Operations

17.3.1 XOR Differentials

Assume that the nabla difference coming out of the *MAP* functions is not being predicted: only the XOR-difference coming out of the *MAP* function is predicted. Bit differences in the MSB act like XOR:

- A bit difference in the MSB only of one of the inputs is guaranteed to result only in a bit difference in the MSB of the output.
- Bit differences in the MSB only of both of the inputs are guaranteed to cancel so that there is no bit difference in of the output.

For the operation $c = a \boxplus b$, the XOR differentials most commonly used are of the form:

$$\Delta_{\oplus}(c', c'') == \alpha \oplus \beta \ : \ \Delta_{\oplus}(b', b'') := \beta, \Delta_{\oplus}(a', a'') := \alpha.$$

The probability of this XOR differential is 2^{-w}, where w is the Hamming weight of

$$((\alpha \ OR \ \beta) \ AND \ \texttt{0x7FF}\ldots\texttt{F}) \, .$$

An attacker is always looking for α and β with low weight. A designers, it was out task to ensure that α and β have high weight.

17.3.2 Linear Approximations

If $c = a \boxplus b$, then the best linear approximations are of the form

$$\alpha \cdot a + \alpha \cdot b + \alpha \cdot c == 0.$$

When using the mask $\alpha = \texttt{0x00}\ldots\texttt{01}$, the linear approximation holds with probability one (bias $=+1$). The bias of this linear approximation in the general case is 2^{-w}, where w is the Hamming weight of ($\alpha \ AND \ \texttt{0xF}\ldots\texttt{FE}$).

17.3.3 Nabla Differences

Nabla differences were especially designed for dealing with the interactions between XOR operations and addition operations. The interaction between nabla differences and the addition operation is best explained by the following two rules.

1. If $c = a \boxplus b$, $\nabla(a', a'') = \alpha$, $\nabla(b', b'') = \beta$, then the addition difference between c' and c'' is

$$\Delta_{\boxplus}(c', c'') := c'' - c' := \sum_{\alpha[k] \neq \text{“}*\text{”}} \alpha[k] \times 2^k + \sum_{\beta[k] \neq \text{“}*\text{”}} \beta[k] \times 2^k \ (\text{mod } 2^{64}).$$

2. If it is known that $\Delta_{\boxplus}(c', c'') = g$, then $\gamma = \nabla(c', c'')$ may be any nabla difference that satisfies

$$\sum_{\gamma[k] \neq \text{“}*\text{”}} \gamma[k] \times 2^k == g (\text{mod } 2^{64}).$$

17.4 Analysis of the XOR Operations

If $c = a \oplus b$, then

$$\alpha \cdot a + \alpha \cdot b + \alpha \cdot c == 0.$$

has bias of 1, and

$$\Delta_{\oplus}(a', a'') := \alpha, \Delta_{\oplus}(b', b'') := \beta \quad \rightarrow \quad \Delta_{\oplus}(c', c'') == (\alpha \oplus \beta),$$

holds with probability one.

If $\nabla(a', a'') = \alpha$ and $\nabla(b', b'') = \beta$, then the following rules can be used to determine the possible values for $\nabla(c', c'')$:

$\alpha[k]$	$\beta[k]$				
	0	1	*	+	-
0	0	1	*	+	-
1	1	0	*	-	+
*	*	*	*	+/-	+/-
+	+	-	+/-	0	1
-	-	+	+/-	1	0

Table 14: Rules for determining $\nabla(c', c'')$ when $c = a \oplus b$.

17.5 Analysis of the Bit Moving Operations

If f is a bit moving operations, and $b := f(a)$, then:

$$f(\alpha) \cdot a + \alpha \cdot b == 0.$$

has bias of 1, while the XOR differential and nabla differentials:

$$\begin{aligned} \Delta_{\oplus}(a', a'') := \alpha \quad &\rightarrow \quad \Delta_{\oplus}(b', b'') == f(\alpha); \\ \nabla(a', a'') := \alpha \quad &\rightarrow \quad \nabla(b', b'') == f(\alpha); \end{aligned}$$

hold with probability one.

18 Differential Paths and the CHI algorithms

The standard technique for differential analysis begins by finding a differential pattern: a single bit difference in one step input is introduced, and the following step inputs attempt to cancel the effect of the resulting bit differences in the working variables, until eventually all of the bit differences in the working variables are canceled out. Our current analysis has been unable to find a differential pattern. Once a bit difference is introduced into the working variables, it is difficult to find an XOR difference in the step inputs that will reduce the number of bit difference: the θ_0 and θ_1 functions ensures that (most of the time) attempting to cancel one set of bit differences will only add more bit differences elsewhere. There will be differential paths that do end up canceling out, but more analysis is required to find these. It is likely that the paths can be found by solving an appropriate set of linear equations: so finding a high probability differential path will become similar problem to finding a low-weight code-word.

18.1 One Bit Difference in Step Input

To illustrate the difficulties, we follow the effect of a single bit difference in CHI-224 and CHI-256. In this description, we will use $a[i, j, k]$ t indicate that variable a has bit differences in position i, j, k.

Assume we are starting at step $t := 0$, with bit difference $W_{2t}[0]$ and no bit difference in W_{2t+1}.

1. Since there are no bit differences in the working variables at this point in time, there will be no bit differences in the values of $preR$, $preS$, $preT$, $preU$.

2. We begin by introducing a bit difference $W_{2t}[0]$

3. The value of W_{2t} is XORed with K_{2t} to make V_0.This bit difference propagates through the XOR with K_{2t} too become a bit difference in $V_0[0]$.

4. The value of V_0 is XORed with $preR$ to make R. Since there is no bit difference in $preR$, the only bit difference in R comes from V_0: this is bit difference $R[0]$.

5. The input mixing function $\theta_0^{\{256\}}(\cdot)$ is applied to V_0 and the result is XORed with $preT$. The bit difference in $V_0[0]$ also propagates through $\theta_0^{\{256\}}(\cdot)$ to become the three bit differences:

   - The bit difference at position 0 is rotated 21 bits to the right (in a 32-bit block) to end up at position 11.
   - The bit difference at position 0 is rotated 26 bits to the right (in a 32-bit block) to end up at position 6.
   - The bit difference at position 0 is rotated 30 bits to the right (in a 32-bit block) to end up at position 2.

   This results in XOR difference $T[11, 6, 2]$.

6. There is no bit difference in W_{2t+1}, and consequently no bit difference in V_1, $\theta_0^{\{256\}}(V_1)$, S or U.

7. The MAP is then applied. The output of the MAP must have bit differences at position 0 (resulting from $R[0]$) and positions 11,6 and 2 (resulting from $T[11, 6, 2]$). We choose the bit difference to occur in the output MAP_0. noting that at $t == 0$, the value of MAP_0 will be multiplexed to X. For each bit position, the probability of resulting in only a bit difference in MAP_0 is 2^{-3} (sub total 2^{-12}). This results in bit differences $X[11, 6, 2, 0]$ and $XX[11, 6, 2, 0]$.

8. The value of XX is added to YY and then rotated by 16 bits to become AA. The bit differences $XX[11, 6, 2, 0]$ become bit differences $AA[59, 54, 50, 48]$ with probability of 2^{-1} for each bit position (sub total 2^{-4}).

9. The value of AA is XORed with F to become the new A. The bit differences $AA[59, 54, 50, 48]$ becomes the bit differences $A[59, 54, 50, 48]$.

10.

We now begin step $t := 1$.

Step	Input Differences	Output Differences	Probability
0	$W_{2t}[0]$ $\theta_0^{\{256\}}(V_0)$	$R[0] = V_0[0]$ $T[11, 6, 2]$	
	$R[0]$ $T[11, 6, 2]$ $MAP_2[11, 6, 2, 0]$	$MAP_2[0]$ $MAP_2[11, 6, 2]$ $X[11, 6, 2, 0]$	2^{-3} $(2^{-3})^3$
	$X[11, 6, 2, 0]$ $XX[11, 6, 2, 0]$	$XX[11, 6, 2, 0]$ $AA[59, 54, 50, 48]$	 $(2^{-1})^4$
1	$AA[59, 54, 50, 40]$	$A[59, 54, 50, 40]$	
	$A[59, 54, 50, 40]$ $A[59, 54, 50, 40]$ $A[59, 54, 50, 40]$	$preS[59, 54, 50, 40]$ $preT[28, 24, 22, 1]$ $preU[63, 42, 37, 33]$	
	Choose $W_{2t+2}[63, 48, 22]$		
	$V_0[63, 48, 22]$	$R[63, 48, 22]$	
	$preS[59, 54, 50, 48]$ $preT[28, 24, 22, 1] \oplus \theta_0^{\{256\}}(V_0)[\ldots]$ $preU[63, 42, 37, 33]$	$S[59, 54, 50, 48]$ $T[59, 54, 50, 42, 37, 33, 22]$ $U[63, 42, 37, 33]$	
	R, S, T, U diffs $MAP_0[63, 42, 37, 33]$	$MAP_0[63, 42, 37, 33]$ $ZZ[31, 10, 5, 1]$	$3 \cdot 2^{-24}$
	$ZZ[31, 10, 5, 1]$	$DD[58, 53, 49, 15]$	2^{-4}
2	$DD[58, 53, 49, 15]$ $A[59, 54, 50, 40]$	$D[58, 53, 49, 15]$ $B[59, 54, 50, 40]$	
	Total		2^{-44}

Table 15: Differential path from a single bit differences in $W_{2t}[0]$. Only variables with bit differences are shown. The probability is 1 unless otherwise noted. See text in Section 18.1 for details.

1. The bit differences $A[59, 54, 50, 48]$ result in bit differences $preS[59, 54, 50, 48]$ (no rotation), $preT[28, 24, 22, 1]$ ($SWAP32$ and right rotation by 26) and $preU[63, 42, 37, 33]$ (right rotation by 17). There are no bit differences $preR$.

2. Input differences $W_{2t}[63, 48, 22]$ which will result in $V_0[63, 48, 22]$ and (via the input mixing function):

$$\begin{aligned}
\theta_0^{\{256\}}(V_0[22]) &\rightarrow \theta_0^{\{256\}}(V_0)[28, 24, 1];\\
\theta_0^{\{256\}}(V_0[48]) &\rightarrow \theta_0^{\{256\}}(V_0)[59, 54, 50];\\
\theta_0^{\{256\}}(V_0[63]) &\rightarrow \theta_0^{\{256\}}(V_0)[42, 37, 33].
\end{aligned}$$

3. The bit differences in R, S, T, U can now be computed:

$$\begin{aligned}
V_0[63, 48, 22] &\rightarrow R[63, 48, 22];\\
preS[59, 54, 50, 48] &\rightarrow S[59, 54, 50, 48];\\
preT[28, 24, 22, 1] \oplus \theta_0^{\{256\}}(V_0)[59, 54, 50, 42, 37, 33, 28, 24, 1] &\rightarrow T[59, 54, 50, 42, 37, 33, 22];\\
preU[63, 42, 37, 33] &\rightarrow U[63, 42, 37, 33].
\end{aligned}$$

4. The MAP is then applied. Note that each of the bit positoins, two of the inputs to the MAP have a bit difference:

   - $R[22], T[22] \rightarrow$ no difference in $MAP[22]$ with probability 2^{-3};
   - $T[33], U[33] \rightarrow MAP_0[33]$ with probability 2^{-2};
   - $T[37], U[37] \rightarrow MAP_0[37]$ with probability 2^{-2};
   - $T[42], U[42] \rightarrow MAP_0[42]$with probability 2^{-2};
   - $R[48], S[48] \rightarrow$ no difference in $MAP[48]$ output with probability 2^{-3};
   - $S[50], T[50] \rightarrow$ no difference in $MAP[50]$ with probability 2^{-3};
   - $S[54], T[54] \rightarrow$ no difference in $MAP[54]$ with probability 2^{-3};
   - $S[59], T[59] \rightarrow$ no difference in $MAP[59]$ with probability 2^{-3};
   - $R[63], U[63] \rightarrow MAP_0[63]$ with probability $3 \cdot 2^{-3}$;
   - $\Rightarrow MAP_0[63, 42, 37, 33]$, with subtotal probability $3 \cdot 2^{-24}$.

   For $t == 1$, the value of MAP_0 is multiplexed to Z, so the differences input to the $POST$-$MIXING$ are $Z[63, 42, 37, 33]$. The $SWAP32$ moves these differences to $ZZ[31, 10, 5, 1]$.

5. The value of ZZ is added to YY and then rotated by 48 bits to become DD. The bit differences $ZZ[31, 10, 5, 1]$ become bit differences $DD[58, 53, 49, 15]$ with probability of 2^{-1} for each bit position (sub total 2^{-4}).

6. The value of DD is XORed with C to become the new D. The bit differences $DD[58, 53, 49, 15]$ becomes the bit differences $D[59, 54, 50, 48]$. The value of A is also moved to B, so the differences now have $B[59, 54, 50, 48]$

After only two steps, the probability is already down to 2^{-44}. A differential pattern over 8 steps for SHA-224 and SHA-256 has probability 2^{-39} [34]. One step of CHI-224 and CHI-256 appears to provide the same resistance to differential attacks as four steps of SHA-224 and SHA-256. This agrees somewhat wit the rate at which the steps inputs are used i the step function: one step of CHI-224 and CHI-256 uses 128-bits of step input, the same amount used by 4 steps of SHA-224 and SHA-256. This makes the full CHI-224 and CHI-256 equivalent in strength to 80 steps of SHA-224 and SHA-256, so the number of steps for CHI-2224 and CHI-256 appears to be quite conservative.

If we conduct a similar analysis for CHI-384 and CHI-512, the main difference would be that the difference in XX or ZZ would propagate to differences in GG, which would introduce twice as many bit differences into the state. This leads us to believe that the differential probabilities for a given number of steps of CHI-384 and CHI-512 will be significantly lower than for the same number of steps of CHI-224 and CHI-256. We do not have a sufficiently detailed analysis to support this statement at this time.

18.2 An Interesting Differential Structure

An interesting thing happens if you input bit differences to W_{2t} corresponding to the rotations of each copy of W_{2t+1}, and vice versa. in this case, we will study CHI-384 and CHI-512. Suppose the differences are:

$$\begin{aligned} W_{2t}[64-49, 64-30, 64-20, 0] &= W_{2t}[44, 34, 5, 0] \\ W_{2t+1}[64-43, 64-6, 64-5, 0] &= W_{2t+1}[59, 58, 21, 0] \end{aligned}$$

The differences in R, T, S, U will be:

$$
\begin{aligned}
W_{2t}[44,34,15,0] &\rightarrow V_0[44,34,15,0];\\
W_{2t+1}[59,58,21,0] &\rightarrow V_1[59,58,21,0];\\
V_0[44,34,15,0] &\rightarrow R[44,34,15,0];\\
V_1[59,58,21,0] &\rightarrow S[59,58,21,0];\\
\theta_0^{\{512\}}(V_0[44,34,5,0]) &\rightarrow T \left\{ \begin{array}{c|ccc} ROTR64 & 43 & 6 & 5 \\ \hline 44 & 1 & 38 & 39 \\ 34 & 55 & 28 & 29 \\ 15 & 36 & 9 & 10 \\ 0 & 21 & 58 & 59 \end{array} \right\}\\
&= T[59,58,55,39,38,36,29,28,21,10,9,1];\\
\theta_1^{\{512\}}(V_0[59,58,21,0]) &\rightarrow U \left\{ \begin{array}{c|ccc} ROTR64 & 49 & 30 & 20 \\ \hline 59 & 10 & 29 & 39 \\ 58 & 9 & 28 & 38 \\ 21 & 36 & 55 & 1 \\ 0 & 15 & 34 & 44 \end{array} \right\}\\
&= U[55,44,39,38,36,34,29,28,15,10,9,1].
\end{aligned}
$$

Now, the inputs to the MAP linear up in pairs:

- $(S, T) : [59, 58, 21]$,
- $(T, U) : [55, 39, 38, 36, 29, 28, 10, 9, 1]$,
- $(R, U) : [44, 34, 15]$,
- $(R, S) : [0]$.

There are several combinations of bit differences in pairs of R, S, T, U that can cancel. If bit differences in each of the pairs above ((S, T), (T, U), (R, U), $R, S)$) could result cancel, then this could be an approach for getting low probability differentials since no bit differences would be input to the working variables.

However, of these pairs, only bit differences in (S, T) and (R, S) can cancel (with probability 2^{-3} for each bit position). In the pairs (T, U), (R, U), there must be at least a single bit difference output. Since are unable to cancel the differences, we do not investigate this differential further (at this time).

Note that, even if all the pairs of bit differences could cancel, then we must account for each of the 16 bit positions, and the probability would be $(2^{-3})^{16} = 2^{-48}$ which is already very low.

18.3 Linear Approximations and CHI

Linear cryptanalysis requires a linear approximation to the algorithm with a bias that can be measured in less time than a generic attack on the algorithm. Linear approximations to the algorithm are constructed from linear applications to the components of the algorithm (such as the linear approximations discussed in Section 17.

Analysis of the CHI algorithms with respect to linear cryptanalysis will use a similar technique to that used for differential attacks. Linear cryptanalysis could be used in preimage attacks, but would be most useful in analyzing applications using secret data, such as HMAC. We expect linear analysis to be unsuccessful in breaking CHI. We expect that the the diffusion in CHI will force linear approximations to use many approximations to the MAP function, quickly decreasing the bias to the point where attacks are no longer possible.

19 Analysis of Second Preimage Attacks

Second pre-image attacks on the compression function are almost equivalent to a kind of related-key attack on the underlying block cipher. These attacks share some similarities with the collision attacks. The significant difference arises in the control of the attacker over the messages: in a collision attack, the attacker can choose both messages in the manner that is most likely to succeed; however, in a second preimage attack, the attacker does not have control over the first message.

A second pre-image attack can use a differential paths as used in collision attacks. For second pre-image attacks, the attacker cannot use nabla differentials in the first few steps (unlike collision attacks) since the attacker has no control over the first "preimage" to be matched. For second pre-image attacks, the differential path is comprised entirely from XOR differences, and the important factor is the probability of the XOR differential path.

The analysis in Section 18 indicates that the probability of such a differential increases rapidly. The message expansion needs to ensure that there are many bit differences in the step inputs. This is investigated briefly in Section 20.

20 Analysis of the Message Expansion

We performed a simple analysis of the the message expansion to find some low-weight codewords of the message expansion.

CHI-224 and CHI-256 To find some low-weight codewords for CHI-224 and CHI-256, we looked at 40 consecutive word64s from the message expansion where 8 consecutive word64s were required to be all zero except for 1,2 or 3 ones. This approach will find low-weight codewords, but these codewords are not guaranteed to be of minimum weight

For collision attacks on CHI-224 and CHI-256, we are interested in codewords that have low weight over the last 32 words (corresponding to the last 16 steps). The weight in the first 8 words (correspond to the fist 4 steps) is not relevant since the attacker will control the nabla differentials in these 4 steps. The complexity is affected most by the number of differences in the last 32 steps.

For second primage attacks on CHI-224 and CHI-256, we are interested in codewords that have low weight over all 40 words since the attacker cannot control nabla differentials in the first few rounds in these attacks.

The results of the search are summarized in Table 16. The minimum weight over 32 word64s is 139 bits, and the minimum weight over 40 word64s is 266 bits. At a very conservative estimate, every non-zero bit in the expanded message results in a factor of 2^{-1} for a differential. Using this estimate:

- Using the minimum-weight 32 word sequence, a 16-step differential has a probability less than 2^{-139}, and differential-based collision attacks have a complexity of at least 2^{139}, which is more

Algorithm	Steps	Lowest Weight Codeword using:		
		1 one	2 ones	3 ones
CHI-224, CHI-256	40 Words	266	364	282
	32 Words	139	186	146
CHI-384,CHI-512	80 Words	834	996	1099
	64 Words	474	589	644

Table 16: Low-weight codeword of the message expansion for the CHI algorithms.

than the complexity of a generic collision attack on CHI-224 and CHI-256. This indicates that the message expansion for CHI-224 and CHI-256 provides good resistance against differential-based collision attacks.

- Using the minimum-weight 40 word sequence, a 20-step differential has a probability less than 2^{-266}, and differential-based second preimage attacks have a complexity of at least 2^{266}, which is more than the complexity of a generic second preimage attack on CHI-224 and CHI-256. This indicates that the message expansion for CHI-224 and CHI-256 provides good resistance against differential-based second preimage attacks.

CHI-384 and CHI-512 To find some low-weight codewords for CHI-384 and CHI-512, we looked at 80 consecutive word64s from the message expansion where 16 consecutive word64s were required to be all zero except for 1,2 or 3 ones. This approach will find low-weight codewords, but these codewords are not guaranteed to be of minimum weight

For collision attacks on CHI-384 and CHI-512, we are interested in codewords that have low weight over the last 64 words (corresponding to the last 32 steps). The weight in the first 16 words (correspond to the fist 8 steps) is not relevant since the attacker will control the nabla differentials in these 8 steps. The complexity is affected most by the number of differences in the last 64 steps.

For second primage attacks on CHI-384 and CHI-512, we are interested in codewords that have low weight over all 80 words since the attacker cannot control nabla differentials in the first few rounds in these attacks.

The results of the search are in Table 16. The minimum weight over 64 word64s is 474 bits, and the minimum weight over 80 word64s is 834 bits. At a very conservative estimate, every non-zero bit in the expanded message results in a factor of 2^{-1} for a differential. Using this estimate:

- Using the minimum-weight 64 word sequence, a 32-step differential has a probability less than 2^{-474}, and differential-based collision attacks have a complexity of at least 2^{474}, which is far more than the complexity of a generic collision attack on CHI-384 and CHI-512. This indicates that the message expansion for CHI-384 and CHI-512 provides good resistance against differential-based collision attacks.

- Using the minimum-weight 80 word sequence, a 40-step differential has a probability less than 2^{-834}, and differential-based second preimage attacks have a complexity of at least 2^{834}, which is far more than the complexity of a generic second preimage attack on CHI-384 and CHI-512. This indicates that the message expansion for CHI-384 and CHI-512 provides good resistance against differential-based second preimage attacks.

21 Conclusion of Cryptanalysis

The analysis in this part of the document demonstrates the strength of the CHI algorithms. The number of steps that were selected for the algorithms (20 steps for CHI-224 and CHI-256, and 40 steps for CHI-284 and CHI-512) was based on a variety of considerations: the factor by which the message block is expanded, the number of step inputs used per step, and the security provided by each step. This analysis indicates that chosen number of steps is perhaps too conservative, and the algorithm would still be secure with fewer steps. For the time being, we are content to have a larger safety factor.

Part IV
Implementation Considerations

22 Introduction to Implementation Considerations

This section addresses the implementation characteristics of the CHI algorithms. We first discuss software implementation on high-end processors (Section 23), mid-range processors (Section 24), low-end processors (Section 25). We conclude by discussing hardware implementations in Section 26. Prior to this discussion, we look at parallelism in the design of the CHI algorithms.

22.1 Parallelism in the CHI Algorithms

There is a good amount of parallelism in the design:

- If desired, two updates of the message expansion feedback can be computed in parallel.
- The rotations and XORs in the message expansion can be implemented in parallel. For example, up to 8 word64s per message expansion update could be rotated in parallel, or a total of 16 word64s could be rotated in parallel for 2 parallel message expansion updates.
- The message expansion update and the step function can be implemented in parallel, with two message expansion updates for each step function.
- The rotations and XORs in the *PRE-MIXING* and *DATA-INPUT* can be implemented in parallel (if supported, 4 word64s could be being processed simultaneously).
- The outputs of the *MAP* phase can be computed in parallel: if sufficient execution units are available, the MAP can be implemented in 5 cycles.
- One limitation of the design is the fact that Y has both a rotation and a $SWAP8$ applied to produce YY. It may have been preferable to apply only the $SWAP8$ to Y, and apply additional rotations to X and Z instead. This is a tweak we are considering for the future.
- The addition operations can be preformed in parallel, as can the final rotations and XORs with the working variables.

Some implementation types (small processors in particular) are not able to take advantage of this parallelism. Larger processors can take advantage of some of this parallelism. Hardware, in particular, can make the most advantage of this parallelism.

23 Implementation Considerations for High-End Processors

This section addresses the implementation of CHI algorithms on high-end 32-bit and 64-bit processors. The NIST-specified reference platform is used as a data point for this discussion. The performance figures for the reference platform are discussed first in Section 23.1. Section 23.2 discusses considerations in implementing the message expansion. Section 23.3 discusses considerations in implementing the step function for CHI-224 and CHI-256, while Section 23.4 discusses considerations in implementing the step function for CHI-384 and CHI-512. SIMD co-processors are discussed in Section 23.5.

23.1 Performance Figures on the NIST SHA-3 Reference Platform

The CHI team built a copy of the reference platform specified in [57, Section 6.B.i]: "NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition." The performance figures given were obtained by booting in either 32- or 64-bit mode as appropriate. Table 34 gives the performance figures in cycles per byte of hash input as measured using the "rdtsc" instruction, for the various sizes of the optimized implementations. In both cases, a large buffer is 1MB, and the figure should approximate the asymptotic performance. A small buffer ("Block") is 64 bytes for CHI-224 and CHI-256, and 128 bytes for CHI-384 and CHI-512 respectively. The implementer was Cameron McDonald, with assistance from Craig Brown.

Algorithm	Buffer Size	Windows Vista Ultimate	
		32-bit Edition	64-bit Edition
CHI-224 and CHI-256	One block: 64B	51	26
	Large Buffer: 1MB	49	24
CHI-384 and CHI-512	One block: 128B	80	18
	Large Buffer: 1MB	78	16

Table 17: Performance figures for the CHI algorithms on the NIST-specified Reference platform. The figures are provided in bytes per cycle.

The relative performance of CHI-224 and CHI-256 on the 32-bit reference platform compared to the 64-bit platform is close to optimal: the ratio is very close to 2 cycles per byte on the 32-bit processors for each cycle per byte on the 64-bit processor. The ratio of 2 to 1 is a good balance because it reflects the number of bytes processed per instruction: 4 bytes per instruction on a 32-bit processor and 8 bytes per instruction on a 64-bit processor.

CHI-384 and CHI-512 perform relatively poorly on the 32-bit reference platform compared to the 64-bit platform: the ratio is approximately 4 cycles on the 32-bit processors for each cycle on the 64-bit processor. Most of the loss in speed can be attributed to the difficulty of performing rotations in 64-bit blocks. Rotations in 64-bit blocks which take one cycle per word64 on a 64-bit processor, but longer on a 32-bit x86 processor. Consider computing the most significant 32-bits of $ROTR64^n(a))$, for a small rotation amount n:

$$MSB32(ROTR64^n(a)) \quad = \quad (al \ll_{32} (32 - n)) \oplus (ah \gg_{32} n)$$

This requires two shift instructions and one XOR instructions: a total of four instructions. Other 32-bit architectures (see Section 24.2) combine shift and rotate operations into a single instruction, which can halve the number of instructions. Interestingly, the ratio of 4 to 1 is close to the ratio between the performance of SHA-512 on 32-bit and 64-bit processors provided by Brian Gladman's optimized SHA-2 code [32].

The bad relative performance on 32-bit processors reflects the design decision to target CHI-384 and CHI-512 at high-end processors (most of which are 64-bit processors), and assume that mid-range and low-end processors do not need fast implementations of CHI-384 and CHI-512.

23.2 Message Expansion

The message expansion uses rotations in 64-bit blocks operations, which are well suited to 64-bit machines, and not well suited to 32-bit machines.

The implementations supplied in the submission package compute all of the step inputs prior to applying the step functions. CHI-224 and CHI-256 require 8 word64s for the message expansion state; CHI-384 and CHI-512 require 12 word64s for message expansion state. In both cases a 32-bit implementation (8 registers) has insufficient registers to store the message expansion state while computing the next word64s in the state, so there must be many operations copying data between the registers and the cache. On the other hand, a 64-bit implementation (16 registers) has sufficient registers to store the message expansion state while computing the next word64s in the state. This means that the only data-moving operations results from the step inputs being copied to the cache, which can typically occur in parallel with the other operations. This is an advantage of computing all of the step inputs prior to applying the step functions, but a disadvantage is that the operations of the message expansion on the critical path.

Another approach is to perform the message expansion in parallel with step function, with two message expansion updates performed for each step function. If there are sufficient execution units to perform the message expansion operations and step function operations in parallel, then this may move the message expansion off the critical path. Unfortunately, neither 32-bit nor 64-bit processors have sufficient registers to store the message expansion state, the working variables and the other required temporary variables used during the computation. This means that some time may lost while copying portions of the message state and working variables between the cache and the registers.

There are advantages and disadvantages to both approaches. We have not had an opportunity to compare the approaches, but hope to do so in the near future. Certainly, since the 32-bit processors are already performing many copies between registers and cache in the former approach, it makes sense that there is less to lose in applying the latter approach.

23.3 Implementing the CHI-224 and CHI-256 Step Function

The operations in CHI-224 and CHI-256 are well suited to 32-bit machines. In addition to the parallelism discussed in Section 22.1 the implementation can use the partitioned description of the step function in Section 7.1.3. Some care is required in forming 64-bit blocks for (a) the rotations in 64-bit blocks operations, and (b) the 64-bit addition operations.

The operations in CHI-224 and CHI-256 are less well suited to 64-bit machines, since the rotations in 32-bit blocks could (potentially) require a single cycle for rotating each 32-bit block. The ratio of 2 to 1 (cycles per byte) indicates a good balance (as discussed above). The 64-bit processors have more registers, resulting in less passing of data in and out of registers. These implementations can exploit the parallelism discussed in Section 22.1, but cannot exploit the parallelism at the 32-bit level.

23.4 Implementing the CHI-384 and CHI-512 Step Function

CHI-384 and CHI-512 perform quite relatively badly on the 32-bit reference platform compared to the 64-bit platform: the ratio is approximately 4 cycles on the 32-bit processors for each cycle on the 64-bit processor. Most of the loss in speed can be attributed to the difficulty of performing rotations in 64-bit blocks. Rotations in 64-bit blocks which take one cycle per word64 on a 64-bit

processor, but longer on a 32-bit x86 processor. Consider computing the most significant 32-bits of $ROTR64^n(a))$, for a small rotation amount n:

$$MSB32(ROTR64^n(a)) \quad = \quad (al \ll_{32} (32 - n)) \oplus (ah \gg_{32} n)$$

This requires two shift instructions and one XOR instructions: a total of three instructions. Other 32-bit architectures (see Section 24.2) combine shift and rotate operations into a single instruction, which can halve the number of instructions. Interestingly, the ratio of 4 to 1 is close to the ratio between the performance of SHA-512 on 32-bit and 64-bit processors provided by Brian Gladman's optimized SHA-2 code [32].

23.5 Notes on Using SIMD Co-Processors

Typically the SIMD registers are distinct from the general purpose registers of the CPU, and timing must account for copying data between the general purpose register and SIMD registers (if required).

The SIMD could be a useful place to perform the message expansion in parallel while the step function is being performed on the main processor The greater amount of storage in the SIMD registers make them useful for storing the message expansion state, and the SIMD operations can be better suited to performing the rotations of the message more efficiently, particularly if the co-processor is attached to a 32-bit machines. The extra latency of the SIMD operations needs to be considered, but otherwise this looks like an attractive option.

A 64-bit processor might be able to use the parallel 32-bit operations in the SIMD for performing the 32-bit operations of CHI-224 and CHI-256. It seems unlikely that that this will result in a significant speed-up, due to the extra latency of SIMD operations (again).

24 Implementation Considerations for Mid-Range Processors

In this section we investigate the ARM architecture processors. There are other architectures available, but it is useful to focus on one architecture. There are manufacturers other than ARM that produce processors conforming to an ARM architecture. We use the term "ARM processors" to include all processors conforming to an ARM architecture, regardless of whether those processors were manufactured by ARM.

There are several ARM architecture version numbers: the latest version is ARM architecture v7. Some ARM architectures support a reduced instruction set denoted "Thumb" which uses instruction codes significantly smaller than the normal ARM instruction codes.

24.1 Architecture

ARM processors have sixteen 32-bit registers available to the programmer, and all processors since ARM architecture v3 have had multi-stage pipelines. ARM processors prior to ARM architecture v5 have concurrency of 1: that is, only one instruction can be processed at a time. Most ARM processors from ARM architecture v5 onwards have concurrency of 3 or more. Two of these execution units are Arithmetic Logical Units (ALUs) which can perform addition, logical operations and bit moving operations. The other execution units are used for moving data.

In the near future, mid-range processors will have concurrency approaching five, and perhaps four of those execution units will be ALUs, but it is more likely that these processors will continue to use two ALUs.

The CHI algorithms, can exploit the concurrency to great effect.

24.2 Instructions

ARM [16] supports all the expected instruction for 32-bit operands: addition, logical operations, *SWAP*8 (since ARM architecture v6), variable-amount shift, variable-amount rotation in 32-bit blocks. Normal ARM instruction set includes 1-bit rotation with carry, but this is excluded from the Thumb instruction sets for architectures prior to ARM Architecture v7.

The normal ARM instructions (not the Thumb instructions) for addition and the logical operations (AND,OR and XOR) include an option to apply a shifting operation prior to performing the addition/AND/OR/XOR. The shifting operation can be a variable shift, variable rotate or 1-bit rotate-right-with-carry. With the exception of the rotation of the bits of Y at the beginning of the *POST-MIXING* phase, every rotation is followed by an XOR operation. This can be exploited by the an implementation of the CHI algorithms.

- An XOR and a rotation in 32-bit blocks (CHI-224 and CHI-256) can be incorporated into a Rotate-then-XOR instruction
- The cost of a XOR and a rotation in 64-bit blocks can be reduced significantly. Consider computing the most significant 32-bits of $ROTR64^n(a)) \oplus b$, for a small rotation amount n

$$MSB32(ROTR64^n(a)) \oplus bh \quad = \quad (al \ll_{32} (32 - n)) \oplus (ah \gg_{32} n) \oplus bh$$

  This requires two shift instructions and two XOR instructions: a total of four instructions when shift-then-XOR is not supported. If supported, this calculation can be performed with only two shift-then-XOR instructions.

- An XOR and the $ROTR64^1(\cdot)$ operation can be performed in two rotate-right-with-carry-and-XOR instructions. If not supported, the same operation takes 4 instructions

This will greatly reduce the number of instructions and have a great impact on the the processing time. We expect an improvement of 20-30%, given the large number of rotation operations in the CHI algorithms.

24.3 NEON SIMD Co-Processors

The ARM11[6] (ARM architecture v6) introduced the ability for the 32-bit processors to be used as a SIMD processor with 8-bit or 16-bit data sets. These operations are not particularly useful for the CHI algorithms, so we do not examine them further.

With the introduction of ARM architecture v7, a 64-bit SIMD instruction set called NEON became available (for example, on the Cortex A8 and Cortex R4). The recently released Qualcomm

[6]Cortex, Cortex A8, Cortex R4, Cortex A9 and NEON are trademarks of ARM Ltd [2].

Scorpion[7] (using ARM architecture v7) has access to a 128-bit SIMD NEON co-processor. The ARM Cortex A8 will introduce 128-bit NEON SIMD co-processors in the near future.

The 32 SIMD registers (64-bit or 128-bit) are distinct from the general purpose registers of the CPU, and timing must account for copying data between the general purpose register and SIMD registers (if required). However, the SIMD co-processor might be appropriate for performing the message expansion (see Section 23.5 for more discussion of this topic).

24.4 Conclusion

The CHI team has not implemented the CHI algorithms on ARM processors, but hope to do so in the near future. The performance figures are likely to be significantly better than for the 32-bit x86 processor, largely due to the increased number of registers and the availability of combined rotate-and-XOR and shift-and-XOR instructions.

25 Implementation Considerations for Low-End Processors

This section addresses the efficiency of the CHI algorithms on 8-bit processors (microcontrollers) and other low-end processors. By "low-end" we are refereing to the price of the processor.

We begin with a survey of some microcontrollers in Section 25.1. Memory requirements are discussed in Section 25.2. An interesting way for reducing the RAM footprint is expounded in Section 25.3. This section ends with a summary.

This sections mentions a variety of low-end processors. Mentioning these processors does not indicate an opinion on these processors: the processors are examined purely from the perspectively of evaluating computational efficiency of the CHI algorithms on low-end processors. The submitter apologizes for any incorrect information contained herein.

25.1 A Survey of Some Low-End Processors

In the 70's and 80's, this class was dominated by 8-bit microcontrollers including Motorola 6800, MOS 6502, Intel 6502, Intel 8048, Zilog Z80, Motorola 6809, Intel 8051, Intel 8080 and Intel 8085[8]. These 8-bit microcontrollers are still wide-spread in use, with a variety of new processors being introduced in recent years.. However, there is a trend towards larger word sizes. A variety of 16-bit and 32-bit microcontrollers are now available.

Freescale[9] produces derivatives of the Motorola 6800, and since 2000 have introduced new 8-bit models (including HCS08 and RS08 processors) and 16-bit models (including HCS12 and S12X processors) and has 32-bit models (Coldfire processors, based on Motorola 68000). Microchip[10] continues to produce 8-bit PIC (dating back to 1975), introduced 16-bit processors such as PIC in recent years, and even more recently announced 32-bit models. Other microcontroller manu-

[7] Qualcomm and Scorpion are trademarks of Qualcomm Inc. [11].

[8] Motorola 6800, MOS 6502, Intel 6502, Intel 8048, Zilog Z80, Motorola 6809 and Intel 8051, Intel 8080 and Intel 8085 are trademarks of their respective owners

[9] Freescale, 6800, CS08,RS08, HCS12, S12X and 56800E, Coldfire, 68000 are trademarks of Freescale Semiconductor, Inc. [5]

[10] Microchip and PIC are trademarks of Microchip Technology Inc. [8].

facturers include Atmel[11] (8 and 32-bit AVR processors), National Semiconductors[12] (8-bit COP8 and 16-bit CompactRISC CR16 processors), Renesas[13] (8-bit H8 derivatives), Zilog[14] (8-bit Z80, Z180, Z8, and more recently, eZ80 and eZ8), Rabbit[15] (Z80-based Rabbit R2000), and NXP[16] (8-bit 80C51-based, XA-core and 32-bit ARM7 and ARM9-based processors[17]).

An interesting change is that the ARM7 and ARM9 processors, which were the mid-range processors of yesteryear, are now affordable enough to be in this price range, as are some 32-bit x86 processors. In 5 years, ARM11 processors[18] may well be in this price range. The ARM processors are covered the discussion regarding implementing on Mid-Range processors (Section 24), so we don't discuss them in any more detail here.

Trends in the architectures of the the smaller microcontrollers include increasing number of registers, and pipelined architectures. Most processors in this range have a single execution unit, although some digital signal processor (DSP) microcontrollers have 2 concurrent execution units. Some of these processors have hardware cryptographic accelerators (see Section 9.4.4). Processors in this class often have limited Read-Only Memory (ROM) for storing the software, and limited RAM for storing variables during processing. The amount of RAM and ROM available continues to increase.

25.1.1 Programming Model of Low End Processors

The programming model varies greatly among low end processors. The older low-end processors had no dedicated registers aside from a single accumulator: most instructions directly references locations in RAM. Where registers are available, there can be limitations on when those registers are used.

Here is a time-line of the programming model of (some) of the processors in this class that include processors

1975 Motorola 6800 has 4 registers.

1976 Intel 8048 processor has 16 registers. The Zilog Z80 processor has 8 registers in two banks: typically one can only access one bank of the registers at a time.

1979-80 Motorola 6809 and Intel 8051 have 8 registers

1987 ARM processors with ARM architecture v2 onwards have 16 registers.

1996 Atmel AVR8 (1996) have 32 registers.

1997 National Semiconductors COP8 and CR16 processors have 16 registers.

2000 The Freescale 56800E DSPs (2000) have 11 registers.

[11] Atmel and AVR are trademarks of Atmel Corp. [3]
[12] National Semiconductors, COP8, CompactRISC and CR16 are trademarks of National Semiconductor Corporation [9]
[13] Renesas and H8 are trademarks of Renesas Technology Corp. [13]
[14] Zilog, Z80, Z180, Z8, eZ80 and eZ8 are trademarks of Zilog, Inc. [15]
[15] Rabbit and R2000 are trademarks of Rabbit Semiconductor [12]
[16] NXP, 80C51 and XA-core are trademarks of NXP Semiconductors
[17] ARM7 and ARM9 are trademarks of the ARM company [2]
[18] ARM11 is a trademark of the ARM company [2]

2001 Microchip PIC24 and dsPIC processors (2001) have 32 registers.

2006 The Atmel AVR32 has 32 registers.

2007 Microchip PIC32 has 32 registers.

Note that there is an increasing number of registers. The CHI algorithm can be implemented in programming models that have no registers. However, the CHI algorithm can take advantage of the increasing number of registers that are available.

The *MAP* function is the main component of the CHI algorithm that is affected by the number of registers. There are an unlimited number of ways to implement the *MAP* function, and some of these implementations may suit some processors more than others. In general, the more registers that are available, the faster the *MAP* function can be performed.

25.1.2 Operations Supported by low end processors

The CHI algorithms can be implemented with reasonable efficiency on processors that can perform the following operations: bit-moving operations such as shift and rotate operations; byte-moving operations; Bit-wise Logical AND, XOR, OR and NOT instructions; and addition operations.

Bit Moving-Operations and Byte-Moving Operations: Most microcontrollers support instructions for shifting by one bit position (to the left and to the right), and 1-bit rotate-with-carry instructions (to the left and to the right), and instructions for copying or moving bytes. For many processors, these are the only instructions of this nature available.

The rotation operations in the CHI algorithms can be performed using a combination of the 1-bit rotate-with-carry operations, left shift, right shift and byte-moving operations. The use of byte moving operations reduces the number of shift operations. Specified rotations that are multiples of eight only require re-addressing of the memory location. Some rotation amounts in the CHI algorithms have been chosen as a multiple of 8 for this very reason.

Only the more expensive processors of this class can afford to have variable-amount shift or variable-amount: Microchip PIC24F[19], National Semiconductor CR16C[20], Freescale Coldfire and Motorola 68000 derivatives, and the ARM processors. The National Semiconductor CR16C can also shift in 32-bit blocks, which may be used to some advantage.

The *SWAP*8 and *SWAP*32 operations only require re-addressing of the bytes in memory, and these operations are always supported. On 16-bit and 32-bit processors, the *SWAP*8 operation can benefit from a dedicated byte-swapping instructions: some processors have these instructions, but others don't.

Logical Operations: The bit-wise XOR operations can perform 64-bit XOR operations as eight independent operations. The bit-wise logical operations are used also to implement the *MAP* function. All processors support these operations: although the National Semiconductor COP8 and CRC16C processors do not support the bit-wise NOT operation. This is not an issue since NOT is equivalent to XORing with all ones.

Some processors, such as the Rabbit R2000, can perform some logical operations on larger word sizes than other logical operations. The R2000 can perform 16-bit wide AND and OR, but only supports 8-bit wide XOR and NOT.

[19]PIC10, PIC12 and PIC14 are trademarks of Microchip Technology Inc. [8].
[20]CR16C is a trademark of National Semiconductor Corporation [9]

Addition Operations: On an 8-bit processor, the 64-bit addition operations (and the 128-bit addition in CHI-384 and CHI-512) are implemented using a sequence of 8-bit addition operations, with each addition including the carry from the preceding addition. Most 8-bit microcontrollers support an add-with-carry operation that uses the carry up from a previous addition. If there is no add-with-carry operation, then there are methods for determining the carry after the addition, but this slows the addition. Of the microcontrollers we investigated, only the Microchip PIC10, PIC12 and PIC14 processors [8][21] do not have an "add-with-carry" operation.

Some processors support addition of words larger than the processors native register size. For example, the Zilog Z80 and the derivative Rabbit processors perform most operations on 8-bit values, but two 16-bit values can be added in a single instruction when the values are placed in the appropriate registers.

Summary: All microcontrollers support the operations required for implementing the CHI algorithms. Some microcontrollers have operations that will perform the computation more efficiently.

operations

25.2 Memory Requirements

The memory requirements are always divided into RAM requirements and ROM requirements. The RAM requirements refer to the memory required to store the variables that may change during the computation. The ROM requirements refer to the memory required for the instructions to implement the algorithm, and any constants used during the computation.

25.2.1 RAM Requirements

The RAM Requirements for the CHI Algorithms are shown in Table 18.

Variables	CHI-224 and CHI-256	CHI-384 and CHI-512
Input Hash Value $H^{(i-1)}$	48	72
`len`	8	16
Message Expansion State	64	128
Working Variables	48	72
Temporary Variables[22]	30	30
Total	198 Bytes	318 Bytes

Table 18: The RAM Requirements for the CHI Algorithms. All values are in bytes.

We examined the products available from a couple of manufacturers (Freescale and Microchip) to see if they supported this amount of internal RAM. This data was obtained from the web sites of the companies. The 16-bit processors always have sufficient internal RAM to implement any of the CHI algorithms, so we focus only on 8-bit processors (in this summary, RAM refers only to internal RAM):

Freescale 8-bit Processors: RS08 HCS08 and HC08 families.

[21] PIC10, PIC12 and PIC14 are trademarks of Microchip Technology Inc. [8].

- The RS08 processors are available with sufficient RAM to implement CHI-224 or CHI-256, and many have sufficient RAM to implement CHI-384 and CHI-512, but not enough RAM to implement CHI-384 and CHI-512.
- Most of the HCS08 processors have sufficient RAM to implement CHI-224 or CHI-256, and many have sufficient RAM to implement CHI-384 and CHI-512.
- The HC08 processors always have sufficient RAM to implement CHI-224 or CHI-256, and are available with sufficient RAM to implement CHI-384 and CHI-512.

Microchip 8-bit Processors: PIC10, PIC12, PIC14,PIC16 and PIC18 families.

- The PIC10 and PIC12 processors are not available with sufficient RAM to implement any of the CHI algorithms.
- The PIC14 processors have 192 bytes of RAM, just sufficient to implement CHI-224 or CHI-256, but but not enough RAM to implement CHI-384 and CHI-512.
- The PIC16 processors are available with sufficient RAM to implement CHI-224 or CHI-256, and only slightly less RAM that required implement CHI-384 and CHI-512.
- The PIC18 processors always have sufficient RAM to implement CHI-224 or CHI-256, and are available with sufficient RAM to implement CHI-384 and CHI-512.

This quick survey shows that there are a wide range of possibilities. Some processors are not available with sufficient internal RAM to implement any of the CHI algorithms (e.g. PIC10 and PIC12 processors). Other processors always have sufficient internal RAM to implement CHI-224 or CHI-256 and are available with sufficient RAM to implement CHI-384 or CHI-512 (e.g. HC08 processors).

In cases, where there is not sufficient internal RAM, external RAM can always be be used. This may slow the speed on the processor.

It is useful to compare the RAM requirements for the SHA-2 algorithms (shown in Table 19) against the RAM requirements for the CHI algorithms. The RAM Requirements for the CHI

Variables	SHA-224 and SHA-256	SHA-384 and SHA-512
Input Hash Value $H^{(i-1)}$	32	64
`len`	8	16
Message Expansion State	64	128
Working Variables	32	64
Minimum Temporary Variables (estimate)	4	8
Total	140 Bytes	280 Bytes

Table 19: The RAM Requirements for the SHA-2 Algorithms. All values are in bytes.

algorithms are larger than the RAM requirements for the SHA-2 algorithms. This is mostly a result of the larger hash state used in CHI algorithms. Every processor that was not available with sufficient RAM for a CHI algorithm, was also not available with sufficient RAM to implement the corresponding SHA-2 algorithm. This indicates that the larger RAM requirements of the CHI algorithms have no more effect that the RAM requirements of the SHA-2 algorithms.

25.2.2 ROM Requirements

The ROM requirements relate to the memory required to store instructions and constants for implementing the algorithm(s). We have not implemented the CHI algorithms on any microprocessors, so it is difficult to predict the ROM requirements.

We can predict the ROM Requirements for the constant values used by the CHI Algorithms; these are shown in Table 20. Note that the Initial Hash Value for CHI-224 is distinct from the Initial Hash Value for CHI-256, which is why the amount of ROM must be doubled. The same applies for CHI-384 and CHI-512 The ROM requirements for constants values are slightly higher than for the SHA-2 counterparts (not shown here), mostly due to the increased hash state size and CHI-224 and CHI-256 using more step constants than SHA-224 and SHA-256. Almost all the microcontrollers we examined had more ROM than required for the constant values in all the CHI algorithms.

Variables	CHI-224 and CHI-256	CHI-384 and CHI-512
Initial Hash Value $H^{(0)}$	$48 \times 2 = 96$	$72 \times 2 = 144$
Step Constants	160	320
Total	256 Bytes	488 Bytes

Table 20: The ROM Requirements for constant values in the CHI Algorithms. All values are in bytes. Note that there is an Initial Hash Value required for each hash size.)

The CHI algorithm breaks down into small components that would make good sub-routines with a small code footprint on 8-bit processors. For example, the step function breaks down into the following components:

`ROTR64` **sub-routine**

`XOR64` **sub-routine** Apply the correspond operation to a word64.

`PREMIX64` **sub-rountine** Apply the `ROTR64` and `XOR64` sub-routines to compute one of the $preR$, $preS$, $preT$ and $preU$ from the inputs

`THETA64` **sub-routine** Apply the θ_0 or θ_1 using the `ROTR64` and `XOR64` sub-routines.

`DATACOMBINE64` **sub-routine** Expand the step inputs and step constants (using the `THETA` sub-routine) and `XOR64` with $preR$, $preS$, $preT$ and $preU$.

`MAP8` **sub-routine** Applying MAP to a byte of the inputs R, S, T, U.

`MAP64` **sub-routine** Applying `MAP8` to each byte of the inputs R, S, T, U.

`MUX8` **sub-routine** Multiplex the outputs of the MAP, and apply `ROTR64` and a $SWAP8$ to obtain XX, YY, ZZ.

`ADD64` **sub-routine**

The message expansion can be implemented by defining `MU64` sub-routines using the `ROTR64` and `XOR64` sub-routines. All of these sub-routines should be fairly simple.

This leads us to believe that the CHI algorithms can be implemented with sufficient efficiently on 8-bit processors.

25.3 Reducing the RAM footprint of the CHI Algorithms

In software, most of the the processing occurs during the rotations and during the MAP function. The following approach may be useful in obtaining an implementation of the step function with small RAM footprint (and possibly small code size too). The idea is to compute the `ByteIndex`-th byte of each of X, Y, Z independently, so that only one byte is required for each temporary variable, rather than 8 bytes (a word64). The following sub-routines would be used.

`PREMIX8` **sub-routine** Perform the shifts and XORs that result in the `ByteIndex`-th byte of $preR$, $preS$, $preT$ and $preU$. The inputs would be only those bytes that contribute to the `ByteIndex`-th byte of $preR$, $preS$, $preT$ and $preU$. This requires 4 bytes of RAM.

`DATACOMBINE8` **sub-routine** Perform the shifts and XORs that result in the `ByteIndex`-th byte of $\theta_0^{\{256\}}(V_0)$ and $\theta_1^{\{256\}}(V_1)$ and XOR directly with the `ByteIndex`-th byte of $preR$, $preS$, $preT$ and $preU$. This requires no additional bytes of RAM.

`MAP8`,`MUX8` **sub-routines** : Defined above. The `MAP8` can implemented efficiently using only one or two bytes of RAM in addition to the bytes of RAM that will be used to store X, Y, Z.

This process requires only 5 or 6 bytes of RAM in addition to the RAM that is used to store X, Y, Z. These additional bytes can be used in rotating Y and applying the $SWAP8$ to get YY. For CHI-224 and CHI-256, the RAM storing X and Z become the destinations for AA and DD. CHI-384 and CHI-512 use the additional bytes when computing GG. The values of AA, DD (and GG for CHI-284 and CHI-512) are XORed in-place to the values of Q, C and F (respectively). The total amount of RAM required is the storage for X, Y, Z ($3 \times 8 = 24$ bytes) plus the 5 or 6 bytes for temporary variables: a total of (at most) 30 bytes.

25.4 Summary

The CHI algorithms can be implemented efficiently on most 8-bit processors and other low-end processors. The RAM requirements are more than for the SHA-2 counterparts, but this does not appear to be a limitation.

26 Hardware Implementation Considerations

This section considers hardware implementations of the CHI algorithms.

We have assumed that the parsing and padding would typically be implemented by the processor managing the hardware, so we have only considered hardware implementation of the compression function.

Three hardware versions are considered

Iterative Implementation in which the hardware implements only one step of the algorithm and the implementation is clocked multiple times when processing each block. A theoretical treatment is discussed in Sections 26.1, 26.2 and 26.3. A simulation is discussed in Section 26.6.

Unrolled Implementation in which all steps of the compression function are implemented in sequence, so the implementation is clocked a single times when processing each block. See Section 26.4.

Partial Implementation in which portions of the compression function are implemented in hardware and the remaining operations are left to be performed by a processor. See Section 26.5.

We conclude the section by examining the performance of a hardware simulation of CHI-224 and CHI-256.

Notes

- We considered the number of gates to implement a function, and the number of gates that were on the critical path of the algorithm.
- Inverter gates (such as in the NOT operations) are usually considered to be free, so they have not been included in this analysis.
- The message expansion feedback must be applied twice (to generate two new word64s) for each application of the step function.
- Theoretical hardware estimates for the *MAP* Function were obtained using the description:

```
INLINE void MAP(WORD R, WORD S, WORD T, WORD U, WORD *X)
{
   WORD i;


   i = (T | U) ^ R;
   X[0] = (i &   S) | ((~T ^ (U & R)) & ~S);
   X[1] = (i & ~S) | ((  U ^ (R & T)) &   S);
   X[2] = (U & (S ^ (T & ~R))) | (~U & (R ^ (T & S)));
}
```

- We list the total in terms of the a variety of gates:

  Flip-Flops A flip-flop can store a single bit of data.

  Multiplexor (MUX) An i-j MUX assigns the the j-outputs to be one of the i-inputs.

  Logical Opertions AND/OR/XOR An i-1 AND/OR/XOR assigns the the output bit to be the AND/OR/XOR of the i-inputs.

  Adders A w-bit Adder computes the addition of two w-bits inputs.

26.1 Theoretical Iterative Implementation: CHI-224 and CHI-256

Table 21 and Table 22 list the gates required for the various components of an iterative implementation of CHI-224 and CHI-256. The set of required gates is summarized in Table 23.

Step Counter An incrementing counter is required to index the step counters and to inform the implementation when to stop. The counter is implemented using an adder. It is not on the critical path.

Storing $H^{(i-1)}$ to feedback after last step Storage for the value of $H^{(i-1)}$ must be provided. An initializing multiplexor either inputs the externally provided value of $H^{(i-1)}$ (at the first step), and otherwise copies the values from the previous step. After the final computation, then stored value is XORed with the value of the working variables. Only the final XOR contributes to the critical path.

Message Expansion Storage for the message expansion state must be provided. An initializing multiplexor either inputs the externally provided message block, or allows the message expansion state to be updated via the LFSR. The LFSR is not on the critical path, but the storage is. The LFSR is implemented using 7-1 XOR gates, since every new bit is the XOR of 7 bits of the message expansion state.

Step Constants The step constants are stored as wires that are grounded or otherwise. The step constants for each of the 20 steps is selected using parallel 20-1 multiplexors. This introduces a 20-1 multiplexor on the critical path.

Step Function Storage for the working variables is required, and an initializing multiplexor either inputs the externally provided value of $H^{(i-1)}$, or allows the working variables to be updated via the step function operations. The value of $K_{2t+k} \oplus W_{2t+k}$ must be computed first (and are on the critical path). Then the following operations are required.

PRE-MIXING*, *DATA-INPUT R, S The values of R (respectively S)is obtain by XORing V_0 (respectively V_1) with copies of bits of B and D (respectively A and D). This requires 3-1 XOR gates.

PRE-MIXING*, *DATA-INPUT T, U The values of T (respectively U)is obtain by XORing three rotations of V_0 (respectively V_1) with copies of bits of A and D (respectively A and E). This requires 5-1 XOR gates. Since 5-1 XOR gates have longer delay than 3-1 gates, the 5-1 XOR gates are on the critical path.

***MAP* Function** Requires AND, OR and XOR operations, some of which are on the critical path.

***MAP* MUX** The three outputs of the *MAP* function are multiplexed to the 3 values X, Y, Z. This requires a 3-3 MUX, which is on the critical path.

POST-MIXING Requires two 64-bit adders, which are implemented in parallel. Hence only 1 64-bit adder is on the critical path.

FEEDBACK The outputs of hte *POST-MIXING* are rotated and XORed with working variables, requiring parallel 2-1 XOR operations, thus adding one 2-1 XOR operation to the critical path.

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Step Counter	5-bit Adder	1	-
Storing $H^{(i-1)}$ to feedback after last step			
Storage	Flip-Flop	6	-
Final XOR with $H^{(i)}$	2-1 XOR	6	1
Message Expansion			
Storage	Flip-Flop	8	-
XOR Feedback	7-1 XOR	2	-
Step Constants			
Storage	Flip-flop	40	1
Multiplexor	20-1 MUX	2	1

Table 21: Gates required for storing $H^{(i-1)}$ (to feedback after last step), the message expansion and the step constants in an iterative implementation of CHI-224 and CHI-256.

26.2 Theoretical Iterative Implementation: CHI-384 and CHI-512

The CHI-384 and CHI-512 algorithms are very similar to the CHI-224 and CHI-256 algorithms, the differences begin:

- CHI-384 and CHI-512 use a 6-bit adder in the message expansion (CHI-224 and CHI-256 use a 5-bit adder).
- Larger storage requirements for $H^{(i-1)}$, the message expansion state, and the working variables.
- More 2-1 XOR gates when XORing $H^{(i-1)}$ with the output of the step function.
- CHI-384 and CHI-512 use 8-1 XOR gates in the message expansion for (CHI-224 and CHI-256 use 7-1 XOR gates).
- CHI-384 and CHI-512 use 40-1 MUX gates to select the step constants (CHI-224 and CHI-256 use 20-1 MUX gates).
- CHI-384 and CHI-512 use 4-1 XOR gates and 6-1 XOR gates in the *PRE-MIXING* and *DATA-INPUT* phases for (CHI-224 and CHI-256 use 3-1 XOR gates and 5-1 XOR gates).
- CHI-384 and CHI-512 use a 3 parallel 64-bit adders in the message expansion (CHI-224 and CHI-256 use a 2 parallel 64-bit adders).

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Storage	Flip-Flop	6	-
$K_{2t+k} \oplus W_{2t+k}$	2-1 XOR	2	1
PRE-MIXING, *DATA-INPUT* R, S	3-1 XOR	2	-
PRE-MIXING, *DATA-INPUT* T, U	5-1 XOR	2	1
MAP Function	2-1 AND	9	2
	2-1 OR	4	1
	2-1 XOR	4	2
MAP MUX	3-3 MUX	1	1
POST-MIXING	64-bit Adders	2	1
FEEDBACK	2-1 XOR	2	1

Table 22: Gates required for the step function in an iterative implementation of CHI-224 and CHI-256.

Class of Operations	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Storage	Flip-flops	20	1
Logic Operations	2-1 AND	9	2
	2-1 OR	4	1
	2-1 XOR	12	3
	3-1 XOR	2	-
	5-1 XOR	2	1
	7-1 XOR	1	-
Multiplexors	3-3 MUX	1	1
	20-1 MUX	1	1
Adders	5-bit Adder	1	-
	64-bit Adder	2	1

Table 23: Summary of gates required for an iterative implementation of CHI-224 and CHI-256 CHI-224 and CHI-256.

26.3 Combined Iterative Implementation Supporting All CHI-384 and CHI-512

Recall that CHI-224 and CHI-256 are very similar to CHI-384 and CHI-512. Some simple additions to the iterative implementation for CHI-384 and CHI-512 can can result iterative implementation that supports all CHI Algorithms. The only changes are the interactions of bits in the message expansion and the step function. Algorithm-dependent 2-1 multiplexors can choose between two variable bit values, while logical AND operations can mask off a variable bit value if it should output zero for one algorithm. Table 27 shows the additional gates required, and Table 28 show the total number of gates of various types.

Message Expansion Suppose that the message expansion state for CHI-384 and CHI-512 is stored in word64s $W16, W15 \ldots, W1$, where $W16, W15$ are the least-recently updated word64s (corresponding the two step inputs output to the step function at this step) and $W02, W01$ are the most-recently updated word64s The Message expansion for CHI-224 and CHI-256 can use

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Step Counter	6-bit Adder	1	1
Storing $H^{(i-1)}$ to feedback after last step			
Storage	Flip-Flop	9	-
Final XOR	2-1 XOR	9	1
Message Expansion			
Storage	Flip-Flop	16	-
XOR Feedback	8-1 XOR	2	-
Step Constants			
Storage	Flip-flops	40	1
Multiplexor	40-1 MUX	2	1

Table 24: Gates required for storing $H^{(i-1)}$ (to feedback after last step), the message expansion and the step constants in an iterative implementation of CHI-384 and CHI-512.

the two most-recently updated word64s ($W02, W01$) and the size least-recently updated word64s ($W16, \ldots, W10$) of the CHI-384 and CHI-512 implementation. The linear feedback for CHI-224 and CHI-256 is similar to that for CHI-384 and CHI-512:

- One set of word64-wide multiplexor selects whether the contents of the $W10$ is updated from $W9$ (in the case of CHI-384 and CHI-512) or $W2$ (in the case of CHI-224 and CHI-384).
- The first logical functions $\mu_0^{\{256\}}(\cdot)$ and $\mu_0^{\{512\}}(\cdot)$ are identical, and both apply to $W15$, so no multiplexor is require there.
- The second logical functions both apply to $W02$, but $\mu_1^{\{256\}}(\cdot)$ and $\mu_1^{\{512\}}(\cdot)$ are distinct so a set of three word64-wide 2-1 multiplexors are required to ensure the correct bits are chosen here.
- Finally, an AND operation is applied to the bits of $W07$: outputting zero bits when CHI-224 and CHI-256 are being used; and outputting the bits of $W07$ when CHI-384 and CHI-512 are used. One set of word64-bit wide AND operations is required.

None of these gates are in the critical path.

Step Function The working variables for CHI-224 and CHI-256 uses the storage for working variables A to F of CHI-224 and CHI-256 .

- In the *PRE-MIXING* and *DATA-INPUT* phases, multiplexors are required to direct bits of (a) the XOR of the step input and step constant and (b) the working variables A, B, D, E to be XORed and be input to the *MAP* function at the correct bits position. This requires:
  - Two sets of word64-wide multiplexors for A, noting that *preS* always includes an unmodified copy of A, so only *preT* and *preU* require multiplexors.
  - One set of word64-wide multiplexors each for B and E.

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Storage	Flip-Flop	9	-
$K_{2t+k} \oplus W_{2t+k}$	2-1 XOR	2	1
PRE-MIXING, *DATA-INPUT* R, S	4-1 XOR	2	-
PRE-MIXING, *DATA-INPUT* T, U	6-1 XOR	2	1
MAP Function	2-1 AND	9	2
	2-1 OR	4	1
	2-1 XOR	4	1
MAP MUX	3-3 MUX	1	1
POST-MIXING	Adder	3	1
FEEDBACK	2-1 XOR	3	1

Table 25: Gates required for the step function of an iterative implementation of CHI-384 and CHI-512.

Class of Operations	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Storage	Flip-flops	34	1
Logic Operations	2-1 AND	9	2
	2-1 OR	4	1
	2-1 XOR	16	3
	4-1 XOR	2	-
	6-1 XOR	2	1
	8-1 XOR	2	-
Multiplexors	3-3 MUX	1	1
	40-1 MUX	1	1
Adders	6-bit Adder	1	1
	64-bit Adder	3	1

Table 26: Summary of gates required for an iterative implementation of CHI-384 and CHI-512.

  - Three sets of word64-wide multiplexors for D.

  One of these multiplexor gates is in the critical path.

- In the *PRE-MIXING* and *DATA-INPUT* phases, an AND operation is applied to the bits of coming from the working variables G and P: outputting zero bits when CHI-224 and CHI-256 are being used; and outputting the appropriate bits of G and P when CHI-384 and CHI-512 are used. Four sets of word64-bit wide AND operations are required. These gates are not in the critical path.

- One set of word64-wide multiplexor selects whether AA is XORed with F (in the case of CHI-224 and CHI-256) or Q (in the case of CHI-384 and CHI-512). One of these multiplexor gates is in the critical path.

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Message Expansion			
Alg-dep MUX	2-1 MUX	4	-
Alg-dep Masking	2-1 AND	1	-
Step Function			
PRE-MIXING, *DATA-INPUT* Alg-dep MUX	2-1 MUX	6	1
PRE-MIXING, *DATA-INPUT* Alg-dep Masking	2-1 AND	4	-
FEEDBACK	2-1 MUX	1	1

Table 27: Algorithm-dependent multiplexors that must be added to the iterative implementation of CHI-384 and CHI-512 in order for the implementation to also support CHI-224 and CHI-256.

26.4 Unrolled Implementation

An unrolled implementation save gates because there is no storage required (no flip flops) and no multiplexors are required. A summary of the gates required for an unrolled implementation of CHI-224 and CHI-256 is shown in Table 29. A summary of the gates required for an unrolled implementation of CHI-224 and CHI-256 is shown in Table 30.

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Storage	Flip-flops	34	3
Logic Operations	2-1 AND	9+8=17	2
	2-1 OR	4	1
	2-1 XOR	16	3
	4-1 XOR	2	-
	6-1 XOR	2	1
	8-1 XOR	2	-
Multiplexors	2-1 MUX	34+14=48	3
	3-3 MUX	1	1
	40-1 MUX	1	?
Adders	6-bit Adder	1	1
	64-bit Adder	3	1

Table 28: Summary of gates for theoretical hardware estimates for a combined iterative implementation of all CHI algorithms.

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Logic Operations	2-1 AND	180	40
	2-1 OR	80	20
	2-1 XOR	240	60
	3-1 XOR	40	-
	5-1 XOR	40	20
	7-1 XOR	20	-
Adders	64-bit Adder	40	20

Table 29: Summary of gates for theoretical hardware estimates for an unrolled implementation of CHI-224 and CHI-256.

26.5 Partial Implementations

In software, most of the cost occurs in the message expansion and the *PRE-MIXING*, *DATA-INPUT* and *MAP* phases of the step function. In hardware, most of the cost occurs in the addition operations in the *POST-MIXING* Phase. It may suit some processors to have a CHI-assistance co-processor implementing only the message expansion and the *PRE-MIXING*, *DATA-INPUT* and *MAP* phases of the step function. The processor can perform the *POST-MIXING* quite efficiently. This would make the processor fast at implementing the CHI algorithms, while decreasing the cost of the hardware implementation.

26.6 Hardware Simulation of CHI-224 an CHI-256

Bijan Ansari (of the CHI team) wrote an iterative implementation of the CHI compression function in the Verilog hardware description language [73]. This implementation was then synthesized using

Function	Type of Elements	Number of Elements	
		Total (÷64)	On Critical Path
Logic Operations	2-1 AND	360	80
	2-1 OR	160	40
	2-1 XOR	640	120
	4-1 XOR	80	-
	6-1 XOR	80	40
	8-1 XOR	80	-
Adders	64-bit Adder	120	40

Table 30: Summary of gates for theoretical hardware estimates for an unrolled implementation of CHI-384 and CHI-512.

Synopsys.[23] Two implementations were generated using first an adder that would provide the fastest possible implementation, and secondly an adder that would provide the the smallest possible implementation. ASIC and FPGA implementations were simulated. A summary of the FPGA and ASIC synthesis is provided in Table 32.

A quick survey of some ASIC and FPGA implementations of SHA-256 is provided in Table 31

The speed of the implementations is about 3.2 times the speed of the SHA-256 implementations at the same clock speed. However, the area is about twice that for SHA-256 implementations. Consequently, the speed/area ratio for CHI is about 1.5 times that for SHA-2 implementations.

[23]Synopsys is a registered trademark of Synopsys, Inc. [14]

Author	Technology	Area	Clock (MHz)	Throughput (MBps)	MBps/MHz
Ocean Logic [10]	ASIC 0.18μ	12.1-14.1K	102-400	78-316	0.76
	Xilinx Virtex E-8	612	71.5	56.5	0.79
	Xilinx Virtex II-5	612	78.8	62.3	0.79
Ting *et. al* [66]	Virtex XCV300E-8 FPGA	1261	88	87	0.99
CAST Inc. [4]	ASIC UMC 0.18 μm	20.5K	280	241*	0.97
	ASICTSMC 0.09 μm	18.0K	500	484*	0.97
	Xilinx Spartan-3 3s400-5	817	75	72.8	0.97
	Xilinx Spartan-3e 3s500e-5	1,007	95	92.2	0.97
	Xilinx Virtex-2 2v250-6	830	107	104*	0.97
	Xilinx Virtex-2Pro 2vp2-6	815	120	116*	0.97
	Xilinx Virtex-4 4vlx15-12	829	144	140*	0.97
	Xilinx Virtex-5 5vlx30-3	433	175	170	0.97
Helion [6]	ASIC 0.18 μm	$\approx$ 23k	150	145	0.97
		$\approx$ 26k	253	245	0.97
	Xilinx Spartan3 -5	752	98	95	0.97
	Xilinx Spartan3E -5	761	105	101.5	0.97
	Xilinx Virtex4 -11	758	162	157	0.97
	Xilinx Virtex5 -3	325	222	215.25	0.97

Table 31: Performance figures for some FPGA and ASIC implementations of SHA-256. Total Area is given in gates (for ASIC implementations) or slices (for FPGA implementations)

Technology	Total Area	Clock (MHz)	Throughput (MBps)	MBps/MHz
ASIC 0.13 μm Area opt.	62988	38	121	3.2
Speed opt.	101462	188	600	3.2
Xilinx Virtex 2-2000	1582	126	400	3.2

Table 32: Performance figures for FPGA and ASIC iterative implementations of CHI-224 and CHI-256. Total Area is given in μm^2 (for ASIC implementations) or slices (for FPGA implementations). One ASIC implementation is optimized for area, and the other is optimized for speed

Part V
NIST Submission Requirements and Evaluation Criteria

27 Introduction

The NIST call for SHA-3 submissions [57] specified a set of submission requirements [57, Section 2], minimum acceptability requirements [57, Section 3] and evaluation criteria [57, Section 4]. The submission documentation has focussed on providing a good explanation for the reader, rather than explaining the relationship to the SHA-3 requirements and evaluation criteria.

This part of the CHI algorithm submission clarifies how this submission fulfills the submission requirements, and explains why we believe the CHI algorithms rate well with respect to the evaluation criteria. Table 33 shows which sections of this document address the requirements and evaluation criteria of [57]. The evaluation criteria are reflections of the submission requirements,

Section of [57]	Description	Section of Submission
2.A	Cover Sheet	27.1
2.B.1	Algorithm Specification and Design Rationale	29
2.B.2	Computation Efficiency	30
2.B.3	Known Answer Tests and Monte Carlo Tests	27.2
2.B.4	Expected Strength	31
2.B.5	Cryptanalysis	32
2.B.6	Advantages and Limitations	33
2.C	Optical Media	27.2
2.D	Intellectual Property Statements / Agreements / Disclosures	27.1
3	Minimum Acceptability Requirements	28
4.A	Security-related Evaluation Criteria	34
4.B	Cost-related Evaluation Criteria	30
4.C	Flexibility-related Evaluation Criteria	35

Table 33: Relationship between the submission requirements and evaluation criteria of [57] and the applicable sections of this document.

27.1 Physical Documents

The cover sheet (Submission requirement 2.A) and intellectual property statements / agreements / disclosures (submission requirement 2.D) require signatures. The original signed physical documents are included with the submission package.

27.2 Optical Media

The necessary software implementations and test vectors (submission requirements 2.B.3 and 2.C) are provided in the NIST-specified directories.

28 Minimum Acceptability Requirements

NIST provided three minimum acceptability requirements [57, Section 3]:

1. The algorithm shall be publicly disclosed and available worldwide without royalties or any intellectual property restrictions.

2. The algorithm shall be implementable in a wide range of hardware and software platforms.

3. The candidate algorithm shall be capable of supporting message digest sizes of 224, 256, 384, and 512 bits, and shall support a maximum message length of at least $2^{64} - 1$ bits. Submitted algorithms may support other message digest sizes and maximum message lengths, and such features will be taken into consideration during the analysis and evaluation period.

The following three subsections address each of the individual minimum acceptability requirements.

28.1 Regarding Minimum Acceptability Requirement 1

Minimum acceptability requirement 1 states: "The algorithm shall be publicly disclosed and available worldwide without royalties or any intellectual property restrictions" [57, Section 3].

The CHI algorithm is described in Part I of this document, and is therefore presumed to be publicly disclosed. The CHI submitter and CHI owners have signed the appropriate intellectual property statements / agreements / disclosures specified in [57, Section 2.D], which relate to the algorithm being available worldwide without royalties or any intellectual property restrictions.

The submitters therefore believe that the CHI algorithms satisfy this minimum acceptability requirement.

28.2 Minimum Acceptability Requirement 2

Minimum acceptability requirement 2 states: "The algorithm shall be implementable in a wide range of hardware and software platforms" [57, Section 3].

To the best of the submitter's knowledge, the CHI algorithms use operations that can be implemented on nearly all hardware platforms. At this point in time, the submitters do not know the minimum area required for a CHI implementation. Hardware implementations are discussed in more detail in Section 26.

The only software platforms that may be unable to support the CHI algorithms are the microcontrollers that have been purchased with less RAM that than the algorithm requires (see Section 25.2 for a discussion of RAM requirements). It should be noted that, typically, the same microcontrollers can be purchased with sufficient RAM at a slightly higher price. With the exception of such microcontrollers, using very limited RAM, the submitters believe that the CHI algorithms can be implemented on all software platforms.

The submitters therefore believe that the CHI algorithms satisfy this minimum acceptability requirement.

For more discussion of the implementation of the CHI algorithms, the reader is referred to Sections 30, 33 and 35.

28.3 Minimum Acceptability Requirement 3

Minimum acceptability requirement 3 states: "The candidate algorithm shall be capable of supporting message digest sizes of 224, 256, 384, and 512 bits, and shall support a maximum message length of at least $2^{64} - 1$ bits. Submitted algorithms may support other message digest sizes and

maximum message lengths, and such features will be taken into consideration during the analysis and evaluation period" [57, Section 3].

The CHI algorithms support message digest sizes of 224, 256, 384, and 512 bits, and support a maximum message length of $2^{64} - 1$ bits (in the case of CHI-224 and CHI-256) and $2^{128} - 1$ bits (in the case of CHI-384 and CHI-512). The output from these algorithms may be truncated to a message digest of shorter length if desired.

The submitters therefore believe that the CHI algorithms satisfy this minimum acceptability requirement.

29 Algorithm Specification and Design Rationale

This section addresses the submission requirements of [57, Section 2.B.1], which we have partitioned into the following set of requirements:

Specification "A complete written specification of the algorithm shall be included, consisting of all necessary mathematical operations, equations, tables, diagrams, and parameters that are needed to implement the algorithm.". Part I contains a complete written specification of the algorithm, so we believe this requirement is satisfied.

Design Rationale "The document shall include design rationale (e.g., the rationale for choosing the specific number of rounds for computing the hashes) and an explanation for all the important design decisions that are made. It should also include 1) any security argument that is applicable, such as a security reduction proof, ..." This is addressed in Section 29.1.

Cryptanalysis "... and 2) a preliminary analysis, such as possible attack scenarios for collision-finding, first-preimage-finding, second-preimage-finding, length-extension attack, multi-collision attack, or any cryptographic attacks that have been considered and their results." This is addressed in Section 29.2.

The Tunable Parameter "In addition, the submitted algorithm may include a tunable security parameter, such as the number of rounds, which would allow the selection of a range of possible security/performance tradeoffs. .... Submissions that do not include such a parameter should include a weakened version of the submitted algorithm for analysis, if at all possible." This is addressed in Section 29.3.

Hashing Model "NIST is open to, and encourages, submissions of hash functions that differ from the traditional Merkle-Damgard model, using other structures, chaining modes, and possibly additional inputs. [Continuing to end of Section 2.B.1]" This is addressed in Section 29.4.

29.1 Design Rationale

This section addresses the documentation requirement: "The document shall include design rationale (e.g., the rationale for choosing the specific number of rounds for computing the hashes) and an explanation for all the important design decisions that are made. It should also include 1) any security argument that is applicable, such as a security reduction proof, ..." [57, Section 2.B.1].

Part II provides the design rationale for the CHI algorithms. Section 9 discusses the considerations that affected the CHI design choices. The choice of operations is argued in Section 10

, and the overall design philosophy is discussed in Section 11. The remaining sections in Part II explain the important design decisions that have been made. The choice for the number of steps is discussed in Section 21.

Informal security proofs have been provided our modifications to the Merkle-Damgaard construction Section 12.1 and the Davies-Meyer construction Section 12.2. The rationale behind the structure of the step function Section 13.1 and the structure of each of the phases in the step function: *PRE-MIXING* 13.3; *DATA-INPUT* 13.4; *MAP* 13.2; *POST-MIXING* 13.5; andchi:crypt:conc *FEEDBACK* 14.4. Design requirements for the details of each of these phases are found in Section 14. The rationale for structure of the message expansion is found in Section 13.6 and the design requirements for the details of message expansion are found in Section 14.5.

This requirement is adequately met by the CHI submission.

29.2 Cryptanalysis

This section addresses the documentation requirement: "... and 2) a preliminary analysis, such as possible attack scenarios for collision-finding, first-preimage-finding, second-preimage-finding, length-extension attack, multi-collision attack, or any cryptographic attacks that have been considered and their results." [57, Section 2.B.1].

Part III conducts a preliminary analysis of the CHI algorithms with respect to these attacks. These are further summarized in Section 34:

- Preliminary analysis with respect to collision-finding is summarized in Section 34.2.1.
- Preliminary analysis with respect to first-preimage-finding is summarized in Section 34.2.2.
- Preliminary analysis with respect to second-preimage-finding is summarized in Section 34.2.3.
- Preliminary analysis with respect to length-extension attack is summarized in Section 34.2.4.
- Preliminary analysis with respect to multi-collision attack is summarized in Section 34.2.5.

This requirement is adequately met by the CHI submission.

29.3 The Tunable Parameter

This section addresses the documentation requirement: "In addition, the submitted algorithm may include a tunable security parameter, such as the number of rounds, which would allow the selection of a range of possible security/performance tradeoffs. .... Submissions that do not include such a parameter should include a weakened version of the submitted algorithm for analysis, if at all possible." [57, Section 2.B.1].

For the submission, the number of steps has been set to 20 (for CHI-224 and CHI-256) and 40 (for CHI-384 and CHI-512) because the CHI team felt that these choices provided an appropriate security/performance tradeoff. A range of possible security/performance tradeoffs for the CHI algorithms can be selected by changing the number of steps.

This requirement is adequately met by the CHI submission.

29.4 Hashing Model

This section addresses the documentation requirement: "NIST is open to, and encourages, submissions of hash functions that differ from the traditional Merkle-Damgard model, using other structures, chaining modes, and possibly additional inputs. However, if a submitted algorithm cannot be used directly in current applications of hash functions as specified in FIPS or NIST Special Publications, the submitted algorithm must define a compatibility construct with the same input and output parameters as the SHA hash functions such that it can replace the existing SHA functions in current applications without any loss of security. The replacement of all SHA functions in any standardized application by this compatibility construct shall require no additional modification of the standard application beyond the alteration of any algorithm specific parameters already present in the standard, such as algorithm name and message block length. Submissions may optionally define other variants, constructs, or iterated structures for specific useful applications.

"It should be noted that standards which refer to a block length are generally designed with the Merkle-Damgard model in mind, and a number of applications make additional assumptions for example HMAC implicitly assumes that the message block length is larger than the message digest size. This is not to say that NIST requires the candidate algorithm to satisfy these assumptions, but in cases where the appropriate choice for a parameter such as message block length is not obvious, the submission package must specify a value that will preserve the security properties and functionality of any of the current standard applications." [57, Section 2.B.1].

The CHI algorithms use a simple modification of the Merkle-Dåmgard model, and this modification is explained in Section 12.1. This modification does not change the external interface to the hash function. From the perspective of an application using a CHI algorithm, the CHI algorithm and FIPS 180-2 algorithms are indistinguishable. The applications of hash functions, as specified in FIPS or NIST Special Publications, use the CHI algorithms in exactly the same manner as they would use the FIPS 180-2 algorithms. Consequently, the comments in the above quote have no bearing on the CHI algorithm submission.

The CHI algorithms all use a message block length that is larger than the message digest's size, so the second paragraph of the above requirement also has no bearing on the CHI algorithm submission.

This requirement is adequately met by the CHI submission.

30 Computational Efficiency and Memory Requirements

This section addresses the submission requirements of [57, Section 2.B.2]:

"**2.B.2** A statement of the algorithms estimated computational efficiency and memory requirements in hardware and software across a variety of platforms shall be included. At a minimum, the submitter shall state efficiency estimates for the "NIST SHA-3 Reference Platform" (specified in Section 6.B) and for 8-bit processors. (Efficiency estimates for other platforms may be included at the submitters discretion.) These estimates shall **each** include the following information, at a minimum:

"a. Description of the platform used to generate the estimate, in sufficient detail so that the estimates could be verified in the public evaluation process (e.g., for software running on a PC, include information about the processor, clock speed, memory, operating system, etc.). For hardware estimates, a gate count (or estimated gate count) should be included.

"b. Speed estimate for the algorithm on the platform specified in Section 6.B. At a minimum, the number of clock cycles required to:

"1. generate one message digest, and

"2. set up the algorithm (e.g., build internal tables)

"shall be specified for each message digest size required in the Minimum Acceptability Requirements section (Section 3) of this announcement.

"c. Any available information on tradeoffs between speed and memory."

Section 30.1 discusses the efficiency of the CHI algorithms on the reference platform. Section 30.2 discusses the efficiency of the CHI algorithms on 8-bit processors. Section 30.3 discusses the efficiency of the CHI algorithms on other processors. Section 30.4 discusses the efficiency of the CHI algorithms in hardware. We are not aware of tradeoffs between speed and memory for the CHI algorithms.

30.1 Efficiency on the Reference Platform

This section addresses the submission requirements of [57, Section 2.B.2.b]:

"b. Speed estimate for the algorithm on the platform specified in Section 6.B. At a minimum, the number of clock cycles required to:

"1. generate one message digest, and

"2. set up the algorithm (e.g., build internal tables)

"shall be specified for each message digest size required in the Minimum Acceptability Requirements section (Section 3) of this announcement. "

Note that no processing is required to set up the algorithm, so point 2.B.2.b.1 does not need to be addressed further.

The speed estimates were obtained using the optimized software implementations included in the submission package. The source was compiled using the NIST-specified compiler: "... the ANSI C compiler in the Microsoft Visual Studio 2005 Professional Edition" [57, Section 6.B.ii]; with the default optimization flags.

The CHI team built a copy of the NIST reference platform specified in [57, Section 6.B.i]: "NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition." The performance figures given were obtained by booting in either 32- or 64-bit mode as appropriate. Table 34 gives the performance figures in cycles per byte of hash input as measured using the "rdtsc" instruction, for the various sizes of the optimized implementations. In both cases, a large buffer is 1MB, and the the performance figure for 1MB should approximate the asymptotic performance. A small buffer ("Block") is 64 bytes for CHI-224 and CHI-256, and 128 bytes for CHI-384 and CHI-512 respectively. These figures are discussed in Section 23.

Algorithm	Buffer Size	Windows Vista Ultimate	
		32-bit Edition	64-bit Edition
CHI-224 and CHI-256	One block: 64B	51	26
	Large Buffer: 1MB	49	24
CHI-384 and CHI-512	One block: 128B	80	18
	Large Buffer: 1MB	78	16

Table 34: Performance figures for the CHI algorithms on the NIST-specified Reference platform. The figures are provided in bytes per cycle.

30.2 Efficiency on 8-bit processors

Section 25 discusses the factors influencing implementations of the CHI algorithms on low-end processors, and in particular, 8-bit microcontrollers.

Required Operations All the processors support instructions that allow implementing the CHI algorithms

Programming Model Some processors have registers and others do not, but there is a trend towards an increasing number of registers. The CHI algorithm can be implemented in programming models that have no registers. However, the CHI algorithm can take advantage of the increasing number of registers that are available to improve the performance.

RAM Requirements Section 25.3 shows how the design of the step function allows implementing with a small number of temporary variables, thus reducing RAM. The RAM requirements for the CHI algorithms are in Table 18. Some processors are not available with sufficient internal RAM to implement CHI algorithms: such processors can be provided with external RAM. Each CHI algorithm requires more RAM than the corresponding SHA-2 algorithm: CHI-224 and CHI-256 require 40% more RAM than SHA-224 and SHA-256; CHI-384 and CHI-512 require 13% more RAM than SHA-384 and SHA-512. This is mostly a result of the larger hash state used in CHI algorithms. However, every processor that was not available with sufficient RAM for a CHI algorithm, also had insufficient RAM to implement the corresponding SHA-2 algorithm. This indicates that the RAM requirements of the CHI algorithms are no more limiting than the SHA-2 algorithms.

ROM Requirements We have not implemented the CHI algorithms on any microprocessors, so it is difficult to predict the ROM requirements. We can predict the ROM Requirements for the constant values used by the CHI Algorithms. The ROM requirements for constants values are slightly higher than for the SHA-2 counterparts mostly due to the increased hash state size and CHI-224 and CHI-256 using more step constants than SHA-224 and SHA-256. We also demonstrated (Section 25.3) how the CHI algorithm breaks down into small components that would make good sub-routines with a small code footprint on 8-bit processors.

30.3 Other Software Platforms

The CHI team is hoping to implement the CHI algorithms on ARM-architecture-based processors in the near future. The performance figures are likely to be significantly better than for the 32-bit

x86 processor, largely due to the increased number of registers and the availability of combined rotate-and-XOR and shift-and-XOR instructions.

30.4 Computational Efficiency in Hardware

Section 26 discusses the factors influencing implementations of the CHI algorithms in hardware. A summary of the required gates and the critical path for an iterative implementation of the CHI algorithms is provided in Table 35. A summary of the required gates and the critical path for an unrolled implementation of the CHI algorithms is provided in Table 36.

The CHI team also synthesized an iterative implementation of the CHI-224 and CHI-256 algorithms (see Section 26.6). The performance figures are provided in Table 37. The speed/area ratio for CHI-224 and CHI-256 is about 1.5 times that for SHA-224 and SHA-256 implementations.

Class	Type of Elements	Number of elements for CHI-224 and CHI-256		Number of elements for CHI-384 and CHI-512	
		Total ÷64	On Critical Path	Total ÷64	On Critical Path
Storage	Flip-flops	20	3	34	3
Logic Operations	2-1 AND	9	2	9	2
	2-1 OR	4	1	4	1
	2-1 XOR	12	3	16	3
	3-1 XOR	2	-	-	-
	4-1 XOR	-	-	2	-
	5-1 XOR	2	1	-	-
	6-1 XOR	-	-	2	1
	7-1 XOR	1	-	-	-
	8-1 XOR	-	-	2	-
Multiplexors	3-3 MUX	1	1	1	1
	20-1 MUX	1	1	-	-
	40-1 MUX	-	-	1	1
Adders	5-bit Adder	1	1	-	-
	6-bit Adder	-	-	1	1
	64-bit Adder	2	1	3	1

Table 35: Summary of gates and critical path for an iterative implementation of the CHI algorithms.

The speed of the implementations is about 3.2 times the speed of the SHA-256 implementations at the same clock speed. However, the area is about twice that for SHA-256 implementations. Consequently, the speed/area ratio for CHI-224 and CHI-256 is about 1.5 times that for SHA-2 implementations.

Class	Type of Elements	Number of elements for CHI-224 and CHI-256		Number of elements for CHI-384 and CHI-512	
		Total ÷64	On Critical Path	Total ÷64	On Critical Path
Logic Operations	2-1 AND	180	40	360	80
	2-1 OR	80	20	160	40
	2-1 XOR	240	60	640	120
	3-1 XOR	40	-	-	-
	4-1 XOR	-	-	80	-
	5-1 XOR	40	20	-	-
	6-1 XOR	-	-	80	40
	7-1 XOR	20	-	-	
	8-1 XOR	-	-	80	-
Adders	64-bit Adder	40	20	120	40

Table 36: Summary of gates for theoretical hardware estimates for an unrolled implementation of CHI the CHI algorithms.

Technology	Total Area	Clock (MHz)	Throughput (MBps)	MBps/MHz
ASIC 0.13 μm Area opt.	62988	38	121	3.2
Speed opt.	101462	188	600	3.2
Xilinx Virtex 2-2000	1582	126	400	3.2

Table 37: Performance figures for FPGA and ASIC iterative implementations of CHI-224 and CHI-256. Total Area is given in μm^2 (for ASIC implementations) or slices (for FPGA implementations). One ASIC implementation is optimized for area, and the other is optimized for speed

31 Statement of Expected Strength

This section addresses the submission requirements of [57, Section 2.B.4]:

"**2.B.4** A statement of the expected strength (i.e., work factor) of the algorithm shall be included, along with any supporting rationale, for each of the security requirements specified in Sections 4.A.ii and 4.A.iii, and for **each** message digest size specified in Section 3."

The security requirements specified in Sections 4.A.ii of [57] relate specific requirements when hash functions are used to support HMAC, Pseudo Random Functions (PRFs), and Randomized Hashing. The CHI algorithms supports HMAC, PRFs and randomized hashing using the constructions currently applied for SHA-2 algorithms. This submission provides no additional constructions for PRFs nor randomized hashing, so some requirements of Sections 4.A.ii do not apply to this submission

31.1 Statement of Expected Strength of CHI-224

The following lists the expected effort for various attacks on CHI-224:

- Collision attacks require an effort of approximately 2^{112}.
- Preimage attacks require an effort of approximately 2^{224}.

- Second-preimage resistance for any message shorter than 2^k bits ($k < 112$) require an effort of approximately $2^{(224-k)}$.
- Any m-bit hash function specified by taking a fixed subset of CHI-224's output bits is expected to meet the above requirements with m replacing 224.
- Length-extension attacks require an effort of approximately 2^{192}.
- Multi-collision attacks require a minimum effort of 2^{192}.
- When CHI-224 is used with HMAC to construct a PRF as specified (using the constructions currently applied for SHA-2 algorithms), that PRF resists any distinguishing attack that requires much fewer than 2^{112} queries and significantly less computation than a preimage attack (preimage attacks are discussed above).

31.2 Statement of Expected Strength of CHI-256

The following lists the expected effort for various attacks on CHI-256:

- Collision attacks require an effort of approximately 2^{128}.
- Preimage attacks require an effort of approximately 2^{256}.
- Second-preimage resistance for any message shorter than 2^k bits ($k < 128$)require an effort of approximately $2^{(256-k)}$.
- Any m-bit hash function specified by taking a fixed subset of CHI-256's output bits is expected to meet the above requirements with m replacing 256.
- Length-extension attacks require an effort of approximately 2^{192}.
- Multi-collision attacks require a minimum effort of 2^{192}.
- When CHI-256 is used with HMAC to construct a PRF as specified (using the constructions currently applied for SHA-2 algorithms), that PRF resists any distinguishing attack that requires much fewer than 2^{128} queries and significantly less computation than a preimage attack (preimage attacks are discussed above).

31.3 Statement of Expected Strength of CHI-384

The following lists the expected effort for various attacks on CHI-384:

- Collision attacks require an effort of approximately 2^{192}.
- Preimage attacks require an effort of approximately 2^{384}.
- Second-preimage resistance for any message shorter than 2^k bits ($k < 192$) require an effort of approximately $2^{(384-k)}$.
- Any m-bit hash function specified by taking a fixed subset of CHI-384's output bits is expected to meet the above requirements with m replacing 384.

- Length-extension attacks require an effort of approximately 2^{288}.
- Multi-collision attacks require a minimum effort of 2^{288}.
- When CHI-384 is used with HMAC to construct a PRF as specified (using the constructions currently applied for SHA-2 algorithms), that PRF resists any distinguishing attack that requires much fewer than 2^{192} queries and significantly less computation than a preimage attack (preimage attacks are discussed above).

31.4 Statement of Expected Strength of CHI-512

The following lists the expected effort for various attacks on CHI-512:

- Collision attacks require an effort of approximately 2^{226}.
- Preimage attacks require an effort of approximately 2^{512}.
- Second-preimage resistance for any message shorter than 2^k bits ($k < 256$) require an effort of approximately $2^{(512-k)}$.
- Any m-bit hash function specified by taking a fixed subset of CHI-512's output bits is expected to meet the above requirements with m replacing 512.
- Length-extension attacks require an effort of approximately 2^{288}.
- Multi-collision attacks require a minimum effort of 2^{288}.
- When CHI-512 is used with HMAC to construct a PRF as specified (using the constructions currently applied for SHA-2 algorithms), that PRF resists any distinguishing attack that requires much fewer than 2^{256} queries and significantly less computation than a preimage attack (preimage attacks are discussed above).

32 Analysis With Respect To Known Attacks

This section addresses the submission requirements of [57, Section 2.B.5]:

"**2.B.5** An analysis of the algorithm with respect to known attacks (e.g., differential cryptanalysis) and their results shall be included.

"To prevent the existence of possible trap-doors in an algorithm, the submitter shall explain the provenance of any constants or tables used in the algorithm, with justification of why these were not chosen to make some attack easier.

"The submitter shall provide a list of known references to any published materials describing or analyzing the security of the submitted algorithm. The submission of copies of these materials (accompanied by a waiver of copyright or permission from the copyright holder for the SHA-3 public evaluation purposes) is encouraged."

Regarding the first paragraph, Section 34 contains a good summary of the analysis contained in this submission. Rather than repeat the results, the reader is direct to that section.

Regarding the last paragraph, the submitter is not aware of any published materials describing or analyzing the security of the submitted algorithm.

It remains to address the middle paragraph: "To prevent the existence of possible "trap-doors" in an algorithm, the submitter shall explain the provenance of any constants or tables used in the algorithm, with justification of why these were not chosen to make some attack easier" [57, Section 2.B.5].

The constants for the CHI algorithms have been generated using a similar technique to the generation of the constants for the SHA-2 algorithms: using the fractional part of the roots of prime numbers. To save memory, the CHI algorithms share some constants. All CHI algorithms use the same set of step constants. The word64s in the initial hash value for CHI-224 are a subset of the word64s in the initial hash value for CHI-384. Similarly, the word64s in the initial hash value for CHI-256 are a subset of the ord64s in the initial hash value for CHI-512. There is no reason to believe that this choice of constants may make some attacks easier.

The *MAP* function (Section 4.1) was chosen from a set of functions that fulfilled the set of requirements outline in Sections 13.2 and 14.1. The designers would happily accept a *MAP* function generated by another party, provided that function fulfilled the same requirements.

33 Statement of Advantages and Limitations

This section addresses the submission requirements of [57, Section 2.B.6], "**2.B.6** A statement that lists and describes the advantages and limitations of the algorithm shall be included."

We list the advantages and limitations under three headings: Security, Software Efficiency, Hardware Efficiency, and Other.

33.1 Advantages

The advantages of the CHI algorithms include:

Security: Familiarity The algorithm components are well analyzed and well understood. The strength of the algorithm can be evaluated using existing techniques.

Security: Simplicity The CHI algorithms are based on the well-known Feistel network, and the step functions use a simple sequence of phases. All CHI algorithms use the same *MAP* function (Section 4.1), so only one complex function needs analyzing. The diffusion is provided by rotation operations and a byte-swapping operation which are easy to understand, and relatively easy to analyze.

Security: Highly Non-linear Component The *MAP* function provides excellent "confusion".

Security: Good Diffusion The rotations used for diffusion have been specifically chosen to prevent pairs of bits from interacting multiple times, so every bit must interact with a large number of other bits, resulting in good diffusion.

Software Efficiency: Suitable for Multiple Execution Units Most of the step function operations can be ordered so that there are three (or more) concurrent operations taking place. Many processors allow a large degree of concurrency, and multiple execution units will only become more prevalent in the future. The CHI algorithms are well positioned to exploit multiple execution units.

Software Efficiency: Flexibility The core of the algorithm is the *MAP* function, which can be implemented efficiently on any processor.

Software Efficiency: SIMD-Friendly Message Expansion The regular structure of message expansion can be exploited on SIMD co-processors.

Hardware Efficiency: Fewer addition operations By limiting the number of addition operations, the size of hardware implementations are reduced, and the speed is increased.

Hardware Efficiency: Addition operations are in Parallel The addition operations occur in parallel, decreasing the critical path.

Hardware Efficiency: *MAP* Takes relatively few Gates, and is parallelizable In software, the *MAP* function is much slower to implement than the addition operations. In hardware, the *MAP* function takes significantly fewer gates (unless a very slow adder is used). Furthermore, each bits of the MAP output can be computed in parallel.

Hardware Efficiency: Suitable for Partial-Hardware Acceleration Some processors may wish to have a hardware accelerator implementing most of the step function, leaving the final additions, rotation operations and XORs to be performed in the processor. This can save implementing a second addition operation on the chip.

Hardware Efficiency: Shared Circuitry A CHI-384/512 hardware implementation can re-use a large portion of the circuitry of the CHI-224/256 implementation.

Hardware Efficiency: Efficient Message Expansion The message expansion is a linear feedback shift register, which can be implemented very efficiently in hardware.

Other: Energy Efficient By limiting the number of addition operations, the energy usage is decreased.

33.2 Limitations

The limitations of the CHI algorithms include:

Security: Complexity of the *MAP* The *MAP* function looks complex in comparison to the *Maj* and *Ch* functions of the SHA-2 algorithms. Some people may see this as a limitation. as this makes analysis of the CHI algorithms more complex than analysis of SHA-2.

Software Efficiency: Rotation Operations The rotation operations can be slow to implement on 8-bit processors. However, we do not know of a more efficient method to provide diffusion.

Software/Hardware Efficiency: Larger Message Block The size of the message block results in a large message expansion state: large message expansion state requires significant amount of RAM on processors and a significant number of gates in flip-flops in hardware. We believe that the size of the message blocks is appropriate for the strengths of the algorithms, and decreasing the size could lead to compromise.

Hardware Efficiency: Large Set of Step Constants This significantly increases the area of hardware implementations.

34 Security-Related Evaluation Criteria

The security-related evaluation criteria in [57, Section 4.A] are provided under five headings:

- [57, Section 4.A.i] Applications of the hash functions. The CHI algorithms can be used in these applications in the same way that the SHA-2 algorithms are used in these applications.
- [57, Section 4.A.ii] Specific requirements when hash functions are used to support HMAC, Pseudo Random Functions (PRFs), and Randomized Hashing. This is discussed in Section 34.1.
- [57, Section 4.A.iii] Additional security requirements of the hash functions. This is discussed in Section 34.2.
- [57, Section 4.A.iv] Evaluations relating to attack resistance. This relates to events after the submission, so we do not address this topic in this submission
- [57, Section 4.A.v] Other consideration factors. Factors such as simplicity are discussed further in Section 35.2.

In our opinion, the CHI algorithms perform well against the security-related evaluation criteria

34.1 Specific Requirements for Certain Applications

The security requirements specified in Sections 4.A.ii of [57] relate specific requirements when hash functions are used to support HMAC, Pseudo Random Functions (PRFs), and Randomized Hashing. The CHI algorithms supports HMAC, PRFs and randomized hashing using the constructions currently applied for SHA-2 algorithms. This submission provides no additional constructions for PRFs nor randomized hashing, so some requirements of Sections 4.A.ii do not apply to this submission

34.2 Additional Requirements for Hash Functions

In addressing these requirements, we consider two classes of attacks: "black-box" attacks and tailored attacks.

Black box attacks exploit the Input/Output Formatting and META structure, and are independent of the internal workings of the underlying block cipher. The Input/Output formatting uses a modified Merkle-Damgård construction, which is analyzed in the design rationale in Section 12.1. The modification to the Merkle-Damgård construction prevents length-extension attacks. The META structure uses a modified Davies-Meyer construction, which is analyzed in the design rationale in Section 12.2. The modification to the Davies-Meyer construction prevents fixed points, which are the only known weakness of the Davies-Meyer construction. (see Section 34.2.4).

Tailored attacks that exploit the specific internal workings of the underlying block cipher. The underlying block cipher uses an unbalanced Feistel network [30], with a non-linear component (the *MAP* phase) layered between diffusion components (the *PRE-MIXING* and *POST-MIXING* phases). The step inputs are generated using a linear feedback shift register,. The step inputs undergo a linear expansion (the *DATA-INPUT* phase) prior to being combined in the step function.

34.2.1 Collision Resistance

Black Box Attacks: We are not aware of black box attacks on the CHI algorithms' Input/Output formatting that have complexity better than $2^{s/2}$. We are not aware of black box attacks on the CHI algorithms' META structure that have complexity better than $2^{s/2}$. The large hash state size (384-bits) of CHI-224 and CHI-256 ensure that collisions in the internal hash state cannot be found with complexity better than 2^{192}, which is significantly higher than the required bound of 2^{112} and 2^{128} respectively.

Tailored Attacks: The Wang *et al.* differential attacks on MD5 [67, 69], SHA-0 [70] and SHA-1[68] have been the basis of the most successful attacks on the internal workings of a hash function. Minor improvements have been achieved using neutral-bit technique [20, 21], tunnels [40] and boomerang attacks [48]. Resistance to these attacks was the primary motivation for the design of the CHI algorithms. Differential attacks are addressed as part of the analysis in Part III. The initial analysis indicates that differential attacks on the CHI algorithms require more effort than required for collision attacks. Consequently, the internal workings of the CHI algorithms are believed to provide sufficient collision resistance.

We further believe that tailored attacks can not take advantage of the rotation in the modified Merkle-Damgård construction and the modified Davies-Meyer construction.

34.2.2 Preimage Resistance

We do not have an analysis of the CHI algorithm with respect to pre-image attack. We imagine that many of the techniques used with block ciphers, such as linear cryptanalysis, can be used to analyze the strength of the CHI functions. We believe that the CHI algorithms will provide adequate protection against these attacks.

34.2.3 Second Preimage Resistance

A pre-image attack on a CHI algorithm is equivalent to performing a related-key attack on the underlying block cipher. We analyzed the underlying block cipher with respect to known attacks on block ciphers in Section 19. For CHI-s ($s \in \{224, 256, 384, 512\}$, the complexity of these attacks appeared to exceed approximately 2^s.

34.2.4 Length Extension Attacks

The CHI algorithms include a countermeasure against length-extension attacks. The analysis of this countermeasures can be found in Section 12.2.

34.2.5 Multi-Collision Attacks

Multi-collision attacks and related attacks are addressed in Section 12.1.2. The hash state of CHI-224 and CHI-256 is larger than in their SHA-2 counterparts, which has the effect that these two CHI algorithms provide significant additional protection against these attacks. The hash state of CHI-384 and CHI-512 is only slightly larger than in their SHA-2 counterparts, so CHI-384 and CHI-512 do not offer significant additional protection against these attacks. However, the minimum complexity of these attacks on CHI-384 and CHI-512 (2^{288}) is so large that we do not consider these attacks to be a significant threat for many years.

35 Flexibility-Related Evaluation Criteria

The flexibility-related evaluation criteria in [57, Section 4.C] are provided under two headings:

- [57, Section 4.C.i] Flexibility. This topic is discussed in Section 35.1.
- [57, Section 4.C.ii] Simplicity. This topic is discussed in Section 35.2.

These sections show that the CHI algorithms performs well against the flexibility-related evaluation criteria.

35.1 Flexibility

This evaluation criteria is explained in the following terms [57, Section 4.C.i]:

"Candidate algorithms with greater flexibility will meet the needs of more users than less flexible algorithms, and therefore, are preferable. However, some extremes of functionality are of little practical use (e.g., extremely short message digest lengths) - for those cases, preference will not be given. Some examples of "flexibility" may include (but are not limited to) the following:

"a. The algorithm has a tunable parameter which allows the selection of a range of possible security/performance tradeoffs.

"b. The algorithm can be implemented securely and efficiently on a wide variety of platforms, including constrained environments, such as smart cards.

"c. Implementations of the algorithm can be parallelized to achieve higher performance efficiency."

Section 29.3 discusses the tunable parameter for the CHI algorithms. We have discussed how the algorithm can be implemented efficiently on a wide variety of platforms (Section 30), and we have avoided operations that could it difficult to implement securely (Section 9.3.2). The ability to parallelize the algorithm is summarized in Section 22.1, with a variety of examples found throughout Part IV.

35.2 Simplicity

This section addresses the evaluation criteria: "A candidate algorithm will be judged according to its relative design simplicity" [57, Section 4.C.ii].

Very few algorithms in symmetric cryptography would be considered simple, so the description "relative design simplicity" is very apt. Design simplicity can be evaluated from various perspectives.

- Familiarity.
- Ease of remembering the algorithm.
- Ease of implementing the algorithm.
- Ease of analyzing the algorithm with respect to attacks.

These perspectives are addressed in Sections 35.2.1, 35.2.2, 35.2.3 and 35.2.4. These sections indicate support our opinion that, although the CHI algorithms perform only relatively well with respect to the ease of remembering the algorithm, the CHI algorithms perform well when evaluated from the other perspectives.

35.2.1 Familiarity

At the Input/Output formatting and META structure levels, the algorithm is very similar to the SHA-3 predecessors (the MD5, SHA-1 and SHA-2 algorithms), and are relatively easy to understand.

At the MACRO structure level, the algorithm consists of the familiar message expansion and step function. The message expansion is a linear feedback shift register using rotation and shift operations for diffusion, (similar to the message expansion used in SHA-2 algorithms) which is a familiar concept in cryptography.

The overall structure of the step function uses a unbalanced Feistel-like structure, similar to that used in the SHA-3 predecessors and block ciphers such as the Data Encryption Standard (DES) [52]. This structure computes the output of a function applied to a subset of of working variables and a set of step inputs, and then XORs this output with the working variables that were not inputs to the function. In a manner similar to SHA-3, the CHI algorithms feedback into non-adjacent positions in the state (the set of working variables). The internal workings for computing the output may appear, at first, to be unfamiliar. However, as explained in Section 13.1, the sequence of phases in CHI follows a similar sequence of phases in DES. The internal workings of those phases are also derived from familiar sources:

1. The *PRE-MIXING* phase uses rotations and XORs to achieve diffusion in a manner similar to the SHA-2 algorithms

2. The *DATA-INPUT* phase expands the step inputs and step constants to improve the resistance to differential attacks. In the SHA-2 Algorithms, there are two feedback points, suggesting that two step inputs should be used. However, a single step input is used instead. This indicates that the approach used in SHA-2 was already used in the SHA-2 algorithms. The σ_0 and σ_1 functions of the SHA-2 algorithms were also the inspiration for the θ_0 and θ_1 functions that are used for expanding the step inputs and step constants.

3. The *MAP* phase uses a bit-sliced non-linear function. The SHA-3 predecessors used such functions with three inputs and one output. The inspiration to used more inputs and more outputs can be traced to the Serpent block cipher [17]. Multiplexing the *MAP* outputs is not used often in cryptography, but it is a familiar concept.

4. The *POST-MIXING* phase uses rotations and addition operations (used in MD5, SHA-1 and the SHA-2 algorithms), and the *SWAP*8 operation is the only unfamiliar operation present here.

The CHI algorithms are comprised of components that are familiar to those in the field of cryptography; hence, we believe that the CHI algorithms perform well when rated against this perspective of simplicity.

35.2.2 Ease of Remembering the Algorithm

The previous section describes how the CHI algorithms are comprised of components that are familiar to those in the field of cryptography. In this section we discuss the ease of remembering the CHI algorithms.

At the Input/Output formatting and META structure levels, the algorithms can be remembered as being identical to the SHA-3 predecessors, with simple modifications:

- When processing the last message block, the modified Merkle-Dåmgard Construction [50, 28] differs in that a one bit rotation is applied to each of the working variables prior to applying the step function.
- The modified Davies-Meyer Construction [46] applies a one bit rotation to the hash input prior to XORing with the output of the step function.

At the MACRO structure level, the CHI algorithm will probably not be the easiest algorithm to remember. However, many aspects of the algorithm are not so difficult to remember.

The structure of the message expansion is easily remembered as being an LFSR similar to the message expansion of the SHA-2 algorithms. The step function uses an approach similar to the SHA-2 algorithms, feeding back into non-adjacent positions: in the case of the CHI algorithms, the feedback positions occur every 3 working variables. The feedback values are computed using a series of phases similar to DES (see Section 13.1). The inner workings of these phases are not too difficult to remember. The *PRE-MIXING* phase must result in four word64s, and uses XOR rotations (32-bit rotations for CHI-224 and CHI-256, 64-bit rotations for CHI-384 and CHI-512). The *DATA-MIXING* phase expands the two step input (XORed with the step constants) to four word64s using functions similar to the σ_0 and σ_1 functions of the SHA-2 algorithms The *MAP* phase uses bit-slicing similar to the Serpent block cipher [17], followed by a multiplexor to reorder the outputs. The *POST-MIXING* phase combines these *MAP* outputs to either two or three feedback values (depending on the choice of algorithm). The *POST-MIXING* first modifies one *MAP* outputs using rotation followed by *SWAP*8. The first two feedback values are obtained by (1) adding this value with one of the other *MAP* output and (2) adding this value to the other *MAP* output which has had 32-bit halves swapped. These two feedback values are rotated (in 64-bit blocks) before begin fed-back. For CHI-384 and CHI-512, the final feedback values is obtain by adding the remaining "un-paired" pair of *MAP* outputs, one values has the 32-bit halves swapped, and the other values is shifted by one bit

This (verbal) description is sufficient to remember most of how the CHI algorithms work. The finer details, such as the rotation amounts and the description of the *MAP* function, are not that easy to remember. These finer details are identical for each step. The *MAP* function simply implements 64 parallel and identical substitution boxes that map from 4-bit to 3 bits. The *MAP* function is as easy to remember as a single row of a DES S-box: DES has six S-boxes with four rows each, so there is far less to remember in comparison to DES.

The CHI algorithms are comprised of components that are familiar to those in the field of cryptography; hence, we believe that the CHI algorithms perform relatively well when rated against this perspective of simplicity.

35.2.3 Ease of Implementing the Algorithm

Regarding implementors knowing what to implement. The algorithm uses familiar operations which

can be implemented in a relatively obvious manner. The most complex operation in the CHI family occurs only in CHI-128: this operation is the 128-bit addition required for computing the message length, and this operation is present in CHI only because it was present in the SHA-384 and SHA-512 algorithms.

Regarding accidental incorrect implementation. The strict ordering of the phases should decrease the chances of accidentally re-ordering operations in an implementation. Making the process in each step identical (with the exception of the step dependent multiplexing and the step constants), should reduce the chances of incorrect implementation.

The CHI team wrote two independent reference implementations of the CHI specification, so that any ambiguity in the specification could be easily identified by inconsistencies in the implementation output. The initial inconsistencies in the two implementations were quickly resolved; evidence of the ease in implementing the algorithm.

Consequently, we believe that the CHI algorithms perform well when rated against this perspective of simplicity.

35.2.4 Ease of Analyzing the Algorithm With Respect to Attacks

The background research at the beginning of the project (Section 9.2) indicated that ease of analysis was essential for a good submission. Ease of analysis is affected by two things: the ease of applying the known techniques to the algorithm

The field of hash function analysis is still immature, and there are few useful techniques available. Those techniques that have been developed are tailored towards hash functions similar to the SHA-3 predecessors: the MD5, SHA-1 and SHA-2 algorithms. Other block-cipher-based techniques can apply in some circumstances, so these two should be accounted for.

The CHI team opted for an algorithm that could be analyzed using either the techniques applied to the SHA-3 predecessors (such as the attack of Wang *et al.* [67, 70, 68] or using the techniques that have been applied to other block ciphers.

The analysis of Wang *et al.* exploits the interaction between addition operations, bit-wise logical operations and operations that re-order bits. The CHI algorithms use exactly these operations. The block cipher analysis techniques that have been most successful in recent years are differential cryptanalysis [19] and linear cryptanalysis [44]. These techniques are particularly easy to apply to algorithms that predominantly use the XOR operation. The CHI algorithms do use addition operations, but the predominant operations are the XOR operation, so these two techniques will be easily applied to the CHI algorithms.

Saying that the CHI algorithms can be easily analyzed by these techniques, is significantly different from saying that the the CHI algorithms can be easily broken using these techniques. Rijndael (the AES block cipher) is easily analyzed with respect to differential cryptanalysis and linear cryptanalysis, and that analysis convinces the cryptanalyst that Rijndael resists these attacks. Similarly, we consider the CHI algorithms to be easily analyzed with respect to the aforementioned forms of analysis, and we hope that the analysis convinces the cryptanalyst that the CHI algorithms resists these attacks.

For these reasons, we believe that the CHI algorithms perform well when rated against this perspective of simplicity.

References

[1] Advanced Micro Devices, Inc. http://www.amd.com.

[2] ARM Ltd. http://www.arm.com.

[3] Atmel Corporation. http://www.atmel.com.

[4] CAST Inc. http://www.cast-inc.com.

[5] Freescale Semiconductor, Inc. http://www.freescale.com.

[6] Helion Technology. http://www.heliontech.com.

[7] Intel Corporation. http://www.intel.com/.

[8] Microchip Technology Inc. http://www.microchip.com.

[9] National Semiconductor Corporation. http://www.national.com.

[10] Ocean Logic Pty Ltd. http://www.ocean-logic.com.

[11] Qualcomm Inc. http://www.qualcomm.com.

[12] Rabbit Semiconductor. http://www.rabbit.com.

[13] Renesas Technology Corp. http://www.renesas.com.

[14] Synopsys, Inc. http://www.synopsys.com.

[15] Zilog, Inc. http://www.zilog.com.

[16] ARM® and Thumb®-2 instruction set quick reference card, March 2007. Document Number ARM QRC 0001L, Release L.

[17] R. Anderson and L. Biham, E.and Knudsen. Serpent: A proposal for the Advanced Encryption Standard, April 1998. http://www.cl.cam.ac.uk/~rja14/serpent.html.

[18] D. Bernstein. Cache-timing attacks on AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[19] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Advances in Cryptology, CRYPTO'90, Lecture Notes in Computer Science, vol. 537, A. J. Menezes and S. A. Vanstone ed., Springer-Verlag*, pages 2–21, 1991.

[20] Eli Biham and Rafi Chen. Near-collisions of sha-0. In Franklin [31], pages 290–305.

[21] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of sha-0 and reduced sha-1. In Cramer [27], pages 36–57.

[22] John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. Cryptology ePrint Archive, Report 2007/223, 2007. http://eprint.iacr.org/.

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

[24] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[25] J-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. *Advances in Cryptology, CRYPTO 2005, Lecture Notes in Computer Science, vol. 3621, V. Shoup ed., Springer-Verlag*, pages 430–448, 2005.

[26] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. Cryptology ePrint Archive, Report 2002/044, 2002. `http://eprint.iacr.org/`.

[27] 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.

[28] I. Damgård. A design principle for hash functions. *Advances in Cryptology, CRYPTO'89, Lecture Notes in Computer Science, vol. 218, G. Brassard ed., Springer-Verlag*, pages 416–427, 1990.

[29] Li Chao Keqin Feng Duo Lei, Da Lin2 and Longjiang Qu. The design principle of hash function with merkle-damgård construction. Cryptology ePrint Archive, Report 2006/135, 2006. `http://eprint.iacr.org/`.

[30] H. Feistel. Cryptography and computer privacy. *Scientific American*, Vol. 228, No. 5:15–23, 1973.

[31] 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.

[32] B. Gladman. SHA1, SHA2, HMAC and key derivation in C. `http://fp.gladman.plus.com/cryptography_technology/sha/index.htm`.

[33] Zheng Gong, Xuejia Lai, and Kefei Chen. A synthetic indifferentiability analysis of block cipher based hash functions. Cryptology ePrint Archive, Report 2007/465, 2007. `http://eprint.iacr.org/`.

[34] Philip Hawkes, Michael Paddon, and Gregory G. Rose. On corrective patterns for the sha-2 family. Cryptology ePrint Archive, Report 2004/207, 2004. `http://eprint.iacr.org/`.

[35] Antoine Joux. Multi-collisions in iterated hash functions. application to cascaded constructions. In Franklin [31], pages 306–316.

[36] Antoine Joux and Thomas Peyrin. Hash functions and the (amplified) boomerang attack. In Menezes [48], pages 244–263.

[37] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. Cryptology ePrint Archive, Report 2005/281, 2005. `http://eprint.iacr.org/`.

[38] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. Cryptology ePrint Archive, Report 2004/304, 2004. `http://eprint.iacr.org/`.

[39] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^{n} work. In Cramer [27], pages 474–490.

[40] Vlastimil Klima. Finding md5 collisions on a notebook pc using multi-message modifications. Cryptology ePrint Archive, Report 2005/102, 2005. `http://eprint.iacr.org/`.

[41] R. Kovacevic. Karnaugh map minimizer. `http://k-map.sourceforge.net/`.

[42] Wonil Lee, Mridul Nandi, Palash Sarkar, Donghoon Chang, Sangjin Lee, and Kouich Sakurai. A generalization of pgv-hash functions and security analysis in black-box model. Cryptology ePrint Archive, Report 2004/069, 2004. `http://eprint.iacr.org/`.

[43] Duo Lei. Revised: Block cipher based hash function construction from pgv. Cryptology ePrint Archive, Report 2005/443, 2005. `http://eprint.iacr.org/`.

[44] M. Matsui. Linear cryptanalysis method for DES cipher. *Advances in Cryptology, EUROCRYPT'93, Lecture Notes in Computer Science, vol. 765, T. Helleseth ed., Springer-Verlag*, pages 386–397, 1994.

[45] M. Matsui and A. Yamaishi. A new method for known plaintext attack of FEAL cipher. *Advances in Cryptology, EUROCRYPT'92, Lecture Notes in Computer Science, vol. 658, R. A. Rueppel ed., Springer-Verlag*, pages 81–91, 1993.

[46] S. Matyas, C. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, pages 5658–5659, 1985.

[47] A. Menezes, P. van Oorschot, and S. Vanstone.

[48] Alfred Menezes, editor. *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*. Springer, 2007.

[49] R. Merkle. Secure communication over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[50] R. Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3:43–58, 1990.

[51] J. Nechvatal, E. Barker, E. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the Advanced Encryption Standard (AES), October 2000. A report of the Computer Security Division, Information Technology Laboratory,

[52] National Institute of Standards and Technology. Data Encryption Standard (DES), October 1999. NIST FIPS PUB 46-3.

[53] National Institute of Standards and Technology. Digital Signature Standard (DSS), January 2000. NIST FIPS PUB 186-2.

[54] National Institute of Standards and Technology. Advanced Encryption Standard (AES), November 2001. NIST FIPS PUB 197.

[55] National Institute of Standards and Technology. The Keyed-Hash Message Authentication Code (HMAC), March 2002. NIST FIPS PUB 198.

[56] National Institute of Standards and Technology. Secure Hash Signature Standard (SHS), August 2002. NIST FIPS PUB 180-2.

[57] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA3) family. *Federal Register*, Vol. 72, No. 212:62212–62220, November 2007.

[58] National Institute of Standards and Technology. Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised), March 2007. NIST Special Publication 800-56A.

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

[60] B. Preneel, G. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: a synthetic approach. *Advances in Cryptology, CRYPTO'93, Lecture Notes in Computer Science, vol. 773, D. Stinson ed., Springer-Verlag*, pages 368–378, 1993.

[61] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[62] R. L. Rivest. The MD4 message digest algorithm. RFC 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[63] R. L. Rivest. The MD5 message digest algorithm. RFC 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.

[64] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[65] Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*. Springer, 2005.

[66] Kurt K. Ting, Steve C. L. Yuen, K. H. Lee, and Philip Heng Wai Leong. An fpga based sha-256 processor. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 577–585, London, UK, 2002. Springer-Verlag.

[67] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/`.

[68] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In Shoup [65], pages 17–36.

[69] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In Cramer [27], pages 19–35.

[70] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient collision search attacks on sha-0. In Shoup [65], pages 1–16.

[71] Wikipedia. Algebraic normal form. `http://en.wikipedia.org/wiki/Algebraic_normal_form`.

[72] Wikipedia. Confusion and diffusion. `http://en.wikipedia.org/wiki/Confusion_and_diffusion`.

[73] Wikipedia. Verilog. `http://en.wikipedia.org/wiki/Verilog`.

A Intermediate Values

Intermediate values in the computation of 1-block and 2-block messages for each of the CHI-224, CHI-256, CHI-384 and CHI-512 are found in the `\int_vals` directory of the optical media. The intermediate values in the computation of 1 block for CHI-224, CHI-256, CHI-384 and CHI-512 are shown here as examples.

A.1 CHI-224 Example

```
CHI-224 Example (1 block message)

Let the message, M, be the 24-bit (len = 24) ASCII string

        "abc"

which is equivalent to the following binary string:

    01100001 01100010 01100011

The message is padded by appending a "1" bit, followed by 423 "0" bits, and ending with the hex value
0000000000000018 (the 64-bit word representation of the length, 24). Thus, the final padded message
consists of 1 block (N=1).

For CHI-224, the initial hash value, H(0), is

    H0(0) = a54ff53a5f1d36f1
    H1(0) = cea7e61fc37a20d5
    H2(0) = 4a77fe7b78415dfc
    H3(0) = 8e34a6fe8e2df92a
    H4(0) = 4e5b408c9c97d4d8
    H5(0) = 24a05eee29922401

The 1st padded message block contains:

    W 0 = 6162638000000000    W 4 = 0000000000000000
    W 1 = 0000000000000000    W 5 = 0000000000000000
    W 2 = 0000000000000000    W 6 = 0000000000000000
    W 3 = 0000000000000000    W 7 = 0000000000000018

The following schedule shows the hex values for a, b, c, d, e, and f after pass i of the "t =: 0 to 19:" loop
described in Sec 7.1.2.

Since this is the last message block to be processed,  the word64s of the input hash value
are each rotated by one bit to the right before any steps occur.

                A               B               C               D               E               F

START  : d2a7fa9d2f8e9b78 e753f30fe1bd106a 253bff3dbc20aefe 471a537f4716fc95 272da0464e4bea6c 92502f7714c91200
t =  0 : b1462ac5a6a520ec d2a7fa9d2f8e9b78 e753f30fe1bd106a 070fc74fbbefa72d 471a537f4716fc95 272da0464e4bea6c
t =  1 : f1d89715bc42cb87 b1462ac5a6a520ec d2a7fa9d2f8e9b78 30570ce950b66bda 070fc74fbbefa72d 471a537f4716fc95
t =  2 : 3dc45dcb7751970b f1d89715bc42cb87 b1462ac5a6a520ec 8993a9a09e3e268e 30570ce950b66bda 070fc74fbbefa72d
t =  3 : 857e3c91222f0155 3dc45dcb7751970b f1d89715bc42cb87 46630cc9658541ff 8993a9a09e3e268e 30570ce950b66bda
t =  4 : da436f9aee09b61c 857e3c91222f0155 3dc45dcb7751970b 5444faf8eca7ab5d 46630cc9658541ff 8993a9a09e3e268e
t =  5 : 9a6993856c33afbf da436f9aee09b61c 857e3c91222f0155 071d939a8cae8181 5444faf8eca7ab5d 46630cc9658541ff
t =  6 : 43e323a26e5c88bf 9a6993856c33afbf da436f9aee09b61c 539a6b632150fa3c 071d939a8cae8181 5444faf8eca7ab5d
t =  7 : ea8a67742a248457 43e323a26e5c88bf 9a6993856c33afbf 3d1a098bda465557 539a6b632150fa3c 071d939a8cae8181
t =  8 : 370e15ae912b691f ea8a67742a248457 43e323a26e5c88bf 15d5d69e9c165d1b 3d1a098bda465557 539a6b632150fa3c
t =  9 : 07ba75db65d52a6b 370e15ae912b691f ea8a67742a248457 86a4be2390acf075 15d5d69e9c165d1b 3d1a098bda465557
t = 10 : 1e2a1450e46e8f0e 07ba75db65d52a6b 370e15ae912b691f 952299df07e3338b 86a4be2390acf075 15d5d69e9c165d1b
t = 11 : 8433a263a8011b97 1e2a1450e46e8f0e 07ba75db65d52a6b 7ff78268eadc0a38 952299df07e3338b 86a4be2390acf075
t = 12 : 0369fe7fe687ba06 8433a263a8011b97 1e2a1450e46e8f0e 144a6de53073b2db 7ff78268eadc0a38 952299df07e3338b
t = 13 : 80ecfa4bc0e86e22 0369fe7fe687ba06 8433a263a8011b97 0219efced9b73362 144a6de53073b2db 7ff78268eadc0a38
```

```
t = 14 : 4dbfdbdf272c7fcc 80ecfa4bc0e86e22 0369fe7fe687ba06 448ff6472bd0b4bf 0219efced9b73362 144a6de53073b2db
t = 15 : 226287862fb32699 4dbfdbdf272c7fcc 80ecfa4bc0e86e22 d0c8101b0091d829 448ff6472bd0b4bf 0219efced9b73362
t = 16 : 1b84fad16bfbbb5b 226287862fb32699 4dbfdbdf272c7fcc 4eaf950c58008f0b d0c8101b0091d829 448ff6472bd0b4bf
t = 17 : 2547a86e76dc2df7 1b84fad16bfbbb5b 226287862fb32699 9571963cd388f98b 4eaf950c58008f0b d0c8101b0091d829
t = 18 : c2b08a3c1ae2cc1e 2547a86e76dc2df7 1b84fad16bfbbb5b ebbd9130f4198bcf 9571963cd388f98b 4eaf950c58008f0b
t = 19 : f8ae01e422e68c90 c2b08a3c1ae2cc1e 2547a86e76dc2df7 a7468432b0b10e43 ebbd9130f4198bcf 9571963cd388f98b

That completes the processing of message block M(1).  The final hash value, H(1), is calculated to be

    H0(1) = ROTR64(H0(0), 1) ^ f8ae01e422e68c90 =  2a09fb790d6817e8
    H1(1) = ROTR64(H1(0), 1) ^ c2b08a3c1ae2cc1e =  25e37933fb5fdc74
    H2(1) = ROTR64(H2(0), 1) ^ 2547a86e76dc2df7 =  007c5753cafc8309
    H3(1) = ROTR64(H3(0), 1) ^ a7468432b0b10e43 =  e05cd74df7a7f2d6
    H4(1) = ROTR64(H4(0), 1) ^ ebbd9130f4198bcf =  cc903176ba5261a3
    H5(1) = ROTR64(H5(0), 1) ^ 9571963cd388f98b =  0721b94bc741eb8b

The resulting 224-bit message digest is

           H0               H1               H3       H4 top bits
    2a09fb790d6817e8 25e37933fb5fdc74 e05cd74df7a7f2d6 cc903176
```

A.2 CHI-256 Example

```
CHI-256 Example (1 block message)

Let the message, M, be the 24-bit (len = 24) ASCII string

        "abc"

which is equivalent to the following binary string:

    01100001 01100010 01100011

The message is padded by appending a "1" bit, followed by 423 "0" bits, and ending with the hex value
0000000000000018 (the 64-bit word representation of the length, 24). Thus, the final padded message
consists of 1 block (N=1).

For CHI-256, the initial hash value, H(0), is

    H0(0) = 510e527fade682d1
    H1(0) = de49e330e42b4cbb
    H2(0) = 29ba5a455316e0c6
    H3(0) = 5507cd18e9e51e69
    H4(0) = 4f9b11c81009a030
    H5(0) = e3d3775f155385c6

The 1st padded message block contains:

    W 0 = 6162638000000000    W 4 = 0000000000000000
    W 1 = 0000000000000000    W 5 = 0000000000000000
    W 2 = 0000000000000000    W 6 = 0000000000000000
    W 3 = 0000000000000000    W 7 = 0000000000000018

The following schedule shows the hex values for a, b, c, d, e, and f after pass i of the "t =: 0 to 19:" loop
described in Sec 7.1.2.

Since this is the last message block to be processed,  the word64s of the input hash value
are each rotated by one bit to the right before any steps occur.

                A                B                C                D                E                F

START  : a887293fd6f34168 ef24f1987215a65d 14dd2d22a98b7063 aa83e68c74f28f34 27cd88e40804d018 71e9bbaf8aa9c2e3
t =  0 : bb0d373a11848366 a887293fd6f34168 ef24f1987215a65d 8651009fa8286bcb aa83e68c74f28f34 27cd88e40804d018
```

```
t =  1 : feaa00fc8f569b0c bb0d373a11848366 a887293fd6f34168 8ced6e41c5b95a50 8651009fa8286bcb aa83e68c74f28f34
t =  2 : 08e15bf0cb7da415 feaa00fc8f569b0c bb0d373a11848366 2d82af261d959396 8ced6e41c5b95a50 8651009fa8286bcb
t =  3 : 104937257e29ca69 08e15bf0cb7da415 feaa00fc8f569b0c fe205785639131af 2d82af261d959396 8ced6e41c5b95a50
t =  4 : d53f14337fd014e8 104937257e29ca69 08e15bf0cb7da415 b4bbb58eefbdec49 fe205785639131af 2d82af261d959396
t =  5 : fe6e888630734223 d53f14337fd014e8 104937257e29ca69 ef3704a53570d85e b4bbb58eefbdec49 fe205785639131af
t =  6 : 666d4f732e11831e fe6e888630734223 d53f14337fd014e8 2a86d4d76c1abb6f ef3704a53570d85e b4bbb58eefbdec49
t =  7 : 30184929677cd1b7 666d4f732e11831e fe6e888630734223 a48d3794d7ade264 2a86d4d76c1abb6f ef3704a53570d85e
t =  8 : 6b5e2ae3e57a1e57 30184929677cd1b7 666d4f732e11831e 7694f2c9c6bc79c5 a48d3794d7ade264 2a86d4d76c1abb6f
t =  9 : 56c0a051ac8733b1 6b5e2ae3e57a1e57 30184929677cd1b7 24ba5443d9712a68 7694f2c9c6bc79c5 a48d3794d7ade264
t = 10 : d362a96ece9d1e7c 56c0a051ac8733b1 6b5e2ae3e57a1e57 d05d91d24878d5ba 24ba5443d9712a68 7694f2c9c6bc79c5
t = 11 : f214373700a7e6a3 d362a96ece9d1e7c 56c0a051ac8733b1 656aa3c859022741 d05d91d24878d5ba 24ba5443d9712a68
t = 12 : 5d72f6eb2005999e f214373700a7e6a3 d362a96ece9d1e7c 64be07acef16ab8a 656aa3c859022741 d05d91d24878d5ba
t = 13 : 7266a20ab8faab44 5d72f6eb2005999e f214373700a7e6a3 e483fbe2720113b6 64be07acef16ab8a 656aa3c859022741
t = 14 : ac4a0828c99e4b6b 7266a20ab8faab44 5d72f6eb2005999e dfd32d6153eb335f e483fbe2720113b6 64be07acef16ab8a
t = 15 : 7ff682d2c1730268 ac4a0828c99e4b6b 7266a20ab8faab44 fb4af7c01fe4154c dfd32d6153eb335f e483fbe2720113b6
t = 16 : de73b07d6db5ab1f 7ff682d2c1730268 ac4a0828c99e4b6b 2eb17618874a32fd fb4af7c01fe4154c dfd32d6153eb335f
t = 17 : b4623267a20a6511 de73b07d6db5ab1f 7ff682d2c1730268 c7692c8f0097ca1b 2eb17618874a32fd fb4af7c01fe4154c
t = 18 : af2771c396776bd4 b4623267a20a6511 de73b07d6db5ab1f 2eab9249751d3d2f c7692c8f0097ca1b 2eb17618874a32fd
t = 19 : 6786f372a05b7da4 af2771c396776bd4 b4623267a20a6511 24a7d93f1e289e36 2eab9249751d3d2f c7692c8f0097ca1b
```

That completes the processing of message block M(1). The final hash value, H(1), is calculated to be

```
    H0(1) = ROTR64(H0(0), 1) ^ 6786f372a05b7da4 =  cf01da4d76a83ccc
    H1(1) = ROTR64(H1(0), 1) ^ af2771c396776bd4 =  4003805be462cd89
    H2(1) = ROTR64(H2(0), 1) ^ b4623267a20a6511 =  a0bf1f450b811572
    H3(1) = ROTR64(H3(0), 1) ^ 24a7d93f1e289e36 =  8e243fb36ada1102
    H4(1) = ROTR64(H4(0), 1) ^ 2eab9249751d3d2f =  09661aad7d19ed37
    H5(1) = ROTR64(H5(0), 1) ^ c7692c8f0097ca1b =  b68097208a3e08f8
```

The resulting 256-bit message digest is

```
           H0               H1               H3               H4
    cf01da4d76a83ccc 4003805be462cd89 8e243fb36ada1102 09661aad7d19ed37
```

A.3 CHI-384 Example

CHI-384 Example (1 block message)

Let the message, M, be the 24-bit (len = 24) ASCII string

```
        "abc"
```

which is equivalent to the following binary string:

```
    01100001 01100010 01100011
```

The message is padded by appending a "1" bit, followed by 871 "0" bits, and ending with the hex value 0000000000000000 0000000000000018 (the 128-bit word representation of the length, 24). Thus, the final padded message consists of 1 block (N=1).

For CHI-384, the initial hash value, H(0), is

```
    H0(0) = a54ff53a5f1d36f1
    H1(0) = cea7e61fc37a20d5
    H2(0) = 4a77fe7b78415dfc
    H3(0) = 8e34a6fe8e2df92a
    H4(0) = 4e5b408c9c97d4d8
    H5(0) = 24a05eee29922401
    H6(0) = 5a8176cffc7c2224
    H7(0) = c3edebda29bec4c8
    H8(0) = 8a074c0f4d999610
```

The 1st padded message block contains:

```
    W 0 = 6162638000000000    W 8 = 0000000000000000
    W 1 = 0000000000000000    W 9 = 0000000000000000
    W 2 = 0000000000000000    W10 = 0000000000000000
    W 3 = 0000000000000000    W11 = 0000000000000000
    W 4 = 0000000000000000    W12 = 0000000000000000
    W 5 = 0000000000000000    W13 = 0000000000000000
    W 6 = 0000000000000000    W14 = 0000000000000000
    W 7 = 0000000000000000    W15 = 0000000000000018
```

The following schedule shows the hex values for a, b, c, d, e, f, g, p, and q
after pass i of the "t =: 0 to 39:" loop described in Sec 7.3.2.
The internal state is too large to beshown on a single line, so the values are presented in
two rows for each step. The first row shows the values a, b, c, d, and e. The second row
shows the values of f, g, p and q.

Since this is the last message block to be processed, the word64s of the input hash value
are each rotated by one bit to the right before any steps occur.

```
                A               B               C               D               E
                                F               G               P               Q

START  : d2a7fa9d2f8e9b78 e753f30fe1bd106a 253bff3dbc20aefe 471a537f4716fc95 272da0464e4bea6c
                          92502f7714c91200 2d40bb67fe3e1112 61f6f5ed14df6264 4503a607a6cccb08
t =  0 : 4014060fd19d720e d2a7fa9d2f8e9b78 e753f30fe1bd106a 6f6e69f026d7c59c 471a537f4716fc95
                          272da0464e4bea6c fd5167d621226652 2d40bb67fe3e1112 61f6f5ed14df6264
t =  1 : 91bcd765aaeed092 4014060fd19d720e d2a7fa9d2f8e9b78 f94b5a9c326104c1 6f6e69f026d7c59c
                          471a537f4716fc95 789311749f2aacea fd5167d621226652 2d40bb67fe3e1112
t =  2 : ce5ae817e3b02407 91bcd765aaeed092 4014060fd19d720e 1895dc186e7b90b3 f94b5a9c326104c1
                          6f6e69f026d7c59c 71395194db65a3c7 789311749f2aacea fd5167d621226652
t =  3 : c17cc7563e218efd ce5ae817e3b02407 91bcd765aaeed092 e4697d0f693d3f76 1895dc186e7b90b3
                          f94b5a9c326104c1 31519bc710207ecb 71395194db65a3c7 789311749f2aacea
t =  4 : 2446995276a62b54 c17cc7563e218efd ce5ae817e3b02407 52a0cc885501fa83 e4697d0f693d3f76
                          1895dc186e7b90b3 65f243dc24bb25f6 31519bc710207ecb 71395194db65a3c7
t =  5 : d6be14464ed22f1f 2446995276a62b54 c17cc7563e218efd d2c96a903be63499 52a0cc885501fa83
                          e4697d0f693d3f76 9efb8cca2e9ff543 65f243dc24bb25f6 31519bc710207ecb
t =  6 : c5979ed4d90ff576 d6be14464ed22f1f 2446995276a62b54 982ed90b16a1601d d2c96a903be63499
                          52a0cc885501fa83 a17874788a0540b8 9efb8cca2e9ff543 65f243dc24bb25f6
t =  7 : 477b1273b883d6bb c5979ed4d90ff576 d6be14464ed22f1f 79e85039bd4cce1e 982ed90b16a1601d
                          d2c96a903be63499 4567617b657f6100 a17874788a0540b8 9efb8cca2e9ff543
t =  8 : 5665a1d96cb5ac70 477b1273b883d6bb c5979ed4d90ff576 0a2861a02b1f24ca 79e85039bd4cce1e
                          982ed90b16a1601d a91d8816cb792496 4567617b657f6100 a17874788a0540b8
t =  9 : c3382b6e52f64737 5665a1d96cb5ac70 477b1273b883d6bb 332505d4801ba3b2 0a2861a02b1f24ca
                          79e85039bd4cce1e 16bc792952d36209 a91d8816cb792496 4567617b657f6100
t = 10 : 5c02f62359b1d603 c3382b6e52f64737 5665a1d96cb5ac70 808a7d2cd849d216 332505d4801ba3b2
                          0a2861a02b1f24ca 0c4ebfa872092ef1 16bc792952d36209 a91d8816cb792496
t = 11 : 5557230798d8cdf9 5c02f62359b1d603 c3382b6e52f64737 891f4e2ea06483b4 808a7d2cd849d216
                          332505d4801ba3b2 e35c26384e2c53c1 0c4ebfa872092ef1 16bc792952d36209
t = 12 : 35265c6f3f48924f 5557230798d8cdf9 5c02f62359b1d603 91b6ac275510a8cc 891f4e2ea06483b4
                          808a7d2cd849d216 67257f292b9b2ff7 e35c26384e2c53c1 0c4ebfa872092ef1
t = 13 : c7f7b13f3407166c 35265c6f3f48924f 5557230798d8cdf9 ee6865fff01c71f3 91b6ac275510a8cc
                          891f4e2ea06483b4 2e8cbfd1327980f5 67257f292b9b2ff7 e35c26384e2c53c1
t = 14 : e5fb8823bcdd5ace c7f7b13f3407166c 35265c6f3f48924f 5ce437b9e93f7a1a ee6865fff01c71f3
                          91b6ac275510a8cc 1bd0a116b874775d 2e8cbfd1327980f5 67257f292b9b2ff7
t = 15 : 8f76e3c789bd1d78 e5fb8823bcdd5ace c7f7b13f3407166c 9cc056e4e614a14b 5ce437b9e93f7a1a
                          ee6865fff01c71f3 4d93a585a70d0f1d 1bd0a116b874775d 2e8cbfd1327980f5
t = 16 : e473d1f4e4876f2a 8f76e3c789bd1d78 e5fb8823bcdd5ace bd33b6e4e4e353f8 9cc056e4e614a14b
                          5ce437b9e93f7a1a 95da5d64e2262fb3 4d93a585a70d0f1d 1bd0a116b874775d
t = 17 : 1aed4226cdc4fd53 e473d1f4e4876f2a 8f76e3c789bd1d78 856f023f93248107 bd33b6e4e4e353f8
                          9cc056e4e614a14b 28ce4ae90dbddb0f 95da5d64e2262fb3 4d93a585a70d0f1d
t = 18 : ad655293700d9cca 1aed4226cdc4fd53 e473d1f4e4876f2a 489ecfbe6a0d40a8 856f023f93248107
                          bd33b6e4e4e353f8 65e382f3f817ff2a 28ce4ae90dbddb0f 95da5d64e2262fb3
t = 19 : a75fae753d9da815 ad655293700d9cca 1aed4226cdc4fd53 781c0792aed7f5ec 489ecfbe6a0d40a8
                          856f023f93248107 55f0d21e1641dd8a 65e382f3f817ff2a 28ce4ae90dbddb0f
t = 20 : 2805d71ee4d65474 a75fae753d9da815 ad655293700d9cca 503bcefd664da866 781c0792aed7f5ec
```

```
                        489ecfbe6a0d40a8 e70017212bd1a2c4 55f0d21e1641dd8a 65e382f3f817ff2a
t = 21 : 374caf13a2d2d3fa 2805d71ee4d65474 a75fae753d9da815 ae0c4bf7cb2c479b 503bcefd664da866
                        781c0792aed7f5ec 756af90347aff8b9 e70017212bd1a2c4 55f0d21e1641dd8a
t = 22 : 105a4775605f3c0b 374caf13a2d2d3fa 2805d71ee4d65474 16fea1a3b9374089 ae0c4bf7cb2c479b
                        503bcefd664da866 97e356e0e9623378 756af90347aff8b9 e70017212bd1a2c4
t = 23 : 28c258e406c9ecb5 105a4775605f3c0b 374caf13a2d2d3fa 3f7e2f1538bbe3b7 16fea1a3b9374089
                        ae0c4bf7cb2c479b 49c47d7333f77746 97e356e0e9623378 756af90347aff8b9
t = 24 : 2b03ad7de9117457 28c258e406c9ecb5 105a4775605f3c0b e21ff9a2d4df064f 3f7e2f1538bbe3b7
                        16fea1a3b9374089 f7f18d62b8ea7f41 49c47d7333f77746 97e356e0e9623378
t = 25 : fbbc9ef2e6be2d43 2b03ad7de9117457 28c258e406c9ecb5 b77adbfea0628c28 e21ff9a2d4df064f
                        3f7e2f1538bbe3b7 3cf8a64cbd7410f1 f7f18d62b8ea7f41 49c47d7333f77746
t = 26 : 222825e328361b56 fbbc9ef2e6be2d43 2b03ad7de9117457 b978cf4b6e5cb22a b77adbfea0628c28
                        e21ff9a2d4df064f 687f040d98bc1e7e 3cf8a64cbd7410f1 f7f18d62b8ea7f41
t = 27 : 08243a9e818e70a5 222825e328361b56 fbbc9ef2e6be2d43 ba9ea26f2f549638 b978cf4b6e5cb22a
                        b77adbfea0628c28 53e75329eb3a3a18 687f040d98bc1e7e 3cf8a64cbd7410f1
t = 28 : 28ac9417158774fa 08243a9e818e70a5 222825e328361b56 7bab0fa700bf5319 ba9ea26f2f549638
                        b978cf4b6e5cb22a e00a8b3b2ad59bc2 53e75329eb3a3a18 687f040d98bc1e7e
t = 29 : 3f56f9f44e3c98b6 28ac9417158774fa 08243a9e818e70a5 763ff1b2f44e7db1 7bab0fa700bf5319
                        ba9ea26f2f549638 cc7f830d5643fc95 e00a8b3b2ad59bc2 53e75329eb3a3a18
t = 30 : 7f6b41a12eeb9812 3f56f9f44e3c98b6 28ac9417158774fa d3a094186a5454ce 763ff1b2f44e7db1
                        7bab0fa700bf5319 f9b8b536096f26e9 cc7f830d5643fc95 e00a8b3b2ad59bc2
t = 31 : 7f2953d3310ad69a 7f6b41a12eeb9812 3f56f9f44e3c98b6 00655341bc488930 d3a094186a5454ce
                        763ff1b2f44e7db1 6d851e91af3b9851 f9b8b536096f26e9 cc7f830d5643fc95
t = 32 : 6e9e762c0ee45979 7f2953d3310ad69a 7f6b41a12eeb9812 2d0e5cfd5032dd7a 00655341bc488930
                        d3a094186a5454ce 9e5da7cbcd02042a 6d851e91af3b9851 f9b8b536096f26e9
t = 33 : 82ca8e1c0248acfe 6e9e762c0ee45979 7f2953d3310ad69a b43d1e1f4aa45302 2d0e5cfd5032dd7a
                        00655341bc488930 ecbe6da53f9161a9 9e5da7cbcd02042a 6d851e91af3b9851
t = 34 : bc14c1b22e08b7b5 82ca8e1c0248acfe 6e9e762c0ee45979 2da22e8e1f2ce0b9 b43d1e1f4aa45302
                        2d0e5cfd5032dd7a 0af1f222fae40cc6 ecbe6da53f9161a9 9e5da7cbcd02042a
t = 35 : 722634879f124300 bc14c1b22e08b7b5 82ca8e1c0248acfe ec90faf377c913e0 2da22e8e1f2ce0b9
                        b43d1e1f4aa45302 d415461851996d77 0af1f222fae40cc6 ecbe6da53f9161a9
t = 36 : 5fefce32ed0e7ac6 722634879f124300 bc14c1b22e08b7b5 09515450707e2ee5 ec90faf377c913e0
                        2da22e8e1f2ce0b9 5fae1e558bbd8eb7 d415461851996d77 0af1f222fae40cc6
t = 37 : 66fd219e5afbec61 5fefce32ed0e7ac6 722634879f124300 bef31c0c674e3590 09515450707e2ee5
                        ec90faf377c913e0 814423a339f1c365 5fae1e558bbd8eb7 d415461851996d77
t = 38 : ca1683c0172213a9 66fd219e5afbec61 5fefce32ed0e7ac6 325237662dad107a bef31c0c674e3590
                        09515450707e2ee5 55c9ab95502c38fa 814423a339f1c365 5fae1e558bbd8eb7
t = 39 : 40ec5bcdfa726dfe ca1683c0172213a9 66fd219e5afbec61 5f9900345ec693f5 325237662dad107a
                        bef31c0c674e3590 7f1fdc0091f86f64 55c9ab95502c38fa 814423a339f1c365

That completes the processing of message block M(1).  The final hash value, H(1), is calculated to be

    H0(1) = ROTR64(H0(0), 1) ^ 40ec5bcdfa726dfe =  924ba150d5fcf686
    H1(1) = ROTR64(H1(0), 1) ^ ca1683c0172213a9 =  2d4570cff69f03c3
    H2(1) = ROTR64(H2(0), 1) ^ 66fd219e5afbec61 =  43c6dea3e6db429f
    H3(1) = ROTR64(H3(0), 1) ^ 5f9900345ec693f5 =  1883534b19d06f60
    H4(1) = ROTR64(H4(0), 1) ^ 325237662dad107a =  157f972063e6fa16
    H5(1) = ROTR64(H5(0), 1) ^ bef31c0c674e3590 =  2ca3337b73872790
    H6(1) = ROTR64(H6(0), 1) ^ 7f1fdc0091f86f64 =  525f67676fc67e76
    H7(1) = ROTR64(H7(0), 1) ^ 55c9ab95502c38fa =  343f5e7844f35a9e
    H8(1) = ROTR64(H8(0), 1) ^ 814423a339f1c365 =  c44785a49f3d086d

The resulting 384-bit message digest is

           H0               H1               H2               H3               H4               H5
    924ba150d5fcf686 2d4570cff69f03c3 43c6dea3e6db429f 1883534b19d06f60 157f972063e6fa16 2ca3337b73872790
```

A.4 CHI-512 Example

```
CHI-512 Example (1 block message)

Let the message, M, be the 24-bit (len = 24) ASCII string
```

"abc"

which is equivalent to the following binary string:

    01100001 01100010 01100011

The message is padded by appending a "1" bit, followed by 871 "0" bits, and ending with the hex value 0000000000000000 0000000000000018 (the 128-bit word representation of the length, 24). Thus, the final padded message consists of 1 block (N=1).

For CHI-512, the initial hash value, H(0), is

    H0(0) = 510e527fade682d1
    H1(0) = de49e330e42b4cbb
    H2(0) = 29ba5a455316e0c6
    H3(0) = 5507cd18e9e51e69
    H4(0) = 4f9b11c81009a030
    H5(0) = e3d3775f155385c6
    H6(0) = 489221632788fb30
    H7(0) = 41921db8feeb38c2
    H8(0) = 9af94a7c48bbd5b6

The 1st padded message block contains:

    W 0 = 6162638000000000   W 8 = 0000000000000000
    W 1 = 0000000000000000   W 9 = 0000000000000000
    W 2 = 0000000000000000   W10 = 0000000000000000
    W 3 = 0000000000000000   W11 = 0000000000000000
    W 4 = 0000000000000000   W12 = 0000000000000000
    W 5 = 0000000000000000   W13 = 0000000000000000
    W 6 = 0000000000000000   W14 = 0000000000000000
    W 7 = 0000000000000000   W15 = 0000000000000018

The following schedule shows the hex values for a, b, c, d, e, f, g, p, and q
after pass i of the "t =: 0 to 39:" loop described in Sec 7.3.2.
The internal state is too large to beshown on a single line, so the values are presented in
two rows for each step. The first row shows the values a, b, c, d, and e. The second row
shows the values of f, g, p and q.

Since this is the last message block to be processed, the word64s of the input hash value
are each rotated by one bit to the right before any steps occur.

```
                A               B               C               D               E
                                F               G               P               Q

START  : a887293fd6f34168 ef24f1987215a65d 14dd2d22a98b7063 aa83e68c74f28f34 27cd88e40804d018
                          71e9bbaf8aa9c2e3 244910b193c47d98 20c90edc7f759c61 4d7ca53e245deadb
t =  0 : 4e3119942b701882 a887293fd6f34168 ef24f1987215a65d d5d2adc6edaa12da aa83e68c74f28f34
                          27cd88e40804d018 cccba063126429c0 244910b193c47d98 20c90edc7f759c61
t =  1 : 8fe3b7c0a8cd070c 4e3119942b701882 a887293fd6f34168 9cdc75fbd0122c5e d5d2adc6edaa12da
                          aa83e68c74f28f34 2b68c56452004669 cccba063126429c0 244910b193c47d98
t =  2 : e5cd2711364d6943 8fe3b7c0a8cd070c 4e3119942b701882 446da76e07796a38 9cdc75fbd0122c5e
                          d5d2adc6edaa12da 5ad3e2b504d66110 2b68c56452004669 cccba063126429c0
t =  3 : b6dc68e006341319 e5cd2711364d6943 8fe3b7c0a8cd070c 9403daac018f34be 446da76e07796a38
                          9cdc75fbd0122c5e ec39016ab25af441 5ad3e2b504d66110 2b68c56452004669
t =  4 : 960675ea34a3cdf3 b6dc68e006341319 e5cd2711364d6943 05034dd097387b1d 9403daac018f34be
                          446da76e07796a38 acc61317d9b11663 ec39016ab25af441 5ad3e2b504d66110
t =  5 : 7d56114d73af2626 960675ea34a3cdf3 b6dc68e006341319 e2f395174893a6c2 05034dd097387b1d
                          9403daac018f34be 70f5c8cb706b52f9 acc61317d9b11663 ec39016ab25af441
t =  6 : 20949622b63458c6 7d56114d73af2626 960675ea34a3cdf3 7130afd12cc6dcce e2f395174893a6c2
                          05034dd097387b1d 446de5d5ab16516d 70f5c8cb706b52f9 acc61317d9b11663
t =  7 : 5d033274a18061eb 20949622b63458c6 7d56114d73af2626 4c0ece415593171d 7130afd12cc6dcce
                          e2f395174893a6c2 e8e94c9f30dbe0a2 446de5d5ab16516d 70f5c8cb706b52f9
t =  8 : f574429c422292ae 5d033274a18061eb 20949622b63458c6 6dffc8ee56ab40b3 4c0ece415593171d
                          7130afd12cc6dcce cd69599fa9b3a358 e8e94c9f30dbe0a2 446de5d5ab16516d
```

```
t =  9 : af9ab105c6b595e2 f574429c422292ae 5d033274a18061eb e76066bb10ea1c9d 6dffc8ee56ab40b3
                          4c0ece415593171d 42fb5d4546e1e5bf cd69599fa9b3a358 e8e94c9f30dbe0a2
t = 10 : 69ad26693627e142 af9ab105c6b595e2 f574429c422292ae 99ef6c2fe3c9bcc4 e76066bb10ea1c9d
                          6dffc8ee56ab40b3 07b751ecd30a505f 42fb5d4546e1e5bf cd69599fa9b3a358
t = 11 : a6eedabd00ed159b 69ad26693627e142 af9ab105c6b595e2 bd808d5022ee9edc 99ef6c2fe3c9bcc4
                          e76066bb10ea1c9d 8d2fe239640b344b 07b751ecd30a505f 42fb5d4546e1e5bf
t = 12 : c0c85ea83ccf43bd a6eedabd00ed159b 69ad26693627e142 6799e9dd41bc6af1 bd808d5022ee9edc
                          99ef6c2fe3c9bcc4 3c85b0dd523c671b 8d2fe239640b344b 07b751ecd30a505f
t = 13 : 7dba1dbf6d52e8f6 c0c85ea83ccf43bd a6eedabd00ed159b df98306455712172 6799e9dd41bc6af1
                          bd808d5022ee9edc 83520ee8366ab4d9 3c85b0dd523c671b 8d2fe239640b344b
t = 14 : 716797b4d942c149 7dba1dbf6d52e8f6 c0c85ea83ccf43bd 49845b2ca4eda116 df98306455712172
                          6799e9dd41bc6af1 b66b972dc9229e6a 83520ee8366ab4d9 3c85b0dd523c671b
t = 15 : ee60c8a630fed371 716797b4d942c149 7dba1dbf6d52e8f6 1707f2c4d4686ced 49845b2ca4eda116
                          df98306455712172 02450f8e5dd5d40f b66b972dc9229e6a 83520ee8366ab4d9
t = 16 : 25d0befb33d33b6b ee60c8a630fed371 716797b4d942c149 dc763adf5cac0dde 1707f2c4d4686ced
                          49845b2ca4eda116 661ee6faf12afeba 02450f8e5dd5d40f b66b972dc9229e6a
t = 17 : 04910072aa06b917 25d0befb33d33b6b ee60c8a630fed371 707385b617bdc143 dc763adf5cac0dde
                          1707f2c4d4686ced 71ad0792e80774d3 661ee6faf12afeba 02450f8e5dd5d40f
t = 18 : c9db9b7660516cd4 04910072aa06b917 25d0befb33d33b6b d11699fd3da620a9 707385b617bdc143
                          dc763adf5cac0dde 3e1f35e40943ba5e 71ad0792e80774d3 661ee6faf12afeba
t = 19 : 76f308bda99612fa c9db9b7660516cd4 04910072aa06b917 aadbfd04abce0b3f d11699fd3da620a9
                          707385b617bdc143 1a51c33d7c37e95f 3e1f35e40943ba5e 71ad0792e80774d3
t = 20 : 61fd86deb9942504 76f308bda99612fa c9db9b7660516cd4 6c35a6fb780131ad aadbfd04abce0b3f
                          d11699fd3da620a9 2d0742a906b6a308 1a51c33d7c37e95f 3e1f35e40943ba5e
t = 21 : 873efd6a4dbd9fb2 61fd86deb9942504 76f308bda99612fa 36c9159d4cc11ebb 6c35a6fb780131ad
                          aadbfd04abce0b3f 588a3792f0865459 2d0742a906b6a308 1a51c33d7c37e95f
t = 22 : e61e329aa3fc7689 873efd6a4dbd9fb2 61fd86deb9942504 5c5ac981892ee016 36c9159d4cc11ebb
                          6c35a6fb780131ad a62c89afccf4abcf 588a3792f0865459 2d0742a906b6a308
t = 23 : 286e4025e1b09b30 e61e329aa3fc7689 873efd6a4dbd9fb2 9a72e4391b92a2bc 5c5ac981892ee016
                          36c9159d4cc11ebb 5d703e58aa53ed54 a62c89afccf4abcf 588a3792f0865459
t = 24 : aa4e8fcab62de884 286e4025e1b09b30 e61e329aa3fc7689 3f7166c8dd370af6 9a72e4391b92a2bc
                          5c5ac981892ee016 ddf7f14aea216314 5d703e58aa53ed54 a62c89afccf4abcf
t = 25 : 65abfcec5a97aa54 aa4e8fcab62de884 286e4025e1b09b30 9e9f3ee134b78233 3f7166c8dd370af6
                          9a72e4391b92a2bc c5030db3b0dfa619 ddf7f14aea216314 5d703e58aa53ed54
t = 26 : c873b1a5d637fb00 65abfcec5a97aa54 aa4e8fcab62de884 b0c9d9c71e65bf47 9e9f3ee134b78233
                          3f7166c8dd370af6 6ddf3b9072d5c78c c5030db3b0dfa619 ddf7f14aea216314
t = 27 : c4ce09e09bb34721 c873b1a5d637fb00 65abfcec5a97aa54 b1f4ea273f58959e b0c9d9c71e65bf47
                          9e9f3ee134b78233 4c920fb6c87359b5 6ddf3b9072d5c78c c5030db3b0dfa619
t = 28 : 59118036810176ff c4ce09e09bb34721 c873b1a5d637fb00 94351ef1c41aa319 b1f4ea273f58959e
                          b0c9d9c71e65bf47 53f218bb81b573da 4c920fb6c87359b5 6ddf3b9072d5c78c
t = 29 : c2c72778eae8c7d4 59118036810176ff c4ce09e09bb34721 10453c8e0e8abeb3 94351ef1c41aa319
                          b1f4ea273f58959e ada8b203025fea9d 53f218bb81b573da 4c920fb6c87359b5
t = 30 : 9f852fab50e619f8 c2c72778eae8c7d4 59118036810176ff c09bb5de52ae203a 10453c8e0e8abeb3
                          94351ef1c41aa319 5f8d1ab4e73f5a9f ada8b203025fea9d 53f218bb81b573da
t = 31 : f6c2829f3fb77e3d 9f852fab50e619f8 c2c72778eae8c7d4 ec9bcd3f2e0bc4b3 c09bb5de52ae203a
                          10453c8e0e8abeb3 8f4acd2c0c10c886 5f8d1ab4e73f5a9f ada8b203025fea9d
t = 32 : 851f109c9c93c9eb f6c2829f3fb77e3d 9f852fab50e619f8 3466d52b9cad540b ec9bcd3f2e0bc4b3
                          c09bb5de52ae203a 328234b463bc4095 8f4acd2c0c10c886 5f8d1ab4e73f5a9f
t = 33 : f89bb88ffe0fa272 851f109c9c93c9eb f6c2829f3fb77e3d f1528ea1af58bd73 3466d52b9cad540b
                          ec9bcd3f2e0bc4b3 0ff93ebc55cd3dd5 328234b463bc4095 8f4acd2c0c10c886
t = 34 : 51e8321180615823 f89bb88ffe0fa272 851f109c9c93c9eb 4b6e7d2db7c53914 f1528ea1af58bd73
                          3466d52b9cad540b 0109731fe05c8089 0ff93ebc55cd3dd5 328234b463bc4095
t = 35 : 1032f83f94c45ab1 51e8321180615823 f89bb88ffe0fa272 2a00cf8ad2978a30 4b6e7d2db7c53914
                          f1528ea1af58bd73 27592e72c8d2322e 0109731fe05c8089 0ff93ebc55cd3dd5
t = 36 : 67a95618cdd4f971 1032f83f94c45ab1 51e8321180615823 15d37bc0cc5dea2d 2a00cf8ad2978a30
                          4b6e7d2db7c53914 34cb049000cccfaa 27592e72c8d2322e 0109731fe05c8089
t = 37 : fefa1c8e4c5293b1 67a95618cdd4f971 1032f83f94c45ab1 cfd7d2188dd3a661 15d37bc0cc5dea2d
                          2a00cf8ad2978a30 8fa1cec3d1735ac1 34cb049000cccfaa 27592e72c8d2322e
t = 38 : bfb986afc8374663 fefa1c8e4c5293b1 67a95618cdd4f971 5a86387a95444246 cfd7d2188dd3a661
                          15d37bc0cc5dea2d 1a456047144f6f82 8fa1cec3d1735ac1 34cb049000cccfaa
t = 39 : a2f6eed854c7a740 bfb986afc8374663 fefa1c8e4c5293b1 b3e7fba72a8373a1 5a86387a95444246
                          cfd7d2188dd3a661 75c1769b3f09f815 1a456047144f6f82 8fa1cec3d1735ac1
```

That completes the processing of message block M(1). The final hash value, H(1), is calculated to be

```
    H0(1) = ROTR64(H0(0), 1) ^ a2f6eed854c7a740 =  0a71c7e78234e628
    H1(1) = ROTR64(H1(0), 1) ^ bfb986afc8374663 =  509d7737ba22e03e
    H2(1) = ROTR64(H2(0), 1) ^ fefa1c8e4c5293b1 =  ea2731ace5d9e3d2
    H3(1) = ROTR64(H3(0), 1) ^ b3e7fba72a8373a1 =  19641d2b5e71fc95
    H4(1) = ROTR64(H4(0), 1) ^ 5a86387a95444246 =  7d4bb09e9d40925e
    H5(1) = ROTR64(H5(0), 1) ^ cfd7d2188dd3a661 =  be3e69b7077a6482
    H6(1) = ROTR64(H6(0), 1) ^ 75c1769b3f09f815 =  5188662aaccd858d
    H7(1) = ROTR64(H7(0), 1) ^ 1a456047144f6f82 =  3a8c6e9b6b3af3e3
    H8(1) = ROTR64(H8(0), 1) ^ 8fa1cec3d1735ac1 =  c2dd6bfdf52eb01a

The resulting 512-bit message digest is

           H0               H1               H2               H3
    0a71c7e78234e628 509d7737ba22e03e ea2731ace5d9e3d2 19641d2b5e71fc95
           H4               H5               H6               H7
    7d4bb09e9d40925e be3e69b7077a6482 5188662aaccd858d 3a8c6e9b6b3af3e3
```

B Change History

A history of changes in the public versions of this document are shown in Table 38.

Date	Description	Changes	Affected Sections
2008-10-29	Original Submission	-	-
2009-01-12	Minor Corrections	Corrected CHI-224 Example intermediate values in Appendix A, and added intermediate values for CHI-256, CHI-384 and CHI-512.	Section 1.1, Appendix A
		Intermediate values are in `\int_vals` directory not `\KAT_MCT` directory as previously stated	Appendix A
		Added Change History as an Appendix	Section 1.1, Appendix B

Table 38: Change History for this Document