New Techniques for Cryptanalysis of Hash Functions and Improved Attacks on Snefru

Eli Biham

Computer Science Department
Technion – Israel Institute of Technology
Haifa 32000, Israel
Email: biham@cs.technion.ac.il
WWW: http://www.cs.technion.ac.il/∼biham/

Abstract. In 1989–1990, two new hash functions were presented, Snefru and MD4. Snefru was soon broken by the newly introduced differential cryptanalysis, while MD4 remained unbroken for several more years. As a result, newer functions based on MD4, e.g., MD5 and SHA-1, became the de-facto and international standards. Following recent techniques of differential cryptanalysis for hash function, today we know that MD4 is even weaker than Snefru. In this paper we apply recent differential cryptanalysis techniques to Snefru, and devise new techniques that improve the attacks on Snefru further, including using generic attacks with differential cryptanalysis, and using virtual messages with second preimage attacks for finding preimages. Our results reduce the memory requirements of prior attacks to a negligible memory, and present a preimage of 2-pass Snefru. Finally, some observations on the padding schemes of Snefru and MD4 are discussed.

1 Introduction

Snefru [7] and MD4 [13] are two hash functions designed in 1989–1990. Soon after, an attack on Snefru based on the newly introduced differential cryptanalysis was published [2, 1]. As a result, MD4 became the de-facto standard. Later, MD5 [14] and SHA-1 [8], which are improved functions of the MD4 family, replaced MD4 as the official and de-facto standards. Following the recent attacks of Wang [15] we know that collisions of MD4 can be found by hand, with complexity between 2^2 and 2^6, so that MD4 is even weaker than Snefru. Snefru has several variants, varying in the number of passes and the hash sizes. The supported hash sizes are 128 and 256 bits. The number of passes in the original 2-pass variant of Snefru is two passes, while a more secure 4-pass version is also available. After the prior attacks were published, an 8-pass version was introduced as well. This 8-pass version is still considered secure.

The most basic and standard attacks on hash functions are the birthday attacks. Their standard implementation requires a large memory (whose complexity is typically the same as the time complexity). Memoryless generic attacks, in which only a negligible amount of memory is required, can also be performed with the same time complexity. The most well known of those is the Floyd [4, Page 7] two-pointer cycle detection algorithm, for which there is a variant that finds a collision. Floyd algorithm is also used in other cryptographic contexts, e.g., the Pollard Rho factoring algorithm [11]. The algorithms of [12, 10] are more efficient and parallelized versions. Another interesting algorithm (which is between three and five times faster than Floyd) was designed by Nivasch [9]. This algorithm uses a single pointer and a small stack, and saves the overhead of advancing two pointers, and applying the iterated function three times at each step.

Soon after the introduction of Snefru, Biham and Shamir applied their newly introduced techniques of differential cryptanalysis [2, 1] to Snefru, and presented efficient second preimage and collision attacks on 2-pass Snefru, as well as many instances of collisions and second preimages. Their attack could also find second preimages and collisions of 3-pass and 4-pass Snefru, but the higher time complexities and memory requirements prevented them from implementing these attacks.

In this paper we present two main techniques that extend these attacks on Snefru: one that combines differential cryptanalysis with generic collision search techniques, in order to get the best of both worlds, i.e., the complexity of differential cryptanalysis, with the memory requirements of the generic memoryless collisions search techniques. Using this technique, the first collision of 3-pass Snefru was found. The other contribution presents a method that allows to use a second preimage attack for finding preimages of the compression function of Snefru, using specially crafted *virtual messages*, which are not preimages of the function, but on which the second preimage attack can be applied, and find a preimage of the function. Many such preimages of the compression function of 2-pass Snefru were found using this technique. The same attack can also be used for 3-pass and 4-pass Snefru, with higher complexities. We also discuss the importance of the particular padding scheme of the hash function, and how it affects the ability to find preimages of the full hash function. Finally, a very long preimage of 2-pass Snefru is presented.

The paper is organized as follows. Section 2 describes Snefru. Prior attacks on Snefru are described in Section 3. Section 4 describes generic memoryless collision search algorithms. In Section 5 memoryless collision attacks based on the combination of differential cryptanalysis with generic attacks are described, and Section 6 describes the preimage attacks on Snefru. Finally, Section 7 summarizes the paper.

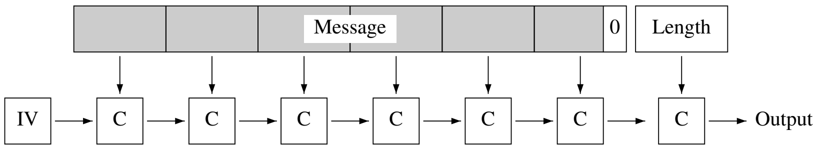

Fig. 1. The Mode of Operation of Snefru

2 Description of Snefru

Snefru [7] is an iterative hash function that follows the Merkle-Damgård construction [5, 6, 3]. It was designed to be a cryptographic hash function which hashes messages of arbitrary length into 128-bit values (a 256-bit variant based on the same design was also introduced). Snefru uses a padding scheme that always adds an additional padding block with the length of the message (unlike the more compact padding scheme of MD4 [13], which adds another block only if necessary). Messages are divided into 384-bit blocks, $M_1, \ldots, M_{n-1}$, where the last block is padded by '0's. An additional block M_n containing only the message length (in bits) is appended. Each block is then mixed with the chaining value (initially using $IV = 0$) by a compression function C. The compression function C takes a 512-bit input composed of the chaining value and the current block, and calculates a new chaining value. More formally, $h_i = \mathrm{C}(h_{i-1} \| M_i)$ for any $1 \le i \le n$, where '$\|$' is the concatenation operator of bit vectors, M_i is block number i, and $h_0 = 0$. The final hash is h_n. This mode of operation is outlined in Figure 1.

The compression function is based on an invertible 512-bit to 512-bit permutation (which can be viewed as a keyless block cipher). The permutation mixes the data in two passes in the standard two-pass version of Snefru, and in four passes in the more secure four-pass Snefru. Each pass is composed of 64 mixing rounds. In round i, the least significant byte of word $i \bmod 16$ is used as an input to an 8x32-bit S box, whose 32-bit output is XORed into the two neighboring data words (all word indices are taken mod 16). After every set of 16 rounds (to which we call a *quarter*), a rotation of each of the words is performed, in order to ensure that each of the 64 bytes is used as an input to an S box once every pass (64 rounds). The output of the compression function is the XOR of the input chaining value with the last words of the output of the permutation. The details of the compression function are described in Figure 2. A complete description on Snefru, including the S boxes, can be found in [7].

3 Prior Attacks

Prior attacks on Snefru are discussed in [2, 1]. These attacks include collision attacks as well as second preimage attacks. Some of the prior attacks even work if the attacker does not know the details of the S boxes. In this section we

```
function C (int32 input[16]) returns int32 output[4]
{
  int32 block[16];
  int32 SBoxEntry;
  int i, index, byteInWord;

  int shiftTable[4] = {16, 8, 16, 24};

  block = input;
  for index = 0 to NO_OF_PASSES-1 do {    (pass index)
    for byteInWord = 0 to 3 do {          (quarter index*4+byteInWord)
      for i = 0 to 15 do {                (round index*64+byteInWord*16+i)
        SBoxEntry = SBOX[2*index+((i/2) mod 2)][block[i] mod 256];
        block[(i + 1) mod 16] ⊕= SBoxEntry;
        block[(i - 1) mod 16] ⊕= SBoxEntry;
      }
      for i = 0 to 15 do
        block[i] = block[i] ⋙ shiftTable[byteInWord];
    }
  }

  for i = 0 to 3 do
    output[i] = input[i] ⊕ block[15-i];

  return(output);
}
```

Fig. 2. The Compression Function of Snefru

describe the main ideas and basic techniques of the attacks, concentrating on the variants with 128-bit hash values.

The second preimage attack is as follows: choose a random block-sized message and prepend the given 128-bit input chaining value to form a 512-bit input to the permutation of the compression function. We create a second message from the first one by modifying two or more bytes in words 5–11, which are used as inputs to the S boxes at rounds 53–59 (i.e., those bytes that do not affect the computation before the fourth quarter). We hash both messages by the compression function and compare the outputs of the two executions. A fraction of 2^{-40} of these pairs of messages are hashed to the same value. Therefore, by hashing about 2^{41} messages we can find a second preimage. As described later in this section, the number can be greatly reduced by using more structured messages.

The characteristic used in this attack is outlined in Figure 3. In the figure each column represents a word of data and each row represents a quarter (16 rounds), where each round is marked by a thin line along the edges. The input appears at the top of the figure, and the calculation is performed downwards. The gray area in the middle represents arbitrary non-zero differences, while the

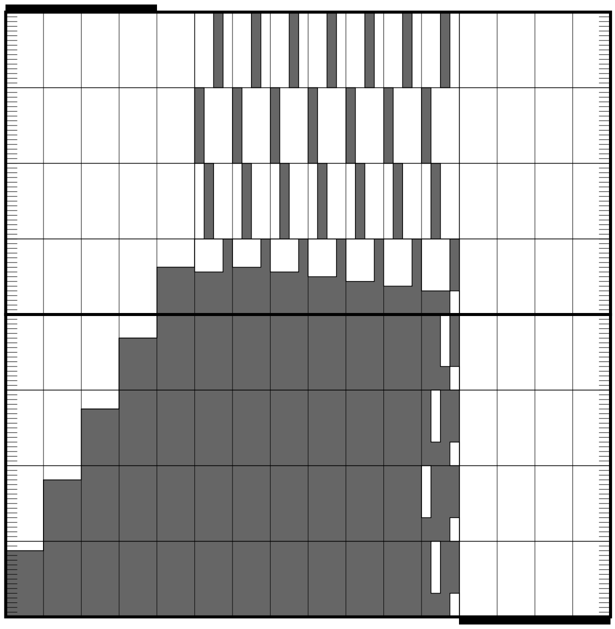

Fig. 3. Graphic Description of the Characteristic

white areas represent zero differences. The two thick black lines at the top-left and the bottom-right corners point to the words which are used in the calculation of the hash value by the compression function. Since both of them occur in the white part of the block, such two messages hash to the same value. In this characteristic, no difference is propagated till round 53 (fourth quarter), in which the difference affects the input of the S box for the first time. The output of the S box then affects the two neighboring words, and so repeatedly in the next rounds till round 58. With some luck, which comes with probability 2^{-8}, the output of the S box of round 58 cancels the difference of the least significant byte in the next word, leading to a zero difference in the input to the S box in round 59, and then to zero differences in the inputs to the S boxes of the following rounds till the next quarter. The difference propagates again in round 68–74 (fifth quarter), and with some luck, the difference in the S box of round 75 is zero again. Similarly, if we are lucky three more times (in rounds 91, 107, and 123), then the difference of the last four words of the permutation is zero, which after XORed with the input chaining value, is leading to a collision of the output. We call this pillar of lucks a *wall* (as all lucks stand on top of each other), and the slow propagation of the differences at the left hand side a *stairs shape*. The total probability of all the five "lucks" in the wall is $2^{-8 \cdot 5} = 2^{-40}$. As the first two "lucks" can be assured deterministically by a simple selection of the changes, the actual probability is 2^{-24}. Thus, a second preimage attack using this characteristic and all these techniques has complexity $2 \cdot 2^{24} = 2^{25}$.

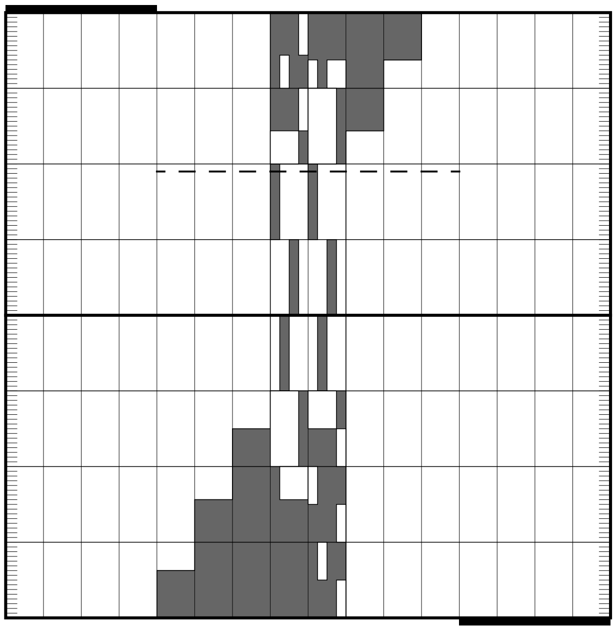

Fig. 4. A Characteristic with Modification at an Intermediate Round

These ideas lead to very efficient collision attacks, by using structures of messages. We randomly choose about $2^{12.5}$ messages in which all the inputs in the white area are the same in all messages, while the messages differ in (up to seven) gray bytes. For each message, we apply the needed (deterministic) modifications to fix the difference of the "lucky" S boxes of quarters 4 and 5. We then hash all these messages. We get about $\frac{(2^{12.5})^2}{2} = 2^{24}$ pairs of messages which are then subjected to the second preimage attack. With high probability such a structure contains a right pair, i.e., a pair whose two messages hash to the same value. Such a pair can be easily found by sorting the $2^{12.5}$ hashed values. The memory complexity of this attack is the same as the time complexity, due to the need to keep all the computed values, for performing the tests for collisions. This attack can also be used when Snefru is considered as a black box, which hides the choice of the S boxes.

An important observation is that whenever the S boxes are known to the attacker, the modification of the bytes may be performed at an intermediate round rather than in the message itself. In this case we choose a message and partially hash it, in order to get the value of the data block at some intermediate round. Figure 4 describes such a characteristic, which modifies the data at the intermediate round denoted by the dashed line. The attack modifies the gray bytes at the marked location, rather than in the message itself. Then, the input of the permutation is calculated by performing the inverse of the compression function

No. of passes	Second Preimage (time)	Collision (time & memory)
2	2^{24}	$2^{12.5}$
3	2^{56}	$2^{28.5}$
4	2^{88}–2^{112}	$2^{44.5}$–$2^{56.5}$

Table 1. Summary of the Complexities of the Prior Attacks

backwards from the marked location, and its output is calculated forward. From the input and the output of the permutation, the output of the compression function is then calculated. The remaining details of the attacks are the same as in the prior case.

Characteristics may also have differences in all the bytes of words in the marked location (rather than just one in each word). Such characteristics are very useful for longer versions of the hash function, i.e., 3-pass and 4-pass Snefru.

A summary of the prior second preimage and collision attacks on Snefru is given in Table 1.

4 Generic Memoryless Collision-search Algorithms

In this section we briefly describe two generic memoryless collision-search algorithms.

4.1 Floyd Algorithm

Floyd algorithm [4, Page 7] traverses the graph generated by iterative application of a function f. It is used by Pollard's Rho factoring algorithm [11], and variants of which are used in the attacks on hash functions of Oorschot and Wiener [10]. In this algorithm, two pointers to the graph are used, one advances by one edge at a time, while the other advances by two at a time. At some moment, the slower pointer enters a cycle, and some time afterwards, their distance would be a multiple of the cycle size, i.e., they will both point to the same location inside the cycle. Once we identify this fact, we deduce the size of the cycle (actually a multiple of the cycle size), and an approximate information on the length of the path from the starting point to the cycle. The collision search variant of Floyd algorithm repeats the process, one pointer starts from the starting point, and the other from the reached location on the cycle, and both advance at the same speed of one edge at a time. After some time they will both point to the first location of the cycle. The previous values of these two pointers form the collision. A detailed description is given in Figure 5. The expected complexity of this attack is about $2^{m/2}$, and it calls the function f up to five times for each value in the path to the collision.

1. Let f be the hash function.
2. Select some starting point v_0 at random.
3. $u = f(v_0)$, $v = f(f(v_0))$.
4. while $u \neq v$ do
   (a) $u = f(u)$.
   (b) $v = f(f(v))$.
5. If $v = v_0$, the starting point v_0 is in the cycle – stop, and try again with another starting point.
6. $u = v_0$.
7. repeat
   (a) $u' = u$, $v' = v$.
   (b) $u = f(u)$.
   (c) $v = f(v)$.
   until $u = v$.
8. Now $u' \neq v'$, and $f(u') = f(v')$.

Fig. 5. Floyd Algorithm with Collision Search

1. Let f be the hash function.
2. Initialize a stack.
3. Select some starting point v_0 at random, and assign $u = v_0$.
4. while stack is empty or $u \neq$ top(stack)
   (a) Push u to the stack.
   (b) Compute $u = f(u)$.
   (c) While stack is not empty and top(stack) $> u$, remove the top entry from the stack.
5. Once this line is reached, u is the minimal value in the cycle.

Fig. 6. Nivasch Cycle Detection Algorithm

4.2 Nivasch Algorithm

In 2004, Nivasch described another cycle detection algorithm [9] that uses only a single pointer into the graph, but keeps a small stack that consists of increasing values of the vertices. At each step, the stack contains all the values in the path that satisfy the property that no smaller value exists in the path anywhere between them and the current point. Therefore, the current value is at the top of the stack, just below it resides the last value in the path that is smaller than it, and so on. The idea is that once we reach the minimal point in the cycle the second time, the minimal point is the only point in the cycle that remains on the stack, so it is easy to identify this point. This cycle detection is described in Figure 6. The expected size of the stack is logarithmic with the number of computations of $f(u)$, which in practical cases is only a few tens up to an hundred. This algorithm is up to three times faster than Floyd's. The difference is especially meaningful when the tail of the path is larger than the cycle. A multi-stack variant of Nivasch algorithm can detect the cycle even earlier by utilizing several stacks without loss of efficiency [9]. An adaptation of the multi-stack

1. Let f be the hash function.
2. Initialize an array of S stacks.
3. Let $g : \text{range}(f) \to \{0, \ldots, S-1\}$ be an efficient balanced function, e.g., a function that truncates the input to $\log_2 S$ bits.
4. Select some starting point v_0 at random, and assign $u = v_0$.
5. Compute $s = g(u)$.
6. Initialize a counter $c = 0$.
7. while stack[s] is empty or $u \neq \text{top}(\text{stack}[s])$
   (a) Push the pair (u, c) to stack[s].
   (b) Compute $u = f(u)$.
   (c) Compute $s = g(u)$.
   (d) Increment c.
   (e) While stack[s] is not empty and $\text{top}(\text{stack}[s]).u > u$, remove the top entry from stack[s].
8. Compute the size of the cycle $p = c - \text{top}(\text{stack}[s]).c$.
9. Assign $c = c_m = \text{top}(\text{stack}[s]).c$.
10. Assign $v = v_0$ and $d = 0$.
11. For each stack[s], $s \in \{0, \ldots, S-1\}$
    – If there is a pair (u', c') in stack[s] (including in popped entries that were not yet overwritten) such that $d < c' < c_m$, assign $v = u'$ and $d = c'$ (if there are several such pairs, use the one with the largest c').
12. For each stack[s], $s \in \{0, \ldots, S-1\}$
    – If there is a pair (u', c') in stack[s] (including in popped entries that were not yet overwritten) such that $c < c' \leq d + p$, assign $u = u'$ and $c = c'$ (if there are several such pairs, use the one with the largest c').
    – Otherwise, there is a pair with a minimal c' such that $c_m \leq c'$, if also $c < c'$ assign $u = u'$ and $c = c'$
13. if $c - p > d$, iteratively compute $v = f(v)$, $c - p - d$ times.
14. if $c - p < d$, iteratively compute $u = f(u)$, $d - c + p$ times.
15. If $u = v$, the starting point v_0 is in the cycle – stop, and try again with another starting point.
16. repeat
    (a) $u' = u$, $v' = v$.
    (b) $u = f(u)$.
    (c) $v = f(v)$.
    until $u = v$.
17. Now $u' \neq v'$, and $f(u') = f(v')$.

Fig. 7. Nivasch Multi-Stack Algorithm with Collision Search

variant of Nivasch's algorithm, with collision search (using a second pointer), which makes an optimal use of the values in the stacks, is given in Figure 7. At first reading, it is advisable to follow the cycle detection part of this algorithm up to Step 8 in order to understand the stacks method, and then follow it again with a single stack (by assuming that $S = 1$ and $g(\cdot) \equiv 0$), and ignore Steps 11–12 (which are optimization steps, and the algorithm would work correctly without them), so that only the collision search is added on top of the original algorithm.

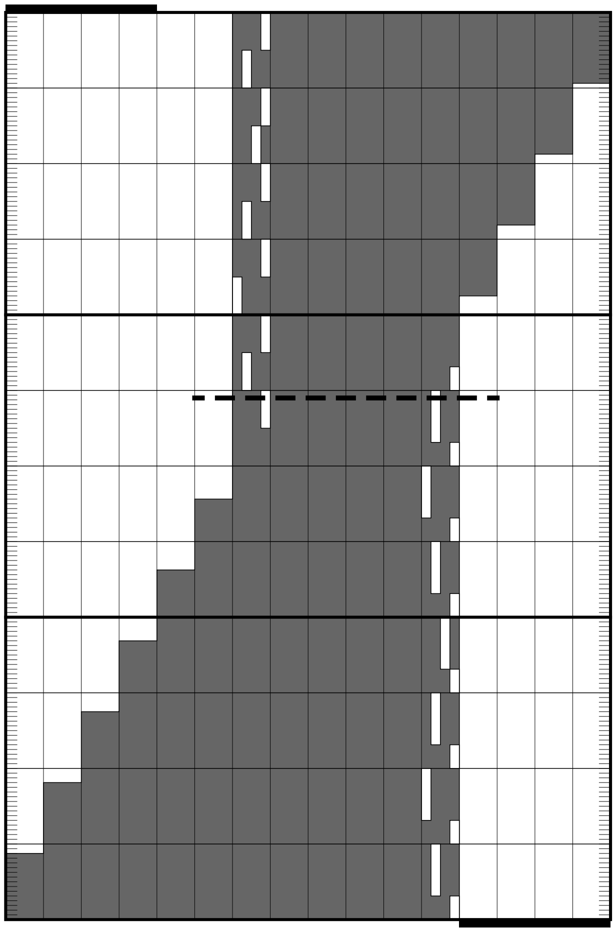

Fig. 8. A Three-Pass Characteristic

5 Using Generic Algorithms for Attacking Snefru

The main drawback of the prior attacks on Snefru is the requirement for a large memory, which in the case of a 3-pass Snefru is about eight gigabytes, and in the case of a 4-pass Snefru is many thousands of terabytes.

In this section we describe a new technique which uses a generic collision search algorithm in conjunction with differential cryptanalysis. This combination was not known in the past, and shows that seemingly unrelated techniques may be combined to form better attacks.

The collision attack on 3-pass Snefru uses the characteristic given in Figure 8. In this characteristic, more than 64 bits may be modified in the marked location. As the conditions of the first six quarters (on the marked location and above it) can be ensured with probability 1 by various simple changes to the message block, the probability of this characteristic is 2^{-56}, i.e., a second preimage can be found after about 2^{56} trials, and a collision may be found after $2^{28.5}$ trials but using $2^{28.5}$ records of memory. Let k be the number of trials required for the collision attack (e.g., $k = 2^{28.5}$ in the case of 3-pass Snefru).

We observe that the prior attack on Snefru makes various tweaks to a message block (or intermediate data) with the hope that some of the tweaks will have the same hash result. The attack can be summarized as follows:

1. Select a block, and compute the values in the location marked by a dashed line.
2. Do about k times:
   (a) Select a "random" tweak for the active bytes in marked location.
   (b) **Assign the tweak to the active bytes in the marked location.**
   (c) **Modify the required bytes to control the first six quarters.**
   (d) **Compute backwards to get the message block.**
   (e) **Compute forward to get the output of the permutation.**
   (f) **Compute the XOR of the last output words with the chaining value, resulting with the output of the compression function for that block.**
   (g) Insert to a hash table.
   (h) If a collision is found: report the collision and stop.

Though it is not impossible these days to apply this attack on 3-pass Snefru using about 8GB of RAM, this is certainly a limit on the practicality of the attack. For 4-pass Snefru the required memory size ensures that the attack would be impossible to apply for many years to follow. We would thus prefer to need a smaller amount of memory.

We observe that Steps 2b–2f can be viewed as a function f of the tweak. With this notation, the attack becomes:

1. Select a block, and compute the values in the location marked by a dashed line.
2. Do about k times:
   (a) Select a "random" tweak for the active bytes in marked location.
   (b–f) **Compute $y = f($tweak$)$.**
   (g) Insert to a hash table.
   (h) If a collision is found: report the collision and stop.

However, once we model the attack as a repetitive application of the function f and a search for a collision of the outputs, we are able to use memoryless collision search techniques instead, using the same function f.

Note that for using the memoryless algorithms, f should have the same number of input and output bits, which is reached by truncating the output to the size of the tweak. In the case of the 3-pass example, f processes 64-bit values into 64-bit values (by truncating the 128-bit output to 64 bits). We expect a random collision (of the truncated values) after 2^{32} iterations — such a collision is a false alarm. We expect a real collision, due to the differential characteristic,

First message:	00000000	00000000	00000000	014A2500	D5717D14	06A9DE9B
	12DB2554	304D2ECE	421F027B	063C73AD	1AF7BDC1	A1654FED
Second message:	00000000	00000000	00000000	9F713600	69B6241A	25DE987C
	D142F521	F1A56064	D9BF9D7E	E03501DA	680D062F	D136E7EA
Common compressed value:	70DE98A5	4FA2634A	E57E0F2D	7F93FCD9		

Table 2. An Example Collision of 3-Pass Snefru (in hexadecimal)

after about $2^{28.5}$ iterations — such a collision is a collision on the full 128-bit original output. As the probability that a random collision would occur within the first $2^{28.5}$ iterations is small, with a very high probability the collision found by the memoryless collision search algorithm is a collision of the attacked hash function.

An example collision of 3-pass Snefru found by this technique is given in Table 2. Using Nivasch algorithm, this collision was found in about half an hour on a personal computer (Intel Core Duo, 1.6GHz), using an unoptimized code.

In the case of a collision of 4-pass Snefru, the time complexity remains as in the prior attack ($2^{44.5}$–$2^{56.5}$), but without the need for a huge memory.

6 Virtual Messages and Preimages of the Compression Function

6.1 A Preimage Attack on the Compression Function

In this section we describe a technique that uses the second preimage attack in order to find preimages of the compression function. The second preimage attack on 3-pass Snefru has complexity 2^{56} using the characteristic of Figure 8. In order to use the second preimage attack, we need to find a message block with the same chaining value and output of a compression function. But this message block would also form a preimage, which would cause the rest of the technique to be redundant. Moreover, it would require to find a preimage in order to find a preimage, which makes such an attack impossible.

We observe that the second preimage attack can actually find blocks with different chaining values and/or different outputs than the given message, as long as they satisfy several *compatibility* criteria related to the values of the bytes in the walls, the input chaining value, and the output of the permutation. We call such a compatible message, which is not a preimage, but can be used with the second preimage attack to find a preimage, a *virtual message*. Thus, in order to find a preimage, we only need to apply the second preimage attack on the virtual message, and control the output so that the attack will find a collision.

We emphasize that the virtual message does not have the required chaining value nor output. In some cases, it could be viewed as a combination of two parts, one corresponding to the chaining value and the wall that protects it (the top left white area of the characteristic), and one to the output and the walls that protects it (the bottom right white area of the characteristic). During the second preimage attack the two parts would be fully combined into a real preimage.

Consider the characteristic of Figure 8. During a compression of a second preimage, the original message and the second preimage should have the same values in all the white areas. In case the original message is a virtual message, the situation is more complicated, as the virtual message does not fulfill the requirements of a real original message. Instead, it should satisfy the following extra conditions

1. It should be possible to replace the input chaining value of the virtual message by the required input chaining value (the one needed by a real preimage) without affecting the other requirements (with minimal additional changes).
2. It should be possible to replace the output area after the last round by the required output (the one needed by a real preimage) without affecting the other requirements.

If those two requirements would be independent, the attack may have been easy. However, these two replacements affect the first words and last words of the data block (words 0 and 15), which in turn affect each other (cyclically). Therefore, they should be performed without disturbing each other, meaning without affecting the input to the S boxes of the first and last words (in rounds 0, 16, ..., 176 for the first word, and rounds 15, 31, ..., 191 for the last word). Therefore, it also should satisfy the following condition

3. When replacing any of the above values, no changes are allowed in the inputs of the S boxes of rounds 0, 16, ..., 176, and 15, 31, ..., 191.

We now observe that once these inputs (and corresponding outputs) of the S boxes of the last word (rounds 15, 31, ..., 191) are fixed, the input chaining value fully controls the inputs to the S boxes of rounds 0, 16, 32, and 48. In the case of round 0, the input to the S box is just one of the bytes of the chaining value. In the case of round 16 the input is an XOR of the first word of the chaining value, the output of the S box at round 1 (whose input is the XOR of the output of the S box of round 0 with another byte of the chaining value), and the fixed output of the S box of round 15. The other two cases (in rounds 32 and 48) are more complex functions of the fixed values and the input chaining value.

The search for a virtual message block starts by selecting the output of the last round (where the last four words are the value needed for the preimage), and computing backwards to receive the input chaining value and block. The probability to receive the required input chaining value is negligible (2^{-128}).

Recall that the original second preimage attack fixes all the white areas in Figure 8, and fixes the wall of the least significant bytes of word 6 in the first six quarters and the wall of the bytes that become least significant bytes of word 11 in the last eight quarters. The values that are fixed in the walls are selected to be their values in the original message.

In our case, we fix the wall of the last rounds to be equal to the message we start with (as the output of the last round has the required value), and we fix the wall of the first rounds to be compatible with both the required input chaining value and the required values of word 0 in all the quarters (which also behaves like a wall of 12 quarters height). This compatibility is not automatic, as the input chaining value reached by computing backwards is not expected to be compatible.

We can view our attack as having two sets of requirements, one ensures the expected behavior in the white left hand side (without difference compared to the message we start with), and ensures compatibility to the required initial value at that side. Similarly, the other ensures the expected behavior in the right hand side, and ensures compatibility of the output. Both are protected from each other by the three requirements mentioned above, including the walls. The virtual message block tries to emulate both requirements simultaneously, the first as if it had the required input chaining value, and the latter as if it had the required output value (while still satisfying the requirements on words 0 and 15).

This latter compatibility is achieved by replacing the chaining value resulting from the backward computation mentioned above, along with fixation of the wall of word 0, and selection of the wall of word 6 as becomes necessary. These changes in the wall in word 6 can then easily be compensated later during the second preimage attack.

Only in one of every 2^{32} trials of a backward computation, the replacement of the chaining value results with compatibility with the required chaining value, without affecting the wall of word 0. The search for the virtual message starts by about 2^{32} such trials, till a message in which we can replace the chaining value to the one we want, and in which the first four quarters of the wall at word 0 remain valid, is found. Once such a message is found, the remainder of the quarters of the wall at word 0 can be directly controlled by modifying words 4 and 5 and by changing the fixation of the wall at word 6. This direct control is very efficient. For example, assume that all the wall of word 0 but the last quarter is already selected as required, then the input to the S box in round 176 is 1-1 related to the least significant byte in the sixth quarter of the wall at word 6 (as the S boxes are byte-wise invertible).

Once all these changes are made, and the values of the wall in word 6 are decided, the computation can be performed in the forward direction from round 0 to the marked round, starting with the modified chaining value and modified values of words 4 and 5. The walls at words 0 and 6 ensure that words outside this range do not affect the values of words 0–5 at the marked location. These six words are now injected to replace the original values of these words at the marked location as received from the original backward computation.

As a result, words 0–5 are computed by this last forward computation, and fit to the change in the chaining value (the left white area). Words 12–15 remain as decided by the original backward computation. If only the wall at word 11 and the wall at word 15 are kept as decided, the result at the output is necessarily

Input chaining value:	00000000	00000000	00000000	00000000 (standard IV)		
Message block:	79F6A75E	0397C368	F60C88DE	3133A55E	6D00251C	8ED3567B
	CA49F82B	A32E5DC4	8F86E479	DD3FF4D6	14DD88C1	A2322E00
Compressed value:	00000000	00000000	00000000	00000000		

Table 3. An Example Preimage of the Compression Function of Snefru (in hexadecimal)

as required, independently of any changes in other words. Words 6–11 may be freely selected by the attacker, just as in the second preimage attack, and whose role is to allow the attacker to control the walls of words 6 and 11 as required.

Note that the received intermediate block, located at the marked location is now a combination of three parts. If we would take this block and compute backwards and forward to receive an input chaining value, and output, without the extra control and changes of the second preimage attack, the received values would not be those that we want, as each side would be affected by the other. But when used with the second preimage attack, with the selected values of the walls, the second preimage attack is able to find a preimage.

We applied this attack on 2-pass Snefru (where the complexity of the second preimage attack is practical), and found many preimages. An example preimage of the compression function of 2-pass Snefru is given in Table 3. Note that this preimage is also a fixpoint of the compression function, and thus it can be easily repeated, without changing the output chaining value.

6.2 Preimage Attacks and the Importance of Padding Schemes

The padding scheme of MD4/SHA optimizes the last block, so that no additional padding block is added unless absolutely necessary (i.e., no extra block is added if the message length and single '1' bit of the padding fit at the end of the last existing block). Thus, the content of the last block can mostly be controlled by the attacker, by selecting messages that are 65-bit short of a multiple of a block. We expect that in such a case, our attack on the compression function would be applicable to the last block, thus leading to a preimage attack on the full hash function (rather than on the compression function only).

However, the padding scheme of Snefru always adds a padding block containing the length only (left filled by '0's). Assuming that the length is bounded by 64 bits, the more restrictive padding scheme of Snefru makes it much more complicated (or maybe impossible using our techniques) to find the preimage of the length block, as the words that the attacker needs to control are fixed to zero. Therefore, the attack on the compression function, as described, cannot work.

It may be the case that an attack can still be found, with a huge complexity (still faster than a generic attack), by changing the locations of the controlled

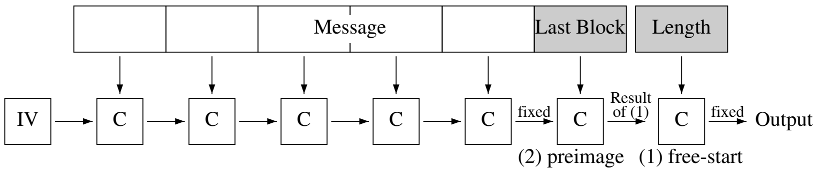

Fig. 9. A free-start preimage of the length block suffices for finding a preimage

words to the area of the input chaining value and the length, and performing a much more complicated computation, but this would require further research.

Though it looks impossible, we observe that such an attack does not have to follow all the requirements of a preimage attack on the compression function. In particular, such an attack does not have to fix the input chaining value in advance — input chaining value can thus be the output of the attack — making the attack a free-start preimage of the compression function on the length block. This kind of attack may be simpler to find, due to the weaker requirements.

A free-start preimage of the length block suffices for finding a preimage of the full hash function, as after this last block along with it's input chaining value are found, we know the output chaining value required from the previous block. All we need to do at that stage, is to apply the preimage attack on the compression function of that previous block. This situation can be seen in Figure 9, which describes the mode of operation used by Snefru, where the attack needs to control only the last two blocks (the last block of the message as well as the length block), but in the reverse order (starting from the last block).

6.3 A Preimage of Snefru

We carefully checked the definition of Snefru, and in particular the definition of the padding block. There is no mentioned limit on the message size, as long as the length fits in the last block. Therefore, the attacker can fully control the last block, at the expense of creating messages with a huge number of blocks. There is no difficulty in creating such huge messages, as fixpoints can be found (e.g., the one from Table 3), and iterated as many times as required.

As the length of the block is 384 bits, the length of the iterated message must be a multiple of 384 bits. We would thus need a length block whose content divides by 384. The block of Table 3 is congruent to 256 modulo 384, thus cannot be used as the length block. On the other hand, the preimage of the compression function described in Table 4 represents a multiple of 384, and is also a fixpoint. Therefore, when this block is iterated 79F6A75E CB8E7368 A8532FD9 81175859 CCE2C60C 734D51CF 5E8B7F23 F48893F9 EE56676D 6E565530 9864E5B1 A2322E00$_x$/384$\approx 2^{374}$ times, it becomes a preimage of the zero hash value! This huge message is a preimage of 2-pass Snefru. By iterating this fixpoint, and using an alternate padding block, it is possible to find preimages for any hash value.

Note that, unfortunately, sending this message would take a huge time. Even verification of this message by the receiver would take a huge time (much more

Input chaining value:	00000000	00000000	00000000	00000000		
Message block:	79F6A75E	CB8E7368	A8532FD9	81175859	CCE2C60C	734D51CF
	5E8B7F23	F48893F9	EE56676D	6E565530	9864E5B1	A2322E00
Compressed value:	00000000	00000000	00000000	00000000		

Table 4. The Block Used for the Preimage of Snefru (in hexadecimal)

Number of Passes	Second preimage of Compression Function (time)	Preimage of Compression Function (time)	Collision Attack (time)
Novelty:	Old	New	Memoryless
2	2^{24}	2^{32}	$2^{12.5}$
3	2^{56}	2^{56}	$2^{28.5}$
4	2^{88}–2^{112}	2^{88}–2^{112}	$2^{44.5}$–$2^{56.5}$

Table 5. Summary of the Attacks on Snefru

than the preimage attack itself). It would even be faster for the receiver to find another single-block preimage (with complexity 2^{128}), than to receive this message and verify it (which takes about 2^{374} time in the standard verification procedure).

7 Summary

In this paper, we described new techniques for cryptanalysis of Snefru:

1. A preimage attack based on the second preimage attack with a virtual message.
2. Differential cryptanalysis using generic algorithms.

Table 5 summarizes the complexities of the attacks on Snefru with 2, 3, and 4 passes (and 128-bit hash values). The second column gives the complexities of the second preimage attacks of [2, 1]. The third column gives the complexities of the preimage attack on the compression function described in this paper. The last column gives the complexity of the collision attacks of [2, 1], which require same size of memory. The memoryless collision attacks described in this paper eliminated the need to that size of memory, without changing the time complexity.

We also discussed the importance of the padding scheme to protect against applying a preimage attack on the compression function for finding preimages of the full hash function.

References

[1] Eli Biham, Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[2] Eli Biham, Adi Shamir, *Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer (extended abstract)*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of CRYPTO'91, pp. 156–171, 1991.

[3] Ivan Bjerre Damgård, *A Design Principle for Hash Functions*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of CRYPTO'89, pp. 416–427, 1989.

[4] Donald E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, third edition, Addison-Wesley, 1997.

[5] Ralph C. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research press, 1982.

[6] Ralph C. Merkle, *One Way Hash Functions and DES*, Advances in Cryptology, proceedings of CRYPTO'89, LNCS, Springer-Verlag, pp. 428–446, 1989.

[7] Ralph C. Merkle, *A Fast Software One-Way Hash Function*, Journal of Cryptology, Vol. 3, No. 1, pp. 43–58, 1990.

[8] National Institute of Standards and Technology, *Secure Hash Standard*, U.S. Department of Commerce, FIPS pub. 180-1, April 1995.

[9] Gabriel Nivasch, *Cycle Detection using a Stack*, Information Processing Letters, Vol. 90, No. 3, pp. 135–140, 2004.

[10] Paul C. van Oorschot, Michael J. Wiener, *Parallel Collision Search with Applications to Hash Functions and Discrete Logarithms*, proceedings of 2nd ACM Conference on Computer and Communications Security, pp. 210–218, ACM Press, 1994.

[11] John M. Pollard, *A Monte Carlo method for factorization*, BIT Numerical Mathematics, 15(3), 1975, pp. 331-334.

[12] Jean-Jacques Quisquater, Jean-Paul Delescaille, *How Easy is Collision Search? Application to DES*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of CRYPTO'89, pp. 429–434, 1989.

[13] Ronald L. Rivest, *The MD4 Message Digest Algorithm*, Lecture Notes in Computer Science, Advances in Cryptology, proceedings of CRYPTO'90, pp. 303–311, 1990.

[14] Ronald L. Rivest, *The MD5 Message Digest Algorithm*, Internet Request for Comments, RFC 1321, April 1992.

[15] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, Xiuyuan Yu, *Cryptanalysis for Hash Functions MD4 and RIPEMD*, advances in cryptology, proceedings of EUROCRYPT 2005, pp. 1–18, 2005.