SMASH - A Cryptographic Hash Function

Lars R. Knudsen

Department of Mathematics, Technical University of Denmark

Abstract. [1] This paper presents a new hash function design, which is different from the popular designs of the MD4-family. Seen in the light of recent attacks on MD4, MD5, SHA-0, SHA-1, and on RIPEMD, there is a need to consider other hash function design strategies. The paper presents also a concrete hash function design named SMASH. One version has a hash code of 256 bits and appears to be at least as fast as SHA-256.

1 Introduction

A *cryptographic hash function* takes as input a binary string of arbitrary length and returns a binary string of a fixed length. Hash functions which satisfy some security properties are widely used in cryptographic applications such as digital signatures, password protection schemes, and conventional message authentication. In the following let $H : \{0,1\}^* \rightarrow \{0,1\}^n$ denote a hash function which returns a string of length n. Most hash functions in use today are so-called *iterated* hash functions, based on iterating a compression function. Examples of iterated hash functions are MD4[19], MD5[20], SHA[13] and RIPEMD-160[7]. For a cryptographic hash function H, one is interested in the complexity of the following attacks[16]:

- **Collision:** find x and x' such that $x' \neq x$ and $H(x) = H(x')$,
- **2nd preimage:** given an x and $y = H(x)$ find $x' \neq x$ such that $H(x') = y$,
- **Preimage:** given $y = H(x)$, find x' such that $H(x') = y$.

Clearly the existence of a 2nd preimage implies the existence of a collision. In a brute-force attack preimages and 2nd preimages can be found after about 2^n applications of H, and a collision can be found after about $2^{n/2}$ applications of H. It is usually the goal in the design of a cryptographic hash function that no attacks perform better than the brute-force attacks.

Often hash functions define an initial value, iv. The hash is then denoted $H(\mathrm{iv}, \cdot)$ to explicitly denote the dependency on the iv. Attacks like the above, but where the attacker is free to choose the value(s) of the iv are called pseudo-attacks. The following assumptions are well-known and widely used in cryptology (where $\oplus$ is the exclusive-or operation).

Assumption 1 *Let $g : \{0,1\}^n \rightarrow \{0,1\}^n$ be a randomly chosen mapping. Then the complexities of finding a collision, a 2nd preimage and a preimage are of the order $2^{n/2}$, 2^n, respectively 2^n. Let $f : \{0,1\}^n \rightarrow \{0,1\}^n$ be a randomly chosen, bijective mapping. Define the function $h : \{0,1\}^n \rightarrow \{0,1\}^n$ by $h(x) = f(x) \oplus x$ for all x. It is assumed that the expected complexity of finding collisions, 2nd preimages and preimages for h is roughly the same as for g.*

[1] After the presentation of SMASH at FSE 2005, the proposal was broken[15].

Most popular hash functions are based on iterating a compression function, which processes a fixed number of bits. The message to be hashed is split into blocks of a certain length where the last block is possibly padded with extra bits. Let $h : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^n$ denote the compression function, where n and ℓ are positive integers. Let $m = m_0 \mid m_1 \mid \ldots \mid m_t$ be the message to be hashed, where $|m_i| = \ell$ for $0 \leq i \leq t$. Then the hash value is taken as h_t, where

$$h_i = h(h_{i-1}, m_i),$$

for $h_0 = \text{iv}$ an initial, fixed value. The values $\{h_i\}$ are called the chaining variables. If a message m cannot be split into blocks of equal length n, i.e., if the last block consists of less than n bits, then a collision-free padding rule is used. If x and y are two arbitrary different strings, then it must hold that the corresponding padded strings are different.

For iterated hash functions the MD-strengthening (after Merkle [11] and Damgård [6]) is as follows. One fixes the iv of the hash function and appends to a message some additional, fixed number of blocks at the end of the input string containing the length of the original message. Then it can be shown that attacks on the resulting hash function implies a similar attack on the compression function.

There has been much progress in recent years in cryptanalysis of iterated hash functions and attacks have been reported on MD4, MD5, SHA-0, reduced SHA-1 and RIPEMD[2, 18, 21]. For these hash functions and for most other popular iterated hash functions, the compression function takes a rather long message and compresses this together with a shorter chaining variable (containing the internal state) to a new value of the chaining variable. E.g., in SHA-0 and SHA-1 the message is 512 bits and the chaining variable 160 bits. One way of viewing this is, that the compression function defines 2^{160} functions from 512 bits to 160 bits (from message to output), but at the same time it defines 2^{512} functions (bijections) from 160 bits to 160 bits (from chaining variable to output). If just a few of these functions are cryptographically weak, this could give an attacker the open door for an attack.

In this paper we consider compression functions built from one, fixed bijective mapping $f : \{0,1\}^n \to \{0,1\}^n$. A related but different approach is in [17]. In our model this leads to hash functions where the compression functions themselves are not cryptographically strong, thus a result similar to the one by Merkle and Damgård, cf. above, cannot be proved. However, the constructions have other advantages and it is conjectured that the resulting hash functions are not easy to break, despite the fact that the compression functions are "weak".

2 Compression functions from one bijective mapping

Our approach is to build an iterated hash function $H : \{0,1\}^* \to \{0,1\}^n$ from one fixed, bijective mapping $f : \{0,1\}^n \to \{0,1\}^n$. If this bijection is chosen carefully, the goal or hope is that such a hashing construction is hard to attack. Such constructions could potentially be built from using a block cipher with a fixed value of the key.

2.1 Motivation for design

Consider iterated hash functions with compression functions $h : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ for which the computation of chaining variables is as follows: $h_i = h(A, B) = f(A) \oplus B$. Here $f : \{0,1\}^n \rightarrow \{0,1\}^n$ is a bijective mapping and the inverse of f is assumed to be as easy to compute as f itself. A and B are variables which depend on the chaining variable h_{i-1} and on the message block m_i. Ideally we would like to have an efficient (easy-to-compute) transformation $e(h_{i-1}, m_i) = (A, B)$. We do not want e to cause collisions so we require that it is invertible. Since we want e to be an invertible function (very) easy to compute, we shall also assume that the inverse of e is easy to compute.

For such compression functions it is possible to invert also h. Given an h_i, simply choose a random value of B, compute $A = f^{-1}(B \oplus h_i)$, then by inverting e, find (h_{i-1}, m_i) which hash to h_i. We shall assume that the complexity of one application of e is small compared to one application of f and thus that inverting h takes roughly time one, where one unit is one application of f (or its inverse). It follows that it is easy to find both collisions and preimages for the compression function. Next we examine what this means for similar attacks on the hash functions (where a fixed value of h_0 is assumed) induced by these compression functions.

Inverting h as above enables a (2nd) preimage attack on H by a meet-in-the-middle approach[10] of complexity about $2^{n/2+1}$, i.e., compute the values of "h_{i-1}" for $2^{n/2}$ messages (of each $i-1$ blocks) and store them. For a fixed value of h_i choose $2^{n/2}$ random values of A, B (as above), which yield $2^{n/2}$ "random" values of "h_{i-1}". The birthday paradox gives the (2nd) preimage. If f is a truly randomly chosen bijection on n bits (which is the aim for it to be) then this (2nd) preimage attack is always possible on the constructions we are considering. So the best we can do regarding (2nd) preimages is try to make sure that the attacker does not have full control over the message blocks when inverting h, in which case such preimages may be of lesser use in practice. Thus, we want to avoid that given (any) h_{i-1}, m_i (and thereby h_i) and m'_i, it is easy to find h'_{i-1} such that $(h_{i-1}, m_i) \neq (h'_{i-1}, m'_i)$ and $h_i = h'_i$, since in this case one can find preimages for the hash function for meaningful messages also in time roughly $2^{n/2}$.

This meet-in-the-middle attack is "irrelevant" regarding collisions, since the complexity of a brute-force attack is $2^{n/2}$ regardless of the nature of the compression function. For collisions it is important that when inverting h the attacker does not have full control over the chaining variable(s) h_{i-1}. If given (any) h_{i-1}, h'_{i-1}, it is easy to find m_i, m'_i such that $(h_{i-1}, m_i) \neq (h'_{i-1}, m'_i)$ and $h_i = h'_i$ then one can find a collision easily also for the hash function. Simply choose two messages $m = m_1, \ldots, m_{i-1}$ and $m' = m'_1, \ldots, m'_{i-1}$ (e.g., with $h_{i-1} \neq h'_{i-1}$), where $i \geq 2$, then the two i-block messages $M = m \mid m_i$ and $M' = m' \mid m'_i$ yield a collision for the hash function.

The above is the motivation for examining the compression functions with respect to the following two attacks:

- I: Given h_{i-1}, h'_{i-1} find m_i, m'_i such that $(h_{i-1}, m_i) \neq (h'_{i-1}, m'_i)$ and $h_i = h'_i$.

– II: Given h_{i-1}, m_i and m'_i, find h'_{i-1} such that $(h_{i-1}, m_i) \neq (h'_{i-1}, m'_i)$ and $h_i = h'_i$.

Consider first the simple e-functions where $A, B \in \{m_i, h_{i-1}, m_i \oplus h_{i-1}\}$. With the requirements for e above, this yields six possibilities for the compression function, see the first column in Table 1. It follows that in all six cases either the first or the second attack is easy to implement, in some cases both. So one needs to consider more complex e-functions to achieve better resistance against the two attacks. There may be many possible ways to build such functions; we believe to have found a simple one.

Scheme	Attack I	Attack II
$h_i = f(h_{i-1}) \oplus m_i$	easy	easy
$h_i = f(m_i) \oplus h_{i-1}$	easy	easy
$h_i = f(h_{i-1}) \oplus m_i \oplus h_{i-1}$	easy	?
$h_i = f(m_i) \oplus m_i \oplus h_{i-1}$	?	easy
$h_i = f(h_{i-1} \oplus m_i) \oplus h_{i-1}$	easy	?
$h_i = f(h_{i-1} \oplus m_i) \oplus m_i$	?	easy

Table 1. Six compression functions.

First we note that there is a natural one-to-one correspondence between bit vectors of length s and elements in the finite field of 2^s elements. We introduce "multiplication by θ" as follows.

Definition 1. *Consider $a \in GF(2)^s$. Let θ be an element of $GF(2^s)$ such that $\theta \notin \{0, 1\}$. Define the multiplication of a by θ as follows. View a as an element of $GF(2^s)$, compute $a\theta$ in $GF(2^s)$, then view the result as an s-bit vector.*

Let $f : \{0,1\}^n \to \{0,1\}^n$ be a bijective mapping and let $\oplus$ denote the exclusive-or operation. Consider the compression function $h : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$:

$$h(h_{i-1}, m_i) = h_i = f(h_{i-1} \oplus m_i) \oplus h_{i-1} \oplus \theta m_i, \tag{1}$$

where θ is as in Definition 1. Multiplication with certain values of θ can be done very efficiently as we shall demonstrate later. Consider Attacks I and II from before.

Attack I: Given h_{i-1} and h'_{i-1} the attacker faces the task of finding m_i and m'_i such that

$$f(h_{i-1} \oplus m_i) \oplus h_{i-1} \oplus \theta m_i = f(h'_{i-1} \oplus m'_i) \oplus h'_{i-1} \oplus \theta m'_i. \tag{2}$$

Or in other words, with $h_{i-1} \oplus h'_{i-1} = \alpha$ and $m_i \oplus m'_i = \beta$ one needs to find two inputs to f of difference $\alpha \oplus \beta$ which yield outputs of difference $\alpha \oplus \theta\beta$ for a fixed value of θ. But if f is "as good as" a randomly chosen mapping, the attacker has no control over the relation between the outputs for two different inputs to f, and he has no better approach than the birthday attack. Note that with $m_i \oplus m'_i = h_{i-1} \oplus h'_{i-1} = \alpha \neq 0$ one never has a collision for h, since in this case the difference in the outputs of f is zero and the difference in the outputs of h is $(\theta + 1)\alpha \neq 0$.

Attack II: For fixed values of h_{i-1}, m_i and m'_i, the attacker faces the task of finding h'_{i-1} such that Eq. 2 is satisfied. But in this case (1) has the form of $g(h_{i-1}) \oplus h_{i-1} \oplus c_1$, where $g(x) = f(x \oplus c_2)$ and where c_1, c_2 are constants. Thus, under Assumption 1 (with sufficiently large n) attacks using a fixed value of m_i seem to be hard to mount.

Although the two attacks above do not seem to be easy to do for the proposed compression function, it is clear that there are properties of it which are not typical for compression functions. These are already discussed above but we highlight them here again.

Inversion: (1) can be inverted. Given h_i, choose an arbitrary value of a, compute $b = f^{-1}(h_i \oplus a) = h_{i-1} \oplus m_i$, then solve for h_{i-1} and m_i. With θ as in Definition 1 this can be accomplished by solving

$$(a \quad b) \;=\; (h_{i-1} \quad m_i) \begin{pmatrix} 1 & 1 \\ \theta & 1 \end{pmatrix}$$

which always succeeds, since $\theta \neq 1$.

Forward prediction: Let h_{i-1} and h'_{i-1} be two inputs to (1) where $\alpha = h_{i-1} \oplus h'_{i-1}$. Choose a value for m_i and compute $m'_i = m_i \oplus \alpha$. Then

$$\begin{aligned} h_i \oplus h'_i &= f(h_{i-1} \oplus m_i) \oplus h_{i-1} \oplus \theta m_i \oplus f(h'_{i-1} \oplus m'_i) \oplus h'_{i-1} \oplus \theta m'_i \\ &= \theta\alpha \oplus \alpha. \end{aligned}$$

The following is a list of potential problems of hash functions based on the proposed compression function.

1. Collisions for the compression function.
2. Pseudo (2nd) preimages for the hash function.
3. (2nd) preimages for the hash function in time roughly $2^{n/2}$.
4. Non-random, predictable properties for the compression function.

Ad 1: It is easy to find collisions for the compression function, so it is not possible to prove a result similar to that of Merkle and Damgård, cf., the introduction. However the simple approach, presented above, does not give the attacker any control over the values of h_{i-1} and h'_{i-1} and it does not appear to be directly useful in attempts to find a collision for the hash function (with a fixed iv).
Ad 2: Since h can be inverted it is trivial to find a (2nd) message and an $\tilde{\text{iv}}$ which is hashed to a given hash value. However, this approach given h_i does not give an attacker control over the value of h_{i-1} and this approach will not directly lead to (2nd) preimages for the hash function (with a fixed iv). Moreover the attacker has no control over m_i.
Ad 3: Let there be given a hash value and an iv. Then since the compression function is easily inverted, it was shown that (2nd) preimages can be found in time roughly $2^{n/2}$ using a meet-in-the-middle attack. One can argue that this is a weakness, however since for any hash function of size n there is a collision

attack of complexity $2^{n/2}$ based on the birthday paradox, one can also argue that if this level of security is too low, then a hash function with a larger hash result should be used anyway.
Ad 4: Consider the "Forward prediction" property above with some $\alpha \neq 0$. It follows that given the difference in two chaining variables one can find two message blocks such that the values of the corresponding outputs of the compression function is $\gamma = \alpha(\theta + 1)$. This approach (alone) will never lead to a collision since $\gamma \neq 0$. Note that the approach extends to longer messages. E.g., assume that for a pair of messages one has $h_{i-1} \oplus h'_{i-1} = \alpha$. Then with $m_{i+s} \oplus m'_{i+s} = h_{i-1+s} \oplus h'_{i-1+s}$ for $s = 0, \ldots, t$ one gets that $h_{i+s} \oplus h'_{i+s} = \alpha(\theta + 1)^{s+1}$. Note that although $\alpha(\theta + 1)^{s+1} \neq 0$ for any s, one can compute a long list of (intermediate) hash values without evaluating h. Also there are applications of hash functions where it is assumed that the output is "pseudorandom" (e.g., HMAC[4]).

2.2 The proposed hash function

To avoid some of the problems of the compression function as listed above, we add some well-known elements in the design of the hash function. Let m be the message to the hashed and assume that it includes padding bits and the message length. Let $m = m_0, m_1, \ldots, m_t$, where each m_i is of n bits. Let iv be initial value to the hash function, compute

$$h_0 = f(\text{iv}) \oplus \text{iv} \tag{3}$$

$$h_i = f(h_{i-1} \oplus m_i) \oplus h_{i-1} \oplus \theta m_i \qquad \text{for } i = 1, \ldots, t \tag{4}$$

$$h_{t+1} = f(h_t) \oplus h_t \tag{5}$$

As seen, we have introduced two applications of a secure compression function based on f, namely one which from the user-selected iv computes h_0 in a secure fashion, and one which from h_t computes the final hash result in a secure fashion.

It is conjectured that this hash function protects against pseudo-attacks, since the attacker has no control over h_0. Moreover because of the final application of a secure compression function it is not possible to predict the final hash value (using the approach of item 4 above). Also, the inclusion of the message length in the padding bits complicates the utilization of long message attacks, e.g., using the approach of item 4 above, see also [16, 9]. Finally, the construction complicates preimage attacks, since the hash results are outputs of a (conjectured) one-way function.

It is claimed that if f is (as good as) a randomly chosen bijective mapping on n bits, then the complexity of the best approach for a preimage, 2nd preimage or a collision attack on the proposed hash function is at least $2^{n/2}$.

2.3 $\theta = 0$ and $\theta = 1$

Consider the compression function above with $\theta = 0$. Then

$$h_i = f(h_{i-1} \oplus m_i) \oplus h_{i-1} \qquad \text{for } i = 1, \ldots, t$$

This variant of the compression function is easy to break. Choose two different messages $m_1, \ldots, m_{i-1}$ and $m'_1, \ldots, m'_{i-1}$ such that $h_{i-1} \neq h'_{i-1}$. Choose a value of $h_i = h'_i$, and compute $m_i = f^{-1}(h_{i-1} \oplus h_i) \oplus h_{i-1}$ and $m'_i = f^{-1}(h'_{i-1} \oplus h'_i) \oplus h'_{i-1}$. Then there is a collision for the messages $m_1, \ldots, m_i$ and $m'_1, \ldots, m'_i$. Therefore, the proposed hash function should not be used with $\theta = 0$. With $\theta = 1$ it follows that the pairs (h_{i-1}, m_i) and (h'_{i-1}, m'_i) collide when $h_{i-1} \oplus m_i = h'_{i-1} \oplus m'_i$.

3 SMASH

In this section a concrete hash function proposal is presented which has been named SMASH.[2] The version presented here has a 256-bit output, hence we refer to it as SMASH-256. Another version with a 512-bit output is named SMASH-512. These are therefore candidate alternatives to SHA-256 and SHA-512 [14]. The designs of SMASH-256 and SMASH-512 are very similar but where the former works on 32-bit words and the latter on 64-bit words. We focus on SMASH-256 next, the details of SMASH-512 is in an appendix.

3.1 SMASH-256

SMASH-256 is designed particularly for implementation on machines using a 32-bit architecture. A 256-bit string y is then represented by eight 32-bit words, $y = y_7, \ldots, y_0$. We shall refer to y_7 and y_0 as the most significant respectively least significant words.

SMASH-256 takes a bit string of length less than 2^{128} and produces a 256-bit hash result. The outline of the method is as follows. Let m be a u-bit message. Apply a padding rule to m (see later), split the result into blocks of 256 bits, $m_1, m_2, \ldots, m_t$ and do the following:

$$\begin{aligned} h_0 &= g_1(\text{iv}) = f(\text{iv}) \oplus \text{iv} && (6)\\ h_i &= h(h_{i-1}, m_i) = f(h_{i-1} \oplus m_i) \oplus h_{i-1} \oplus \theta m_i \qquad \text{for } i = 1, \ldots, t && (7)\\ h_{t+1} &= g_2(h_t) = f(h_t) \oplus h_t, && (8) \end{aligned}$$

where iv is an initial value. The hash result of a message m is then defined as $\text{Hash}(\text{iv}, m) = h_{t+1}$. The subfunctions g_1, g_2, and f all take a 256-bit input and produce a 256-bit output and h takes a 512-bit input and produces a 256-bit output. g_1 is called the input transformation, g_2 the output transformation, h is called the compression function and f the "core" function, which is a bijective mapping. g_1 and g_2 are of the same form, constructed under Assumption 1.

As a target value of the iv use the all zero 256-bit string.

Padding rule Let m be a t-bit message for $t > 0$. The padding rule is as follows: append a '1'-bit to m, then append u '0'-bits, where $u \geq 0$ is the minimum integer value satisfying

$$(t + 1) + u \equiv 128 \bmod 256.$$

[2] **smash** /smaesh/: to break (something) into small pieces by hitting, throwing, or dropping, often noisily

$$H_1 \circ H_3 \circ H_2 \circ L \circ H_1 \circ H_2 \circ H_3 \circ L \circ H_2 \circ H_1 \circ H_3 \circ L \circ H_3 \circ H_2 \circ H_1(\cdot)$$

Fig. 1. SMASH-256: Outline of f, the core function.

Append to this string a 128-bit string representing the binary value of t.

The compression function, *h* The function takes two arguments of each 256 bits, h_{i-1} and m_i. The two arguments are exclusive-ored and the result evaluated through f. The output of f is then exclusive-ored to h_{i-1} and to θm_i.

"Multiplication" by *θ* This section outlines one method to implement the multiplication of a particular value of θ. As already mentioned there is a natural one-to-one correspondence between bit vectors of length 256 with elements in the finite field $GF(2^{256})$. Consider the representation of the finite field defined via the irreducible polynomial $q(\theta) = \theta^{256} \oplus \theta^{16} \oplus \theta^3 \oplus \theta \oplus 1$ over $GF(2)$. Then multiplication of a 256-bit vector y by θ can be implemented with a linear shift by one position plus an exclusive-or. Let $z = \theta y$, then

$$z = \begin{cases} \text{ShiftLeft}(y, 1), & \text{if } \text{msb}(y) = 0 \\ \text{ShiftLeft}(y, 1) \oplus \text{poly}_1, & \text{if } \text{msb}(y) = 1 \end{cases},$$

where poly_1 is the 256-bit representation of the element $\theta^{16} \oplus \theta^3 \oplus \theta \oplus 1$, that is, eight words (of each 32 bits) where the seven most significant ones have values zero and where the least significant word is $0001000b_x$ in hexadecimal notation. In a 32-bit architecture the multiplication can be implemented as follows. Let $y = (y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0)$, where $|y_i| = 32$, then $\theta y = z = (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)$, where for $i = 1, \ldots, 7$

$$z_i = \begin{cases} \text{ShiftLeft}(y_i, 1), & \text{if } \text{msb}(y_{i-1}) = 0 \\ \text{ShiftLeft}(y_i, 1) \oplus 1, & \text{if } \text{msb}(y_{i-1}) = 1, \end{cases}$$

and where

$$z_0 = \begin{cases} \text{ShiftLeft}(y_0, 1), & \text{if } \text{msb}(y_7) = 0 \\ \text{ShiftLeft}(y_0, 1) \oplus 0001000b_x, & \text{if } \text{msb}(y_7) = 1. \end{cases}$$

The core function, *f* The core function in SMASH-256 consists of several rounds, some called H-rounds and some called L-rounds, see Figure 1. There are three different H-rounds. In each of them a 4×4 bijective S-box is used together with some linear diffusion functions. The S-box is used in "bit-slice" mode, which was used also in the block cipher designs Three-way[5] and Serpent[3]. In the following let $\mathbf{a} = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ be the 256-bit input to an H-round, where each a_i is of 32 bits. The outline of all H-rounds is the same, see Figure 2, where $a^{<<r}$ is the word a rotated r positions to the left. $(x, y, z, w) = \text{Sbs}(x, y, z, w)$ means that for all $i = 0, \ldots, 31$, the four ith bits from x, y, z, w

$$
\boxed{
\begin{aligned}
&(a_7, a_6, a_5, a_4) = \mathrm{Sbs}(a_7, a_6, a_5, a_4) \\
&a_{i+4} = a_{i+4} + a_i^{<<r_i} \quad \text{for } i = 0, \ldots, 3 \\
&(a_3, a_2, a_1, a_0) = \mathrm{Sbs}(a_3, a_2, a_1, a_0) \\
&a_i = a_i + a_{i+4}^{<<r_{i+4}} \quad \text{for } i = 0, \ldots, 3 \\
&(a_7, a_6, a_5, a_4) = \mathrm{Sbs}(a_7, a_6, a_5, a_4) \\
&a_{i+4} = a_{i+4} + a_i^{<<r_{i+8}} \quad \text{for } i = 0, \ldots, 3 \\
&(a_3, a_2, a_1, a_0) = \mathrm{Sbs}(a_3, a_2, a_1, a_0) \\
&a_i = a_i + a_{i+4}^{<<r_{i+12}} \quad \text{for } i = 0, \ldots, 3,
\end{aligned}
}
$$

Fig. 2. SMASH-256: Outline of an H-round.

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
S_1 :	6	13	12	7	15	1	3	10	8	11	5	0	2	4	14	9
S_2 :	1	11	6	0	14	13	5	10	12	2	9	7	3	8	15	4
S_3 :	4	2	9	12	8	1	14	7	15	5	0	11	6	10	3	13

Fig. 3. The SMASH-256 S-boxes.

are evaluated through a 4-bit bijective S-box (Sbs is short for S-box bit-slice) using the convention that the bit from x is the most significant bit. In one H-round the same particular S-box is used in all four bitslice applications. The differences between H_1, H_2, and H_3 are in the S-box used and in the rotations used. For H_i the S-box used is S_i, and the rotations are R_i, see Figures 3 and 4. The L-round (there is only one) is defined as in Figure 5, where $\mathrm{ShiftLeft}(x, 8)$ is the 32-bit quantity x shifted eight positions to the left and $\mathrm{ShiftRight}(x, 8)$ is x shifted eight positions to the right.

3.2 Some ideas behind the design

In this section some further details of the design of the core function of SMASH-256 are explained. Let $\mathbf{a} = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ be a 256-bit variable, where the a_is are of 32 bits each. $\mathbf{a}$ represents the internal state of the compression function. We concentrate first of the design of the H-rounds and L-rounds in the core function. Arrange the 256 bits of the internal state in a matrix as follows.

a_7	a_6	a_5	a_4
a_3	a_2	a_1	a_0

Consider Figure 2. First a bitslice S-box is applied to the top row. Rotated versions of the words in the top row are then added to words in the second row. Then a bitslice S-box is applied to the second row, and rotated versions of words of this result added to words in the top row. This is repeated once, such that in total in one H-round, four bitslice S-box applications and four diffusion layers are performed. The rotations in the H-rounds have been chosen such that each

	r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
R_1 :	19	18	17	7	1	7	26	20	0	16	20	5	28	2	20	4
R_2 :	22	29	12	4	18	2	13	29	26	20	16	29	18	4	10	9
R_3 :	4	21	19	5	24	20	12	16	14	30	3	4	23	15	13	12

Fig. 4. The SMASH-256 rotations.

$$\boxed{\begin{array}{l} a_3 = a_3 \oplus \text{ShiftLeft}(a_7, 8) \\ a_2 = a_2 \oplus \text{ShiftLeft}(a_6, 8) \\ a_1 = a_1 \oplus \text{ShiftRight}(a_5, 8) \\ a_0 = a_0 \oplus \text{ShiftRight}(a_4, 8) \end{array}}$$

Fig. 5. SMASH-256: Outline of an L-round.

of the 256 bits in the internal memory is mixed with all other bits as quickly as possible (relative to this design!). It is clear that with rotations and modular additions all bits depend on all bits after a few steps. However, the H-rounds were designed such that the dependencies between the bits are stronger than just via the carry bits of the addition.

The diffusion layer. For the purpose of studying optimum diffusion functions replace all additions in the H-rounds by exclusive-ors. Also, it shall be assumed that each of the four output bits of an S-box depend on all four input bits. Because of the bitslice method and the assumption on the S-box (that all output bits depend on all input bits), it is convenient to consider only the 32 bit positions in words of the top row and 32 bit positions in the words of the second row when discussing dependencies of bits. Consider one bit in the top row of the input to an H-round. After the first bitslice S-box, this bit affects still only one bit position (of 32 in total) in the top row. After the diffusion layer, the bit affects in the best cases four bit positions in the second row (if the four rotations r_1, r_2, r_3, r_4 are all different). After the bitslice of the second row, the bit still only affects four bit positions in the second row, however after the next diffusion layer, the bit affects up to 17 positions in the top row, where we have counted also the initial single position from the beginning. After the subsequent bitslice of the top row together with the diffusion layer, the bit affects in the best cases all 32 positions in the second row, but still only 17 positions in the top row. The fourth and last bitslice and diffusion layer of the H-round ensures that the initial bit affects in the best cases all 32 positions of both the top and the second row.

Consider next one bit in the second row of the input to an H-round. After the first bitslice S-box and diffusion layer, this bit affects still only one bit position in the second row and zero in the top row. After the next bitslice S-box and diffusion layer, the bit affects (in the best case) four bit positions in the top row (if the four rotations r_5, r_6, r_7, r_8 are all different) and one bit position in the second row. After the subsequent bitslice of the top row together with the diffusion layer, the bit affects 17 positions in the second row in the best case, but still only 4 positions in the top row. After the fourth and last bitslice and diffusion layer of the H-round the initial bit affects in the best cases all 32 positions of the top row and 17 positions of the second row. It is a simple matter to implement a search algorithm which finds values of $r_0, \ldots, r_{15}$ such that the diffusion is optimum as outlined here. All H-rounds in SMASH-256 are designed according to this strategy.

The L-round. Consider variants of SMASH-256 where the modular additions are replaced by exclusive-ors. Let a be the 256-bit input to the core function f. Then the following property holds for the H-rounds:

$$H(a)^{<<c} = H(a^{<<c}).$$

This property does not always hold when modular additions are used in the H-rounds, that is,

$$(a+b)^c = a^{<<c} + b^{<<c}, \tag{9}$$

does not always hold, since there is no carry bit in the least significant bit of a modular addition and since the carry in the most significant bit of a modular addition is thrown away. However, empirical results show that equality holds in (9) with a probability of about 1/4. Therefore we introduce the L-round, which uses the shift operation. The shift operation is not invariant under rotations. We believe that the L-round together with modular additions prevent exploitable properties like (9) for the core functions in SMASH-256.

The S-boxes. The S-boxes are chosen as in the design of the block cipher Serpent [3]. These are 4-bit permutations with the following properties:

- each differential characteristic has a probability of at most 1/4, and a one-bit input difference will never lead to a one-bit output difference;
- each linear characteristic has a probability in the range $1/2 \pm 1/4$, and a linear relation between one single bit in the input and one single bit in the output has a probability in the range $1/2 \pm 1/8$;

The three S-boxes used in SMASH-256 are derived as linear variants of the S-boxes S_0, S_2, and S_4 from Serpent [3]. An implementation of SMASH-256 [1] uses the bitslice implementations of the Serpent S-boxes from [12], which were modified slightly to reduce the number of variables used in the program.

4 Short analysis of SMASH

There is very little theory in the design of cryptographic hash functions today and it is very difficult to prove much about the security of these. Therefore it is not possible to give a precise analysis of cryptographic hash functions like SMASH. In this section we consider a few general attacks and (try to) argue that they are unlikely to succeed.

SMASH-256 consists of a total of 48 S-box layers. Differential characteristics with one active S-box per S-box layer are not possible due to the above design criteria. A very crude estimate is that there are at least three active S-boxes per every two S-box layers. Since the most likely differential characteristic for one layer has probability 2^{-2} this leads to a complexity of $((2^{-6})^{24})^{-1} = 2^{144}$ for a differential characteristic for the function f. A linear characteristic for one S-box layer has a bias of at most 2^{-2}. An analogue crude estimate for linear cryptanalysis gives a complexity of 2^{144}. Since the aim for SMASH-256 is a security level of 2^{128} it is believed that (traditional) approaches in differential and linear cryptanalysis are unlikely to be very efficient when applied to SMASH-256.

Dobbertin's attacks on MD4 and MD5 as well as the recent attacks[2, 18] on SHA-0 and SHA-1 exploit that the attacker has much freedom to influence many of the individual steps of the respective compression functions, namely through the message blocks. SMASH is different from the SHA-designs in that the message is input at the beginning (at step 0) only, and it seems this gives an attacker much less room to play. This is not a proof that these attacks will not work and the readers are invited to apply them (or variants of them) to SMASH.

5 Performance

An implementation of SMASH-256 [1] shows a performance of about 30 cycles per byte in a pure C-implementation. For comparison the implementation of SHA-256 by B. Gladman [8] produced a speed of 40 cycles per byte on the same platform using the same compiler. Speeds of about 21 cycles per byte for SHA-256 have been reported in an assembler implementation. It is expected that an assembler implementation of SMASH-256 would likewise increase the performance.

6 Finishing remarks

We have presented a new approach in hash function design together with a concrete proposal for a hash function. The proposal deviates from the most popular hash function designs in use today, in that only one, fixed and bijective, (supposedly strong) cryptographic mapping is used. After the presentation at FSE 2005 SMASH was broken. In [15] it is shown that it is possible to find messages with 256 blocks which collide when compressed through SMASH-256. There appears to be a similar attack on SMASH-512 for messages of 512 blocks. The attack makes use of the "forward prediction" together with some differential techniques. It appears that there are several ways to modify SMASH to thwart the new attacks. One is to use different f functions for every iteration[15]. Another is to use a secure compression function not only in the first and last iteration (see (3)-(5)) but after the processing of every n blocks of the message for, say, $n = 8$ or $n = 16$.

One interesting avenue for further research is compression function designs using two (or more) fixed, bijective mappings.

7 Acknowledgments

The author would like to thank Martin Clausen for many discussions regarding this paper and for implementing SMASH-256.

References

1. Martin Clausen. An implementation of SMASH-256. Private communications.
2. E. Biham, R. Chen. Near-Collisions of SHA-0. In Matt Franklin, editor, *Advances in Cryptology: CRYPTO'2004, Lecture Notes in Computer Science 3152*. Springer Verlag, 2004.
3. R.J. Anderson, E. Biham, and L.R. Knudsen. SERPENT - a 128-bit block cipher. A candidate for the Advanced Encryption Standard. Documentation available at `http://www.ramkilde.com/serpent`.
4. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology: CRYPTO'96, Lecture Notes in Computer Science 1109*, pages 1–15. Springer Verlag, 1996.
5. J. Daemen. A new approach to block cipher design. In R. Anderson, editor, *Fast Software Encryption - Proc. Cambridge Security Workshop, Cambridge, U.K., Lecture Notes in Computer Science 809*, pages 18–32. Springer Verlag, 1994.
6. I.B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology: CRYPTO'89, Lecture Notes in Computer Science 435*, pages 416–427. Springer Verlag, 1990.
7. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strenghened version of RIPEMD. In Gollmann D., editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 1996, Lecture Notes in Computer Science 1039*, pages 71–82. Springer Verlag, 1996.
8. Brian Gladman. Available at `http://fp.gladman.plus.com/cryptography_technology/sha/index.htm`
9. X. Lai. On the design and security of block ciphers. In J.L. Massey, editor, *ETH Series in Information Processing*, volume 1. Hartung-Gorre Verlag, Konstanz, 1992.
10. X. Lai and J.L. Massey. Hash functions based on block ciphers. In *Advances in Cryptology - EUROCRYPT'92, Lecture Notes in Computer Science 658*, pages 55–70. Springer Verlag, 1993.
11. R. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology - CRYPTO'89, Lecture Notes in Computer Science 435*, pages 428–446. Springer Verlag, 1990.
12. D.A. Osvik. Speeding Up Serpent, *Third Advanced Encryption Standard Candidate Conference, April 13–14, 2000, New York, USA*, pp. 317-329, NIST, 2000.
13. NIST. Secure hash standard. FIPS 180-1, US Department of Commerce, Washington D.C., April 1995.
14. NIST. Secure hash standard. FIPS 180-2, US Department of Commerce, Washington D.C., August 2002.
15. N. Pramstaller, C. Rechberger, and V. Rijmen. *Smashing SMASH.* The IACR Eprint Archive, 2005/081.
16. B. Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, January 1993.
17. B. Preneel, R. Govaerts, and J. Vandewalle. On the power of memory in the design of collision resistant hash functions. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology: AusCrypt 92, Lecture Notes in Computer Science 718*, pages 105–121. Springer Verlag, 1993.
18. V. Rijmen. Update on SHA-1. Accepted for presentation at CT-RSA'2005.
19. R.L. Rivest. The MD4 message digest algorithm. In S. Vanstone, editor, *Advances in Cryptology - CRYPTO'90, Lecture Notes in Computer Science 537*, pages 303–311. Springer Verlag, 1991.
20. R.L. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.
21. X. Wang, D. Feng, X. Lai, H. Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199. Available at `eprint.iacr.org/2004/199`.

$$H_{1,3,2} \circ L \circ H_{2,3,1} \circ L \circ H_{1,2,3} \circ L \circ H_{2,1,3} \circ L \circ H_{3,2,1} \circ L \circ H_{3,1,2}(\cdot)$$

Fig. 6. SMASH-512: Outline of f, the core function, where $H_{a,b,c}$ denotes $H_a \circ H_b \circ H_c$.

	r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
R_1 :	56	40	24	8	55	48	61	14	37	13	25	17	61	29	13	45
R_2 :	24	8	48	32	12	62	57	35	1	45	33	13	4	60	12	20
R_3 :	8	56	48	0	22	21	7	44	34	30	62	2	58	50	34	10

Fig. 7. The SMASH-512 rotations.

SMASH-512

SMASH-512 takes a bit string of length less than 2^{256} and produces a 512-bit hash result. The outline of the method is as follows. Let m be a u-bit message. Apply a padding rule to m (see later), split the result into blocks of 512 bits, $m_1, m_2, \ldots, m_t$ and do as in (6), (7), and (8). The hash result of a message m is the defined as $\mathrm{Hash}(\mathrm{iv}, m) = h_{t+1}$. The subfunctions g_1, g_2, and f all take a 512-bit input and produce a 512-bit output and h takes a 1024-bit input and produces a 512-bit output. As a target value of the iv use the all zero 512-bit string. The design is very similar to that of SMASH-256, the main difference is that the latter is designed for 32-bit architectures whereas SMASH-512 is for best suited for 64-bit architectures.

Consider the representation of the finite field $GF(2^{512})$ defined via the irreducible polynomial $q(\theta) = \theta^{512} \oplus \theta^8 \oplus \theta^5 \oplus \theta^2 \oplus 1$ over $GF(2)$. Then multiplication by θ can be defined by a linear shift by one position and an exclusive-or. In a 64-bit architecture the multiplication can be implemented as follows. Let $y = (y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0)$, where $|y_i| = 64$, then $\theta y = z = (z_7, z_6, z_5, z_4, z_3, z_2, z_1, z_0)$, where for $i = 1, \ldots, 7$

$$z_i = \begin{cases} \mathrm{ShiftLeft}(y_i, 1), & \text{if } \mathrm{msb}(y_{i-1}) = 0 \\ \mathrm{ShiftLeft}(y_i, 1) \oplus 1, & \text{if } \mathrm{msb}(y_{i-1}) = 1, \end{cases}$$

and where

$$z_0 = \begin{cases} \mathrm{ShiftLeft}(y_0, 1), & \text{if } \mathrm{msb}(y_7) = 0 \\ \mathrm{ShiftLeft}(y_0, 1) \oplus 0000000000000125_x, & \text{if } \mathrm{msb}(y_7) = 1. \end{cases}$$

The core function in SMASH-512 consists of a mix of 18 H-rounds and five L-rounds, see Figure 6. The differences between the H-rounds of SMASH-256 and of SMASH-512 are in the rotations used. The outline is the same as for SMASH-256, see Figure 2, as are the S-boxes. The rotations for SMASH-512 are in Figure 7. The definition of the L-round is the same as the one for SMASH-256, see Figure 5.

Padding rule Let m be a t-bit message for $t > 0$. The padding rule is as follows: append a '1'-bit to m, then append u '0'-bits, where $u \geq 0$ is the minimum integer value satisfying

$$(t+1)+u \equiv 256 \bmod 512.$$

Append to this string a 256-bit string representing the binary value of t.