The TIB3 Hash

Miguel Montes Daniel Penazzi[1]

October 2008

Abstract

We describe here the family of hash functions TIB3-224,256,384,512. We give their specifications first, then some motivations for the designs, their resistance against attacks, and performance figures.

[1] Universidad Nacional de Córdoba, Facultad de Matemática, Astronomía y Física, Córdoba, Argentina, Haya de la Torre y Medina Allende, +54-351-4334051/363,e-mail: miguel.montes@gmail.com,penazzi@mate.uncor.edu

Contents

1 Introduction

1.1 Notation

$GF(q)$ denotes the finite field with q elements.

$\mathbb{Z}$ denotes the integers.

$\mathbb{Z}/(n)$ the integers mod n.

$+$ means the sum in $\mathbb{Z}/(n)$ for $n = 2^{64}$ or $n = 2^{32}$.

$\oplus$ is the sum of $GF(2)$ or of $GF(2)^n$ depending on the context (i.e., the xor bit a bit).

$a \ll r$ means right shift by r bits, i.e., multiplication by 2^r.

$a \gg r$ means left shift by r bits, i.e., the integer part of the division of a by 2^r.

$a||b$ is the concatenation of words a and b.

2 Specification

Here we specify the design of the new family of hash functions TIB3-224,256,384 and 512.

2.1 Overview

The hash functions described here all are based on a generalization of the Merkle-Damgård construction.

We explain in more detail below, but here we give a general idea of the hash function.

Let's review first the current standard, SHA-256. As a high level SHA-256 can be described as an iterative hash function with the Merkle-Damgård scheme in which the underlying compression function is based on the Davies-Meyer scheme applied to a 256-bit block cipher constructed following an unbalanced Feistel design, source-heavy. The cipher uses a mix of xors, sums and special non-linear compression functions. The expansion of the key is done by means of an LFSR-like expansion but with a mix of xors, sums and rotations. The expansion is invertible in the sense that it uses 16 32-bit words that are expanded into 64 32-bits words in such a way that given for example the last 16 words the recursion can be worked backwards to get again all the words.

TIB3 shares some high level characteristics with SHA256, but with added security features so an attack on SHA256 is unlikely to extend to TIB3 . The main security features are:

1) Besides the usual previous hash and current message block, the compression function also uses the number of bits processed and the previous message block.

2) The last iteration is done using a different scheme than Davies-Meyer, to prevent extension attacks.

3) The underlying block cipher is an SPN cipher instead of an unbalanced Feistel, and it uses, besides xors and sums, non-linear bijective Sboxes.

4) The expansion of the key is done in such a way that a backward recursion is unlikely to succeed.

We will go now into the details.

2.2 The General Scheme

As outlined above, TIB3 uses a block cipher, which we describe in the next section. Here we described how it is used.

The block cipher is a block cipher that can be salted, i.e., we have a family of functions $\{E^s : \{0,1\}^k \times \{0,1\}^n \mapsto \{0,1\}^n : (K,P) \mapsto E^s_K(P)\}_{s\in S}$, where for each $s \in S$, $E^s_K(P)$ encrypts P with the key K. (the idea is that s does not change the security, and it can be under the control of an adversary. In our scheme s is used to change slightly the expansion of the key into the round keys). We use $S = \{0,1\}^L$, and assume that k is even and call $r = \frac{k}{2}$.

Let M be a message of bitlength ℓ. Set $t = \lceil \frac{\ell}{r} \rceil$.(here $\lceil x \rceil$ is the least integer greater than or equal to x). Divide the first $(t-1)r$ bits of M into blocks $m_1, ..., m_{t-1}$, each of length r. (if $t = 1$, then there is nothing here). If ℓ is a multiple of r (i.e. $t = \frac{\ell}{r}$ exactly), then let m_t be the last r bits of M. Otherwise, construct m_t by taking the last $\ell - r(t-1)$ bits of M, append a 1, and then 0s as needed to complete r bits. Finally, construct a last block m_{t+1} that consists of ℓ mod 2^L in the first L bits,followed by $r - L$ zeroes.

Define:

$$\ell_i = \begin{cases} i.r \bmod L & i = 1, ..., t-1 \\ \ell \bmod L & i = t \\ 0 & i = t+1 \end{cases}$$

Let $\hat{h} = \overbrace{0...0}^{r-n} || h$ be the extension of an element of $\{0,1\}^n$ to an element of $\{0,1\}^r$ by appending zeroes to the left.

Let $h_0 \in \{0,1\}^n$ and and $m_0 \in \{0,1\}^r$ be IVs and define for $i \geq 1$:

$$h_i = \begin{cases} E^{\ell_i}_{m_i || m_{i-1}}(h_{i-1}) \oplus h_{i-1} & \text{if } i \leq t \\ E^{\ell_i}_{m_i \oplus \hat{h}_{i-1} || m_{i-1}}(h_{i-1}) \oplus h_{i-1} & \text{if } i = t+1 \end{cases}$$

Then $H(M)$ is defined to be h_{t+1} (in the cases of length 256 and 512) or the truncation to the leftmost 224 (resp. 384) digits of h_{t+1} in the case of length 224. (resp 384).

In all our cases, $L = 64$, and in the case of $TIB3$-256 (and $TIB3$-224), $k = 1024$ and $n = 256$. In the case of $TIB3$-521 (and $TIB3$-384), $k = 2048$ and $n = 512$.

The 224 bit version is the 256 version with different IVs and truncated, but the block cipher used is the same. The 384 version is the 512 version with different IVs and truncated, but the block cipher is the same. The block cipher for the 512 version is based on the 256 version. So we will start by explaining the 256 version.

2.3 The Block Cipher for the 256 bit case

Now we are going to describe a block cipher with encryption block of 256 bits, and key size 1024 bits with salt of 64 bits. We want to emphasize that the block cipher was designed taking into account that it was going to be used as a building block within a hash function and not as a block cipher per se. In particular, we are not claiming 1024-bit security, and because of the way we are going to use it, the key expansion treats some bits of the key in a different manner than others, which for a general purpose block cipher it would be a bad idea.

The cipher has 16 rounds, and all the rounds are equal. The round is a substitution permutation network: xor a round key, pass bits through Sboxes, spread the local changes by means of a series of xors, shifts and 32-bit sums.

The structure of the round is not symmetrical: if we think of the 256 bits as four 64-bit words, then not every word is treated in the same manner. A permutation of the words at the end of the rounds ensures that at the end of each block of 4 rounds all words have been treated similarly. Let's denote the four sixty-words as A,C,E and G.

At the beginning of the round C is xored to G. (this in fact was thought originally to be part of the diffusion of the previous round, but for efficiency reasons is put here.)

A, C, E and G are xored with some round keys,and then the bits of A,C and E are passed through sixty-four 3-by-3 Sboxes.

The passage through the Sboxes is done in a bitslice way, like Serpent does with their 4 by 4 Sboxes. Namely, bits 0 of A,C and E go through one Sbox, bits 1 of A,C and E go through another Sbox, etc. The Sboxes are all the same: if we represent 000 as 0, 001 as 1, 010 as 2, etc, then it is the Sbox 64170352. i.e., 0 goes to 6, 1 goes to 4, 2 to 1, etc. We denote this as $Sbox(A, C, E)$. Meanwhile, G is subjected to a transformation $PHTX(G)$ that mixes its bits.

The transformation is a mix of xors, 32-bit sums and shifts. Given a 64 bit word D, viewed as an integer in $\{0, 1, ..., 2^{64} - 1\}$, we denote $D \ll a$ the left shift by a, i.e., the multiplication modulo 2^{64} of D by 2^a, and $D \gg a$ is the right shift by a, i.e., the integer part of the division of D by 2^a. $+$ denotes the sum modulo 2^{64} and $\oplus$ the xor bit a bit.

Given a 64 bit word D, we define $D^* = PHTX(D)$ to be the function:

$$\begin{aligned} \tilde{D} = & \quad D + (D \ll 32) + (D \ll 47) \\ D^* = & \quad \tilde{D} \oplus (\tilde{D} \gg 32) \oplus (\tilde{D} \gg 43) \end{aligned}$$

We call this function a "PHTX" function, because of its similarity with the usual PHT function of two words $(L, H) \mapsto (L + 2H, L + H)$, except that we have not only $+$ but also some shifts and xors.

After A, C, and E have gone through the Sboxes and G through the PHTX function, we spread the changes across some of the words: we pass the new C through the $PHTX$ function, and then add G to A and E to G. The addition used here is not the addition of $\mathbb{Z}/(2^{64})$, but rather the addition of the group $(\mathbb{Z}/(2^{32}))^2$, i.e., two parallel 32-bit additions. After this, we shift the words to the left for the next round: $(A, C, E, G) = (C, E, G, A)$.

If we denote the addition of $(\mathbb{Z}/(2^{32}))^2$ as $\tilde{+}$, then the entire round is: (denoting assignment by :=)

$$\begin{aligned}
G &:= G \oplus C \\
(A, C, E, G) &:= (A, C, E, G) \oplus \mathit{roundkeys} \\
(A, C, E) &:= \mathit{Sbox}(A, C, E) \\
G &:= PHTX(G) \\
C &:= PHTX(C) \\
A &:= A \tilde{+} G \\
G &:= E \tilde{+} G \\
(A, C, E, G) &:= (C, E, G, A)
\end{aligned}$$

2.3.1 Key Expansion

Because of the way the cipher is going to be used, we think of the key as consisting of a left part and a right part: $K = (LK, RK)$, each of 512 bits. (recall that the current block will be put into LK and the previous block into RK)

$LK \oplus RK$ is expanded to 2048 bits under the control of RK, in a way described below. Also, LK and RK by themselves are used as part of the round keys.

If we denote the expansion of $LK \oplus RK$ under the control of RK as sixty-four-bits words $D_0, ..., D_{31}$, and denote LK as the sixty-four-bit words $LK_0, ..., LK_7$ and similarly with RK, then the 64 round keys are:

Round 1 Keys:	D_0, LK_0, D_1, LK_0	Round 2 Keys:	D_2, LK_1, D_3, LK_1
Round 3 Keys:	D_4, LK_2, D_5, LK_2	Round 4 Keys:	D_6, LK_3, D_7, LK_3
Round 5 Keys:	D_8, LK_4, D_9, LK_4	Round 6 Keys:	$D_{10}, LK_5, D_{11}, LK_5$
Round 7 Keys:	$D_{12}, LK_6, D_{13}, LK_6$	Round 8 Keys:	$D_{14}, LK_7, D_{15}, LK_7$
Round 9 Keys:	$RK_0, D_{16}, RK_1, D_{16}$	Round 10 Keys:	$RK_2, D_{17}, RK_3, D_{17}$
Round 11 Keys:	$RK_4, D_{18}, RK_5, D_{18}$	Round 12 Keys:	$RK_6, D_{19}, RK_7, D_{19}$
Round 13 Keys:	$D_{20}, D_{21}, D_{22}, D_{21}$	Round 14 Keys:	$D_{23}, D_{24}, D_{25}, D_{24}$
Round 15 Keys:	$D_{26}, D_{27}, D_{28}, D_{27}$	Round 16 Keys:	$D_{29}, D_{30}, D_{31}, D_{30}$

2.3.2 Expansion of $LK \oplus RK$

Expansion of $LK \oplus RK$ is done by means of a modified LFSR: Consider the following function ψ that takes as inputs four 64-bit words W, X, Y, Z and outputs one 64 bit word $V = \psi(W, X, Y, Z)$ by means of the following transformations:

$$V \quad := \quad (Y + (Z \ll 32)) \oplus W \oplus X \oplus (Z \gg 32)$$

$$\begin{aligned} V &:= V + (V \lll 32) + (V \lll 43) \\ V &:= V \oplus (V \ggg 39) \end{aligned}$$

If $LK \oplus RK$ is loaded into $D_0, ..., D_7$, we define D_8 and D_9 to be:

$$\begin{aligned} D_8 &= \psi(D_3 \oplus RK_0, D_4 \oplus RK_1, D_5 \oplus RK_2, D_1 \oplus RK_3) \\ D_9 &= \psi(D_2 \oplus RK_4 \oplus const, D_7 \oplus RK_5 \oplus salt, D_6 \oplus RK_7, D_0 \oplus RK_6) \end{aligned}$$

where *salt* is the salt value and *const* is the constant 0x428a2f98d728ae22 (the first round constant of SHA512). Once obtained these ten 64-bit words $D_0, D_1, ..., D_9$, we do a recursion for $i \geq 10$ by means of $D_i = \psi(D_{i-10}, D_{i-8}, D_{i-3}, D_{i-2})$ (the polynomial $x^{10} + x^8 + x^3 + x^2 + 1$ is primitive).

2.3.3 I/O specification

Each block of 512 bits=64 bytes is read as follows: given the block $b_0||b_1||....||b_{63}$, then $b_0 + b_1 2^8 + b_2(2^8)^2 + ... + b_7(2^8)^7$ is loaded into the first 64-bit word, $b_8 + b_9 2^8 + ... + b_{15}(2^8)^7$ is loaded into the second 64-bit word, etc.

The final hash is the content of the registers A, C, E, G at the end of all the iterations. This is extracted as the following bytes: $A\&0xFF$, $(A \ggg 8)\&0xFF$, $(A \ggg 16)\&0xFF$, etc,

2.4 The Block Cipher for the 512 bit case

The block cipher in this case is essentially the same as the one for the 256 bit case except that all the 64-bits words of the 256 bit case are now 128-bit words. The 32-bit sums are now all 64-bit sums, and there are some other small differences in the shift amounts and in the diffusion, because due to the larger size of the block, more effort is needed in order to ensure a good mix. Also, because 128-bit registers are not common, we prefer to explain the cipher in terms of 64-bit words. Basically it is the transformation to 64-bit words of the implementation of the 256 case in 32-bit words.

As before there are 16 rounds.

We write the state now as eight 64-bit words A, B, C, D, E, F, G, H. Each round is now: (+ is the 64-bit sum in all cases)

$$\begin{aligned} G &:= G \oplus C \\ H &:= H \oplus D \\ (A, B, C, D, E, F, G, H) &:= (A, B, C, D, E, F, G, H) \oplus roundkeys \\ (A, C, E) &:= Sbox(A, C, E) \\ (B, D, F) &:= Sbox(B, D, F) \\ (G, H) &:= PHTXD(G, H) \end{aligned}$$

$$\begin{array}{rcl}
(C,D) & := & PHTXD(C,D) \\
A & := & A+G \\
B & := & B+H \\
G & := & E+G \\
H & := & F+H \\
(A,B,C,D,E,F,G,H) & := & (C,D,E,F,G,H,A,B)
\end{array}$$

where $PHTXD$ is a "double" version of PHTX:

$$\begin{array}{l}
PHTXD(L,H): \\
\qquad \begin{array}{rcl}
H & := & H \oplus L \\
H & := & PHTX(H) \\
L & := & L \oplus H \\
L & := & PHTX(L)
\end{array}
\end{array}$$

(here $PHTX$ is the same function defined in the 256 case).

The expansion is again based on a function φ which is basically the ψ function of the 256 case, but with 128-bit arguments instead of 64-bit arguments. There are some differences in the shift amounts needed because of the larger size. However, since we have not implemented the code in 128-bit machines, and in order to have an easier "map" into the reference implementation (of 64 bits), we list the function with eight 64-bit arguments instead of four 128-bit arguments and producing two 64-bit outputs instead of one 128-bit output:

$$(V,V^*) \quad = \quad \varphi(W,W^*,X,X^*,Y,Y^*,Z,Z^*):$$

$$\begin{array}{rcl}
V & := & W \oplus X \oplus Y \oplus Z^* \\
V^* & := & ((Y^*+Z) \oplus W^* \oplus X^*) + V + (V \ll 23) \\
V & := & V \oplus (V^* \gg 15)
\end{array}$$

As in the 256 case, we xor the leftmost 512 bits of the key (written $LK_0,..,LK_{15}$) with the rightmost 512 bits of the key (written $RK_0,...,RK_{15}$) and put this into (in this case) sixteen 64-bit words $W_0,...,W_{15}$.

We compute $W_{16},...,W_{19}$ by:

$$\begin{array}{rcl}
(W_{16},W_{17}) & := & \varphi(\diamondsuit_0,\diamondsuit_1,\diamondsuit_2,\diamondsuit_3) \\
(W_{18},W_{19}) & := & \varphi(\spadesuit_0,\spadesuit_1,\spadesuit_2,\spadesuit_3)
\end{array}$$

where:

$$\begin{aligned}
\diamondsuit_0 &= (W_6 \oplus LK_0, W_7 \oplus LK_1) \\
\diamondsuit_1 &= (W_8 \oplus LK_2, W_9 \oplus LK_3) \\
\diamondsuit_2 &= (W_{10} \oplus LK_4, W_{11} \oplus LK_5) \\
\diamondsuit_3 &= (W_2 \oplus LK_6, W_3 \oplus LK_7) \\
\spadesuit_0 &= (W_4 \oplus LK_8 \oplus const, W_5 \oplus LK_9) \\
\spadesuit_1 &= (W_{14} \oplus LK_{10} \oplus salt, W_{15} \oplus LK_{11}) \\
\spadesuit_2 &= (W_{12} \oplus LK_{14}, W_{13} \oplus LK_{15}) \\
\spadesuit_3 &= (W_0 \oplus LK_{12}, W_1 \oplus LK_{13})
\end{aligned}$$

where *const* is again 0x428a2f98d728ae22.

Then we compute W_i, $i = 20, ..., 63$ by:

$$(W_i, W_{i+1}) = \varphi(W_{i-20}, W_{i-19}, W_{i-16}, W_{i-15}, W_{i-6}, W_{i-5}, W_{i-4}, W_{i-3})$$

The round keys are:
Round 1 Keys: $W_0, W_1, LK_0, LK_1, W_2, W_3, LK_0, LK_1$
Round 2 Keys: $W_4, W_5, LK_2, LK_3, W_6, W_7, LK_2, LK_3$
Round 3 Keys: $W_8, W_9, LK_4, LK_5, W_{10}, W_{11}, LK_4, LK_5$
Round 4 Keys: $W_{12}, W_{13}, LK_6, LK_7, W_{14}, W_{15}, LK_6, LK_7$
Round 5 Keys: $W_{16}, W_{17}, LK_8, LK_9, W_{18}, W_{19}, LK_8, LK_9$
Round 6 Keys: $W_{20}, W_{21}, LK_{10}, LK_{11}, W_{22}, W_{23}, LK_{10}, LK_{11}$
Round 7 Keys: $W_{24}, W_{25}, LK_{12}, LK_{13}, W_{26}, W_{27}, LK_{12}, LK_{13}$
Round 8 Keys: $W_{28}, W_{29}, LK_{14}, LK_{15}, W_{30}, W_{31}, LK_{14}, LK_{15}$
Round 9 Keys: $RK_0, RK_1, W_{32}, W_{33}, RK_2, RK_3, W_{32}, W_{33}$
Round 10 Keys: $RK_4, RK_5, W_{34}, W_{35}, RK_6, RK_7, W_{34}, W_{35}$
Round 11 Keys: $RK_8, RK_9, W_{36}, W_{37}, RK_{10}, RK_{11}, W_{36}, W_{37}$
Round 12 Keys: $RK_{12}, RK_{13}, W_{38}, W_{39}, RK_{14}, RK_{15}, W_{38}, W_{39}$
Round 13 Keys: $D_{40}, D_{41}, D_{42}, D_{43}, D_{44}, D_{45}, D_{42}, D_{43}$
Round 14 Keys: $D_{46}, D_{47}, D_{48}, D_{49}, D_{50}, D_{51}, D_{48}, D_{49}$
Round 15 Keys: $D_{52}, D_{53}, D_{54}, D_{55}, D_{56}, D_{57}, D_{54}, D_{55}$
Round 16 Keys: $D_{58}, D_{59}, D_{60}, D_{61}, D_{62}, D_{63}, D_{60}, D_{61}$

2.4.1 I/O specification

They are the same as in the 256 case, save that the block is 1024 bits long.

2.5 IVs

For each length we need two IVs: one for h_0, the other for m_0 (the first "previous block"). Our IVs are based on the IVs for SHA512. Basically the initial hash h_0 for TIB3-512 takes the same words used in the IV for SHA-512. (i.e., eight words), while the h_0 for TIB3-384 are those same words but putting the last four words first and viceversa. m_0 for TIB3-512 is the h_0 for $TIB3-384$ repeated, and m_0 for TIB3-384 is the h_0 for TIB3-512 repeated. h_0 for TIB3-256 is the first four words of the h_0 of TIB3-512, while the h_0 of TIB3-224 is the last four words of the h_0 of TIB3-512. (equivalently, the first four words of TIB3-384). m_0 for TIB3-256 is the h_0 for TIB3-224 repeated and viceversa.

Explicitly: (all words are to be read left to right, then top to bottom)

TIB3-256:

h_0 consists of the following four 64-bit words:

$$\begin{array}{cc}
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1
\end{array}$$

m_0 consists of the following eight 64-bit words:

$$\begin{array}{cc}
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179 \\
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179
\end{array}$$

TIB3-224:

h_0 consists of the following four 64-bit words:

$$\begin{array}{cc}
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179
\end{array}$$

m_0 consists of the following eight 64-bit words:

$$\begin{array}{cc}
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1 \\
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1
\end{array}$$

TIB3-384:

h_0 consists of the following eight 64-bit words:

$$\begin{array}{cc}
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179 \\
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1
\end{array}$$

m_0 consists of the following sixteen 64-bit words:

$$\begin{array}{cc}
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1 \\
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179 \\
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1 \\
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179
\end{array}$$

TIB3-512:
h_0 consists of the following eight 64-bit words:

$$\begin{array}{cc}
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1 \\
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179
\end{array}$$

m_0 consists of the following sixteen 64-bit words:

$$\begin{array}{cc}
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179 \\
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1 \\
0x510e527fade682d1 & 0x9b05688c2b3e6c1f \\
0x1f83d9abfb41bd6b & 0x5be0cd19137e2179 \\
0x6a09e667f3bcc908 & 0xbb67ae8584caa73b \\
0x3c6ef372fe94f82b & 0xa54ff53a5f1d36f1
\end{array}$$

3 Rationale for the design

We consider three possibilities for the top structure of a hash function:

1) A traditional structure that use a block cipher in one of the secure schemes of [PGV94] as compression function, with a Merkle-Damgård strengthening.

2) Use the same Merkle-Damgård iteration, but with a stream cipher instead of a block cipher as underlying compression function.

3) Use a different approach altogether. (for example, a belt and mill structure like Panama or Radio-Gatun).

We felt that approach 3), trying a new kind of structure, was too risky for the SHA-3 competition, although in the particular case of the belt and mill structure, it already has some years of exposure, so to use that structure would not have been that risky. However, we did not "felt at home" with that structure.

The structure 2) has many good advantages, but altogether, we felt that keeping a traditional approach was the right choice. However, we added to the usual design two security features: one: re-use of the previous input block in the iterative step. Two, use the number of bits processed in the input function.

Of the 12 secure schemes of [PGV94], only four resist all attacks. However, they are all of the form $h_i = E_{h_{i-1}}(*) \oplus (\star)$, where $*$ and $\star$ can each be either m_i or $m_i \oplus h_{i-1}$. However, we wanted to read our message blocks in chunks of 512 bits, and any of these schemes would force us to construct a 512-bit cipher for a hash function of length 256, which we felt was a misuse of resources. So, we settled on the popular Davies-Meyer scheme $h_i = E_{m_i}(h_{i-1}) \oplus h_{i-1}$, used on all the members of the MDx family, including SHA256 and friends. The Davies-Meyer scheme is one of the 12 secure schemes of PGV, with the only problem of vulnerability to a fixed point attack. This is not a problem in the MDx family due to the Merkle-Damgård strengthening at the end of the message, but just in case we decided to add the processed bits as a salt into the block cipher, which makes then even this possible problem vanish.

We decided also to use the previous block in the recursion. In the Davies-Meyer scheme, the attacker has control of the key. In a collision attack, the attacker will try to get a collision by choosing appropriate message blocks. However, if each message block is used in two iterations, the attacker will have to get a collision for the last time the block is used. This mean two things: first, in the last block for which the attacker tries to get a collision, half the values of the key cannot be used in the attack (because they are going to be reused in the next iteration). Second, the values that are used in the attack have already been used in the previous iteration, which means that the attacker has to set up a system of equations for the current iteration and the previous one also. This in fact forces the attacker to work through 32 rounds of the cipher, instead of just 16.

3.1 The 3×3 S-box

We wanted in the design to have Sboxes, and not rely only on the non linearity provided by a combination of xors and sums. Large Sboxes are very good, but they require the use of huge tables. We wanted the hash to be easily implemented in low resources environments, so we wanted to use only sums,shifts and a combination of logical operators like xor, or, and, etc.

This naturally pointed the way for small Sboxes that could be implemented in a bitsliced way by means of their component boolean functions. The natural one would be a 4 by 4 Sbox, like Serpent, but the boolean logic of any good 4by4 Sbox is extensive. 3 by3 Sboxes have very short boolean logics.

We asked our S-box to be invertible, to minimize the greatest non trivial value in the XOR difference table (resistance against differential cryptanalysis), to minimize the highest non trivial correlation among linear combinations of input and output bits. (resistance against linear cryptanalysis), to have no fixed points or opposite fixed points ($x \mapsto \overline{x}$), and to minimize the number of times that a difference of one bit in the input translates into a difference of one bit at the output. These were studied in [FP05], and it turns out they are all "algebraic", i.e., they all result form the construction of [Nyb94] of taking the inverse function over $GF(2^3)$:

$$x \mapsto \begin{cases} x^{-1} & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases}$$

and then pre-compose and/or post-compose with an affine transformation. We chose the Sbox used in [FP05], which is one of the ones obtained by further asking that all possible representations as $S(x) = (Q_M.(P_M x + P_b)^{-1} + Q_b)$ (and with any of the two ways of representing $GF(8)$ as $GF(2)[x]/p(x)$) either have Q_M or P_M different from the identity matrix, and such that if $X_M = I$ then $X_b \neq 0$; that it should sent 0 to either 3,5 or 6 (numbers with hamming weight two) and that its cycle representation has length 8. Representing $0 = 000$, $1 =$

001, $2 = 010$, etc, then the Sbox is $(0,1,2,3,4,5,6,7) \mapsto (6,4,1,7,0,3,5,2)$ with cycle representation (06537214). Its weakest representation as a composition of affine functions and the inverse map is $S(x) = A(x+1)^{-1} + 4$ with

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

and when $GF(8)$ is viewed as $GF(2)[x]/(x^3+x+1)$ As a polynomial over $GF(8)$ it is $6x^6+3x^3+5x^4+7x^3+x^2+4x+6$ when $GF(8)$ is viewed as $GF(2)[x]/(x^3+x+1)$ and $7x^6+4x^4+x^2+6$ when $GF(8)$ is viewed as $GF(2)[x]/(x^3+x^2+1)$

Since we are going to implement it in bitsliced mode, its component boolean functions are relevant: they are:

$$\begin{aligned} f(x,y,z) &= \overline{x} \oplus (y \wedge \overline{z}) \\ g(x,y,z) &= z \oplus (\overline{x} \wedge \overline{y}) \\ h(x,y,z) &= y \oplus (x \wedge z) \end{aligned}$$

The difference table given by:

$$(\Delta x, \Delta y) \rightarrow (\#\{x \in \{0,1\}^3 : S(x) \oplus S(x \oplus \Delta x) = \Delta y\})$$

is:

	1	2	3	4	5	6	7
1	0	2	2	0	0	2	2
2	2	0	2	0	2	0	2
3	2	2	0	0	2	2	0
4	0	0	0	2	2	2	2
5	0	2	2	2	2	0	0
6	2	0	2	2	0	2	0
7	2	2	0	2	0	0	2

If $a \cdot b$ is the parity (0 or 1) of the bitwise product of a and b, then the linear table

$$(\Gamma x, \Gamma y) \rightarrow (\#\{x \in \{0,1\}^3 : x \cdot \Gamma x = S(x) \cdot \Gamma y\} - 4)$$

is:

	1	2	3	4	5	6	7
1	0	2	-2	0	0	2	2
2	2	0	2	0	-2	0	2
3	2	-2	0	0	2	2	0
4	0	0	0	-2	-2	2	-2
5	0	-2	-2	2	-2	0	0
6	2	0	-2	-2	0	-2	0
7	-2	-2	0	-2	0	0	2

3.2 Diffusion

We wanted a diffusion that would involve operations outside of $GF(8)$, to prevent algebraic attacks. A very good idea would be to use some combinations of multiplications and perhaps variable rotations. However, these operations, although reasonably fast in the target Core2Duo machine, became unreasonable in constrained environments. So we wanted the diffusion to consists only on a combination of sums, xors and shifts. Sums were necessary to add extra nonlinearity to the small 3by3 Sboxes and to prevent algebraic attacks. We excluded rotations because we wanted the code to work well on both 32 bit and 64 bits machines, and rotations of 64 bit words are slow in 32 bits machines and viceversa. However, shifts of at least 32 bits work well on both machines, although it is true that the diffusion could have been better using other shifts (and rotations). The diffusion we settled on is simple enough but quickly produce a number of active Sboxes if small differences are not deal with immediately.

3.3 Key Expansion

The key expansion was a balance act between security and speed. The more bits we used that come from a complex expansion, the better security, but the worst the speed. So we settled on expanding part of the key, and using other parts just by repeating them. For the expansion itself we used originally a linear code, because in that case the minimum nonzero distance equals the minimum weight. This is an advantage to both the attacker and the designer, but to the designer is an advantage only if he can prove lower bounds. For the codes we tried either we could not prove lower bounds or they were too slow. We decided to use a nonlinear code. The minimum nonzero hamming distance of a linear code is not the minimum weight and makes the attacker job more difficult. (in a linear expansion, the attacker can try vectors that have some small Hamming weight and satisfy some conditions, then simply add it to an existing message to find another message that collides. In a nonlinear expansion, he or she has to work with pairs of messages from the start). Again, we had to balance speed and security. The one we settled on passed some test we threw at it that showed that apparently any small difference at the start will grow to a big difference by the end, which is the critical portion of a a differential attack, since a collision has to be produced there. The attacker then might try to find a small weight at the end, and simply run the recursion backward. But here we added the extra security feature of having a recursion of length 10, but only provide 8 inputs. That is, in order to work the recursion backward, the attacker has to fix a target $D_{23}, ..., D_{31}$ that s/he wants to obtain. Working backwards, he or she can find suitables $D_0, .., D_9$ that will produce the targets. However, these are not necessarily acceptable, since s/he can only control $D_0, ..., D_7$, while D_8 and D_9 depend on them. So, after working the recursion backward, the attacker has to verify that the results are compatible verifying the equations that produce D_8 and D_9 from $D_0, ..., D_7$. This will happen only with probability 2^{-128} if the targets are random. So actually the attacker will have to develop a method to work this out.

Moreover, this feature of widening the window of recursion allow us to introduce constants in the recursion, that prevent slide attacks.

3.4 Round keys

The order of the round keys was chosen again with regard to implementation considerations: to increase speed, as we said above, we reuse parts of the key instead of expanding. But we wanted to reuse them in the same round, so as to not reload them. Since there is a difference in the treatment of A, C, E with that of G, we felt it was safe to use the same key once on one of A, C, E, and a second time on G.

If we were designing a cipher solely as a cipher, the treatment of LK and RK should have been more symmetric. However, because of the way the cipher is going to be used, we actually wanted them to be asymmetric, so that the way a message block is used the first time is different from the way it is used the second time.

3.5 The number of Rounds

Although after 4 rounds every part of the state has been treated similarly, so in theory we could take for example an odd number of rounds, it seems better to have a multiple of 4 rounds, so that after the whole rounds every part of the cipher is treated As explained below, if the attacker lets a small difference propagate, very soon a critical mass of active Sboxes makes the attack pass the 2^{128} work factor. Once the number of active Sboxes became critical, two or three rounds are enough. However, the attacker will obviously try to control the differences. Since the attacker has control of the key, there can be 1024 bits with very low weight. The expansion of the key will multiply this low weight into heavier weights, but it needs "time" to do so. In order to properly give the expansion the needed time, and taking into account that we also wanted more use of both the current and previous block we felt that 16 rounds allow the expansion of the key to do its job, and give an adequate margin of security given that the attacker has control of the key. For comparison, the 256 bit version of Rijndael (also an SPN cipher) has 14 rounds. Since the first multiple of 4 greater than 14 is 16 this gave us another reason for this number. However, taking into account that each block is used through 32 rounds the safety margin is higher.

A weakened version of the algorithm would be a version with less numbers of rounds. However, for a proper assessment of a weakened version that takes into account the expansion, the rounds that are eliminated should be the first ones, e.g a 12-round weakened version of the algorithm should have the expansion as it is, and simply eliminate rounds 1-4. (then $D_0, ..D_7$ for example would not be used directly in the key rounds).

On the other hand the algorithm can be readily strengthened if the need arise. A strengthened version of the algorithm would include 4 more rounds (or a multiple of 4), and the round keys for those rounds would came from simply continuing the expansion of D into $D_{32}, ...$, etc.

We think that the 20-round version of the algorithm would be much stronger with respect to the 16 round algorithm than the strength of the 16 round version versus the 12 round version. The slow in speed due to four more rounds plus 12 more rounds of expansion however could be too much, and overall we felt that the 16 round version is strong enough.

3.6 The padding and the final iteration

A problem with the design that we chose is that since we want each message block to be processed twice we need to necessarily add an extra block, regardless of the length of the message. We do the usual padding of appending a 1 followed by 0s as needed, with one exception: if the length of the message is exactly a multiple of 512, we do not pad. We could do that, but in that case the 1 would go necessarily in the next block, which anyway will have a codification of the length of the message. Thus, adding or not an extra 1 merely changes the codification of the length, so it is not needed.

As for the last iteration, we change the way it is processed to avoid any kind of length extension attack: if we were not to do that, an attacker that knows the final hash (but not the whole message) and the length of the message would know the "previous block", and can append any new blocks as needed, since in that case the original final hash will be an intermediate hash. The attacker can claim that since he knows the new final hash he knows the message. This can be of relevance in some applications. But since we change the way the last iteration is done with respect to the previous ones, the final hash is not an intermediate hash, and this attack cannot be mounted.

4 Ease of Replacement of SHA2

Despite the internal differences, TIB3 as a black box behaves exactly like SHA2. By this we mean that it also has an iterated block structure, the size of the blocks (versions 224/256 and 384/512) are the same for the SHA2 family and the size of the partial hashes is also the same. This means that if there is a need for a replacement of the SHA2 family TIB3 should be able to do it straightforwardly. (the only basic difference in a concrete application is that TIB3 uses more internal memory to hold the previous block and the heavier expansion, but externally it still reads one block at a time).

In particular, from the point of view of applications, any general construction that uses the SHA2 family in a general way (e.g, digital signatures, key derivation, pseudorandom bit generators, etc) can also use the TIB3 family in exactly the same way. In particular, the HMAC construction is applicable to TIB3 in the same way as with SHA2.

It may have other specific constructions that take advantage that there is a "previous block" IV that can be used as part of a secret for example, but we have not investigated its suitability and are not making any claims at the moment.

5 Security

5.1 Resistance of the general scheme

5.1.1 Collision Resistance of the general scheme in the black-box model

We could divide the proof in two parts: one showing the resistance of the general scheme given a compression resistant compression function of the type we use, doing an analogous to the Merkle-Damgård theorem, and a second part proving the resistance of the compression function we use, given a general block cipher with the characteristics we need, doing an analogous of the security proof of the Davies-Meyer scheme.

If one looks at the original proofs of the Merkle-Damgård and Davies-Meyer schemes one sees that the differences in the salt and the previous block do not affect them negatively, so they hold for our scheme.

In the interest of completeness however (and to explicitly show that our statement above is true), we include here an integrated proof of the security of the whole scheme, under the assumption of a random block cipher. (we repeat that this is basically the same classical proofs).

That is, we have a collection $\{E^s\}_s$ of random block ciphers $E^s : \{0,1\}^k \times \{0,1\}^n \mapsto \{0,1\}^n$. We denote by D^s their inverses. (i.e., for each $K \in \{0,1\}^k$, $P \mapsto E^s_K(P)$ is a permutation (selected randomly from all permutations) of $\{0,1\}^n$ and D^s_K is its inverse.

An adversary A is given access to oracles E^s and D^s for each s, and we bound its probability of success in terms of the number of queries made to the oracles.

The random block cipher can be modeled in the following way: for each s and each k, initially E^s_k is undefined, with domain and range empty. Whenever a query to encrypt an element x by E^s_k is made, the oracle checks to see if x is in the domain of E^s_k. It it is, it returns the element of the range associated to x. Otherwise, it returns a random element y of the complement of the range, and add x to the domain, y to the range and associates x with y. Whenever a query to decrypt an element y with D^s_k is made, the oracle checks if it is in the range, in that case it returns the x associated, otherwise it returns a random element x of the complement of the domain, adds x to the domain and y to the range and associate x with y. (this can be simplified if one assumes that the adversary never does pointless queries, i.e., if the adversary has already queried the oracle for the encryption of x (and getting y as an answer), the adversary will not query again for x, nor will it query for the decryption of y. (and similarly if the first query was actually for the decryption of y). From now on we make this assumption.

Recall that our scheme makes a hash function H^E out of a block cipher E in the manner that we described earlier. $\mathbf{Adv}(q)$ will denote the maximum possible advantage an adversary has in finding collisions if he or she does at most q queries i.e., the maximum (over all adversaries that make at most q queries to an oracle that selects E at random) of the probability that such an adversary will find $M \neq M^*$ with $H^E(M) = H^E(M^*)$.

Theorem: $\mathbf{Adv}(q) \leq \frac{q(q-1)}{2^n}$

Proof: The proof is basically the same as the one on [BRS02], adapted to our case.

By hypothesis A does (at most) q queries. If $q > 2^{n-1}$, then $\mathbf{Adv}(q) \leq 1 < \frac{q(q+1)}{2^n}$ so there is nothing to prove. So we may assume $q \leq 2^{n-1}$.

Let Q_i denote (s_i, k_i, x_i, y_i) if the ith query is a query to E^{s_i} to encrypt x_i under the key k_i, and the oracle returns y_i, or if the ith query is a query to D^{s_i} to decrypt y_i under the key k_i and the oracle returns x_i.

If A is successful it will produce messages $M \neq M^*$ with $H^E(M) \neq H^E(M^*)$.

Consider two cases:

Case 1: The messages M and M^* have different length.

In this case, since the the last block contains the length we have that $m_{t+1} \neq m^*_{t^*+1}$.

The last iteration always use the block cipher E^0. The key that is used is the xor of the last block with the zero extension of the previous hash. The length of the message is put in the last block in a place where the xor with the previous hash does not affect it, so the keys will remain different.Since $H^E(M) = H^E(M^*)$, we must have then that:

$$E^0_{m_{t+1}\oplus\hat{h}_t||m_t}(h_t) \oplus h_t = E^0_{m^*_{t^*+1}\oplus\hat{h}^*_{t^*}||m^*_{t^*}}(h^*_{t^*}) \oplus h^*_{t^*}$$

Now, at some moment A must have queried the oracle in order to get the left hand of that equation. Let Q_i be that query, i.e., $s_i = 0$,$k_i = m_{t+1} \oplus \hat{h}_t||m_t$, $x_i = h_t$ and $y_i = E^0_{k_i}(x_i)$.

Similarly at some other moment j A must have queried to get the right hand side. (they must be different since $k_i \neq k_j$). Without loss of generality we can assume that $j < i$. So, on the ith query both $s_i = 0$ and Q_j are already fixed. When making the query, A can do one of the following:

a) Give the oracle $x_i = h_t$ and the key $k_i = m_{t+1} \oplus \hat{h}_t||m_t$. The oracle will randomly return y_i from the complement of the range. For the collision to hold this random value must satisfy $y_i = x_i \oplus y_j \oplus x_j$. Note that if $x_i \oplus y_j \oplus x_j$ is already in the range so far defined, this cannot happen, but A can make sure that $x_i \oplus y_j \oplus x_j$ is not in the range by carefully selecting x_i. The oracle selects y_i at random from a set of size at least $2^n - (i-1)$, so the probability that $y_i = x_i \oplus y_j \oplus x_j$ for a **fixed** $j < i$ is at most $\frac{1}{2^n-(i-1)} < \frac{1}{2^n-q} \leq \frac{1}{2^n-2^{n-1}} = \frac{1}{2^{n-1}}$. However, any $j < i$ for which $s_j = 0$ will be a "good" j, so in the worst case we have to sum this over all $j < i$, and we get a probability bounded by $\frac{1}{2^{n-1}} \sum_{j<i} j \leq \frac{1}{2^{n-1}} \sum_{j<q} j = \frac{q(q-1)}{2^n}$ b) Give the oracle y_i and key $k_i =$

$m_{t+1} \oplus \hat{z}_{i,j}||m_t$, where $z_{i,j} = y_i \oplus y_j \oplus x_j$. The oracle will randomly return x_i from the complement of the domain. For the collision to hold this x_i must satisfy $x_i = y_i \oplus y_j \oplus x_j$, which again happens with probability bounded by $\frac{1}{2^n-(i-1)} < \frac{1}{2^{n-1}}$ for each fixed j and $\frac{q(q-1)}{2^n}$ overall.

Case 2: M **and** M^* **have the same length.** In this case $t = t^*$ and we have the following subcases: *Subcase 2a:* $h_t \neq h_t^*$ Since there is a collision, we must have then as in the previous case that

$$E^0_{m_{t+1}\oplus\hat{h}_t||m_t}(h_t) \oplus h_t = E^0_{m^*_{t^*+1}\oplus\hat{h}^*_{t^*}||m^*_{t^*}}(h^*_{t^*}) \oplus h^*_{t^*}$$

Since $h_t \neq h_t^*$, the queries for the left hand and the right hand of the equation are different queries. Using the same argument as in the previous case we again bound the probability of this case by $\frac{q(q-1)}{2^n}$.

Subcase 2b: $h_t = h_t^*$ In this case there must be a collision before the last iteration: Since we are assuming $M \neq M^*$, there must be some b with $m_b \neq m_b^*$. Then either $h_b = h_b^*$ and we have a collision here, or $h_b \neq h_b^*$, but since $h_t = h_t^*$, then there must be some $c > b$ with $h_{c-1} \neq h^*_{c-1}$ but with $h_c = h_c^*$. In any case, since $c < t$, A must find a collision of the form

$$E^{\ell_c}_{m_c||m_{c-1}}(h_{c-1}) \oplus h_{c-1} = E^{\ell_c}_{m_c^*||m^*_{c-1}}(h^*_{c-1}) \oplus h^*_{c-1}$$

Since either $m_c \neq m_c^*$ or $h_c \neq h_c^*$, the queries in order to get data for the left part and the right part of the equation must be different. Using a similar argument than in case 1 (except that here the key k_i is independent of x_i) we get again that the probability of obtaining a collision is bounded by $\frac{q(q-1)}{2^n}$ QED.

5.1.2 Preimage Resistance of the general scheme in the black-box model

Any attacker that wants now to find preimages with at most q queries has advantage bounded by $\frac{q}{2^{n-1}}$. The proof is again basically the classical one, and is similar to the previous one except that now since the attacker wants preimage rather than collision, basically the sums over j that are done in the previous argument are not there, and the bounds are simply $q/2^{n-1}$ instead of the summatory obtained there that was quadratic in q. Since it is similar to both the classical proofs and the previous, we skip the details.

5.1.3 Resistance of the general scheme against Joux's attack

In [Jou04], A. Joux introduced an attack to find multicollisions in less time than what was expected. Namely, to find two messages that collide on a hash function of length n by the birthday paradox one requires time $O(2^{\frac{n}{2}})$. To find J messages that all collide one would expect time exponential in both n and J, however if the hash function is constructed using the Merkle-Damgård scheme, Joux proved that it can be done in time exponential in n but logarithmic in J. Namely, the attack finds 2^k messages that collide in $O(k2^{\frac{n}{2}})$. Namely, if f is the compression function used in the iteration $h_i = f(h_{i-1}, m_i)$, find $m_1^0 \neq m_1^1$ with $f(h_0, m_1^0) = f(h_0, m_1^1)$, that is $h_1^0 = h_1^1$. Then find $m_2^0 \neq m_2^1$ with $f(h_1^0, m_2^0) = f(h_1^0, m_2^1)$, that is $h_2^0 = h_2^1$. Continue k times. Then all the 2^k messages $m_1^{r_1}||...||m_k^{r_k}$ where $(r_1, ..., r_k) \in \{0,1\}^k$ collide.

Our recursion is not of the form $h_i = f(h_{i-1}, m_i)$, so the attack "as is" cannot be applied and it needs to be modified.

The "obvious" modification would be, if we denote by f_i the compression function in the ith iteration, to find $m_1^0 \neq m_1^1$ such that $f_1(h_0, m_1^0, m_0) = f_1(h_0, m_1^1, m_0)$. Calling that h_1, then find a collision between the two functions $\star \mapsto f_2(h_1, \star, m_1^0)$ and $\star \mapsto f_2(h_1, \star, m_1^1)$, which is not harder than finding a collision of the same function. i.e, find $m_2^0 \neq m_2^1$ with $f_2(h_1, m_2^0, m_1^0) = f_2(h_1, m_2^1, m_1^1)$. Then, the messages $m_1^0||m_2^0$ and $m_1^1||m_2^1$ do produce a partial internal collision, i.e., $h_2^0 = h_2^1$. However it is NOT true that they also collide with the messages $m_1^0||m_2^1$ and $m_1^1||m_2^0$. So the exponential explosion of the Joux attack does not happen. For this modification of the attack to succeed, one would have to find $m_2^0 \neq m_2^1$ such that $f_2(h_1, m_2^0, m_1^0) = f_2(h_1, m_2^0, m_1^1) = f_2(h_1, m_2^1, m_1^0) = f_2(h_1, m_2^1, m_1^1)$. If f_2 is random this cannot be done in time $O(2^{\frac{n}{2}})$, so a Joux-like attack of this form that doesn't take the internals of f into consideration apparently cannot succeed. The equation $f_2(h_1, m_2^0, m_1^0) = f_2(h_1, m_2^0, m_1^1) = f_2(h_1, m_2^1, m_1^0) = f_2(h_1, m_2^1, m_1^1)$ can be modeled in the folowing way: let $random0(x) = f_2(h_1, x, m_1^0)$ and $random1(x) = f_2(h_1, x, m_1^1)$. Then one needs $x \neq y$ with $random0(x) = random0(y) = random1(x) = random1(y)$. This four-way collision is different than quadruple collision (i.e. a multicollision of cardinality 4) for $random0$ (where we would simply ask for x, y, z, w all different with $random0(x) = random0(y) = random0(z) = random0(w)$). The latter happens with high probability when taking 2^{cn} samples with c about $\frac{3}{4}$. The four way collision $random0(x) = random0(y) = random1(x) = random1(y)$ happens only with negligible probability, even when taking more than 2^n samples: when taking m samples (for each of $random0$ and $random1$) the probability of NOT finding a four way collision is

$$\prod_{i=1}^{m} \left((1 - \frac{i}{2^n}) + \frac{i}{2^n}.(1 - \frac{1}{2^n}) + \frac{i}{2^{2n}}.(1 - \frac{1}{2^n}) \right)$$

which is very near 1 even for high values ($\sim 2^n$) of m. (for example, when $n = 10$ the probability goes below 0.999000 only when $m > 1465$, which is actually higher than 2^n. With $m = 2^{1.5n}$ the probability of not finding a four way collision is still 60%) When $n = 20$ and $m = 2^{28}$ the probability of not finding a 4 way collision is 96.9233%

As an example of a test run we did using a couple of random functions, when $n = 10$, taking a sample of $m = 768 (\sim 2^{0.958n})$ one finds 268 collisions for one function, 293 for the other, 275 collisions between them, 68 triple collisions for one and 78 triple collisions for the other, and 13 quadruple collisions for one function and 22 for the other, but no four-way collisions. When taking

$m = 100000$ (this is about $2^{1.6n}$) we found TWO four-way collisions. When $m = 2^{19} (= 2^{2n-1})$ we found 119 four way collisions.

So it appears that at least this generalization of the Joux attack cannot succeed with work less than $O(2^n)$. However, another modification of the Joux attack does work: from our hash function H the attacker defines a new hash function: H^* given by $H^*(m_1, m_2, ...) = H(m_1, q, m_2, q, m_3, q, ...)$ where q is a fixed block. Any collision or multicollision of H^* is clearly also a collision or multicollision of H. And H^* can now be (almost) put in the framework of the Merkle-Damgård scheme. (the "almost" part has to do with the fact that we use the bits processed. However, this can be bypassed in the Joux attack). The difference is that now each iteration of H^* is two iterations of H. From the point of view of a black box attack, it does not make any difference between using for example $\star \mapsto f_1(h0, \star, m_0)$ or of using $\star \mapsto f_2(f_1(h0, \star, m_0), q, \star)$ and so the Joux attack, applied to H^* produces multicollisions for H. (if we abandon the black box model however, and instead of trying to find collisions by the birthday paradox the attacker uses some of the innerworkings of the function, then there is clearly a difference. In our case, the attacker will have to work with twice the block size, 32 rounds, and will only be able to control one of the input blocks).

5.1.4 Resistance of the general scheme against Kelsey-Schneier's attack

In [KS05], Kelsey and Schneier introduced a theoretical attack that shows that on hashes that use the Merkle-Damgård iteration it is possible to find a second preimage for a message of length 2^k with $k \times 2^{\frac{n}{2}+1} + 2^{n-k+1}$ work. The basic idea is, given a target message, to build expandable messages that can be linked to one of the intermediate hash values of the hash function for the target message.

As in the Joux attack, the Kelsey-Schneier attack is not directly applicable to our scheme, but again it can be modified and applied to the hash H^*. However, it seems that this would translate into an attack on the hash function H only if the target message is also of the form $(m_1, q, m_2, q, m_3, q, ...)$. Nevertheless there might be a further modification of the attack for the general case.

5.2 Differential Attacks

The attacker tries to find a collision by trying to track differentials, either forward or backward. The attacker will try to generate a non-zero input difference in the block messages that will get cancelled, leading to a collision.

The attacker might try for a one block-one iteration collision: some difference in one message block that gets cancelled within one iteration. This is however impossible in our scheme because, if two messages differ in only one block, say the ith, if the attacker finds a collision in the ith iteration, since the block will be reused in the $(i+1)$th iteration, the attacker has to get a second collision for that iteration too. Or else, he has to get a differential in the ith iteration that will get canceled in the $(i+1)$th iteration. In either way, the attacker has to work through at least two iterations. (and 32 rounds)

Even so, if the attacker does not want to have to work with more iterations, all other blocks will have to be the same. But then he cannot control several key rounds of the ith iteration and several key rounds of the $(i+1)$th iteration. If the portion of the state paired up with those portions of the key rounds does not have a difference, then the attacker is very happy, but if not, he or she cannot control it, and the difference will propagate. So the attacker has to try to carefully select the trail the differences will take.

Also, the double use of a block effectively multiplies the expansion of the block: for example, in MD4 the block was expanded by three, by simple repetitions. This proved to be inadequate. In MD5 the block is expanded by four, again with simple repetitions, and this simple extra expansion proved to be far more resistant (more or less a decade more than MD4), altough eventually broken. SHA-1 expands by 5, and in a more complex way than MD5, but still it was broken. SHA-256 expands by 4, but in a far more complex way than SHA-1. Our expansion of the 1024-bit key is also by a factor of 4, but we must take into consideration the way the cipher is going to be used in the hash function, and ask how many times a single block of 512 bits is expanded, to get an idea of the work of the attacker in trying to control differences. The first time a block is used, it will be xored with the previous block and expanded into 32 sixty-four-bit words, but 8 of those words are used twice, so this is in fact an expansion into 40 words. The block itself is used “as is” twice, so the first time it is used we have that it has been used 7 times in one form or another. The second time it is used it is again expanded 5 times, and used “as is” once, for a total of 6 times, and 13 times overall. Further, in the expansion itself it is used in a different way than the first time. Moreover, due to the expansion, any single change propagates to many different round keys.

Let’s take a closer look at the cipher: We can picture the cipher as this:

$$\begin{array}{ccccccccc} A & \star & \star & \star & \dots & \dots & \star & \star & \star \\ C & \star & \star & \star & \dots & \dots & \star & \star & \star \\ E & \star & \star & \star & \dots & \dots & \star & \star & \star \\ G & \star & \star & \star & \dots & \dots & \star & \star & \star \end{array}$$

The passage through the Sboxes produce changes (restricted to A, C, E) in a vertical, column-wise way. G suffers in the meantime horizontal changes due to the PHTX transform, as well as C post-passage through the Sboxes. The sums of G into A and E into G, although they are “ vertical”, produce horizontal changes due to the carries, so they are kind of a mixed change. In every 4-round cycle each word passes through 2 individual horizontal changes, three vertical changes and is involved in two more mixed changes. This ensures that the number of active Sboxes multiply rapidly.

As we said above, the attacker might try to leave some round keys unchanged, at least through some rounds, and try to folllow the evolution of any small change in the state through some probabilistic trail. However, after a few rounds of even a 1 bit change in the state, there are more than 40 active Sboxes per round, in fact usually there are more than 50. Since the passage of any specific non zero difference through the Sbox to reach another specific non zero difference hold with probability either zero or 2^{-2}, it follows that after just two rounds with just 32 active Sboxes in each of them (and, as said above, there are usually 50 after a couple of rounds if the key rounds are the same) the attacker will face a probability of $(2^{-2})^{64} = 2^{-128}$, and since anyway a birthday attack of complexity $O(2^{128})$ can always be mounted against a hash function of 256 bits, there is no point in trying this attack. We are not even counting the probabilities involved through passage by the sums.

So the attacker has to try to have changes in the message blocks that ensure that there are some change in the key rounds to offset the changes in the state, not necessarily every round, but he/she cannot afford to let too many rounds pass without change. However, the changes in the keys should not be too high, otherwise again the number of active Sboxes will explode beyond the 64 threshold. The key expansion, being non-linear, is difficult to analyze. However, it seems doubtful to find any input words that would produce an expanded key of hamming weight of less than 200. (Actually, we have never found any with less than 400). To these one must add the other part of the expansion of the key, plus the fact that each block is used twice, so it seems safe to say that finding a collision should be well beyond the 2^{128} work factor.

Similar considerations apply to the 512 bit version:here the threshold is now 2^{256}, but there are 128 Sboxes per round, and the expansion plus the diffusion ensure that very quickly one gets at least 80-90 active Sboxes per round.

5.3 Algebraic Attacks

Here, the attacker tries to create a system of nonlinear (usually quadratic) equations, that represent the internal operations of the hash, and then solve them by means of of some algorithm, for example a Grobner base reduction or a more specialized algorithm like the XSL algorithm. The Sboxes used in TIB3 are susceptible to this attack because they are 3 by 3 Sboxes. (The original attack against some versions of Serpent and Rijndael made use of two things: in the case of Rijndael, that the Sboxes were algebraic, and in the case of Serpent, that the 4 by 4 Sboxes were small. The Sbox of TIB3 is both small and algebraic, so they should be easy targets. This is offset by the fact that there are 64 of them in each round for a total of 1024 for the whole cipher, plus the use of the shifts that move the bits around, so the system of equations of a cipher similar to TIB3 but that uses only xors and shifts would be very complicated, similar to the system of Serpent. (Serpent has 32 rounds but only 32 Sboxes per round, the total is again 1024 Sboxes). But in TIB3 besides xors and shifts, 32-bit sums are also used. (although we described the cipher with both 32-bit sums in parallel and 64-bits sums, the 64-bits sums are only on the high bits, so they are effectively 32-bit sums). Each sum requires tens of equations of the bits to describe. In the main portion of the cipher, there are 8 sums in each round, for a total of 128 sums for the whole 16 rounds. Also, in the expansion of the key there are 3 sums in each iteration, which gives a total of 72 more sums. In the whole cipher then there are 200 sums, so there is adequate protection for the algebraic attacks.

Other types of algebraic attacks, like attacks of interpolation, also require that the degree of the polynomial that describes the function is either not very high or sparse. The polynomial that represents the Sbox is $6x^6 + 3x^3 + 5x^4 + 7x^3 + x^2 + 4x + 6$ and since there are 64 of them in each round the complete polynomial is sufficiently complicated. Plus, as before, the use of 32 bit sums complicate the picture. In general these attacks work best for functions that work essentially within one field or group. TIB3 is not one of those functions.

5.4 Fixed Points

In a usual construction $h_i = f(h_{i-1}, m_i)$ a fixed point is a pair (h, m) with $h = f(h, m)$, which allow the attacker to repeat m at will.

Because we use the bits processed and previous block, we have a recursion that can be described as $h_i = f_i(h_{i-1}, m_i, m_{i-1})$. Then even finding h, m, m^* with $h = f_i(h, m, m^*)$ does not guarantee that we can keep the fixed point one more round. First of all there is the problem that the next input would be of the form $(h, \star, m)$. But assume that $\star$ can be chosen to be m^* so as to be able to repeat the message m^*mm^*m.... Even so, now this must be also a fixedpoint for f_{i+1}. So the attacker would have to find a multiple fixed point, which seems rather improbable.

6 Performance

As can be seen below, TIB3 has excellent performance on long messages. On messages of one block length or less, it suffers from the fact that we always process at least two blocks, unlike other hashes in which if the length is, say, half a block, they pad, add the length and process just that block. Instead we pad, process the block, and then process the last block with the length.

Briefly stated, on the reference platform Core2Duo 2.4 GHz running Vista 64-bits, TIB3 hashes at about 290-300 MB/sec for the 224 and 256 versions and at about 350 MB/sec for the 384 and 512 versions for long messages but with a significant reduction for shorter messages, and running Vista 32 bits the 224 and 256 versions run at about 173 MB/sec for long messages, and the 384/512 versions at about 130 Mb/sec, again with significant reduction for short messages.

So in 64- bits either TIB3-256/224 or TIB3-512/384 have excellent performance, with TIB3-512/384 better, while on 32 bits there is a sharp drop in the performance,though still very good, but in this case the 256/224 versions are better than the 512/384.

6.1 Performance on 64-bit processors

6.1.1 Performance on the reference platform

On the reference platform Core2Duo 2.4 GHz with 4GB of RAM running Vista Ultimate 64-bit, we obtain the results shown below with the optimized version for 64 bits compiled using Visual Studio.

We did a test of the number of cycles needed to hash one message of length 256, 504, 512, 513, 1016, 1024, 5120, 10240, 102400 and 1024000 bits. We ran the test 100 times, took the minimum and a sample of 8 of the 100 results to see variability. The tables below show in the first line the length of the message, the second the minimum number of cycles per bit hashed, then the 8 samples, and then the actual number of cycles for the minimum. The last line is the speed estimate based on the 2,4GHz clock, measured in MBytes/sec. (Mega=2^{20}).

```
Cycle Test on 224 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 4.36    2.21    2.13    3.13    1.58    1.57    1.08    1.02    0.96    0.96
 4.41    2.26    2.23    3.20    1.64    1.61    1.08    1.03    2.05    0.97
 4.36    2.33    2.16    3.13    1.61    1.58    1.10    1.03    0.97    0.97
 4.36    2.26    2.23    3.23    1.64    1.62    1.08    1.03    0.97    0.97
 4.41    2.31    2.18    3.16    1.61    1.59    1.10    1.02    0.97    0.97
 4.36    2.26    2.25    3.25    1.64    1.63    1.09    1.03    0.97    0.97
 4.41    2.33    2.16    3.16    1.59    1.58    1.10    1.03    0.97    0.96
 4.36    2.26    2.23    3.23    1.64    1.63    1.09    1.03    0.97    0.96
 4.41    2.31    2.16    3.16    1.59    1.59    1.10    1.03    0.97    0.96
 1116    1116    1092    1608    1608    1608    5532   10440   98460  979368
65.63  129.21  134.14   91.27  180.77  182.19  264.79  280.62  297.55  299.14
```

```
Cycle Test on 256 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 4.41    2.24    2.18    3.16    1.58    1.56    1.09    1.03    0.97    0.96
 4.41    2.31    2.25    3.25    1.64    1.61    1.09    1.03    0.98    0.97
 4.45    2.24    2.18    3.18    1.61    1.57    1.09    1.03    0.98    0.97
 4.45    2.31    2.25    3.23    1.64    1.57    1.09    1.03    0.98    0.96
 4.41    2.24    2.18    3.16    1.58    1.57    1.09    1.03    0.98    0.96
 4.45    2.31    2.25    3.23    1.59    1.57    1.09    1.03    0.98    0.96
 4.45    2.24    2.18    3.18    1.59    1.57    1.09    1.03    0.98    1.01
 4.41    2.31    2.25    3.25    1.61    1.57    1.09    1.03    0.97    0.96
 4.45    2.24    2.18    3.18    1.59    1.57    1.09    1.03    0.98    0.96
 1128    1128    1116    1620    1608    1596    5556   10536   99552  979992
64.93  127.83  131.26   90.60  180.77  183.56  263.65  278.06  294.28  298.95
```

```
Cycle Test on 384 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 6.98    3.57    3.54    3.53    1.77    1.75    0.98    0.89    0.80    0.79
 7.03    3.62    3.66    3.58    1.80    1.79    0.99    0.89    0.80    0.79
 6.98    3.62    3.59    3.60    1.81    1.77    0.99    0.89    0.80    0.79
 6.98    3.64    3.68    3.56    1.78    1.78    0.99    0.89    0.80    0.79
 6.98    3.62    3.59    3.53    1.81    1.76    0.99    0.90    0.80    0.79
 6.98    3.57    3.63    3.56    1.80    1.79    0.99    0.89    0.80    0.79
 6.98    3.62    3.59    3.60    1.81    1.76    0.99    0.89    0.80    0.79
```

```
   6.98    3.57    3.63    3.60    1.78    1.79    0.98    0.89    0.80    0.79
   6.98    3.64    3.56    3.58    1.82    1.76    0.99    0.89    0.80    0.79
   1788    1800    1812    1812    1800    1788    5016    9072   81552  804300
  40.96   80.11   80.84   81.00  161.49  163.85  292.03  322.93  359.24  364.25

  Cycle Test on 512 version
    256     504     512     513    1016    1024    5120   10240  102400 1024000
   7.13    3.64    3.59    3.58    1.78    1.76    0.99    0.89    0.79    0.78
   7.13    3.71    3.68    3.67    1.78    1.82    0.99    0.89    0.80    0.79
   7.13    3.67    3.59    3.60    1.82    1.76    0.99    0.89    0.80    0.79
   7.13    3.74    3.68    3.67    1.80    1.80    0.99    0.89    0.80    0.79
   7.13    3.67    3.61    3.60    1.83    1.76    0.99    0.90    0.80    0.79
   7.13    3.71    3.68    3.67    1.78    1.79    0.99    0.89    0.80    0.78
   7.13    3.64    3.61    3.60    1.82    1.77    0.99    0.90    0.80    0.78
   7.13    3.71    3.66    3.67    1.80    1.79    0.99    0.89    0.80    0.80
   7.13    3.67    3.61    3.58    1.83    1.77    0.99    0.90    0.80    0.78
   1824    1836    1836    1836    1812    1800    5052    9120   81204  800964
  40.15   78.54   79.78   79.94  160.42  162.76  289.95  321.24  360.78  365.77
```

Other test measured the speed directly, hashing many messages of the required length and taking the actual time as measured by the clock counter. We obtained the following results:

TIB3-224:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
62	122	127	90	179	180	255	278	284	284	*MB/sec*

TIB3-256:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
60	122	127	81	170	180	255	279	284	284	*MB/sec*

TIB3-384:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
40	78	78	78	161	161	280	307	348	347	*MB/sec*

TIB3-512:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
39	77	77	76	155	161	280	307	349	348	*MB/sec*

The call for SHA-3 requires to measure the number of cycles to set-up the algorithm, e.g. in constructing internal tables. We do not have any such table constructions, the only thing would be the settings of the IVs. So we measured the cycles needed to do the “Init” procedure in the specifications, which sets up the initial state. Because the number of cycles is very low we did 10000 tests and saved the minimum and the median.

In the 224/256 versions the minimum is 18 cycles and the median 27 cycles.

In the 384/512 versions the minimum is 36 cycles and the median 45 cycles.

6.1.2 Performance On Linux 64-bits

On the reference Core2Duo 2.4 GHz with 4GB of RAM running Mandriva of 64 bits, compiling with gcc, similar tests give:

```
Cycle Test on 224 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 4.78    2.43    2.37    3.42    1.71    1.69    1.16    1.09    1.04    1.03
 4.83    2.43    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 4.83    2.43    2.37    3.42    1.71    1.69    1.16    1.09    1.04    1.07
 4.83    2.43    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 4.83    2.45    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 4.78    2.43    2.37    3.44    1.71    1.69    1.16    1.10    1.04    1.03
 4.78    2.43    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 4.78    2.43    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 4.83    2.43    2.37    3.42    1.71    1.69    1.16    1.10    1.04    1.03
 1224    1224    1212    1752    1740    1728    5940   11208  106080 1053072
59.84  117.81  120.86   83.77  167.06  169.54  246.61  261.39  276.18  278.20

Cycle Test on 256 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 4.78    2.40    2.37    3.42    1.72    1.71    1.16    1.09    1.03    1.03
 4.83    2.40    2.39    3.44    1.72    1.72    1.16    1.09    1.03    1.03
 4.83    2.40    2.37    3.44    1.74    1.71    1.16    1.09    1.03    1.03
 4.83    2.40    2.39    3.44    1.74    1.72    1.16    1.09    1.03    1.03
 4.83    2.40    2.39    3.42    1.72    1.72    1.16    1.09    1.03    1.03
 4.78    2.40    2.37    3.44    1.72    1.72    1.16    1.09    1.03    1.03
 4.83    2.40    2.39    3.42    1.74    1.72    1.16    1.09    1.03    1.03
 4.83    2.40    2.39    3.44    1.74    1.72    1.16    1.09    1.03    1.03
 4.83    2.40    2.37    3.44    1.72    1.71    1.16    1.09    1.03    1.03
 1224    1212    1212    1752    1752    1752    5964   11208  105720 1053036
59.84  118.97  120.86   83.77  165.91  167.22  245.61  261.39  277.12  278.21

Cycle Test on 384 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 8.25    4.21    4.12    4.14    2.07    2.04    1.15    1.04    0.92    0.91
 8.30    4.21    4.15    4.14    2.08    2.05    1.15    1.04    0.92    0.92
 8.30    4.21    4.15    4.14    2.08    2.04    1.15    1.04    0.92    0.92
 8.30    4.21    4.15    4.14    2.08    2.04    1.15    1.04    0.92    0.92
 8.30    4.21    4.15    4.14    2.08    2.04    1.15    1.04    0.92    0.92
 8.30    4.21    4.15    4.14    2.08    2.05    1.15    1.04    0.92    0.92
 8.34    4.21    4.15    4.14    2.08    2.05    1.15    1.04    0.92    0.92
 8.30    4.21    4.15    4.14    2.08    2.05    1.15    1.04    0.92    0.93
 8.34    4.21    4.15    4.14    2.07    2.04    1.15    1.04    0.92    0.93
 2112    2124    2112    2124    2100    2088    5880   10620   94392  936840
34.68   67.89   69.36   69.10  138.42  140.31  249.12  275.87  310.37  312.72
```

```
Cycle Test on 512 version
   256      504      512      513     1016     1024     5120    10240   102400 1024000
  8.39     4.21     4.22     4.21     2.10     2.06     1.16     1.04     0.94     0.92
  8.44     4.24     4.22     4.21     2.11     2.07     1.16     1.04     0.94     0.93
  8.39     4.21     4.22     4.21     2.11     2.07     1.16     1.04     0.94     0.93
  8.39     4.24     4.22     4.21     2.13     2.07     1.16     1.04     0.94     0.93
  8.39     4.21     4.24     4.21     2.11     2.06     1.16     1.04     0.94     0.92
  8.39     4.24     4.22     4.21     2.10     2.07     1.16     1.04     0.94     0.92
  8.39     4.21     4.22     4.21     2.11     2.07     1.16     1.04     0.94     0.92
  8.39     4.24     4.22     4.21     2.10     2.07     1.16     1.04     0.94     0.92
  8.39     4.21     4.24     4.21     2.11     2.07     1.16     1.04     0.94     0.92
  2148     2124     2160     2160     2136     2112     5916    10656    95844  937092
 34.10    67.89    67.82    67.95   136.09   138.72   247.61   274.93   305.67  312.64
```

TIB3-224:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
54	108	111	78	154	160	240	252	271	271	MB/sec

TIB3-256:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
54	108	111	78	154	160	240	252	271	271	MB/sec

TIB3-384:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
33	64	65	65	129	132	240	281	305	305	MB/sec

TIB3-512:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
32	63	63	64	126	129	240	266	305	305	MB/sec

6.2 Performance on 32-bit processors

The algorithm was build in such a way to be adaptable to both 32 and 64 bit processors. The sums are, except for some in the 384/512 versions, actually all 32-bit sums, and the shift are actually shifts of 32-bit words.

On the reference platform Core2Duo 2.4 GHz with 2GB of RAM running Vista 32-bit, we obtain the following results with the optimized version for 32 bits compiled using Visual Studio.

As before, the first line indicate the number of bits of the message, the second the minimum number of cycles per bit over all tests, then a sample to show regularity, the next to last line is the actual number of cycles and the last line is estimated speed based on the speed of the processor.

```
Cycle Test on 224 version
   256      504      512      513     1016     1024     5120    10240   102400 1024000
```

```
 7.59    3.86    3.75    5.40    2.73    2.68    1.83    1.73    1.63    1.62
 7.78    3.90    3.82    5.47    2.76    2.68    1.84    1.74    1.63    1.62
 7.64    3.88    3.77    5.47    2.76    2.70    1.84    1.74    1.63    1.62
 7.83    3.93    3.75    5.47    2.76    2.70    1.83    1.74    1.63    1.62
 7.64    3.90    3.77    5.47    2.74    2.68    1.84    1.74    1.63    1.62
 7.59    4.10    3.75    5.47    2.75    2.70    1.84    1.74    1.63    1.62
 7.83    3.88    3.77    5.47    2.76    2.73    1.84    1.74    1.63    1.62
 7.64    3.90    3.75    5.47    2.75    2.73    1.84    1.74    1.63    1.62
 7.78    3.90    3.77    5.47    2.74    2.70    1.84    1.74    1.63    1.62
 1944    1944    1920    2772    2772    2748    9372   17712  166620 1663524
37.68   74.17   76.29   52.95  104.86  106.61  156.30  165.41  175.83  176.11


Cycle Test on 256 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
 7.78    3.83    3.80    5.45    2.73    2.68    1.83    1.72    1.63    1.62
 7.83    3.93    3.82    5.47    2.75    2.72    1.84    1.73    1.63    1.62
 7.83    3.93    3.82    5.47    2.83    2.84    1.84    1.73    1.63    1.62
 7.83    3.83    3.82    5.47    2.75    2.71    1.84    1.73    1.63    1.62
 7.83    3.90    3.82    5.47    2.75    2.71    1.84    1.73    1.63    1.62
 7.83    3.90    3.82    5.47    2.75    2.70    1.84    1.73    1.63    1.62
 7.83    4.14    3.82    5.47    2.74    2.71    1.84    1.73    1.63    1.62
 7.83    3.93    3.82    5.47    2.75    2.77    1.84    1.73    1.63    1.66
 7.83    3.93    3.82    5.47    2.75    2.71    1.84    1.73    1.63    1.62
 1992    1932    1944    2796    2772    2748    9360   17640  166608 1662948
36.77   74.64   75.35   52.49  104.86  106.61  156.50  166.08  175.84  176.17


Cycle Test on 384 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
18.89    9.57    9.42    9.40    4.71    4.66    2.69    2.44    2.21    2.21
18.94    9.60    9.45    9.43    4.72    4.68    2.69    2.44    2.21    2.39
18.89    9.57    9.42    9.43    4.71    4.68    2.69    2.44    2.21    2.21
18.89    9.57    9.45    9.43    4.71    4.68    2.69    2.44    2.21    2.21
18.89    9.57    9.42    9.43    4.81    4.69    2.69    2.44    2.21    2.25
18.89    9.60    9.45    9.43    4.71    4.68    2.69    2.44    2.21    2.21
18.94    9.57    9.42    9.43    4.72    4.68    2.69    2.44    2.21    2.21
18.89    9.57    9.45    9.43    4.71    4.69    2.69    2.44    2.21    2.27
18.89    9.60    9.42    9.43    4.71    4.69    2.69    2.44    2.21    2.21
 4836    4824    4824    4824    4788    4776   13752   24960  226764 2262060
15.15   29.89   30.37   30.43   60.71   61.34  106.52  117.38  129.20  129.51


Cycle Test on 512 version
  256     504     512     513    1016    1024    5120   10240  102400 1024000
19.03    9.64    9.49    9.47    4.74    4.73    2.69    2.44    2.22    2.21
```

```
19.03    9.64    9.52    9.50    4.84    4.73    2.70    2.44    2.22    2.21
19.08    9.64    9.52    9.66    4.75    4.75    2.70    2.44    2.22    2.21
19.08    9.64    9.52    9.47    4.76    4.75    2.70    2.44    2.22    2.21
19.08    9.64    9.52    9.47    4.75    4.73    2.70    2.44    2.22    2.21
19.08    9.64    9.52    9.50    4.76    4.73    2.71    2.44    2.22    2.21
19.08    9.64    9.52    9.50    4.76    4.73    2.69    2.44    2.22    2.21
19.08    9.64    9.52    9.50    4.75    4.73    2.70    2.44    2.22    2.21
19.03    9.64    9.52    9.50    4.75    4.73    2.70    2.44    2.22    2.21
 4872    4860    4860    4860    4812    4848   13788   25008  226848 2262372
15.03   29.67   30.14   30.20   60.41   60.43  106.24  117.15  129.15  129.50
```

Running actual tests for speed we obtain:
TIB3-224:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
36	73	74	52	102	105	153	162	173	173	MB/sec

TIB3-256:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
36	73	74	51	102	105	153	161	173	173	MB/sec

TIB3-384:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
15	29	30	30	60	61	106	117	130	130	MB/sec

TIB3-512:

256	504	512	513	1016	1024	5120	10240	102400	1024000	*bits*
15	30	30	29	59	60	106	118	125	130	MB/sec

The call for SHA-3 requires to measure the number of cycles to set-up the algorithm, e.g. in constructing internal tables. We do not have any such table constructions, the only thing would be the settings of the IVs. So we measured the cycles needed to do the "Init" procedure in the specifications, which sets up the initial state. Because the number of cycles is very low we did 10000 tests and saved the minimum and the median.

In the 224/256 versions the minimum is 99 cycles and the median 108 cycles.

In the 384/512 versions the minimum is 117 cycles and the median 126 cycles.

6.2.1 32-bit restricted environment

The Optimized 32 bit version was compiled for Windows Mobile 5.0 for ARMV4i and tested on an iPaq with processor ARM92OT S3C2442A with 53.88 MB RAM obtaining 3.17 Mb/sec for the 256 bit version.

6.3 Performance on 8-bit processors

As we explained above, TIB3-256 can be easily thought in terms of 32-bit registers, so we model this operations now on 8-bit registers. Since TIB3 uses only logical operations, sums and shifts, it should be easily implemented on 8-bit processors.

6.3.1 Data Memory requirements

The data memory requirements are: 64 bytes for the current block, 64 bytes for the previous block, 32 bytes for the hash, 32 bytes to hold the hash state at the start of the encryption in order to do the Davies-Meyer update, 80 bytes for the recursion of the expansion and 8 bytes to keep the bits processed, so a total of 280 bytes.

6.3.2 General Considerations

We consider each part of the encryption step separately.

The Expansion of the key:

Here we have the ψ function. When viewed as operating on 32-bit word, ψ uses six 32-bit xors, three 32-bit sums and two shifts, one of 11 bits, the other of 7.

ψ is used 24 times in the expansion, to which we have to add 8 more xors in calculating the input to the expansion, plus 20 more xors when calculating the inputs to the first two uses of ψ. So the whole expansion uses 72 sums, 100 xors and 48 shifts.

Rounds of Encryption

Each round of encryption has 8 xors, 4 sums, two PHTXs and a passage through the Sboxes. Each PHTX uses 2 sums, 2 xors and 2 shifts. So we have a total for the round of 10 xors, 8 sums and 2 shifts plus the Sbox.

As we said earlier the Sbox can be modeled bitsliced with the boolean functions:

$$\begin{aligned} f(x,y,z) &= \overline{x} \oplus (y \wedge \overline{z}) \\ g(x,y,z) &= z \oplus (\overline{x} \wedge \overline{y}) \\ h(x,y,z) &= y \oplus (x \wedge z) \end{aligned}$$

That would be 3 xors, 3 ands and 4 negations plus two copies into temporary registers. In fact it can be modeled taking temporary registers $t1$ and $t2$ as:

$$\begin{aligned} t1 &= \overline{z} \\ t2 &= y \\ z &= x \\ y &= x \vee y \end{aligned}$$

$$\begin{aligned} x &= \overline{x} \\ z &= z \oplus t2 \\ t2 &= t2 \wedge t1 \\ y &= y \oplus t1 \\ x &= x \oplus t2 \end{aligned}$$

so a total of 9 operations which have to be repeated on 32-bit registers, thus we have 18 logical operations.

Thus each round has 8 sums, 28 logical operations, and 2 shifts. There are 16 rounds, so the encryption proper after the expansion of the key takes 128 sums, 448 logical operations and 32 shifts.

The total for the encryption (rounds+expansion) is 200 sums, 548 logical operations and 80 shifts. We may be undercounting some moves between registers.

After the encryption we have 8 xors for the Davies-Meyer update, and the update of the bits processed, which is the sum of 512 (or less) to a 64-bit register, so this is 2 more sums.

To this we have to add the initialization, which is just the copy of 768 bits=24 32-bit registers plus the last iteration. The last iteration is like the others, but there is an extra xor of the hash and the block, this affects only 256 bits

So, to hash a message of length t blocks we need approximately $202t + 200$ sums, $556t + 558$ logical operations and $80t + 80$ shifts.

6.3.3 Efficiency on an ideal model

If we assume a processor model with several 8-bit registers (for example four) in which the logical and arithmetic operations (and,or, xors, sums) take one cycle on each register, and such that the sum on 32-bit words then takes 8 cycles to implement, shifts take 12 cycles and other logical operations take 4 cycles, then the total number of cycles for a message of length t blocks would be $4800t+4792$ cycles (on code that should be optimized for 8-bit processor)

If instead we have also 8 cycles per logical operation then the total is $7024t+7024$ cycles. (these estimates and the general model are similar to the ones the MARS team did in their presentation. See [BCD$^+$])

Assuming 20MHz, for long messages where we can disregard the overhead of 4792 (7024) cycles, we are hashing about 4000 blocks (2800) of 512 bits per second, about 2Mbit/sec. (1.4Mbit/sec), if we assume one instruction cycle=one clock cycle like MARS did. In practice however it could be one instruction cycle=4 clock cycles, so the speed would be in the order of 350Kbit/sec. If instead of hashing a long message one has to hash many one-block messages, then each such message has the overhead of the extra encryption, so the speed is halved.

However, in many cases one does not use many 8-bit registers but only one. In that case the cost of moving data in and out of the register is a great factor.

For example, the cost of doing one xor of two 8-bit quantities is at least 3 cycles: one cycle to move one of the quantities into the register, another to xor the other, then a third cycle to move the content of the register into another location. On the other hand if one makes several xors like $a = b \oplus c \oplus d \oplus e$, we still have only two moves and three xors.

With only one register the cost of doing the same operation on a 32-bit word grows fast, because one need to constantly move in and out of the register 8-bit quantities. In particular a xor of two 32-bit words cannot be made in parallel. We estimate that sums and xors will take 16 cycles, a shift of 1 five cycles, and others correspondingly higher.

So in such a model we expect the estimate above to be much higher so we did a simulated implementation

6.3.4 Simulated implementation

We implemented the code on a simulated 8-bit micro-controller.

We simulated MicroChip PIC18F452 running at 40 Mhz. We used:

Developing environment MPLAB IDE v8.10,

Compiled with compiler MPLAB C18

In the simulation of that micro-controller each assignment takes between 2 or 3 instruction cycles, logical operations and sums of 8-bit registers take 7 cycles and shifts of 8-bit registers take 5 cycles, instead of the 1 cycle we assumed on the general model.

Logical operations or the sum on 32-bit registers take both 36 cycles. (about four times the cost of the corresponding operation on 8-bit).

Shifts are very dependent on the amount shifted. The shifts used in our algorithm are:

$\ll 11$: 114 cycles.

$\ll 15$: 146 cycles.

$\gg 11$: 113 cycles.

$\gg 7$: 80 cycles.

(with better optimization this numbers can be wrought down onto the 40-50 range). This numbers came from the fact that we used a C code in terms of 32-bit registers to implement it. The simulator optimized area, not speed. If we were to hand code it, then a shift by 11 can be done in about 10-20 instructions, but the simulator did not do it that way.

Assuming an average of 120 cycles for each shift, plus the 36 cycles per logical operation or sum, we would expect about 36888 cycles per encryption plus several more cycles due to temporary assignments, on an optimized code. Since the code is not optimized for speed we expected more cycles, and effectively the simulation obtained 45468 instruction cycles per encryption. In this processor (or at least in the simulator) each instruction cycle takes 4 clock cycles, that is the 40Mhz speed translates into 10 million instruction cycles per second. Effectively the time the simulator gave was 4.5468 mSecs per encryption, roughly 113Kbit/sec.

TIB3-384 and TIB3-512 are basically doubled versions of TIB3-256. The only difference is that TIB3-512 uses also some 64-bit sums, so the numbers would be somewhat larger than twice the numbers for TIB3-256.

6.4 Hardware Implementation

We made a preliminary implementation on a FPGA Xilinx Virtex4 xc4vsx35ff668-12 t requiring 6062 slices from a total of 15360 (39%) and 11141 LUTs from a total of 30720 (36%) with a maximum propagation path of 70.797ns (14,12 MHz).

```
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 176
 32-bit adder                                          : 32
 64-bit adder                                          : 144
# Xors                                                 : 2744
 1-bit xor2                                            : 416
 1-bit xor3                                            : 1440
 1-bit xor4                                            : 768
 512-bit xor2                                          : 1
 64-bit xor2                                           : 118
 64-bit xor3                                           : 1
```

We thank Jorge Naguil for translating the algorithm to VHDL.

References

[BCD$^+$] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic. Mars — a candidate cipher for AES.

[BRS02] 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.

[FP05] Sebastián Fontana and Daniel Penazzi. A study of 3 by 3 s-boxes and its application on a bitsliced multiplicative cipher: Quetzalcoatl. Actas del tercer Congreso Iberoamericano de Seguridad Informática, (CIBSI05), Valparaiso, Chile., 2005.

[Jou04] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[KS05] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^{n} work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[Nyb94] Kaisa Nyberg. Differentially uniform mappings for cryptography. In *Advances in Cryptology - EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[PGV94] Bart Preneel, Ren Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: a synthetic approach. In *Advances in Cryptology - CRYPTO 93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.