A Secure Hash Algorithm with only 8 Folded SHA-1 Steps

Danilo Gligoroski†, Smile Markovski†† and Svein J. Knapskog†

† "Centre for Quantifiable Quality of Service in Communication Systems", Norwegian University of Science and Technology, Trondheim, Norway
†† Faculty of Sciences, Institute of Informatics, "Ss Cyril and Methodius" University, Skopje, Republic of Macedonia [1]

Summary
We propose a new design principle for construction of iterated cryptographic hash functions: *computations in the iterative part of the compression function to start with variables produced in the message expansion part that have complexity level of a random Boolean function*. Then we show that to reach the cryptographic strength that will withstand all currently known techniques for finding collisions, much lower number of iterations is necessary. Concretely we use the recently proposed nonlinear technique "Quasigroup Fold" together with the mentioned principle to design a hash function that has only 8 iterative steps. Besides increasing the security, the reference C code for the obtained hash function shows that it is at least 3% faster than original reference code for SHA-1.

Key words: *SHA-1,SHA-2, hash, quasigroup folding*

Introduction

Since the successful attacks on MDx family of functions were announced by Wang at. al [1-4] a significant part of the cryptographic community devoted its current scientific interest in revising the principles of constructing cryptographically secure hash functions. Thus, the Merkle-Damgård design of hash functions [5,6] came under careful revision, and some weaknesses have been already found. Besides the classical birthday attack (and some variants such as Yuval's attack [7]), in a recent paper from 2004 by Joux [8] the concepts of multi-collisions and expandable messages were introduced and it was shown that the workload for finding second preimage collisions with equal length for iterated one-way hash functions is about $\log(k)\times 2^{n/2}$, where k is the number of computed hash values. Kelsy and Schneier in 2005 [9] extended the approach for finding expandable messages with different length, and they showed that the workload is about $k\times 2^{n/2+1}$. Then Coron at al. [10] made several suggestions how to strengthen the Merkle-Damgård design. On the other hand, in recent ePrint paper, Gauravaram, Millan and Neito [11] give an interesting discussion about the possibilities that Merkle-Damgård design for MDx family was in fact not properly implemented. That is especially true for the so called pseudo-collisions attack, for which MD5, SHA-0 and SHA-1 were not designed to be secure. These thoughts for the design criteria are also present in Preneel works [12, 13].

Previous Work – Having a completely linear message expansion part, SHA-1 reaches the level of complexity of a random nonlinear multivariate Boolean function over the field $GF(2)$ in about 20 steps in the iterative part of its compression function. Basically, all known methods for finding collisions on SHA-0 and SHA-1 exploit heavily that weak part of the design. Another characteristic that all known methods for finding collisions on MDx family of hash functions have in common is that they use the invertibility and the linearity of addition modulo 2^{32} and left rotation, that are frequently used in the iterative part of the compression function. Recently Gligoroski, Markovski and Knapskog, presented at NIST Cryptographic Workshop [14] a technique called "Quasigroup Folding" that eliminates that linear characteristic of SHA-1 and, tracing the computations of the compression function, after several steps becomes infeasible. However, the proposed solution was around 85% slower than the original SHA-1.

Facing the recent successful cryptanalysis of SHA-1, NIST has updated their information and recommendations for the policy of using cryptographic hash functions. Their recommendations now include the use of SHA-2 family, and they propose smooth transfer from SHA-1 to SHA-2. However, having in mind that SHA-2 is two times slower than SHA-1, for many industrial applications, SHA-1 will continue its life. From that point of view, there is still big industrial interest for fast cryptographic hash functions (similar or even faster than SHA-1, and with digest sizes of 160 bits).

Our Work – In this paper we define a measure of the similarity of an iterated hash functions to the random function. We examine how SHA-1 and SHA-2 behave according to that measure and propose a new design

[1]This work was carried out during the visit of prof. Smile Markovski to the Faculty of information technology, mathematics and electrical engineering - Department of Telematics, The Norwegian University of Science and Technology - Trondheim, Norway, in the framework of Erasmus Mundus program and NordSecMob Master's Programme in Security and Mobile Computing.

Manuscript received October 5, 2006.
Manuscript revised October 25, 2006.

principle using that measure. Further on, following that principle and using the technique of Quasigroup Folding, we define a new hash function with 160 bits message digest size. It has only 16 steps in the message expansion part and has only 8 internal iterative steps. We claim that the new hash function SHA-1Q2 is cryptographically secure and can withstand all known successful attacks on MDx family of hash functions. For comparison, Wang at al. claim that they can find collisions in 58 steps of original SHA-1 with complexity of 2^{33}. Moreover, SHA-1Q2 is faster than the original SHA-1 for at least 3.0%.

The paper is organized as follows. Some preliminaries about Quasigroup Folding and SHA-1 are given in Section 2. The algorithm SHA-1Q2 is given in Section 3, security analysis is given in Section 4 and we close our paper with conclusions in Section 5.

2. Preliminaries and notation

In this paper we will use the same notation as that of NIST: FIPS 180-2 description of SHA-1 [15].
The following operations are applied to 32-bit words in SHA-1Q2:

1. Bitwise logical word operations: $\wedge$, $\vee$, $\oplus$ and $\neg$.
2. Addition modulo 2^{32}.
3. The rotate left (circular left shift) operation, $ROTL^n(x)$, where x is a 32-bit word and n is an integer with $0 \le n < 32$.
4. The operation of quasigroup folding of a 32-bit word $x = x_1x_2x_3x_4x_5x_6x_7x_8$ (represented as a concatenation of eight 4-bit variables $x_1,\ldots,x_8$) is defined by the following equations:

$$\begin{aligned} QFOLD(x) &= x_1' \, x_2' \, x_3' \, x_4' x_5 x_6 \, x_7 \, x_8, \\ x_1' &= x_1 * x_5, \\ x_2' &= x_6 * x_2, \\ x_3' &= x_3 * x_7, \\ x_4' &= x_8 * x_4, \end{aligned} \qquad (1)$$

where the operation $*$ is a 16×16 quasigroup operation defined as in Table 1.

SHA-1Q2 uses a sequence of 8 logical functions, f_0, f_1,..., f_7. Each function f_t, where $0 \le t \le 7$, operates on three 32-bit words, x, y, and z, and produces a 32-bit word as output. The function $f_t(x,y,z)$ is defined as follows:

$f_t(x,y,z) =$

$$\begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z), & 0 \le t \le 1 \\ Parity(x,y,z) = x \oplus y \oplus z, & 2 \le t \le 3 \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z), & 4 \le t \le 5 \\ Parity(x,y,z) = x \oplus y \oplus z, & 6 \le t \le 7 \end{cases}$$

SHA-1Q2 does not use any sequence of predefined constant 32-bit words. Preprocessing in SHA-1Q2 is exactly the same as that of SHA-1. That means that these three steps: padding the message M, parsing the padded message into message blocks, and setting the initial hash value, $H^{(0)}$ are the same as in SHA-1. In the parsing step the message is parsed into N blocks of 512 bits, and the i-th block of 512 bits is a concatenation of sixteen 32-bit words denoted as $M_0^{(i)}, M_1^{(i)}, \ldots, M_{15}^{(i)}$.

The initial hash value, $H^{(0)}$ for SHA-1Q2 is the same as that of SHA-1 and consist of the following five 32-bit words:

$$\begin{aligned} &H_0^{(0)} = 67452301,\ H_1^{(0)} = efcdab89, \\ &H_2^{(0)} = 98badcfe,\ H_3^{(0)} = 10325476, \\ &H_4^{(0)} = c3d2e1f0. \end{aligned} \qquad (2)$$

*	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	a	4	5	9	6	0	e	1	2	c	d	f	3	8	b	7
1	5	b	c	8	4	e	0	7	3	2	f	a	1	9	d	6
2	c	5	2	d	f	8	a	e	1	3	6	7	b	0	9	4
3	7	d	3	e	2	1	b	c	5	9	4	8	0	f	6	a
4	1	2	4	a	b	7	8	9	0	d	3	e	6	c	5	f
5	4	a	8	b	d	2	c	6	e	f	5	9	7	3	1	0
6	0	e	d	2	8	3	6	5	c	b	7	4	9	a	f	1
7	b	6	0	5	9	d	4	8	7	a	2	3	f	1	e	c
8	d	8	6	1	c	a	f	0	b	5	9	2	4	7	3	e
9	2	f	1	0	7	c	5	b	9	6	8	d	a	e	4	3
a	6	c	b	7	a	f	1	3	4	8	e	0	d	5	2	9
b	8	1	f	6	3	9	7	4	a	e	c	5	2	d	0	b
c	f	3	9	4	e	6	2	d	8	7	0	1	c	b	a	5
d	e	9	7	3	1	b	d	f	6	0	a	c	5	4	8	2
e	3	0	e	c	5	4	9	a	f	1	b	6	8	2	7	d
f	9	7	a	f	0	5	3	2	d	4	1	b	e	6	c	8

Table 1: A quasigroup (Q, *) of order 16

3. SHA-1Q2 Hash Computation

The SHA-1Q2 hash computation uses functions and constants defined in Section 2. Addition (+) is performed modulo 2^{32}. After preprocessing is completed, each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed in order, using the steps described algorithmically in the Table 2.

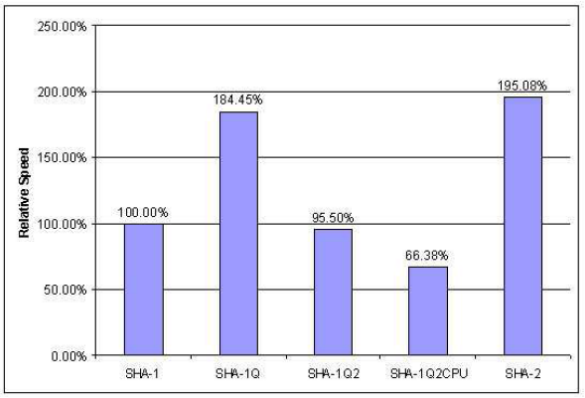

Fig. 1 Speed comparison (higher bars mean slower algorithm) between different hash functions - relative to the speed of SHA-1.

In Figure 1 we give a speed comparison between SHA-1, SHA-1Q, SHA-1Q2 and SHA-2. To obtain the values for SHA-2 we have taken the relative speed comparison between SHA-1 and SHA-2 in Crypto++ library [16]. The label SHA-1Q2CPU is estimation what would be the speed of SHA-1Q2 if the operation of Quasigroup Folding is implemented in the microcode of the CPU as an assembler instruction. Logical bitwise operations NOT, XOR, AND and OR are normally implemented in every modern CPU as a single assembler instruction and the same can be done for the operation of Quasigroup Folding in future versions of the modern CPUs. The operation Quasigroup Folding is univariate, and can be implemented in a parallel fashion. Its execution will take only one or two CPU cycles.

For $i = 1$ to N:
{
1. Message expansion part for obtaining 512 working bits from 512 message bits:

$$W_t = \begin{cases} M_i^{(t)}, & 0 \le t \le 15 \\ \text{QFOLD}(ROTL^7(W_{t-1} \oplus W_{t-3} + W_{t-6} \oplus W_{t-8} + W_{t-12} \oplus W_{t-14} + W_{t-13} \oplus W_{t-16} + W_{t-2} \oplus W_{t-7} + W_{t-4} \oplus W_{t-10} + W_{t-5} \oplus W_{t-9} + W_{t-11} \oplus W_{t-15})), & 16 \le t \le 31 \end{cases}$$

2. Initialize the five working variables a, b, c, d and e, with the $(i\text{-}1)^{\text{th}}$ hash value and the values of W_{31}, W_{30}, W_{29}, W_{28}, W_{27}:
$a = H_0^{(i-1)} + W_{31}$
$b = H_1^{(i-1)} + W_{30}$
$c = H_2^{(i-1)} + W_{29}$
$d = H_3^{(i-1)} + W_{28}$
$e = H_4^{(i-1)} + W_{27}$
3. For t=0 to 7
{
$T = \text{QFOLD}(ROTL^5(a) + f_t(b,c,d) + e + (W_t \oplus W_{t+16}) + (W_{t+8} \oplus W_{t+22}))$
$e = d$
$d = c$
$c = ROTL^{30}(b)$
$b = a$
$a = T$
}
4. Compute the i^{th} intermediate hash value $H^{(i)}$:
$H_0^{(i)} = a + H_0^{(i-1)}$
$H_1^{(i)} = b + H_1^{(i-1)}$
$H_2^{(i)} = c + H_2^{(i-1)}$
$H_3^{(i)} = d + H_3^{(i-1)}$
$H_4^{(i)} = e + H_4^{(i-1)}$
}

Table 2: Algorithmic description of SHA-1Q2 hash function.

At the end of this section we give SHA-1Q2 test hash values for the following messages:

1. `abc',
   Hash: D3 17 3E B6 8E E4 3C 10 D8 B6 BB A3 53 AC BB 5A 35 EF 33 30
2. `abcdbcdecdefdefgefghfghighijhijkijkljklmklmnl mnomnopnopq',
   Hash: 3B 2B 12 77 42 B9 67 A9 F0 F6 BD 0F D6 87 7A C8 DB 0B A0 F8
3. 1,000,000 `a',
   Hash: 8E 36 55 F8 A9 7E 3B 12 58 38 D5 32 FD 6A DF 07 E1 FB E2 E3
4. 10 `01234567012345670123456701234567 01234567012345670123456701234567',
   Hash: D4 EC F9 82 33 77 9E E5 71 AE B9 61 19 8A C0 14 27 5E C6 C2

The complete source code for SHA-1Q2 is given in the Appendix.

4. Security of SHA-1Q2

In this section we will make an initial analysis of how strongly collision resistant, preimage resistant and second preimage resistant SHA-1Q2 is. In the first subsection we will analyze the properties of the message expansion part and in the rest of the chapter we will discuss the strength of the function against finding different types of collisions. The chosen methodology for that part will be to define a measure of complexity of its compression function $F:\{0,1\}^{512} \rightarrow \{0,1\}^{160}$ expressed as a normalized average number $\overline{L_F}$ of terms (monomials) in the Algebraic

Normal Form (ANF) of that function and to examine a reduced version of that function.

4.1 Properties of message expansion part

It is relatively easy to prove the following Theorem:

Theorem 1. *The message expansion part of the SHA-1Q2 is a bijection* $\xi : \{0,1\}^{512} \rightarrow \{0,1\}^{512}$.

Proof. It is enough to show that the message expansion part is surjection, i.e. for every 16-tuple $W=(W_{16},W_{17}, \ldots,W_{31})$ there exist a 16-tuple preimage $M=(M_0,M_1,\ldots,M_{15})$ such that $\xi(M)=W$.

First we should note that the quasigroup folding operation $QFOLD:\{0,1\}^{32} \rightarrow \{0,1\}^{32}$ is a bijection. That means that from the recurrent equation that describes the message expansion part for a given 16-tuple $W=(W_{16},W_{17},\ldots,W_{31})$ we have the relation:

$W_{31}=QFOLD(ROTL^7(W_{30} \oplus W_{28} + W_{25} \oplus W_{23} + W_{19} \oplus W_{17} + W_{18} \oplus W_{15} + W_{29} \oplus W_{24} + W_{27} \oplus W_{21} + W_{26} \oplus W_{22} + W_{20} \oplus W_{16}))$.

From there it is straightforward to compute the unique value for W_{15} as:

$W_{15} = (QFOLD^{-1}(ROTR^7(W_{31})) - W_{30} \oplus W_{28} - W_{25} \oplus W_{23} - W_{19} \oplus W_{17} - W_{29} \oplus W_{24} - W_{27} \oplus W_{21} - W_{26} \oplus W_{22} - W_{20} \oplus W_{16}) \oplus W_{18}$.

Now, having the new 16-tuple $W =(W_{15}, W_{16}, \ldots, W_{30})$ we can proceed further to compute the unique value for W_{14}, and so on until we compute the unique value for W_0.□

4.2 Properties of iterative part

The iterative part of SHA-1Q2 introduces a new design principle: *For the initialization of the working variables (that will give the final output) the last values produced in the message expansion part are combined together with some predefined initial values.*

Namely, instead of the original design proposal in SHA-1 where the initialization of the five working variables *a*, *b*, *c*, *d*, and *e* is done with the $(i\text{-}1)^{th}$ hash value (and where $H_0^{(0)}$, $H_1^{(0)}$, $H_2^{(0)}$, $H_3^{(0)}$, $H_4^{(0)}$ are fixed and predetermined) we initialize the five working variables both with the values of $(i\text{-}1)^{th}$ hash value and with the values of W_{31}, W_{30}, W_{29}, W_{28}, W_{27}. According to the analysis about the algebraic complexity of the hash functions in their iterative part, that is given in the following subsection, we argue that this design principle enables us both to reduce the number of iteration steps in the hash function, and to thwart any of the current successful attacks on MDx family of hash functions, since the iterative part starts immediately with algebraic complexity of a random Boolean function.

SHA-1Q2 in its iterative part uses quasigroup folding similarly, but slightly different than SHA-1Q. Namely, instead of the assignment $T = \text{QFOLD}(ROTL^5(a) + f_t(b,c,d) + e + W_t)$ in SHA- 1Q2, in order to achieve mixing of all 32 variables W_i in just 8 steps we use the assignment $T = \text{QFOLD}(ROTL^5(a)+f_t(b,c,d)+e+(W_t \oplus W_{t+16})+(W_{t+8} \oplus W_{t+22}))$. In such a way, all 32 variables W_i, $0 \le i \le 31$, are used in the iterative part after 8 steps. In the Table 3 we give the order of how the iterative part of the SHA-1Q2 combines the variables W_i, $0 \le i \le 31$.

Step	Used W_i
Initialisation	W_{31}, W_{30}, W_{29}, W_{28}, W_{27}
t = 0	W_0, W_8, W_{16}, W_{22}
t = 1	W_1, W_9, W_{17}, W_{23}
t = 2	W_2, W_{10}, W_{18}, W_{24}
t = 3	W_3, W_{11}, W_{19}, W_{25}
t = 4	W_4, W_{12}, W_{20}, W_{26}
t = 5	W_5, W_{13}, W_{21}, W_{27}
t = 6	W_6, W_{14}, W_{22}, W_{28}
t = 7	W_7, W_{15}, W_{23}, W_{29}

Table 3: The order of using the working variables W_i, $0 \le i \le 31$ in the compression function of SHA-1Q2.

4.3 Normalized Average Number of Terms – NANT

Let $1 \le r \le n$ be integers and let $F:\{0,1\}^n \rightarrow \{0,1\}^r$ be a vector valued Boolean function. The vector valued function F can be represented as an r-tuple of Boolean functions $F = (F^{(1)}, F^{(2)}, \ldots, F^{(r)})$, where $F^{(s)}:\{0,1\}^n \rightarrow \{0,1\}$ ($s = 1, 2, \ldots, r$), and the value of $F^{(s)}(x_1, \ldots, x_n)$ equals the value of the s-th component of $F(x_1, \ldots, x_n)$. The Boolean functions $F^{(s)}(x_1, \ldots, x_n)$ can be expressed in the Algebraic Normal Form (ANF) as polynomials with n variables $x_1, \ldots, x_n$ of kind $a_0 \oplus a_1x_1 \oplus \ldots \oplus a_nx_n \oplus a_{1,2}x_1x_2 \oplus \ldots \oplus a_{n-1,n}x_{n-1,n} \oplus \ldots \oplus a_{1,2,\ldots,n}x_1x_2\ldots x_n$, where $a_\lambda \in \{0,1\}$. Each ANF have up to 2^n terms (i.e. monomials), depending of the values of the coefficients a_λ. Denote by $L_{F(s)}$ the number of terms in the ANF of the function $F^{(s)}$. Then the number of terms of the vector valued function F is defined to be the number

$$L_F = \sum_{s=1}^{r} L_{F^{(s)}}.$$

Defnition 1. *Let* $F:\{0,1\}^n \rightarrow \{0,1\}^r$ *be a vector valued Boolean function. For any* $k \in \{1,\ldots,n\}$ *and any assembly of S subsets* $\sigma_j = \{i_1,\ldots,i_k\} \subset \{1, \ldots, n\}$ *chosen uniformly at random* ($1 \le j \le S$), *let* $F_{\sigma j}$ *denote the restriction of F defined by* $F_{\sigma j}$ $(x_1,\ldots,x_n)=F(0,\ldots,0, x_{i1}, 0,\ldots,0, x_{i2}, 0,\ldots,0, x_{ik},\ldots,0)$. *We define a random variable* $\overline{L_F}$ *– the Normalized Average Number of Terms (NANT) as:*

$$\overline{L_F} = \overline{L_F}(r,k) = \frac{1}{r} \cdot \frac{1}{2^{k-1}} \cdot \lim_{S \to \infty} \frac{1}{S} \sum_{s=1}^{r} L_{F\sigma_j}$$

Since the subsets σ_j are chosen uniformly at random, the average values of $L_{F_{\sigma j}^{(s)}}$ (s = 1, 2,…,r) are 2^{k-1}, the average value of $L_{F_{\sigma j}}$ is $r2^{k-1}$ and $L_{F_{\sigma j}^{(s)}} \leq 2^k$. So, the following theorem is true:

Theorem 2. *For any function* $F:\{0,1\}^n \to \{0,1\}^r$ *chosen uniformly at random from the set of all such functions, for any value of* $r \geq 1$ *and for any* $k \in \{1, \ldots, n\}$, *it is true that*

$$0 \leq \overline{L_F} \leq 2$$

and that the expected value is

$$0 \leq EX(\overline{L_F}) \leq 2$$

□

For our purposes we take n = 512 and we consider functions $F : \{0,1\}^{512} \to \{0,1\}^r$, where $r \in \{32, 160, 256\}$ depending on whether we measure the complexity in the message expansion part (32-bit variables), the iterative part of SHA-1 and SHA-1Q2 (the hash is 160 bits), or the iterative part of SHA-2 (the hash is 256 bits).

Remark: Obviously, there is very close analogy between our introduced measure of Normalized Average Number of Terms – $\overline{L_F}(r,k)$ for a specific value k, and the propagation criterion of degree l and order k ($PC(l)$ of order k) introduced by Preneel at al. [17]. (Stronger mathematical relations between NANT and $PC(l)$ of order k are also interesting research topic but are out of scope of this paper.)

The reasons why we choose to apply averaging in the definition of NANT were that we wanted to have a tool that will give us quantitative measure how close to random Boolean function some iteratively defined hash function is. We find that having such a measure, rather than just a mathematical definition that will give us Yes/No answers, is much preferable in the analysis of iteratively defined hash functions. Moreover, having concretely defined compression functions (such as those of SHA-1, SHA-1Q2 and SHA-2) with several hundreds of input bits, it was practically impossible directly to apply the definitions for the balanced Boolean function, for high order of resiliency or for high propagation degree. Still, we want to stress that we do not consider the value of $\overline{L_F}$ as an ultimate measure that will unconditionally prove security claims for the hash functions. To repeat once again, NANT is used just as a tool to conjecture how close and how fast some iterated Boolean functions obtain a property that is true for a random Boolean function.

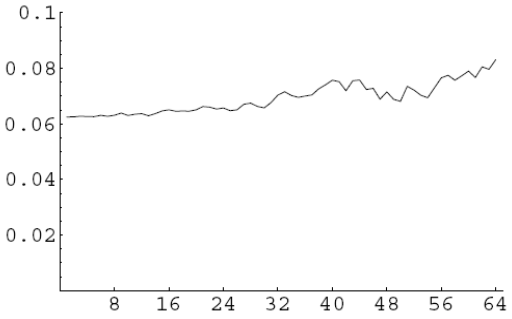

Fig. 2a SHA-1 Message Expansion

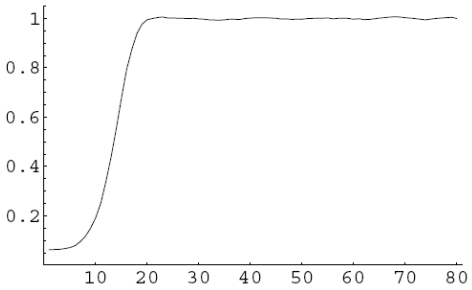

Fig. 2b SHA-1 Iterations

For small values of k (k = 3, 4, 5, 6, 7, 8), the values $\overline{L_F}(r,k)$ are easily computable. In Figures 2–4 we give graphs for SHA-1, SHA-2 and SHA-1Q2 for their message expansion part and for their iterative part when k = 5. Similar graphs can be obtained for other values of k. In Figure 2a it can be seen that the message expansion part of SHA-1, being completely linear, never reaches the complexity of a random Boolean function. Further, in Figure 2b we can see that SHA-1 reaches the complexity of a random Boolean function after 20 steps in its iterative part. Here we can again express our opinion that linearity of its message expansion part, in combination with the linear and invertible operations applied in iteration part, are the essential reasons for the successes of finding collisions of reduced SHA-1 function up to 58 steps and for the breaking of the security level of 2^{80} SHA-1 computations for the collision resistance.

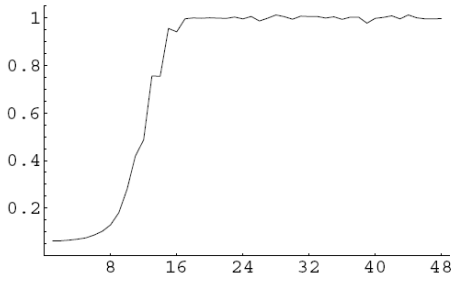

Fig. 3a SHA-2 Message Expansion

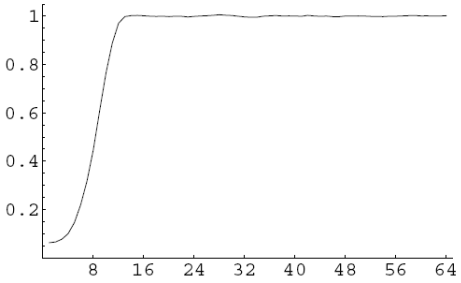

Fig. 3b SHA-2 Iterations

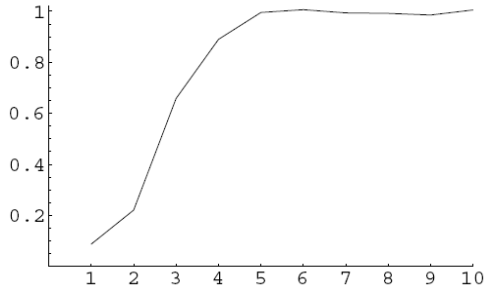

Fig. 4a SHA-1Q2 Message Expansion

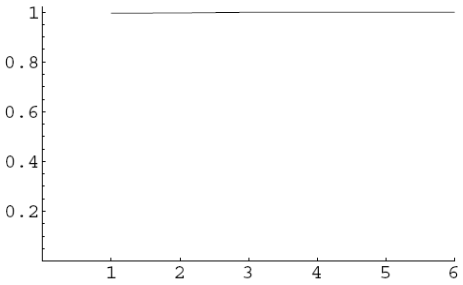

Fig. 4b SHA-1Q2 Iterations

The situation with SHA-2 is significantly different. From Figure 3a we see that the message expansion part of SHA-2 is much better designed and it reaches the same complexity as a random Boolean function after 16 steps, which reflects afterwards in the iterative part of SHA-2 that achieves the complexity level of a random Boolean function after 13 steps (Figure 3b).

In Figures 4a and 4b we show the complexity levels of our proposal SHA-1Q2. Having similar design principles as SHA-2 in the design of SHA-1Q2 we have introduced one new principle that is not present in the design of SHA-2: *computations in the iterative part of the compression function starts with variables produced in the message expansion part that have complexity level of a random Boolean function*. As a consequence of this additional design detail SHA-1Q2 starts immediately with a complexity of a random Boolean function in its iterative part.

4.4 Finding Collisions in Variants of Reduced Compression Function of SHA-1Q2

In this subsection we will analyze a reduced compression function SHA-1Q2 with only one or two steps, and we will show how to find collisions with workload less than 2^{80}.

Together with the initialization in which we use the values W_{31}, W_{30}, W_{29}, W_{28}, W_{27} we can divide the set of all 32 variables W_i, $0 \le i \le 31$, into four disjunctive subsets:

1. Used words from the extension part. $X_1 = \{W_{31}, \ldots, W_{27}, W_{22}, W_{16}\}$
2. Unused words from the extension part. $X_2 = \{W_{26}, \ldots, W_{23}, W_{21}, \ldots, W_{17}\}$
3. Used words from the message part. $Y_1 = \{W_8, W_0\}$
4. Unused words from the message part. $Y_2 = \{W_1, \ldots, W_7, W_9, \ldots, W_{15}\}$

So, if we fix the values of the set X_1 then, with the procedure described in the proof of the Theorem 1, we can search trough the values of the set X_2 in order to find collisions in the set Y_1. Since the total number of bits of the variables in the set Y_1 is 64, from the birthday paradox we can expect that after 2^{32} attempts of different values from the set X_2 we will find a collision in the set Y_1. Consequently, the values in the set Y_2 will be different,

i.e., we will find a collision of reduced SHA-1Q2 on one step after 2^{32} attempts.

For a two step reduction we will have the following situation:

1. Used words from the extension part. $X_1 = \{W_{31}, \ldots, W_{27}, W_{23}, W_{22}, W_{17}, W_{16}\}$
2. Unused words from the extension part. $X_2 = \{W_{26}, W_{25}, W_{24}, W_{21}, W_{20}, W_{19}, W_{18}\}$
3. Used words from the message part. $Y_1 = \{W_9, W_8, W_1, W_0\}$
4. Unused words from the message part. $Y_2 = \{W_2, \ldots, W_7, W_{10}, \ldots, W_{15}\}$.

Again if we fix the values of the set X_1 we can search trough the values of the set X_2 in order to find collisions in the set Y_1. Since now the total number of bits of the variables in the set Y_1 is 128 from the birthday paradox we can expect that after 2^{64} attempts of different values from the set X_2 we will find a collision in the set Y_1. Consequently, we will find a collision of reduced SHA-1Q2 on two steps after 2^{64} attempts.

Obviously, the above strategy does not work for three step reduced SHA-1Q2 since it needs workload in the range of 2^{96}, which is more than simple exhaustive search of size 2^{80}.

4.5 Finding Collisions in Full SHA-1Q2

We will discuss the strength of the iterated hash function SHA-1Q2 as a collision resistant function in the light of the previous work [14], the discussion that was given there and in the light of the known successful attacks on MDx family of hash functions.

Finding collisions in MDx family of hash functions was always based on the fact that in the iterative step they use linear and invertible functions: eXclusive OR and addition modulo 2^{32}. That allowed numerous researchers (for example den Boer and Bosselaers in [21, 22], Dobbertin in [23] and at the rump session of Eurocrypt '96, Wang at al in [1-4] and many others) to mount successful attacks for finding collisions. The basic principle that was used was the following: Setting up some system of equations obtained from the definition of the hash function, then tracing forward and backward some initial bit differences that will result in fine tuning and annulations of those differences and, finally, obtaining collisions. As it is stressed in [14], Quasigroup Folding prevents those attacks, since the operation is highly nonlinear and its tracing (and solving the nonlinear systems of equations) becomes infeasible after few steps. The message expansion of SHA-1Q2 uses 16 QFOLD operations, it is bijective function on the set $\{0,1\}^{512}$ and the last obtained five 32-bit variables obtained in the expansion part are the starting values for the iterative part. So, our claim that SHA-1Q2 is collision resistant (i.e. the workload for finding collisions is 2^{80} SHA-1Q2 computations) is based on the nonlinearity and complexity of the used Quasigroup Folding technique.

More precisely, since every application of Quasigroup Folding introduces at least 4 nonlinear quasigroup equations with 8 unknown 4-bit variables, after 16 steps in the expansion part we will obtain at least 64 quasigroup equations with 128 unknown 4-bit variables. The relations between those 4-bit variables are not in some finite field since the used quasigroup is non-associative and non-commutative. Solving that system will need the usage of quasigroup parastrophes of the quasigroup defined in Table 1 (for more details see [14]). Except exhaustive combinatorial search, in this moment there is no mathematical knowledge for fast finding of solutions for such complex nonlinear systems of quasigroup equations. The current level of the mathematical knowledge for solving such nonlinear systems of equations can only reduce the problem to a system of multivariate polynomial equations in $GF(2)$. In such a case the most advanced mathematical technique for solving such systems is by the Buchberger's method of Gröbner's bases [18]. However, if the number of monomials in such systems is exponential (on the number of variables) then that algorithm has exponential running time (see for example [19, 20]). Since we have designed the compression function of SHA-1Q2 to be conjectured that obtained systems of nonlinear equations in $GF(2)$ will have an exponential number of monomials (see the NANT level in Table 4b. that is around 1.0, that is equivalent to the claim that the number of monomials is exponential on the number of output bits), we conjecture that the compression function of SHA-1Q2 is collision resistant, i.e., for finding a collision the needed computational workload is at least 2^{80} SHA-1Q2 computations.

4.6 Finding Preimages and Second Preimages of SHA-1Q2

From the definition of SHA-1Q2 similarly as with SHA-1 from a given 160-bit hash digest it is possible to perform backward steps by guessing values for the message and for the working variables of the extension part. However, since the message expansion part is bijection over $\{0,1\}^{512}$ if the attacker guess some values for W_7 and W_{15} he(she) can not just put arbitrary values for W_{23} and W_{29}. In fact, because of that bijective property of the message expansion part, the attacker have to guess the whole initial

message of 512 bits, compute the message expansion and then perform backward steps. At the end of that backward procedure the attacker will have to find a message that, by going backward, will end by giving the initial 160 bits defined by (2) and that search effort will need 2^{160} attempts.

4.7 Cryptographic properties of the used quasigroup

In this subsection we would like to give several remarks about the used quasigroup. Basically here we can repeat the claims of Gligoroski at. al from the paper [14]. The operations of quasigroup-folding are performed by 16×16 quasigroup that is non-commutative, non-associative, non-idempotent, non-involutory and without neutral elements. That is quite different to the Boolean functions conjunction, disjunction, negation, addition (*mod* 2^{32}) and exclusive disjunction, which are used in the MDx family of hash functions. All of these Boolean functions satisfy many algebraic laws suitable for making reductions and for solving equations and systems of equations. On the other hand, solving equations that introduce quasigroup operations can not lead to the reduction of variables and successful tracing of differences, that is the core of every successful differential attack on the hash functions that we know of today. There are huge numbers of quasigroups of order 16 that satisfy the above mentioned properties. In fact the number of such quasigroups is of order 2^{430}, any such a quasigroup can be used in a definition of SHA-1Q2, and their construction can be done in very fast manner.

5. Conclusion

In this paper we have constructed a new hash function SHA-1Q2 based on the hash function SHA-1 and its security improvement SHA-1Q. The proposed hash function uses recently introduced technique of nonlinear bijective operation "Quasigroup Folding" as a tool for obtaining highly complex and nonlinear relations of the output bits. The design of SHA-1Q2 introduces a new principle: the reached complexity level of a random Boolean function in the message expansion part to be used immediately in the iterative part of the compression function. It has 16 steps in the Message Expansion part, 8 iterative steps and it is at least 3% faster than SHA-1.

References

[1] X. Wang, X. Lai, D. Feng, H. Chen and X. Yu, "Cryptanalysis of the Hash Functions MD4 and RIPEMD", EuroCrypt 2005, Springer LNCS 3494 (2005), 118
[2] X. Wang and H. Yu , "How to Break MD5 and Other Hash Functions", EuroCrypt 2005, Springer LNCS 3494 (2005), 1935
[3] X. Wang, Y. L. Yin, H. Yu, "Collision Search Attacks on SHA-1", Crypto 2005, Springer LNCS 3621 (2005), pp. 17-36
[4] X. Wang, H. Yu, Y. L. Yin :"Efficient Collision Search Attacks on SHA-0", Crypto 2005, Springer LNCS 3621 (2005), pp. 1-16
[5] R. Merkle, "One way hash functions and DES", Advances in Cryptology-Crypto'89, LNCS 435, G. Brassard, Ed., Springer-Verlag, 1990, pp. 428-446
[6] I.B. Damgård, "A design principle for hash functions", Advances in CryptologyCRYPTO 89 LNCS 435, G. Brassard, Ed., Springer-Verlag, 1990, pp. 416-427
[7] G. Yuval, "How to swindle Rabin", Cryptologia, 3 (1979), pp. 187-190
[8] A. Joux, "Multicollisions in iterated hash functions. Application to cascaded constructions", In M. Franklin, editor, Advances in Cryptology CRYPTO 2004, volume 3152 of Lecture Notes in Computer Science, pp. 306-316. Springer-Verlag, Berlin, Germany, 2004
[9] J. Kelsey and B. Schneier, "Second preimages on n-bit hash functions for much less than 2^n work", In R. Cramer, editor, Advances in Cryptology EUROCRYPT 2005, volume 3494 of Lecture Notes in Computer Science, pp. 474-490. Springer-Verlag, Berlin, Germany, 2005
[10] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisisted: How to construct a hash Function", in V. Shoup, editor, Advances in Cryptology CRYPTO 2005, volume 3621 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2005
[11] P. Gauravaram, W. Millan and J. G. Nieto, "Some thoughts on Collision Attacks in the Hash Functions MD5, SHA-0 and SHA-1", Cryptology ePrint Archive: Report 2005/391.
[12] B. Preneel, "Analysis and design of Cryptographic Hash Functions", PhD thesis, Katholieke Universiteit Leuven, 1993
[13] B. Preneel, "The State of Cryptographic Hash Functions", Lectures on Data Security, Lecture Notes in Computer Science 1561, I. Damgård (ed.), pp. 158-182, 1999
[14] D. Gligoroski, S. Markovski and S. J. Knapskog, "A Fix of the MD4 Family of Hash Functions – Quasigroup Fold", NIST Cryptographic Hash Workshop, 2005, http://www.csrc.nist.gov/pki/HashWorkshop/program.htm
[15] Secure Hash Signature Standard (SHS) (FIPS PUB 180-2), United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-2, 2002 August 1
[16] Crypto++ library, http://www.cryptopp.com
[17] B. Preneel, R. Govaerts and J. Varitlevalle, "Boolean functions satisfying higher order propagation criteria", Advances in Cryptology - EUROCRYPT91, Lecture Notes in Computer Science 547, D. W. Davies (ed.), Springer-Verlag, pp. 141-152, 1991

[18] B. Buchberger, “Buchbergers PhD thesis 1965 - An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal”, Journal of Symbolic Computation, Volume 41, Issues 3-4, March-April 2006, pp. 475-511, Elsevier Ltd.
[19] M. Clegg, J. Edmonds and R. Impagliazzo, “Using the Groebner basis algorithm to find proofs of unsatisfiability”, in Proc. 28th ACM Symp. on Theory of Computing, pp. 174-183, ACM 1996
[20] R. Impagliazzo, P. Pudlák, J. Sgall, “Lower Bounds for the Polynomial Calculus and the Groebner Basis Algorithm”, Electronic Colloquium on Computational Complexity (ECCC) 4(42), (1997)
[21] B. den Boer, and A. Bosselaers: \An attack on the last two rounds of MD4", Advances in Cryptology, CRYPTO91,
Lecture Notes in Computer Science, vol. 576, Springer-Verlag, Berlin, 1992, pp. 194-203
[22] B. den Boer, and A. Bosselaers, “Collisions for the compression function of MD5”, Advances in Cryptology, EUROCRYPT93, Lecture Notes in Computer Science, vol. 765, Springer-Verlag, Berlin, 1994, pp. 293-304
[23] H. Dobbertin, “Cryptanalysis of MD4”, J. Cryptology (1998) 11, pp. 253-271

Danilo Gligoroski received the PhD degree in Computer Science from Institute of Informatics, Faculty of Natural Sciences and Mathematics, at University of Skopje – Macedonia in 1997. His research interests are Cryptography, Computer Security, Discrete algorithms and Information Theory and Coding. Currently he is PostDoc at Q2S – Centre for Quantifiable Quality of Service in Communication Systems at Norwegian University of Science and Technology - Trondheim, Norway.

Smile Markovski received his PhD in Mathematics from University of Skopje in 1980 in the field of algebra. He has been elected as full professor in 1991 at the same university. His research interests are Universal algebras, *n*-ary and vector valued groupoids, quasigroup theory, discrete mathematics, cryptography and coding theory.

Svein Johan Knapskog received his MS in Electrical Engineering from Norwegian University of Science and Technology – Trondheim in 1972. His research interests are Network Security, Cryptography, and Security Standards. Currently he is a head of the “Centre for Quantifiable Quality of Service in Communication Systems – Q2S”, at the Norwegian University of Science and Technology, Trondheim, Norway.

Appendix:
C source code of SHA-1Q2 hash function

```
/* SHA1Q2.c -
An altered SHA-1 algorithm named as SHA-1Q2 that fixes the weaknesses of SHA-1
message-digest algorithm. Version 2 has only 16 message expansion rounds, 8
iteration rounds and is about 3% faster then original SHA-1.
*/
/* The modification for SHA-1Q2 made by Danilo Gligoroski 05.05.2006. */
/* The modification for SHA-1Q made by Danilo Gligoroski 20.08.2005.

Danilo Gligoroski makes no representations concerning either the
merchantability of this software or the suitability of this software for any
particular purpose. It is provided "as is" without express or implied warranty
of any kind.

This work was carried out during the Postdoc research position ot Q2S -Centre
for Quantifiable Quality of Service in Communication Systems at Norwegian
University of Science and Technology - Trondheim, Norway.
*/
/*
* sha1.c
*
* Description:
* This file implements the Secure Hashing Algorithm 1 as
* defined in FIPS PUB 180-1 published April 17, 1995.
*
* The SHA-1, produces a 160-bit message digest for a given
* data stream. It should take about 2**n steps to find a
* message with the same digest as a given message and
* 2**(n/2) to find any two messages with the same digest,
* when n is the digest size in bits. Therefore, this
* algorithm can serve as a means of providing a
* "fingerprint" for a message.
*
* Portability Issues:
* SHA-1 is defined in terms of 32-bit "words". This code
* uses <stdint.h> (included via "sha1.h" to define 32 and 8
* bit unsigned integer types. If your C compiler does not
* support 32 bit unsigned integers, this code is not
* appropriate.
*
* Caveats:
* SHA-1 is designed to work with messages less than 2^64 bits
* long. Although SHA-1 allows a message digest to be generated
* for messages of any number of bits less than 2^64, this
* implementation only works with messages with a length that is
* a multiple of the size of an 8-bit character.
*
*/
#include "sha1.h"
/* This is the definition of the quasigroup Q of order 16x16. */
unsigned char Q[256] ={ 10, 4, 5, 9, 6, 0, 14, 1, 2, 12,
13, 15, 3, 8, 11, 7,
5, 11, 12, 8, 4, 14, 0, 7, 3, 2, 15, 10, 1, 9, 13, 6,
12, 5, 2, 13, 15, 8, 10, 14, 1, 3, 6, 7, 11, 0, 9, 4,
7, 13, 3, 14, 2, 1, 11, 12, 5, 9, 4, 8, 0, 15, 6, 10,
1, 2, 4, 10, 11, 7, 8, 9, 0, 13, 3, 14, 6, 12, 5, 15,
4, 10, 8, 11, 13, 2, 12, 6, 14, 15, 5, 9, 7, 3, 1, 0,
0, 14, 13, 2, 8, 3, 6, 5, 12, 11, 7, 4, 9, 10, 15, 1,
11, 6, 0, 5, 9, 13, 4, 8, 7, 10, 2, 3, 15, 1, 14, 12,
13, 8, 6, 1, 12, 10, 15, 0, 11, 5, 9, 2, 4, 7, 3, 14,
2, 15, 1, 0, 7, 12, 5, 11, 9, 6, 8, 13, 10, 14, 4, 3,
6, 12, 11, 7, 10, 15, 1, 3, 4, 8, 14, 0, 13, 5, 2, 9,
8, 1, 15, 6, 3, 9, 7, 4, 10, 14, 12, 5, 2, 13, 0, 11,
15, 3, 9, 4, 14, 6, 2, 13, 8, 7, 0, 1, 12, 11, 10, 5,
14, 9, 7, 3, 1, 11, 13, 15, 6, 0, 10, 12, 5, 4, 8, 2,
3, 0, 14, 12, 5, 4, 9, 10, 15, 1, 11, 6, 8, 2, 7, 13,
9, 7, 10, 15, 0, 5, 3, 2, 13, 4, 1, 11, 14, 6, 12, 8
};
/* First 16 bits of a variable 'a' will be changed by 16 bits
obtained by quasigroup transformation defined below. */ #define
QUASIGROUP_FOLD(a) {\
(a) = ((a)&0x0fffffff)|(((unsigned char)Q[( ((a)&0xf0000000)>>28)| \
( ((a)&0x0000f000)>> 8)])<<28);\
(a) = ((a)&0xf0ffffff)|(((unsigned char)Q[( ((a)&0x0f000000)>>20)| \
( ((a)&0x00000f00)>> 8)])<<24);\
(a) = ((a)&0xff0fffff)|(((unsigned char)Q[( ((a)&0x00f00000)>>20)| \
( ((a)&0x000000f0) )])<<20);\
(a) = ((a)&0xfff0ffff)|(((unsigned char)Q[( ((a)&0x000f0000)>>12)| \
( ((a)&0x0000000f) )])<<16);\
}
/*
* Define the SHA1 circular left shift macro
*/
#define SHA1CircularShift(bits,word) \
(((word) << (bits)) | ((word) >> (32-(bits))))
/* Local Function Prototyptes */ void SHA1PadMessage(SHA1Context
*); void SHA1ProcessMessageBlock(SHA1Context *);
/*
* SHA1Reset
*
* Description:
* This function will initialize the SHA1Context in preparation
* for computing a new SHA1 message digest.
*
* Parameters:
* context: [in/out]
* The context to reset.
*
* Returns:
* sha Error Code.
*
*/
int SHA1Reset(SHA1Context *context) {
if (!context)
{
return shaNull;
}
context->Length_Low = 0;
context->Length_High = 0;
context->Message_Block_Index = 0;
context->Intermediate_Hash[0] = 0x67452301;
context->Intermediate_Hash[1] = 0xEFCDAB89;
context->Intermediate_Hash[2] = 0x98BADCFE;
context->Intermediate_Hash[3] = 0x10325476;
context->Intermediate_Hash[4] = 0xC3D2E1F0;
context->Computed = 0;
context->Corrupted = 0;
return shaSuccess;
}
/*
* SHA1Result
*
* Description:
* This function will return the 160-bit message digest into the
* Message_Digest array provided by the caller.
* NOTE: The first octet of hash is stored in the 0th element,
* the last octet of hash in the 19th element.
*
* Parameters:
* context: [in/out]
* The context to use to calculate the SHA-1 hash.
* Message_Digest: [out]
* Where the digest is returned.
*
* Returns:
* sha Error Code.
*
*/
int SHA1Result( SHA1Context *context,
uint8_t Message_Digest[SHA1HashSize])
{
int i;
if (!context || !Message_Digest)
{
return shaNull;
}
if (context->Corrupted)
{
return context->Corrupted;
}
if (!context->Computed)
{
SHA1PadMessage(context);
for(i=0; i<64; ++i)
{
/* message may be sensitive, clear it out */
context->Message_Block[i] = 0;
}
context->Length_Low = 0; /* and clear length */
context->Length_High = 0;
context->Computed = 1;
}
for(i = 0; i < SHA1HashSize; ++i)
{
Message_Digest[i] = context->Intermediate_Hash[i>>2]
>> 8 * ( 3 - ( i & 0x03 ) );
}
return shaSuccess;
}
/*
* SHA1Input
*
* Description:
* This function accepts an array of octets as the next portion
* of the message.
*
* Parameters:
* context: [in/out]
* The SHA context to update
* message_array: [in]
* An array of characters representing the next portion of
* the message.
* length: [in]
* The length of the message in message_array
*
* Returns:
* sha Error Code.
*
*/
int SHA1Input( SHA1Context *context,
const uint8_t *message_array,
unsigned length)
{
if (!length)
{
return shaSuccess;
}
if (!context || !message_array)
{
return shaNull;
}
if (context->Computed)
{
context->Corrupted = shaStateError;
return shaStateError;
}
if (context->Corrupted)
{
return context->Corrupted;
}
while(length-- && !context->Corrupted)
{
context->Message_Block[context->Message_Block_Index++] =
(*message_array & 0xFF);
context->Length_Low += 8;
if (context->Length_Low == 0)
{
context->Length_High++;
if (context->Length_High == 0)
{
```

```
/* Message is too long */
context->Corrupted = 1;
}
}
if (context->Message_Block_Index == 64)
{
SHA1ProcessMessageBlock(context);
}
message_array++;
}
return shaSuccess;
}
/*
* SHA1ProcessMessageBlock
*
* Description:
* This function will process the next 512 bits of the message
* stored in the Message_Block array.
*
* Parameters:
* None.
*
* Returns:
* Nothing.
*
* Comments:
* Many of the variable names in this code, especially the
* single character names, were used because those were the
* names used in the publication.
*
*
*/
void SHA1ProcessMessageBlock(SHA1Context *context) { /*
Intervention by Danilo Gligoroski */ /* SHA-1Q2 does not use any
predefined constants, so the next lines will
be put in comment.
*/
// const uint32_t K[] = { /* Constants defined in SHA-1 */
// 0x5A827999,
// 0x6ED9EBA1,
// 0x8F1BBCDC,
// 0xCA62C1D6
// };
int t; /* Loop counter */
uint32_t temp; /* Temporary word value */
/* Intervention by Danilo Gligoroski */
/* Instead of 80 uint32_t words we need only 32 */
uint32_t W[32]; /* Word sequence */
uint32_t A, B, C, D, E; /* Word buffers */
/*
* Initialize the first 16 words in the array W
*/
for(t = 0; t < 16; t++)
{
W[t] = context->Message_Block[t * 4] << 24;
W[t] |= context->Message_Block[t * 4 + 1] << 16;
W[t] |= context->Message_Block[t * 4 + 2] << 8;
W[t] |= context->Message_Block[t * 4 + 3];
}
/* Intervention by Danilo Gligoroski */
/* The following changes are made:
1. Only 16 steps are performed.
2. Instead of bitwise XORing, a complex operation of addition (mod 2^32)
and XOR are used.
3. In every step, 7 bits circular shift is performed.
4. One QUASIGROUP_FOLD operation is performed per new W[t].
*/
for(t = 16; t < 32; t++)
{
W[t] = SHA1CircularShift(7,\
(W[t - 1] ^ W[t - 3]) + (W[t - 6] ^ W[t - 8]) + \
(W[t - 12] ^ W[t - 14]) + (W[t - 13] ^ W[t - 16]) + \
(W[t - 2] ^ W[t - 7]) + (W[t - 4] ^ W[t - 10]) + \
(W[t - 5] ^ W[t - 9]) + (W[t - 11] ^ W[t - 15])
);
QUASIGROUP_FOLD((W[t]))
}
/* Intervention by Danilo Gligoroski */
/* The following changes are made:
1. Instead of original initialization - an initialization involves
the last 5 words obtained in the message expansion part.
*/
A = context->Intermediate_Hash[0] + W[31];
B = context->Intermediate_Hash[1] + W[30];
C = context->Intermediate_Hash[2] + W[29];
D = context->Intermediate_Hash[3] + W[28];
E = context->Intermediate_Hash[4] + W[27];
/* Intervention by Danilo Gligoroski */
/* The following changes are made:
1. Instead of original 4x20=80 steps, only 4x2=8 steps
are performed.
2. Those transformations are made on words:
W[0], W[1], ..., W[29]
3. We don't use addition of any constants K[].
*/
/* t=0; */
temp = SHA1CircularShift(5,A) + ((B & C) ^ ((~B) & D)) +\
E + (W[0]^W[16]) + (W[8]^W[22]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
15

/* t=1; */
temp = SHA1CircularShift(5,A) + ((B & C) ^ ((~B) & D)) +\
E + (W[1]^W[17]) + (W[9]^W[23]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=2 */
temp = SHA1CircularShift(5,A) + (B ^ C ^ D) +\
E + (W[2]^W[18]) + (W[10]^W[24]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=3 */
temp = SHA1CircularShift(5,A) + (B ^ C ^ D) +\
E + (W[3]^W[19]) + (W[11]^W[25]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=4 */
temp = SHA1CircularShift(5,A) + ((B & C) ^ (B & D) ^ (C & D)) +\
E + (W[4]^W[20]) + (W[12]^W[26]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=5 */
temp = SHA1CircularShift(5,A) + ((B & C) ^ (B & D) ^ (C & D)) +\
E + (W[5]^W[21]) + (W[13]^W[27]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=6 */
temp = SHA1CircularShift(5,A) + (B ^ C ^ D) +\
E + (W[6]^W[22]) + (W[14]^W[28]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
/* t=7 */
temp = SHA1CircularShift(5,A) + (B ^ C ^ D) +\
E + (W[7]^W[23]) + (W[15]^W[29]);
E = D;
D = C;
C = SHA1CircularShift(30,B);
B = A;
A = temp;
QUASIGROUP_FOLD((A));
context->Intermediate_Hash[0] += A;
context->Intermediate_Hash[1] += B;
context->Intermediate_Hash[2] += C;
context->Intermediate_Hash[3] += D;
context->Intermediate_Hash[4] += E;
context->Message_Block_Index = 0;
}
/*
* SHA1PadMessage
*
* Description:
* According to the standard, the message must be padded to an even
* 512 bits. The first padding bit must be a '1'. The last 64
* bits represent the length of the original message. All bits in
* between should be 0. This function will pad the message
* according to those rules by filling the Message_Block array
* accordingly. It will also call the ProcessMessageBlock function
* provided appropriately. When it returns, it can be assumed that
* the message digest has been computed.
*
* Parameters:
* context: [in/out]
* The context to pad
* ProcessMessageBlock: [in]
* The appropriate SHA*ProcessMessageBlock function
* Returns:
* Nothing.
*
*/
void SHA1PadMessage(SHA1Context *context) {
/*
* Check to see if the current message block is too small to hold
* the initial padding bits and length. If so, we will pad the
* block, process it, and then continue padding into a second
* block.
*/
if (context->Message_Block_Index > 55)
{
context->Message_Block[context->Message_Block_Index++] = 0x80;
while(context->Message_Block_Index < 64)
{
context->Message_Block[context->Message_Block_Index++] = 0;
}
SHA1ProcessMessageBlock(context);
while(context->Message_Block_Index < 56)
{
context->Message_Block[context->Message_Block_Index++] = 0;
}
}
else
{
context->Message_Block[context->Message_Block_Index++] = 0x80;
while(context->Message_Block_Index < 56)
{
context->Message_Block[context->Message_Block_Index++] = 0;
}
}
/*
* Store the message length as the last 8 octets
*/
context->Message_Block[56] = context->Length_High >> 24;
```

```
context->Message_Block[57] = context->Length_High >> 16;
context->Message_Block[58] = context->Length_High >> 8;
context->Message_Block[59] = context->Length_High;
context->Message_Block[60] = context->Length_Low >> 24;
context->Message_Block[61] = context->Length_Low >> 16;
context->Message_Block[62] = context->Length_Low >> 8;
context->Message_Block[63] = context->Length_Low;
SHA1ProcessMessageBlock(context);
}
```