Caligo, an Extensible Block Cipher
-and-
CHash, a Caligo Based Hash

Alexis W. Machado
email: alexis.machado(@task.com.br @telemigcelular.com.br)
Mar/09/2006

Abstract: The *Caligo* operations are performed on whole blocks only. No subdivision passes through an s-box or a Feistel network. The cipher definition is the same for any block size, allowing exhaustive search for statistical deviations on small block variants. I also propose *CHash*, a hash function that takes advantage of the cipher extensibility and resists the *extension attack*.

Keywords: Caligo, extensible, block cipher, symmetric key, CHash, extension attack.

1) Introduction

Currently, there is no provably secure block cipher. After withstanding many attempts of cryptanalysis, the algorithm is assumed to be secure. With a single definition for any block size, *Caligo* permits full inspection of small block variants, simplifying the search for defects and associated cryptanalytic attacks.

In sections 2 and 3, the algorithm is described. In section 4, the definition of *CHash* is given. In sections 5 and 6, the results of seeking for *frequently differentials, impossible differentials* and *linear correlations* are presented. In section 7, competitive performance on two 64-bit processors is shown.

2) Encryption and Decryption

Given a natural number n, each element of the set $\Omega = \{0, 1, 2, .. , 2^n - 1\}$ have a distinct binary representation in the set $\Psi = \{0, 1\}^n$. The term *block* identifies an integer from Ω or the correspondent bit-sequence of Ψ (n is called the *block size*). Therefore, we can do algebraic or bitwise operations on a block by considering it an element of Ω or Ψ respectively.

Been r the *number of rounds* of the cipher, the encryption and decryption *round functions* f_i, g_i: $\Omega^{3r} \times \Omega \to \Omega$ are defined, respectively, by

$$f_i(K, X) = (K[3i] \times (X \oplus K[3i+1]))'' + K[3i+2] \quad (\text{mod } 2^n) \quad (0 \le i < r) \quad (2.1)$$
$$g_i(K, X) = (K[3i]^{-1} \times (X - K[3i+2])'') \oplus K[3i+1] \quad (\text{mod } 2^n) \quad (0 \le i < r) \quad (2.2)$$

where

a) The *subkey* vector $K = \langle K[0], K[1], .. K[3r-1] \rangle$ have 3r odd-blocks derived from a secret block called *masterkey* (see section 3). $K[3i]^{-1}$ is the multiplicative inverse (mod 2^n) of $K[3i]$ (these elements should be calculated during the subkey setup and stored together with K).

b) X is an input block.

c) $''$ is the bit-reversal of a block. Example for $n=8$: $(01001101)'' = 10110010$.

d) $\oplus$ is the bitwise *xor* of two blocks.

e) $\times$, $+$ and $-$ are the integer *multiplication*, *addition* and *subtraction*, reduced mod 2^n.

A composition (in the second parameter) of the round functions is performed to construct the encryption and decryption functions :

$$f(r, K, X) = f_{r-1} \circ f_{r-2} \circ \ldots \circ f_1 \circ f_0 (K, X) \qquad (2.3)$$
$$g(r, K, X) = g_0 \circ g_1 \circ \ldots \circ g_{r-2} \circ g_{r-1} (K, X) \qquad (2.4)$$

The above **n**-bit block round functions (2.1 and 2.2) can embed the encryption/decryption of smaller blocks with **n−m** bit length ($0 \le m < n$). The input/output $n-m$ least significant bits represent the smaller block been processed. The most significant $n-m$ bits must be shifted to the least significant $n-m$ positions **after** the bit-reversal (it's equivalent to shift the least significant $n-m$ bits to the most significant $n-m$ positions **before** the bit-reversal). The modified (and more general) round functions are

$$f_i(K, X) = (2^m \times K[3i] \times (X \oplus K[3i+1]))'' + K[3i+2] \quad (\text{mod } 2^n) \quad (0 \le i < r) \quad (2.5)$$
$$g_i(K, X) = (K[3i]^{-1} \times (2^m \times (X - K[3i+2]))'') \oplus K[3i+1] \quad (\text{mod } 2^n) \quad (0 \le i < r) \quad (2.6)$$

The "user" interacts with the cipher by passing and receiving n-bit strings, which are interpreted in accord with the *little endian convention* (least significant bits in lower addresses).

3) Subkey Setup

First consider the sequence of blocks based on a T-function [4] :

$$R_0 = 0$$
$$R_{i+1} = R_i + ((R_i \times R_i) \vee 5) \qquad (i \ge 0)$$

where $\vee$ is the bitwise *or* operation. Note that R_i have the same parity of i. Let C_e and C_o be constant vectors formed by 3r even-blocks and 3r odd-blocks respectively:

$$C_e[i] = R_{20+2i} \qquad (0 \le i < 3r)$$
$$C_o[i] = R_{20+2i+1} \qquad (0 \le i < 3r)$$

The cipher *masterkey* is a single block M provided by the user. From M we derive the "weak" subkey vector W, formed by 3r odd-blocks:

$W[i] = M \oplus C_e[i] \quad (0 \le i < 3r) \qquad$ when M is odd

or

$W[i] = M \oplus C_o[i] \quad (0 \le i < 3r) \qquad$ when M is even

The subkey vector K, used for encryption and decryption, have 3r odd-blocks. Is derived from M, C_o and W in the following way :

$$M' = f(r, C_o, M)$$
$$K[i] = f(r, W, M' + i) \vee 1 \qquad (0 \le i < 3r)$$

The *or* operation forces K[i] to be odd. Since W is derived from M in a very simple way, it's prudent to avoid a direct interaction between them to compute K[i]. The combination of W and M′ guarantees that even related masterkeys (M) will produce uncorrelated subkey vectors (K). Furthermore, it's hard for an attacker to find a subkey K[i] based on the knowledge of other subkeys of K.

4) CHash, a Caligo Based Hash

Forced by the *birthday paradox*, cryptographic hashes must generate very large digests. Due to the *block extensibility*, Caligo can be used in hash modes with any suitable block size, without redefining the algorithm. A particular hash mode to be used with the cipher is proposed in this section. The resulting function is called *CHash*.

By using the weak vector W (instead of K), taking into account that W depends on the masterkey ($W = \omega(M)$) and choosing $r = 6$, we define the encryption function *w* as

$$w(M, X) = f(6, W, X) = f(6, \omega(M), X)$$

The L-bit string to be hashed, $S = S_1 \| S_2 \| \,..\, \| S_m$, is viewed as a concatenation of n-bit substrings or blocks. S_m must be end-padded with zeros if L is not a multiple of *n*. The implementor may choose any suitable block size (generally $160 \le n \le 512$). The hash construction is

$$H_0 = 0$$
$$H_i = w(H_{i-1}, S_i) \oplus H_{i-1} \oplus S_i \qquad (1 \le i \le m) \qquad (4.1)$$
$$H_{m+1} = w(1, H_m \oplus L) \oplus H_m \oplus L \qquad (m \ge 0) \qquad (4.2)$$

where

a) The Preneel-Miyaguchi mode is applied to compute $H_1 \,..\, H_m$. This scheme compensates the use of W (instead of K), since an attacker can't control the masterkey (H_{i-1}) directly. For a discussion on Preneel-Miyaguchi security, see ref. [6] and [7].

b) H_{m+1} is the hash of S.

c) The use of L in H_{m+1} is equivalent to the *Merkle-Damgård strengthening*.

d) H_{m+1} breaks the chain on the first parameter by using the fixed masterkey 1. Moreover, it doesn't depend on S directly. Therefore, H_{m+1} is unlikely to appear in (or used to calculate) an intermediate state of another string. This avoids the *extension attack*, in which the adversary can compute the hash of S||S' without knowing S (or a part of it), but only the hash of S (for Merkle-Damgård strengthened strings, given only the hash of S and the block L representing the length of S, he can calculate the hash of S||L||S') (see [5, section 6.3.1]).

e) $m = \lfloor (L + n - 1) / n \rfloor$. For the empty string, $L = 0 \Rightarrow m = 0 \Rightarrow H_{m+1} = H_1 = w(1, 0)$.
f) The input size limitation is $0 \leq L < 2^{n/2}$.

Since the W setup is faster than a block encryption, *CHash* is not supposed to be much slower than an encryption mode like CBC. The compression function (4.1) timings are given in section 7.

5) Differential Properties

For the cipher, the relation between an input block X, an input difference U and the resulting output difference V is given by

$$V = f(r, K, X) \oplus f(r, K, X \oplus U) \qquad (5.1)$$

In the encryption and decryption functions, the addition and subtraction of a constant operand K[3i+2] protects the cipher against differentials $\langle U, V \rangle$ when $U \oplus V$ have high Hamming weight or $U \wedge V$ (bitwise *and*) have long runs of 1's (see [2]). The multiplication by K[3i] and $K[3i]^{-1}$ gives little protection against differentials $\langle U, V \rangle$ when U and V have low Hamming weight concentrated in the **most** significant bits. The bit-reversal swaps these bits to the **least** significant positions, and the next multiplication can provide a good diffusion.

For M = 0, all possible 16-bit input blocks (X) and input differences (U) had been tested to count the occurrences of all difference pairs or *differentials* $\langle U, V \rangle$. The most frequent differential, $\langle U_{max}, V_{max} \rangle$, for up to 10 rounds was acquired.

For a round function where the *addition* operation is replaced by *xor* (table 5.1), the best pair found is formed by the *iterative palindromic* difference $U_p = U = V = 2^{15} - 2 = 0111...1110$. The pair $\langle U_p, U_p \rangle$ occurs with probability 1/2 in a round. Analyzing this experimental result, we can see that the multiplication have no effect against $U_p = 2^{n-1} - 2$, for all *n*, when X is odd (hence probability 1/2 for any X). In fact, for an odd subkey k and an odd block X, the congruences $X \oplus U_p \equiv U_p + 2 - X \pmod{2^n}$ and $k(U_p + 2) \equiv U_p + 2 \pmod{2^n}$ implies $k(X \oplus U_p) \oplus kX \equiv U_p \pmod{2^n}$. The difference U_p can be concatenated to built a high probability $(1/2^r)$ *r-round differential characteristic* [1].

For the addition operation, the probability of the above pair $\langle U_p, U_p \rangle$ falls drastically to $1/2^{n-2}$, since $U_p \wedge U_p$ have a run of $n-2$ binary 1's [2]. Accordingly, tables 5.2 and 5.3 show how the distribution of differentials $\langle U, V \rangle$ flattens fast as the number of rounds increases, but stops at round **four**. This same behavior was observed on up to 28-bit blocks. It's reasonable to conjecture that we gain no additional protection against conventional differential cryptanalysis with more than four rounds.

Rounds	U_{max}	V_{max}	$\langle U_{max}, V_{max}\rangle$ occurrences
1	0x8000	0x0001	0x10000
2	0x7FFE	0x7FFE	0x4000
3	0x7FFE	0x7FFE	0x1FF0
4	0x7FFE	0x7FFE	0x1028
5	0x7FFE	0x7FFE	0x07F2
6	0x7FFE	0x7FFE	0x03AE
7	0x7FFE	0x7FFE	0x01E0
8	0x7FFE	0x7FFE	0x00E4
9	0x7FFE	0x7FFE	0x0072
10	0x7FFE	0x7FFE	0x0040

Table 5.1: Max. differential occurrences for n = 16 and **addition replaced by xor**

Rounds	U_{max}	V_{max}	$\langle U_{max}, V_{max}\rangle$ occurrences
1	0x8000	0x0003	0x8000
2	0x0824	0x0004	0x180A
3	0x2000	0x0120	0x0050
4	0x5200	0x0003	0x002A
5	0x3B82	0x1AFB	0x0012
6	0xF048	0x1B39	0x0014
7	0xC454	0xFF89	0x0014
8	0xDA97	0x5774	0x0014
9	0x6EC6	0xAEE4	0x0014
10	0x0823	0x28EB	0x0012

Table 5.2: Max. differential occurrences for n = 16 and **correct** round functions

Rounds	U_{max}	V_{max}	$\langle U_{max}, V_{max}\rangle$ occurrences
1	0x20000	0x00003	0x20000
2	0x29091	0x00003	0x01F7C
3	0x02440	0x010A0	0x000DE
4	0x3B236	0x2DB86	0x00016
5	0x08A1E	0x1C2C7	0x00014
6	0x17672	0x0165E	0x00016
7	0x3147B	0x12CB1	0x00016
8	0x02AB5	0x1A822	0x00014
9	0x00A10	0x133BA	0x00014
10	0x01089	0x31860	0x00014

Table 5.3: Max. differential occurrences for n = 18 and **correct** round functions

In the tables 5.2 and 5.3, $U = 2^{n-1}$ (0x8000 and 0x20000) seems to be good input differences (they pass through multiplication with probability 1 and the addition transforms the (bit-reversed) difference 0x01 into 0x03 with probability 1/2). Therefore, it's interesting to verify what the fixed $U = 2^{24-1}$ (0x800000) and $U = 2^{28-1}$ (0x8000000) can do with 24 and 28-bit blocks respectively (tables 5.4 and 5.5).

Round s	U	V_{max}	⟨U, V_{max}⟩ occurrences
1	0x800000	0x000003	0x800000
2	0x800000	0xA92160	0x002280
3	0x800000	0x1F0210	0x00011A
4	0x800000	0x82897F	0x000010
5	0x800000	0x9CE780	0x00000E
6	0x800000	0xECD8F7	0x000010
7	0x800000	0xE99DFD	0x000010
8	0x800000	0x8A7D68	0x000012
9	0x800000	0x555D5A	0x000010
10	0x800000	0x3FE0FA	0x000010

Table 5.4 : Max. differential occurrences for n = 24 and U = 0x800000

Rounds	U	V_{max}	⟨U, V_{max}⟩ occurrences
1	0x8000000	0x0000003	0x8000000
2	0x8000000	0x58221E2	0x0003624
3	0x8000000	0x9400128	0x0000430
4	0x8000000	0x0000030	0x000002E
5	0x8000000	0x17A3DCA	0x0000010
6	0x8000000	0x36A3B96	0x0000010
7	0x8000000	0xD41C966	0x0000010
8	0x8000000	0x5F69B6C	0x0000010
9	0x8000000	0x06CBC79	0x0000012
10	0x8000000	0x8EFE2DE	0x0000014

Table 5.5 : Max. differential occurrences for n = 28 and U = 0x8000000

A noteworthy detail in round four of tables 5.2 to 5.5, is how close are the occurrences of the most frequent differential, despite the block size variation.

Impossible Differentials

For a given non-zero input difference U_0, we see in equation 5.1 that X and $X' = X \oplus U_0$ give the same V. So there are at most 2^{n-1} possible differentials ⟨U_0, V⟩ and 2^{n-1} impossible ones. But after changing the masterkey we may get differentials that could not be found with the previous one.

The following tests had been done with 16 and 18-bit blocks and up to 10 rounds. In equation 5.1, for each value of U, the keys M = 0..63 had been combined with all values of X to compute the number N of **not found** values of V. Tables 5.6 and 5.7 show the larger value of N (N_{max}), the associated input difference U_{max} and the probability $P_{max} = N_{max}/(2^n-1)$ (V=0 excluded) of finding impossible differentials under these keys.

Rounds	U_{max}	N_{max}	P_{max}
1	0x8000	0xFFF0	0.999771
2	0x8000	0x1A68	0.103151
3	0x0800	0x0005	0.000076
4	0x0001	0x0000	0.000000
5	0x0001	0x0000	0.000000
6	0x0001	0x0000	0.000000
7	0x0001	0x0000	0.000000
8	0x0001	0x0000	0.000000
9	0x0001	0x0000	0.000000
10	0x0001	0x0000	0.000000

Table 5.6: Max. impossible differentials for n = 16 and M = 0..63

Rounds	U_{max}	N_{max}	P_{max}
1	0x20000	0x3FFEE	0.999935
2	0x20000	0x0B0BC	0.172593
3	0x10000	0x00044	0.000259
4	0x00001	0x00000	0.000000
5	0x00001	0x00000	0.000000
6	0x00001	0x00000	0.000000
7	0x00001	0x00000	0.000000
8	0x00001	0x00000	0.000000
9	0x00001	0x00000	0.000000
10	0x00001	0x00000	0.000000

Table 5.7: Max. impossible differentials for n = 18 and M = 0..63

These tables prove that, in 16 and 18-bit cases, there are no impossible differentials from 4 to 10 rounds of the cipher. This is coherent with the fact that, for round 4 an beyond, the differential distributions are equally closer to a *normal distribution*, as tables 5.2 to 5.5 suggest.

6) Linear Properties

To verify linear dependencies in the cipher, fixed bit groups from the input (X) and output (Y) blocks are xored together. This is equivalent to xor the bits of the number

$$Z = (m_X \wedge X) \oplus (m_Y \wedge Y)$$

where

a) m_X and m_Y are the bit group selecting masks
b) $\wedge$ is the bitwise *and* operator
c) $\oplus$ is the bitwise *exclusive-or (xor)* operator
d) $Y = f(r, K, X)$

Ideally, the resulting bit should be odd with probability 1/2 for a randomly chosen X.

An exhaustive search was done with the 16-bit variant (n = 16) and M = 0. For each mask pair $\langle mx, my \rangle$, all X and Y was generated. The bits of each Z was xored and the number of odd results accumulated in N. The odd parity probability was approximated by $p = N/2^{16}$ and the bias (deviation from 1/2) by $|p - 1/2|$. The mask pairs with higher bias, for up to 10 rounds, are in table 6.1. The bias for 18 and 20-bit blocks with fixed m_X (0x00001) can bee seen in tables 6.2 and 6.3.

These tables suggest that no additional protection against a linear attack is achieved with more than four rounds.

Rounds	m_X	m_Y	N	p	Bias
1	0x0001	0x8000	0x26EA	0.1520	0.3480
2	0x400F	0xB000	0x5E18	0.3676	0.1324
3	0x0C01	0x8D30	0x865C	0.5248	0.0248
4	0x07E3	0xB000	0x83AE	0.5144	0.0144
5	0xBD75	0x57E4	0x7CD6	0.4876	0.0124
6	0xDDB5	0xDC68	0x7CB0	0.4871	0.0129
7	0x1C6E	0x5D44	0x7CBA	0.4872	0.0128
8	0xFD1A	0xBD9A	0x7CD4	0.4876	0.0124
9	0xB9F2	0x0492	0x7CD0	0.4875	0.0125
10	0x4106	0x0D98	0x8304	0.5118	0.0118

Table 6.1: Higher bias for n = 16 and M = 0

Rounds	m_x	m_y	N	Bias
1	0x00001	0x20000	0x01F66	0.4693
2	0x00001	0x1A67E	0x21922	0.0245
3	0x00001	0x3CF63	0x20816	0.0079
4	0x00001	0x147F9	0x1FB42	0.0046
5	0x00001	0x1206A	0x1FAFC	0.0049
6	0x00001	0x384DA	0x1FA82	0.0054
7	0x00001	0x12A2C	0x204B8	0.0046
8	0x00001	0x0EE66	0x1FB0C	0.0048
9	0x00001	0x347C0	0x2051E	0.0050
10	0x00001	0x1A54E	0x2048E	0.0044

Table 6.2: Higher bias for n = 18, M = 0 and fixed input mask 0x00001

Rounds	m_x	m_y	N	Bias
1	0x00001	0x80000	0xE6296	0.3991
2	0x00001	0xA3685	0x82CD4	0.0109
3	0x00001	0xBDC23	0x80EFC	0.0037
4	0x00001	0x291DC	0x7F42E	0.0029
5	0x00001	0x7DA35	0x80A5C	0.0025
6	0x00001	0x210E1	0x7F5FC	0.0024
7	0x00001	0xC5A45	0x7F5DC	0.0025
8	0x00001	0x37ACF	0x7F5A0	0.0025
9	0x00001	0x27CDF	0x7F5F8	0.0024
10	0x00001	0xB5AE2	0x7F532	0.0026

Table 6.3: Higher bias for n = 20, M = 0 and fixed input mask 0x00001

7) Performance

To be competitive, this algorithm needs a processor with fast 64-bit multiplication like Alpha 21264, Itanium or Athlon64. Table 7.1 compares Caligo with Rijndael C code running on Alpha 21264 processor (see ref. [3]).

Cipher	Rounds	Key size	Block size	Encryption cycles/byte
Rijndael	10	128	128	18
Caligo	6	128	128	23
Caligo	6	256	256	20
Caligo	6	320	320	23
Caligo	6	512	512	39

Table 7.1: Alpha 21264, C code performance of Caligo.

Table 7.2 shows the timings on the AMD Athlon64 processor under Red-Hat Linux. In this case, the *mul* and *bswap* 64-bit assembly instructions was embedded in the C code.

Rounds	Block size	Encryption cycles/byte	Decryption cycles/byte
6	128	21	20
6	256	23	23
6	320	26	26
6	512	38	39

Table 7.2: AMD Atlhon64, C+assembly code performance of Caligo.

Table 7.3 gives the CHash compression function (4.1) performance on the Alpha 21264 processor. The code had been written in C.

Hash	Block size	Cycles/block	Cycles/byte
CHash-256	256	926	28
CHash-320	320	1247	31
CHash-512	512	0	45

Table 7.3: Alpha 21264, C code performance of the CHash compression function.

References

[1] V. Furman "Differential Cryptanalysis of Nimbus", Fast Software Encryption: 8th International Workshop

[2] A. Machado "Differential Probability of Modular Addition with a Constant Operand" http://eprint.iacr.org/2001/052

[3] R. Weiss and N. Binkert "A comparison of AES candidates on the Alpha 21264" http://csrc.nist.gov/encryption/aes/round2/conf3/papers/18-rweiss.pdf

[4] A. Klimov "Applications of T-functions in cryptography" http://www.wisdom.weizmann.ac.il/~ask/th.ps.gz

[5] N. Ferguson and B. Schneier "Practical Cryptography". Wiley Publishing, 2003.

[6] B. Preneel, R. Govaerts and J. Vandewalle, "Hash functions based on block ciphers: A synthetic approach" http://www.cosic.esat.kuleuven.be/publications/article-48.pdf

[7] J. Black, P. Rogaway, and T. Shrimpton, "Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV" http://www.cs.ucdavis.edu/~rogaway/papers/hash.pdf

Appendix A: Caligo Test Vectors

The masterkey (M), plaintext (X_0) and ciphertext (X_r) blocks are given in hexadecimal.

A.1) n = 256, r = 6

```
M  = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Xr = 4F3311B6 A9B391B2 AD0D74E6 F55296F2 911BA9F7 18833BCC 0FD9FCE4 E134AC7C

M  = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X0 = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Xr = 32C66775 46371E6F 6E124370 D2149A02 9B61096D 5637AA0E D909AC2C 777D5931
```

M = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_r = DB87C4DE 5314A39B E79F7B65 E294A9B7 30A03BDD 60F98784 3FEE940F B3E38C09

A.2) n = 320, r = 6

M = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_r = 28554800 088778B4 C7A14690 3876A6EB 6A663BF2 63C4C131 78C9E23C E40ABA5A
97F1976F D5AD179B

M = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_0 = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_r = 5BCC5EB3 E21526C3 774CED6E C5B60448 B7471983 1C7BE9CA 04D2078D 1A543DD4
5C1CAA47 0C46CE01

M = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
X_r = AC701C56 9B31D28B 76D91C02 4AFE4858 FECC054B 5BC877EB BEAA3954 6DE2A95C
3EA44B4E 4B3303F6

A.3) n = 512, r = 6

M = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_r = 9129FD59 3DCB4430 0097D220 7D4F2384 C07B4365 C7226B7E BEB01779 1E8ED80F
D6807D91 6C253196 2130D365 1A931443 57C4EE1A EE4E2AC7 6022A2D1 B6338DA4

M = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_0 = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_r = 4A72B4BB 14D98F1A C0A61B69 B5ADC92D B141D71C 96A737C6 D97ACED0 2D175821
19E59037 8689DA08 455ADC00 033E9671 36CE374F E7987FA7 59243F47 958119B7

M = 01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_0 = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
X_r = 385B012D E0BCF842 C00BC9CD FD567EE2 B971323A 29DF26CE 37202E83 634B2466
E5688E3E C785C42F 870E9D31 17DD6101 617880EF B0AE0231 2A2B8B67 0BBED0DD

Appendix B: CHash Test Vectors

B.1) The empty string

```
n=256: A78FD14E 92A1B6A2 3CA9B32B F87D1560 908F7241 675C3F33 356B863F 55DB056A

n=320: BB248F5A 4428391F 38BACB08 B7FE21C1 2C3D338A AC865AB2 5366FD74 3AAE2CED
       8F07F6F0 A2D0DDFD

n=512: DBEE2656 D4E48C27 167B59EB 25D596EA B09A36F3 DCDD634A 7975AE99 6ED9D1D1
       09D3C093 685A7687 E08A36BE AEF3CC4F 7FF27288 23A6EBD1 89FEC156 29536EE9
```

B.2) The string “abc”

```
n=256: 5BB659EE 309766BA C445C26E 943839D2 E833F3BF 343AEA39 449F5AEF B9F2D404

n=320: 72DD6C73 EBF679A2 17086626 C4BBC793 74D5DE6E 576B3D48 E9977AA2 CFE2352D
       C8E4A75F 71CA2B0D

n=512: D89E708A 2EE4537A 801B56CF 5318DC31 F0A134D0 28EDB69A D4645C54 02688769
       49F377E4 F977002F E7F68420 6E3F82D9 58E78FDF 9CA45E69 70D3B0BC C2FEFE02
```

B.2) The string formed by “a” repeated 1000 times

```
n=256: B7B8F460 CB4F5A54 9334CD86 644B49DE 4F4EAB4B 6F9D54DE 75FEA58A E7760566

n=320: 954F57F7 FBAB9AC5 C32815AB 4A1111E7 AE892ED6 66B93DB3 91A23588 4EEC0693
       712A269B A7686CE9

n=512: 47E10EE0 E28613C5 83A35EB9 9CE63B99 E9E5A02A 9DEF59BC 26DD0F4A B389E1CA
       6DBAE7BD 14F7998C 115523D8 D4F2F8FE 3C4C8CA5 DD51363A D53F6F3B F1557BF7
```