A New Hash Family Obtained by Modifying the SHA-2 Family

Somitra Kumar Sanadhya and Palash Sarkar

Applied Statistics Unit,
Indian Statistical Institute,
203, B.T. Road, Kolkata,
India 700108.
somitra_r@isical.ac.in, palash@isical.ac.in

Abstract. In this work, we study several properties of the SHA-2 design which have been utilized in recent collision attacks against reduced round SHA-2. Small modifications to the SHA-2 design are suggested to thwart these attacks. The modified round function provides the same resistance to linearization attacks as the original SHA-2 round function, but, provides better resistance to non-linear attacks. Our next contribution is to introduce the general idea of "multiple feed-forward" for the construction of cryptographic hash functions. This can provide increased resistance to the Chabaud-Joux type "perturbation-correction" collision attacks. The idea of feed-forward is taken further by introducing the idea of feed-forward across message blocks leading to resistance against generic multi-collision attacks. The net effect of the suggested changes to the SHA-2 design has insignificant impact on the efficiency of computing the digest.

1 Introduction

Following the attacks on SHA-0 [1] and SHA-1 [20], the attention of the cryptanalysis community has been directed to the SHA-2 family. Recent attacks against SHA-2 starting with [12], and followed by [18, 16, 7, 17] have utilized certain previously unknown properties in the round function of SHA-2. These have led to upto 24-step attacks against both SHA-256 and SHA-512. A unified combinatorial description of the recent attacks on reduced-round SHA-2 can be found in our technical report [14]. While none of these attacks threaten any of the security properties of the full SHA-2 hash functions, it is also true that some features of the compression function have been exploited in the attack. The following four properties have been used in the recent step-reduced attacks.

1. The works in [5], [6], [10, 11] and [13] show that 9-round local collisions obtained using XOR differentials hold with probability 2^{-39} or less for SHA-256. In contrast, the work of [12] shows that 9-round local collisions using additive differentials can be obtained with probability $1/3$ (and improved to probability 1 in [18]). This suggests that the round function resists XOR differentials better than additive differentials.
2. SHA-2 design uses 8 registers a to h, where a and e registers are nonlinearly updated while the rest are simply copied. It turns out that there is a *very simple* relation by which the e-register value at Step i can be controlled using only the a-register values at Step i to $i-4$.
3. The round update function for the a-register uses an invertible linear transformation Σ_0 while that of the e-register uses an invertible linear transformation Σ_1. Both Σ_0 and Σ_1 have 0 and -1 as fixed points.
4. The technique of perturbation-correction [1] is used to build the attacks. A 9-step local collision is suitably placed between steps i and $i+8$ and it is ensured that all message word differences after Step $i+8$ are zero.

We suggest methods to get around the above issues. The linear maps Σ_0, Σ_1 are modified to affine maps Γ_0, Γ_1 to ensure that Γ_0, Γ_1 and $\Gamma_0 \oplus \Gamma_1$ do not have any fixed points (along with a few other properties). This takes care of the third point above. Simple but carefully considered changes are suggested to the update function of the a and e-registers. These changes help in avoiding local collisions of high probability based on additive differentials which have been obtained in [12] and [18]. These changes also lead to cancelling out the simple relation between the a and e-registers. As a result, both the first and second points above are eliminated.

The resistance of the new round function to linearization attacks is the same as that provided by the SHA-2 round function. In particular, the best linear local collision for the 256-bit case holds with probability 2^{-39} as in the case of SHA-256. We argue later that the suggested changes provide better resistance to recent non-linear attacks.

To tackle the fourth point, we introduce a new hash function design called multiple feed-forward. This provides additional resistance to the perturbation-correction technique for finding collisions. The SHA-family design uses a single feed-forward where the chaining value is added to the output of composition of all the round functions. We suggest introducing several other feed-forward steps where the feed-forward is alternately provided using addition and XOR. A consequence is that if any 9-round local collision is placed within the first 16 steps, then there will be a step i within these 16 steps such that there will be a perturbation in the registers at Step i and this perturbation will necessarily extend to steps beyond the first 16 steps. Since message words beyond the first 16 steps are obtained using the message recursion, it will be difficult to cancel out the effect of such cascaded perturbation. This improves the resistance to perturbation-correction attacks. Such resistance is achieved at a marginal cost. The amortized cost of the new feed-forward steps is less than one t-bit operation (add/XOR) per step, where $t = 32$ for SHA-256 and $t = 64$ for SHA-512.

The idea of feed-forward is taken one step further. We suggest the idea of providing feed-forward across message blocks. The intuitive justification is that this provides an additional mechanism for allowing the processing of the current block to depend on earlier blocks. Concrete suggestions are given for the SHA-2 family. These improve the resistance of SHA-2 hash functions against generic multi-collision attacks introduced in [8]. At a general level, our idea of feed-forward across message blocks can be seen as a practical version of the wide-pipe design strategy suggested in [9].

We put together all the ideas and make a new proposal called SShash. A description is given in Section 7 along with comparison of timing results to SHA-2. The implementation has been made by modifying the implementation of SHA-2 available from [3]. Test vectors and the code for the 256-bit variant of SShash are given in Section A.

Relation to the SHA-3 Competition. The NIST of USA is currently running a competition to select a new hash standard called SHA-3 [4]. Around 60 candidates have been submitted to the SHA-3 competition. Our proposal SShash has not been submitted to the SHA-3 competition.

According to NIST documentation [4], "NIST does not currently plan to withdraw SHA-2 or remove it from the revised Secure Hash Standard". So, scientific interest in SHA-2 is still very much alive. Our proposal should be seen in this light. We have suggested some modifications to SHA-2 so as to resist the recent reduced-round attacks and also to achieve other desirable features. As is the case in all designs of symmetric primitives, confidence in a primitive grows with the failure of cryptanalytic efforts. The same is also true for our proposal. If SShash gets broken in the future, then the procedure may throw some light on possible attacks on SHA-2. On the other hand, if the design survives, then some lessons would have been learnt.

The new proposal is not a competitor for SHA-3 and hence will not be a standard. But, NIST standards are of great interest worldwide. Consequently, there are users who would be interested in knowing how to avoid the recent reduced-round attacks on SHA-2. If SShash can indeed achieve this, then such users may consider using SShash for possible proprietary purposes.

2 The SHA-2 Hash Family

The newest members of SHA family of hash functions were standardized by US NIST in 2002 [19]. There are two functions in this standard: SHA-256 and SHA-512. In addition, the standard also specifies their truncated versions as SHA-224 and SHA-384. The number in the name of the hash function refers to the length of message digest produced by that function. Next we describe SHA-256 and SHA-512 in detail.

Eight registers are used in the evaluation of SHA-2. The initial values in the registers are specified by an $8 \times n$ bit IV, n=32 for SHA-256 and 64 for SHA-512. In Step i, the 8 registers are updated from $(a_{i-1},\ b_{i-1},\ c_{i-1},\ d_{i-1},\ e_{i-1},\ f_{i-1},\ g_{i-1},\ h_{i-1})$ to $(a_i,\ b_i,\ c_i,\ d_i,\ e_i,\ f_i,\ g_i,\ h_i)$ according to the following equation:

$$\left.\begin{aligned}
a_i &= \Sigma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) + \Sigma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i + W_i\\
b_i &= a_{i-1}\\
c_i &= b_{i-1}\\
d_i &= c_{i-1}\\
e_i &= d_{i-1} + \Sigma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i + W_i\\
f_i &= e_{i-1}\\
g_i &= f_{i-1}\\
h_i &= g_{i-1}
\end{aligned}\right\} \quad (1)$$

The f_{IF} and the f_{MAJ} are three-variable boolean functions defined as:

$$\begin{aligned}
f_{IF}(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z),\\
f_{MAJ}(x, y, z) &= (x \wedge y) \oplus (y \wedge z) \oplus (z \wedge x).
\end{aligned}$$

For SHA-256, the functions Σ_0 and Σ_1 are defined as:

$$\begin{aligned}
\Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x),\\
\Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x).
\end{aligned}$$

For SHA-512, the corresponding functions are:

$$\begin{aligned}
\Sigma_0(x) &= ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x),\\
\Sigma_1(x) &= ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x).
\end{aligned}$$

Round i uses a t-bit word W_i which is derived from the message and a constant word K_i. There are 64 steps in SHA-256 and 80 in SHA-512. The hash function operates on a 512-bit (resp. 1024-bit) message specified as 16 words of 32 (resp. 64) bits for SHA-256 (resp. SHA-512). Given the message words $m_0,\ m_1,\ \ldots m_{15}$, the W_i 's are computed using the Equation:

$$W_i = \begin{cases} m_i & \text{for } 0 \leq i \leq 15\\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & \text{for } 16 \leq i \leq 63 \text{ (or 80)} \end{cases} \quad (2)$$

For SHA-256, the functions σ_0 and σ_1 are defined as:

$$\begin{aligned}
\sigma_0(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x),\\
\sigma_1(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x).
\end{aligned}$$

And for SHA-512, they are defined as:

$$\begin{aligned}
\sigma_0(x) &= ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x),\\
\sigma_1(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x).
\end{aligned}$$

The $\mathsf{IV} = (a_{-1}, b_{-1}, c_{-1}, d_{-1}, e_{-1}, f_{-1}, g_{-1}, h_{-1})$ is defined differently for SHA-224, SHA-256, SHA-384 and SHA-512. For details, see [19].

The output hash value of a one block (512-bit for SHA-256 and 1024-bit for SHA-512) message is obtained by chaining the IV with the register values at the end of the final round. A similar strategy is used for multi-block messages, where the IV for next block is taken as the hash output of the previous block.

3 Some Insights into Recent Attacks on SHA-2

Local collisions for linearized version of SHA-2 were studied by Gilbert and Handschuh [5] and by Sanadhya and Sarkar [13]. All these local collisions hold for the actual SHA-256 with probabilities of about 2^{-39} or less. Using these local collisions, Mendel et al [10, 11] and later Sanadhya and Sarkar [15] obtained collisions for 18 step SHA-256. We call attacks using local collisions for the linearized version of SHA-2 as *linear attacks*.

Nikolić and Biryukov [12] presented a local collision which is valid for the actual SHA-256 function. The important point to note is that this local collision holds with probability of about 1/3. Using similar methods, Sanadhya and Sarkar [18] obtained another local collision which holds with probability 1. Extension of these attacks to obtain upto 24-round collisions for both SHA-256 and SHA-512 have been given in [7, 17]. We call attacks using local collisions valid for the actual SHA-2 as *nonlinear attacks*.

We now discuss certain features of the SHA-2 design which facilitated collision attacks against reduced round versions discussed earlier.

Linear vs. Nonlinear attacks. The only places where XOR is used in the SHA-2 design are in the design of the transformations Σ_0, Σ_1 and σ_0, σ_1. Modular addition is much more extensively used in the round function. To a certain extent, this explains the difference in probabilities between linear and non-linear attacks.

Choice of the Transformations Σ_0 and Σ_1. Two transformations Σ_0 and Σ_1 are used in the round function of SHA-2. These transformations are given in Section 2.

Consider the equations $\Sigma_0(x) = x$ and $\Sigma_1(x) = x$. Any solution to these equations will give a "fixed point" for the transformations Σ_0 and Σ_1. Since both these transformations use only XORs, we can equivalently look at the equations $(\Sigma_0 \oplus I_{32})(x) = 0$ and $(\Sigma_1 \oplus I_{32})(x) = 0$ for SHA-256, where I_{32} is the identity matrix of order 32. For SHA-512, the I_{32} needs to be replaced by I_{64}.

For SHA-256, $\Sigma_0 \oplus I_{32}$ has rank 31 but $\Sigma_1 \oplus I_{32}$ has rank 29. The null space of $\Sigma_0 \oplus I_{32}$ has basis {`0xffffffff`}, whereas the null space of $\Sigma_1 \oplus I_{32}$ has basis {`0x99999999`, `0xaaaaaaaa`, `0xcccccccc`}. For SHA-512, the ranks of both $\Sigma_0 \oplus I_{64}$ and $\Sigma_1 \oplus I_{64}$ are 63 and the null space has basis {`0xffffffffffffffff`}.

Fixed points of Σ_0 and Σ_1 for both SHA-256 and SHA-512 are shown in Table 1.

The analysis above shows that both Σ_0 and Σ_1 have common fixed points for both the functions in the SHA-2 family. Moreover, the common fixed points have very simple structure as well: all the bits are either zero or one when they are expressed as 32-bit (or 64-bit) quantities. The numeric value of these common fixed points are 0 and -1. *This is one of the crucial issues which provides the nonlinear local collisions of high probability utilized in recent attacks.*

Cross Dependence Equation. In the calculation of new register values, at each step of the SHA-2 hash family, registers b, c and d are merely copies of register a values of previous steps. Registers e

Table 1. Fixed points of Σ_0 and Σ_1 for SHA-256 and SHA-512.

Hash function	Transformation	Fixed Points
SHA-256	Σ_0	{0x00000000, 0xffffffff}
	Σ_1	{0x00000000, 0xffffffff, 0x33333333, 0x55555555,
		0x66666666, 0x99999999, 0xaaaaaaaa, 0xcccccccc}
SHA-512	Σ_0	{0x0000000000000000, 0xffffffffffffffff}
	Σ_1	{0x0000000000000000, 0xffffffffffffffff}

and a are also related since most of the terms in their computation are common. Thus, we note that e_i can be computed solely from the register a values as shown below.

$$\begin{aligned} e_i &= d_{i-1} + \Sigma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i + W_i \\ &= d_{i-1} + a_i - \Sigma_0(a_{i-1}) - f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) \\ &= a_{i-4} + a_i - \Sigma_0(a_{i-1}) - f_{MAJ}(a_{i-1}, a_{i-2}, a_{i-3}). \end{aligned} \tag{3}$$

This relationship between these two register values, which we call the Cross Dependence Equation (CDE), implies that if the a register values for five consecutive steps are known then the e register for the last of these steps can be determined. This fact means that we can control the value of e_i from a_{i-4}, a register value which was computed 4 steps earlier. The CDE was given this name in [16] and it was earlier mentioned in [18]. A consequence of CDE is that it can be used to provide an alternate description of SHA-2 round function. This was independently observed in [7].

The CDE allows an attacker to get simple relations between a and e registers by ensuring suitable behaviour of f_{MAJ}. Note that it is rather easy to control the differential behavior of f_{MAJ} as utilized in [12] and other related works. The CDE, therefore, reduces the utility of two register updates in each round.

Local Collision and Message Expansion. The idea of perturbation-correction from [1] is used to obtain a local collision. If a message difference (perturbation) is introduced at Step i, then it is possible to define subsequent message differences such that the perturbation is cancelled at Step $i+8$. This leads to a 9-step local collision. The idea of the NB attack and its extensions for obtaining an r-round collision is the following. Choose a suitable i and place a local collision from Step i to $i+8$. Then ensure that $\delta W_j = 0$ for $j = i+9, \ldots, r-1$ leading to an r-round collision. This approach succeeds because a perturbation introduced at some step can be "quickly" cancelled within a few steps. Viewed another way, the introduced perturbation need not affect registers at Step j if j is somewhat far from i, i.e., the perturbation does not have long-range effects.

4 Suggestions for Improvements to the SHA-2 Design

4.1 Using Affine Transformations in the Update Function

We suggest that the linear functions Σ_0, Σ_1 be replaced by affine functions Γ_0 and Γ_1 respectively such that the following conditions are satisfied.

1. For all $x \in \{0,1\}^t$, $\Gamma_i(x) \neq x$ or $\overline{x}$, $i = 0, 1$.
2. For all $x \in \{0,1\}^t$, $\Gamma_0(x) \neq \Gamma_1(x)$ or $\overline{\Gamma_1(x)}$.

The first property is similar to one of the design criteria for the AES S-box [2]. This property ensures that Γ_i have no fixed points or complementary fixed point. On the other hand, the second property ensures that Γ_0 and Γ_1 do not agree upon any input; and also the output of one on any input cannot be obtained by complementing the output of the other on the same input. The formulation of these properties are motivated by the desire to avoid the situation where 0 and -1 are fixed points of both Σ_0 and Σ_1.

Achieving the above properties requires a bit of linear algebra. Suppose, we define $\Gamma_i(x) = \Sigma_i(x) \oplus \mathbf{b}_i$, where $\mathbf{b}_0, \mathbf{b}_1$ are to be chosen such that the above two conditions are satisfied. Additionally, we would like each of $\mathbf{b}_0$, $\mathbf{b}_1$ and $\mathbf{b}_0 \oplus \mathbf{b}_1$ to be either balanced or nearly balanced.

Consider the first point for SHA-256: $\Gamma_i(x) = x$ for some x implies $(\Sigma_i \oplus I_{32})x = \mathbf{b}_i$ and $\Gamma_i(x) = \overline{x}$ for some x implies $\Gamma_i(x) = (\Sigma_i \oplus I_{32})x = \overline{\mathbf{b}_i}$. In other words, the first point holds if both $\mathbf{b}_i$ and $\overline{\mathbf{b}_i}$ are not in the column space of $(\Sigma_i \oplus I_{32})$. In a similar manner, it can be shown that the second point holds if both $\mathbf{b}_0 \oplus \mathbf{b}_1$ and $\overline{\mathbf{b}_0 \oplus \mathbf{b}_1}$ are not in the column space of $(\Sigma_0 \oplus \Sigma_1)$.

We computed many choices of $\mathbf{b}_0$ and $\mathbf{b}_1$ satisfying these constraints. Examples for SHA-256 are

$$\mathbf{b}_0 = \texttt{0xdcb2344c} \text{ and } \mathbf{b}_1 = \texttt{0x9b097671}.$$

For SHA-512, examples are

$$\mathbf{b}_0 = \texttt{0x1762e66a04d6be32} \text{ and } \mathbf{b}_1 = \texttt{0x12135c7549e2fcdd}.$$

4.2 Mix of $+$ and $\oplus$

In the design of the SHA-2 round function, the XOR operation is used a relatively lesser number of times compared to modular addition. A better mix of $+$ and $\oplus$ can be obtained by using $+$ to add W_i to obtain a_i and using $\oplus$ to XOR W_i to obtain e_i. This will mean that if we work with XOR differentials, then it will be difficult to analyze the XOR differentials of the a-register; on the other hand, if we work with modular differentials, then it will be difficult to analyze the modular differentials of the e-register.

A property of the SHA-2 round function is that it is easy to fix $a_{i-4}, \ldots, a_i$ (using words $W_{i-4}, \ldots, W_i$) to simple values and ensure that e_i is also fixed to a simple value. This is because of the CDE (3). An example of the simplification that is possible using the CDE is when $a_{i-1} = a_{i-2} = 0$. In this case, $e_i = a_i + a_{i-4}$, which is a rather simple relation.

To remove the above issues, we suggest the update functions to be the following.

$$\left.\begin{aligned}
a_i &= h_{i-1} \oplus (\Gamma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + K_i + W_i) \\
b_i &= a_{i-1} \\
c_i &= b_{i-1} \\
d_i &= c_{i-1} \\
e_i &= (d_{i-1} + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i) \oplus W_i \\
f_i &= e_{i-1} \\
g_i &= f_{i-1} \\
h_i &= g_{i-1}.
\end{aligned}\right\} \qquad (4)$$

Note that the term $T_{i-1} = \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + K_i$ is present in the computation of both a_i and e_i. This common sub-expression need to be computed only once for each step. In the SHA-2 compression function, the term $\Sigma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i + W_i$ is common to both a_i and e_i. Computing Γ requires one extra t-bit XOR operation, where $t = 32$ for SHA-256 and $t = 64$ for SHA-512. By our estimate, the round function given in (4) requires 6 extra t-bit operations when compared to the SHA-2 round function. We expect this to have insignificant effect on the efficiency. The actual efficiency depends on a large number of parameters such as cache size, instruction pipelining, number of processor cores, etcetera.

New Cross-Dependence Relation. We have

$$\begin{aligned} T_{i-1} &= (a_i \oplus h_{i-1}) - (\Gamma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) - W_i) \\ &= (e_i \oplus W_i) - (d_{i-1} + h_{i-1}). \end{aligned}$$

Consequently the new cross-dependence relation is the following.

$$e_i = W_i \oplus ((a_i \oplus h_{i-1}) + (d_{i-1} + h_{i-1}) - (\Gamma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) - W_i)) \tag{5}$$

The dependences on both W_i and h_{i-1} do not directly cancel out. As a result, it is impossible to express e_i solely in terms of a_{i-4} to a_i. Going back to the previous example, if $a_{i-1} = a_{i-2} = 0$ (note $b_{i-1} = a_{i-2}$) then $e_i = W_i \oplus ((a_i \oplus h_{i-1}) + (d_{i-1} + h_{i-1} + W_i - \mathbf{b}_0))$ which has two alternations of $+$ and $\oplus$ and is a more complicated relation compared to the simple $e_i = a_i + a_{i-4}$ that is obtained for the SHA-2 compression function.

Resistance to Linear Attacks. Local collisions using linear approximations involve two steps. In the first step, all additions are replaced by XORs and in the second step, the functions f_{MAJ} and f_{IF} are replaced by suitable linear approximations [5, 13]. Then one considers the differential behaviour of the resulting linear function and tries to obtain a local collision for the linearized version of the round function.

Let $\ell_1(x, y, z)$ and $\ell_2(x, y, z)$ be linear approximations of $f_{MAJ}(x, y, z)$ and $f_{IF}(x, y, z)$. Further, suppose all additions are replaced by XORs in the SHA-2 round function. Define $\Delta a_i = a'_i \oplus a_i$, where a'_i and a_i correspond to two different messages. Similarly, define Δ of the other registers. Then for the SHA-2 round function, we have,

$$\Delta a_i = \Sigma_0(\Delta a_{i-1}) \oplus \ell_1(\Delta a_{i-1}, b_{i-1}, c_{i-1}) \oplus \Sigma_1(\Delta e_{i-1}) \oplus \ell_2(\Delta e_{i-1}, f_{i-1}, g_{i-1}) \oplus \Delta h_{i-1} \oplus \Delta W_i; \tag{6}$$
$$\Delta e_i = \Delta d_{i-1} \oplus \Sigma_1(\Delta e_{i-1}) \oplus \ell_2(\Delta e_{i-1}, f_{i-1}, g_{i-1}) \oplus \Delta h_{i-1} \oplus \Delta W_i. \tag{7}$$

For the round function shown in (4), we have

$$\begin{aligned} \Gamma_0(a'_{i-1}) \oplus \Gamma_0(a_{i-1}) &= \Sigma_0(a'_{i-1}) \oplus \mathbf{b}_0 \oplus \Sigma_0(a_{i-1}) \oplus \mathbf{b}_0 \\ &= \Sigma_0(a'_{i-1}) \oplus \Sigma_0(a_{i-1}) \\ &= \Sigma_0(a'_{i-1} \oplus a_{i-1}) \\ &= \Sigma_0(\Delta a_{i-1}). \end{aligned}$$

A similar calculation shows that $\Gamma_1(e'_{i-1}) \oplus \Gamma_1(e_{i-1}) = \Sigma_1(\Delta e_{i-1})$. Now, it is easy to see that the expression for Δa_i and Δe_i for the round function shown in (4) are exactly those given by (6). As a result, the entire analysis of linear approximations [5, 13] done for SHA-2 compression function apply without any change to the new suggestion. In particular, this shows that the best linear local collision holds with probability 2^{-39} for the new suggestion as for the SHA-256 compression function.

Resistance to Non-Linear Attacks. In the SHA-2 round function, we have

$$\delta e_i = \delta \Sigma_1(e_{i-1}) + \delta f_{IF}(\delta e_{i-1}, \delta f_{i-1}, \delta g_{i-1}) + \delta d_{i-1} + \delta h_{i-1} + \delta W_i$$
$$\delta a_i = \delta \Sigma_0(a_{i-1}) + \delta f_{MAJ}(\delta a_{i-1}, \delta b_{i-1}, \delta c_{i-1}) + \delta e_i - \delta d_{i-1}.$$

As a result, introducing a message difference of w at the i-th step, i.e., $\delta W_i = w$, makes $\delta a_i = \delta e_i = w$. This is the starting step of the NB and later attacks. The next few message differences are used to cancel out the difference introduced in a_i and e_i. Let us now consider the differential form for a and e registers using the new suggestion.

$$\begin{aligned}
\delta a_i &= a'_i - a_i \\
&= (h'_{i-1} \oplus (\Gamma_0(a'_{i-1}) + f_{MAJ}(a'_{i-1}, b'_{i-1}, c'_{i-1}) + \Gamma_1(e'_{i-1}) + f_{IF}(e'_{i-1}, f'_{i-1}, g'_{i-1}) + K_i + W'_i)) \\
&\quad -(h_{i-1} \oplus (\Gamma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + K_i + W_i)) \\
\delta e_i &= e'_i - e_i \\
&= ((d'_{i-1} + \Gamma_1(e'_{i-1}) + f_{IF}(e'_{i-1}, f'_{i-1}, g'_{i-1}) + h'_{i-1} + K_i) \oplus W'_i) \\
&\quad -((d_{i-1} + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i) \oplus W_i).
\end{aligned}$$

Suppose that we now introduce an additive difference of w in δW_i, i.e., $\delta W_i = W'_i - W_i = w$. Then

$$\delta a_i = (c_1 \oplus (c_2 + W_i + w)) - (c_1 \oplus (c_2 + W_i)) \neq (c_3 \oplus (W_i + w)) - (c_3 \oplus W_i) = \delta e_i$$

where

$$\begin{aligned}
c_1 &= h_{i-1}, \\
c_2 &= \Gamma_0(a_{i-1}) + f_{MAJ}(a_{i-1}, b_{i-1}, c_{i-1}) + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + K_i, \\
c_3 &= d_{i-1} + \Gamma_1(e_{i-1}) + f_{IF}(e_{i-1}, f_{i-1}, g_{i-1}) + h_{i-1} + K_i.
\end{aligned}$$

Since i is the first place where the perturbation is introduced, the quantities corresponding to primed and unprimed variables are same for $(i-1)$st step, i.e., the quantities c_1, c_2 and c_3 take the same value for both primed and unprimed values.

One can notice that for the new suggestion, the expressions for δa_i and δe_i are not as simple as that for SHA-2 compression functions. Introducing a perturbation of w through W_i does not introduce the same perturbation in the a and e registers and neither of these perturbations is equal to w. The analysis of a few more steps does not provide any means to cancel out the perturbations of δa_i and δe_i. Consequently, there is no way to apply the NB type attack on this round function. This is an improvement over the SHA-2 round function.

5 Multiple Feed Forward: A New Design Construct

We introduce the idea of multiple feed-forward, i.e., feed-forward at several places. This provides a possible additional layer of resistance against perturbation-correction attacks.

The computation starts with a C (the initial value of which is IV) and suppose that the output of step i is $S^{(i)}$. The SHA-2 compression function employs a single feed-forward, i.e., the output of H is $C + S^{(N-1)}$ where $N = 64$ for SHA-256 and $N = 80$ for SHA-512.

Let $t_1 = 7$, $t_2 = 14$ and $s = 16$. We introduce new feed-forwards at steps $t_1 + s \times k_1$ and $t_2 + s \times k_2$, where k_1, k_2 are positive integers. Let $S^{(-1)} = C$. We use two sets of temporary registers T_1 and T_2, i.e., each T_i consists of $n = 8$ t-bit words, where $t = 32$ for SHA-256 and $t = 64$ for SHA-512.

The algorithm for the compression function is shown in Figure 1. In the figure assume for the moment that in the input $T_1 = T_2 = 0^{nt}$ and the output does not contain T_1 and T_2. The general description allows the introduction of feed-forward across message blocks as explained in the next section.

Fig. 1. Modified compression function with two feed-forward threads. Here $t_1 = 7$, $t_2 = 14$ and $s = 16$; G_i is the step function.

$\mathsf{Compress}(\mathsf{reg}, T_1, T_2, W)$

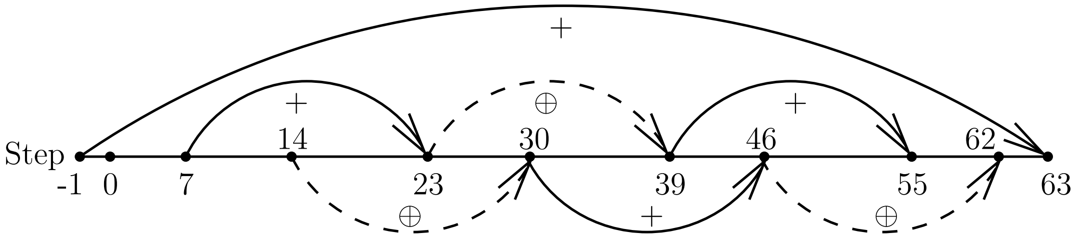
parse W into 16 t-bit words $W_0, \ldots, W_{15}$;
$(W_0, \ldots, W_{N-1}) = \mathsf{msgExpn}(W_0, \ldots, W_{15})$;
$S^{-1} = \mathsf{reg}$;
for $i = 0, \ldots, 15$ do
    $S^{(i)} \leftarrow G_i(S^{(i-1)}, W_i)$;
    if $(i = 7)$ then $T_1 \leftarrow T_1 \oplus S^{(i)}$; if $(i = 14)$ then $T_2 \leftarrow T_2 + S^{(i)}$;
end for;
for $i = 16, \ldots, N-1$ do
    $S^{(i)} = G_i(S^{(i-1)}, W_i)$;
    if $((i - t_1) \bmod s = 0)$ then
        if $((i - t_1) \bmod 2s = 0)$ then
            $S^{(i)} \leftarrow S^{(i)} \oplus T_1$; $T_1 = S^{(i)}$;
        else
            $S^{(i)} \leftarrow S^{(i)} + T_1$; $T_1 = S^{(i)}$;
    if $((i - t_2) \bmod s = 0)$ then
        if $((i - t_2) \bmod 2s = 0)$ then
            $S^{(i)} \leftarrow S^{(i)} + T_2$; $T_2 = S^{(i)}$;
        else
            $S^{(i)} \leftarrow S^{(i)} \oplus T_2$; $T_2 = S^{(i)}$;
end for;
output $(S^{(N-1)} + \mathsf{reg}, T_1, T_2)$.

The following points are to be noted.

1. The feed-forward with C at Step $N - 1$ remains unchanged.
2. There are two feed-forward threads starting at t_1 and t_2. Both threads alternate the type of feed-forward between $+$ and $\oplus$. The first thread starts with $+$, while the second thread starts with $\oplus$.
3. The choice of t_1 and t_2 ensures that if we take any interval of length 9 in the range 0 to 15, then at least one of t_1 or t_2 will lie properly within the interval.
4. The choices of t_1, t_2 and s ensure that the sets $\{t_1 + s \times k_1 : k_1 \geq 1\}$ and $\{t_2 + s \times k_2 : k_2 \geq 1\}$ are disjoint. (A sufficient condition for this is to have s to be co-prime to $(t_1 - t_2)$.) So, at no step will two feed-forward threads starting at t_1 and t_2 meet.
5. New feed-forward is not introduced at Step $(N - 1)$. Again, the choices of t_1, t_2 and s ensure this.
6. The amortized cost of the new feed-forwards is less than one t-bit operation (addition or XOR) per step, where t is 32 for SHA-256 and 64 for SHA-512. In an efficient implementation with loop unrolling, the conditional statements will not be required. The temporary registers T_1 and T_2 will simply be added or XORed to the internal registers at the appropriate step.

For SHA-256, the line starting at $t_1 = 7$ introduces feed-forwards at Steps 23, 39 and 55 and the line starting at $t_2 = 14$ introduces feed-forwards at Steps 30, 46 and 62. Additive feed-forward is used at Steps 23 and 55 due to the first line and at Step 46 due to the second line. XOR feed-forward is used at Step 39 due to the first line and at Steps 30 and 62 due to the second line. There is a total of 6 feed-forward steps requiring a total of $6 \times 8 = 48$ 32-bit add/XOR operations. Amortized over the 64 steps, the cost is 3/4th 32-bit operation per step. The feed-forwards for SHA-256 are illustrated in Figure 2.

Fig. 2. Illustration of multiple feed-forward for SHA-256. There are two threads of feed-forwards, based on XOR and modular addition. The original additive feed-forward from the input (marked as Step -1) to Step 63 is also retained.



For SHA-512, the feed-forwards of the first line occur at Steps 23, 39, 55 and 71 with additive feed-forwards at Steps 23 and 55 and XOR feed-forwards at Steps 39 and 71. The second line of feed-forwards is introduced at Steps 30, 46, 62 and 78, with XOR at Steps 30 and 62 and addition at Steps 46 and 78. Amortized over the 80 steps, the cost is 4/5th 64-bit operation (add/XOR) per step.

The idea of multiple feed-forward is generic and can be used with other hash designs. Depending on the number of registers, the number of free words and the total number of steps N, the value of t_1, t_2 and s has to be chosen appropriately so as to ensure the properties described above.

Resistance to Perturbation-Correction Attacks. The basic idea of such an attack has been discussed earlier. By our choice of t_1 and t_2, at least one of these will be properly contained in any 9-step local collision. Consequently, the registers at the corresponding step will have a difference. Due to feed-forwards, this difference will be pushed out into the steps where the message words are defined through message recursion. Since these words cannot be directly controlled, it will be difficult to cancel out the introduced perturbation at these steps. The overall effect will be that the local collision based perturbation-correction attack will become substantially more difficult to apply.

A Possible Drawback. There is a possibility that the idea of multiple feed-forward can be used to attack the design. The feed-forward steps may be seen as providing possible control in the message expansion region which can in fact be used to correct perturbations in the higher rounds. However, we could not actually see how this could be done. At this point of time, we think multiple feed-forwards actually create perturbations in the message expansion region rather than cancel them.

6 Feed-Forward Across Message Blocks

The idea of feed-forward can be extended to different message blocks. Let us go back to the construction in Section 5. After the first message block has been processed, three quantities are produced, the chaining variable C and the quantities T_1 and T_2. But, T_1 and T_2 are not used further. These two quantities are "lightweight" digests of the first message block. We suggest that these be used in the processing of the second message block.

The processing of the second block starts with C as the IV and by the construction of Section 5, the output of Step 7 is taken to be new T_1 and the output of Step 14 is taken to be the new T_2. We modify this as follows. The new T_1 is the XOR of the old T_1 and the output of Step 7; and the new T_2 is the sum of the old T_2 and the output of Step 14. The rest of the two feed-forward threads are as before. Figure 1 shows the complete description. Note that at a general level, this idea of feed-forward across message blocks is similar to the wide-pipe design strategy introduced in [9].

The modification can still be considered to be within the MD framework. Let $M^{(0)}, \ldots, M^{(\ell)}$ be the message blocks (including padding with length). Let $C^{(-1)} = \mathsf{IV}$ and $T_1^{(-1)} = T_2^{(-1)} = 0^{nt}$. There are ℓ invocations of the compression function, where the $(i+1)$st invocation of the compression function takes $(C^{(i)}, T_1^{(i)}, T_2^{(i)}, M^{(i+1)})$ as input and produces a $(C^{(i+1)}, T_1^{(i+1)}, T_2^{(i+1)})$ as output. Here $C^{(i)}, C^{(i+1)}$ are the chaining variables; and $T_1^{(i)}, T_2^{(i)}, T_1^{(i+1)}, T_2^{(i+1)}$ are the feed-forward quantities.

Viewed in this way, it is easy to prove as usual by backward induction that a collision for the hash function leads to a collision for the compression function. Finding a pseudo collision for the compression function defined in this manner may be easier than the compression function where there is no feed-forward across message blocks. This is because one may choose T_1 and T_2 as suitable values. However, the hashing starts with $C^{(-1)} = \mathsf{IV}$ and $T_1^{(-1)} = T_2^{(-1)} = 0^{nt}$, which fixes the initial choice of T_1 and T_2.

Resistance to Multi-Collision Attacks. Let us now consider the effect of multi-collision attacks [8]. This is a generic technique which applies to the MD type construction. Using r invocations of generic collision finding algorithm on the compression function, it is possible to construct 2^r messages which map to the same value. Suppose M_1, M_1' are two distinct message blocks which (starting from IV) map to the same value C_1; and M_2, M_2' are two distinct message blocks which starting from C_1 map to the same value C_2. Then the four messages (M_1, M_2), (M_1, M_2'), (M_1', M_2) and (M_1', M_2') map to the same value C_2. Since the output of the compression function consists of nt bits, the generic algorithm will require $2 \times 2^{nt/2}$ invocations of the compression function to find a collision.

Suppose we apply this to the new construction. The output of the compression function is the chaining variable C as well as T_1 and T_2. Then we need distinct M_1, M_1' which starting from IV and $T_1 = T_2 = 0^{nt}$ map to same value $C^{(1)}, T_1^{(1)}, T_2^{(1)}$; and distinct M_2, M_2' which starting from $C^{(1)}, T_1^{(1)}, T_2^{(1)}$ map to the same value $C^{(2)}, T_1^{(2)}, T_2^{(2)}$. Then as before, we will have four messages which map to the same value $C^{(2)}, T_1^{(2)}, T_2^{(2)}$. The advantage here is that the generic collision finding algorithm now needs to be applied to a compression function whose output is $3nt$ bits. As a result, the generic algorithm will require $2 \times 2^{3nt/2}$ invocations of the new compression function to find a collision. This is more than 2^n invocations and hence is not useful. (If a hash function produces n-bit digests, then one can *surely* obtain a collision by applying the hash function to $2^n + 1$ distinct inputs; and one can obtain a collision with *high probability* by applying the hash function to $2^{n/2}$ inputs.)

On the other hand, suppose that M_1, M_1' are such that starting from IV and $T_1 = T_2 = 0^{nt}$, they produce the same value for $C^{(1)}$ but not necessarily the same value for $T_1^{(1)}, T_2^{(1)}$, i.e., $(T_1^{(1)}, T_2^{(1)}) \neq ((T_1^{(1)})', (T_2^{(1)})')$. Further, suppose that M_2, M_2' are such that M_2 starting from $C^{(1)}$ and $T_1^{(1)}, T_2^{(1)}$ produces the same $C^{(2)}$ that M_2' starting from $C^{(1)}$ and $(T_1^{(1)})', (T_2^{(1)})'$ produces. This results in single two-block collision between (M_1, M_1') and (M_2, M_2'). But, now (M_1, M_2) does not produce the same value as (M_1, M_2'). More generally, no two of the four possible messages produce the same value. This is due to the difference in the intermediate feed-forward values, i.e., $(T_1^{(1)}, T_2^{(1)}) \neq ((T_1^{(1)})', (T_2^{(1)})')$.

To summarize, in a manner similar to the wide-pipe design strategy [9], extending feed-forward across message blocks provides resistance to generic multi-collision attacks.

7 Design Specification and Implemenation

We describe the complete hash algorithm, which we call SShash. There are two main variants – SShash-256 and SShash-512 with truncated versions SShash-224 and SShash-384 being obtained as in SHA-2 [19].

Suppose M is a message to be hashed. We consider M to be a binary string of length λ and this string is padded as in SHA-2, i.e., append the bit 1 to the message, followed by k zeros where k is the smallest non-negative integer such that $\lambda + 1 + k \equiv 14t \bmod 16t$, with $t = 32$ for SShash-256 and $t = 64$ for SShash-512. Then the $2t$-bit binary representation of λ is appended. This makes the total length of the padded message to be a multiple of $2t$. The padded message is parsed into q $16t$-bit blocks $M^{(1)}, \ldots, M^{(q)}$.

The algorithm starts with 8 t-bit registers called the initialization vector IV; the IV for SShash-256 is same as the IV for SHA-256 and the IV for SShash-512 is same as the IV for SHA-512. A compression function Compress is used to process the message blocks and produce the digest in the following manner.

SShash.
1. $\mathsf{reg} = \mathsf{IV}$; $T_1 = T_2 = 0^{8t}$;
2. for $i = 1$ to q do $(\mathsf{reg}, T_1, T_2) = \mathsf{Compress}(\mathsf{reg}, T_1, T_2, M^{(i)})$;
4. return reg;

The function Compress is shown in Figure 1. It takes as input 8 t-bit registers and a $16t$-bit message block. The message block is parsed into 16 t-bit words $W_0, \ldots W_{15}$. These words are then expanded into N t-bit words $W_0, \ldots, W_{N-1}$ using msgExpn. The message expansion for SShash-256 is the same as that for SHA-256 and the message expansion for SShash-512 is the same as that for SHA-512. In particular, the value of N is 64 for SShash-256 and 80 for SShash-512. (Note. In the above q refers to the number of message blocks, while N refers to the number of t-bit words after the expansion of a single message block.)

The step function G is applied N times. This function G takes as input 8 t-bit registers $(a_{i-1}, \ldots, h_{i-1})$ and updates them using W_i to obtain $(a_i, \ldots, h_i)$. This updation is done as shown in (4). These updations require the values of the t-bit constants $K_0, \ldots, K_{N-1}$. Again, these constant values are as specified in SHA-2.

The above completes the description of the modified algorithm SShash. We have implemented both SShash-256 and SShash-512. For this implementation, we modified the SHA-2 code available at [3]. Macros for the compression function of SShash-256 is given in Appendix A. The code for SShash-512 is similar and is not shown. Appendix A also provides test vectors for SShash.

Speed comparison of SShash with SHA-2 is shown in Table 2. The obtained speeds correspond with our earlier stated intuition that the suggested modifications of SHA-2 to obtain SShash do not affect the efficiency too much.

8 Conclusions

In this work, we studied several properties of the SHA-2 design which were used in recent collision attacks against reduced SHA-2. We have suggested modifications to the SHA-2 design so as to make these attacks inapplicable. The modified SHA-2 design is almost as efficient as the original one.

Table 2. Speed of SShash compared to SHA-2. For SHA-2, the implementation from [3] has been used and SShash implementations have been obtained by modifying the SHA-2 implementations of [3]. The speed measurements are given in cycles/byte and were obtained on a dual-core Pentium having two CPUs at 2.2 GHz and running Fedora.

SHA-256	SShash-256	SHA-512	SShash-512
19.3	22	13.6	14.0

We provided a generic construction of multiple feed-forward. This can be used to strengthen a design against perturbation-correction collision finding attacks. Further, the idea of feed-forward over several message blocks is suggested and shown to provide resistance to multi-collision attacks.

Acknowledgements

The authors wish to thank anonymous reviewers of ASIACCS 2009 for giving useful suggestions.

References

1. Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO 1998, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.
2. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
3. Wei Dai. Crypto++ Library 5.5.2. `http://www.cryptopp.com/`.
4. Federal Register Vol. 72, No. 212. *Announcing Request for Candidate Algorithm Nominations for a new Cryptographic Hash Algorithm (SHA-3) Family*. U.S. Department of Commerce, National Institute of Standards and Technology(NIST), November 2, 2007. Available at `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`.
5. Henri Gilbert and Helena Handschuh. Security Analysis of SHA-256 and Sisters. In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography, 10th Annual International Workshop, SAC 2003, Ottawa, Canada, August 14-15, 2003, Revised Papers*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2003.
6. Philip Hawkes, Michael Paddon, and Gregory G. Rose. On Corrective Patterns for the SHA-2 Family. *Cryptology eprint Archive*, August 2004. Available at `http://eprint.iacr.org/2004/207`.
7. Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In *Selected Areas in Cryptography, 15th Annual International Workshop, SAC 2008, Revised Papers*, 2008. To appear.
8. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.
9. Stefan Lucks. A failure-friendly design principle for hash functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.
10. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2006.
11. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. *Cryptology eprint Archive*, March 2008. Available at `http://eprint.iacr.org/2008/130`.
12. Ivica Nikolić and Alex Biryukov. Collisions for Step-Reduced SHA-256. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, March 26-28, 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.
13. Somitra Kumar Sanadhya and Palash Sarkar. New Local Collisions for the SHA-2 Hash Family. In Kil-Hyun Nam and Gwangsoo Rhee, editors, *Information Security and Cryptology - ICISC 2007, 10th International Conference, Seoul, Korea, November 29-30, 2007, Proceedings*, volume 4817 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2007.

14. Somitra Kumar Sanadhya and Palash Sarkar. A Combinatorial Analysis of Recent Attacks on Step Reduced SHA-2 Family. Cryptology ePrint Archive, Report 2008/271, 2008. http://eprint.iacr.org/2008/271.
15. Somitra Kumar Sanadhya and Palash Sarkar. Attacking Reduced Round SHA-256. In Steven Bellovin and Rosario Gennaro, editors, *Applied Cryptography and Network Security - ACNS 2008, 6th International Conference, New York, NY, June 03-06, 2008, Proceedings*, volume 5037 of *Lecture Notes in Computer Science*. Springer, 2008.
16. Somitra Kumar Sanadhya and Palash Sarkar. Deterministic Constructions of 21-Step Collisions for the SHA-2 Hash Family. In Editors, editor, *Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, September 2008, Proceedings*, volume 5222 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2008.
17. Somitra Kumar Sanadhya and Palash Sarkar. New Collision Attacks Against Up To 24-step SHA-2 . In D.R. Chowdhury, V. Rijmen, and A. Das, editors, *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India*, volume 5365 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2008.
18. Somitra Kumar Sanadhya and Palash Sarkar. Non-Linear Reduced Round Attacks Against SHA-2 Hash family. In Yi Mu and Willy Susilo, editors, *Information Security and Privacy - ACISP 2008, The 13th Australasian Conference, Wollongong, Australia, 7-9 July 2008, Proceedings*, volume 5107 of *Lecture Notes in Computer Science*. Springer, 2008.
19. Secure Hash Standard. *Federal Information Processing Standard Publication 180-2*. U.S. Department of Commerce, National Institute of Standards and Technology(NIST), 2002. Available at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.
20. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

A Compression Function for SShash-256

The required macros and the description of the compression function for SShash-256 are provided below. Corresponding macros for SShash-512 are similar and hence not provided. Test vectors for SShash-256 and SShash-512 are given in Table 3.

Table 3. Test vectors for SShash. The given digests are the outputs of SShash on the string "aaa".

SShash-256	0x27ef472acd480e556be88c4b320008b278d1819fe297abdd97ed947a295e3eb4
SShash-512	0x47cd2dafcc070c317d242d027a2b3ec65345065dadbeff05cf88745a2a759c8f ff67b38965dbd9fbc15280fe41415b8364fe8f46baf9dc60a5173912480ce916

```
#define word32 unsigned int

#define ROTR(x, n) (((x) >> (n)) | ((x) << (32-(n))))
#define SHR(x, n)   ( (x) >> (n) )

#define NEW_SIG_0(x) ( ( (ROTR(x,2)) ^ (ROTR(x,13)) ^ (ROTR(x,22)) ) ^ 0xdcb2344c )
#define NEW_SIG_1(x) ( ( (ROTR(x,6)) ^ (ROTR(x,11)) ^ (ROTR(x,25)) ) ^ 0x9b097671 )

#define sig_0(x) ( (ROTR(x,7)) ^ (ROTR(x,18)) ^ (SHR(x,3)) )
#define sig_1(x) ( (ROTR(x,17)) ^ (ROTR(x,19)) ^ (SHR(x,10)) )

#define CH(x,y,z) ((z)^((x)&((y)^(z))))
#define MAJ(x,y,z) (((x)&(y))|((z)&((x)|(y))))
```

```
#define a1(i) T[(0-i)&7]
#define b1(i) T[(1-i)&7]
#define c1(i) T[(2-i)&7]
#define d1(i) T[(3-i)&7]
#define e1(i) T[(4-i)&7]
#define f1(i) T[(5-i)&7]
#define g1(i) T[(6-i)&7]
#define h1(i) T[(7-i)&7]


#define FF_XOR_START(T) { \
 T[0]^=a1(i); T[1]^=b1(i); T[2]^=c1(i); T[3]^=d1(i); \
 T[4]^=e1(i); T[5]^=f1(i); T[6]^=g1(i); T[7]^=h1(i); \
}

#define FF_ADD_START(T) { \
 T[0]+=a1(i); T[1]+=b1(i); T[2]+=c1(i); T[3]+=d1(i); \
 T[4]+=e1(i); T[5]+=f1(i); T[6]+=g1(i); T[7]+=h1(i); \
}

#define FF_ADD(T) {\
 a1(i) +=T[0]; b1(i) +=T[1]; c1(i) +=T[2]; d1(i) +=T[3]; \
 e1(i) +=T[4]; f1(i) +=T[5]; g1(i) +=T[6]; h1(i) +=T[7]; \
 T[0]=a1(i); T[1]=b1(i); T[2]=c1(i); T[3]=d1(i); T[4]=e1(i); \
 T[5]=f1(i); T[6]=g1(i); T[7]=h1(i); \
}

#define FF_XOR(T) { \
 a1(i) ^=T[0]; b1(i) ^=T[1]; c1(i) ^=T[2]; d1(i) ^=T[3]; \
 e1(i) ^=T[4]; f1(i) ^=T[5]; g1(i) ^=T[6]; h1(i) ^=T[7]; \
 T[0]=a1(i); T[1]=b1(i); T[2]=c1(i); T[3]=d1(i); T[4]=e1(i); \
 T[5]=f1(i); T[6]=g1(i); T[7]=h1(i); \
}

#define FF_FINAL(REG) { \
 REG[7] += h1(0); REG[6] += g1(0); REG[5] += f1(0); REG[4] += e1(0); \
 REG[3] += d1(0); REG[2] += c1(0); REG[1] += b1(0); REG[0] += a1(0);  \
}

#define R_0(i) { \
 temp1 = NEW_SIG_1(e1(i))+CH(e1(i),f1(i),g1(i))+K[i]; \
 temp2 = NEW_SIG_0(a1(i)) + MAJ(a1(i),b1(i),c1(i)); \
 d1(i) += temp1 + h1(i); d1(i) ^= data[i]; \
 h1(i) ^= temp1 + temp2 + data[i]; \
}
```

```
#define R_1(i) { \
 temp1 = NEW_SIG_1(e1(i))+CH(e1(i),f1(i),g1(i))+K[i+16]; \
 temp2 = NEW_SIG_0(a1(i)) + MAJ(a1(i),b1(i),c1(i)); \
 d1(i) += temp1 + h1(i); d1(i) ^= W[i+16]; \
 h1(i) ^= temp1 + temp2 + W[i+16]; \
}

#define R_2(i) { \
 temp1 = NEW_SIG_1(e1(i))+CH(e1(i),f1(i),g1(i))+K[i+32]; \
 temp2 = NEW_SIG_0(a1(i)) + MAJ(a1(i),b1(i),c1(i)); \
 d1(i) += temp1 + h1(i); d1(i) ^= W[i+32]; \
 h1(i) ^= temp1 + temp2 + W[i+32]; \
}

#define R_3(i) { \
 temp1 = NEW_SIG_1(e1(i))+CH(e1(i),f1(i),g1(i))+K[i+48]; \
 temp2 = NEW_SIG_0(a1(i)) + MAJ(a1(i),b1(i),c1(i)); \
 d1(i) += temp1 + h1(i); d1(i) ^= W[i+48]; \
 h1(i) ^= temp1 + temp2 + W[i+48]; \
}

compression_func(word32 *reg, const word32 *data){
  word32 i, W[64], temp1, temp2, T[8];
  static word32 T1[8]={0,0,0,0,0,0,0,0}, T2[8]={0,0,0,0,0,0,0,0};

  memcpy(W,data,64);
  for(i=16; i<64; i++)
    W[i] = sig_1(W[i-2])+W[i-7]+sig_0(W[i-15])+W[i-16];

  memcpy(T, reg, sizeof(T));

  {
    R_0( 0); R_0( 1); R_0( 2); R_0( 3);
    R_0( 4); R_0( 5); R_0( 6); R_0( 7); FF_XOR_START(T1);
    R_0( 8); R_0( 9); R_0(10); R_0(11);
    R_0(12); R_0(13); R_0(14); FF_ADD_START(T2); R_0(15);
  }
  {
    R_1( 0); R_1( 1); R_1( 2); R_1( 3);
    R_1( 4); R_1( 5); R_1( 6); R_1( 7); FF_ADD(T1);
    R_1( 8); R_1( 9); R_1(10); R_1(11);
    R_1(12); R_1(13); R_1(14); FF_XOR(T2); R_1(15);
  }
  {
    R_2( 0); R_2( 1); R_2( 2); R_2( 3);
```

```
    R_2( 4); R_2( 5); R_2( 6); R_2( 7); FF_XOR(T1);
    R_2( 8); R_2( 9); R_2(10); R_2(11);
    R_2(12); R_2(13); R_2(14); FF_ADD(T2); R_2(15);
  }
  {
    R_3( 0); R_3( 1); R_3( 2); R_3( 3);
    R_3( 4); R_3( 5); R_3( 6); R_3( 7); FF_ADD(T1);
    R_3( 8); R_3( 9); R_3(10); R_3(11);
    R_3(12); R_3(13); R_3(14); FF_XOR(T2); R_3(15);
  }
  FF_FINAL(reg);
}
```