J. Cryptology (1990) 3: 43–58

Journal of Cryptology

A Fast Software One-Way Hash Function[1]

Ralph C. Merkle
Xerox PARC, 3333 Coyote Hill Road,
Palo Alto, CA 94304, U.S.A.

Abstract. One way hash functions are an important cryptographic primitive, and can be used to solve a wide variety of problems involving authentication and integrity. It would be useful to adopt a standard one-way hash function for use in a wide variety of systems throughout the world. Such a standard one-way hash function should be easy to implement, use, and understand; resistant to cryptographic attack, and should be fast when implemented in software. We present a candidate one-way hash function which appears to have these desirable properties. Further analysis of its cryptographic security is required before it can be considered for widespread use.

Key words. One-way hash function, Message digest algorithm, One-way function, Manipulation detection code, MDC, Authentication, Integrity.

1. Introduction

Authentication and integrity are central goals in computer systems today. Widespread concern over virus attacks has highlighted the need to protect computer programs against unauthorized or unintended changes. More generally, as computers become more pervasive and more critical, the requirement that changes both to computer programs and to the data stored in computers be controlled has become more important [15].

A fundamental requirement in controlling such changes is the ability to detect them. If programs and data can be undetectably changed, then there is no way to verify their correctness. If you cannot detect the changes made to the binary form of your computer program by a computer virus, you cannot protect yourself against the virus.

This fundamental desire to detect changes (of any form, whether authorized or not) has motivated the use of *one-way hash functions* (also called MDCs (Manipulation Detection Codes), fingerprints, cryptographically secure checksums, message digests, or simply one-way functions). One-way hash functions provide a rapid and convenient method of detecting any change at all, no matter how trivial.

Unfortunately, the design of one-way hash functions has not produced entirely satisfactory results [4], [5], [11], [12], [13]. A recent proposal by IBM [22], [23]

[1] Date received: November 7, 1989. Date revised: July 18, 1990.

based on DES [9], [14], [16], [19] is apparently secure (a significant accomplishment!) but is slower than Snefru. A previous proposal by the author (also based on DES) is likewise significantly slower than Snefru [18].

At the present time there is no widely accepted standard one-way hash function, despite its obvious desirability.

There is now a consensus on some of the major design principles for one-way hash functions [1], [3], [6], [18], [24] which allows greater confidence in at least some of the technical aspects of the design. In addition, because one-way hash functions are relatively unaffected by export regulations, it should be possible to obtain fairly wide if not universal adoption of a single standard one-way hash function.

This happy confluence of circumstances motivated the present work, which describes a candidate one-way hash function: Snefru.

Soon after the design and publication of Snefru, Rivest proposed MD4 [26]. The fundamental design criteria for MD4 (e.g., good security and the ability to hash data rapidly on a 32-bit RISC processor) are the same as those for Snefru, although the design itself is radically different. Neither the security of Snefru nor the security of MD4 has been adequately established at the present time. Further review will be required to determine which, if either, of these two candidate one-way hash functions should see widespread use.

2. Snefru

Snefru was named after an Egyptian Pharaoh (following a suggestion by Dan Greene).

A one-way hash function is a function F which accepts an arbitrarily large input x, and produces a small fixed-size output y:

$$y = F(x).$$

Further, no other input x' can be found (although many such inputs almost certainly exist) which will generate y. Because of this, a small y (perhals 128 bits) can authenticate an arbitrarily large x. This property is crucial for the convenient authentication of large amounts of information.

As an example, x might be a computer program whose integrity we wish to check. The output, y, is a "checksum" that is stored in a safe place. (A wide variety of issues are involved in providing a "safe" place for y. We do not touch on these issues in this paper. One-way hash functions do not, in and of themselves, provide a solution to integrity and authentication problems: they should be viewed as a fundamental tool which can be used to build any one of many possible solutions.) If the program, x, has been changed, then the output, y, will also have changed. Thus, any time we wish to verify the integrity of x, we can recompute $F(x)$ and compare the result against the (previously known) value of y. If the two values differ, then someone has changed x and we are alerted to the existence of a problem. If the two values are the same, then x has not been changed.

Somewhat more formally, a one-way hash function is a function F with the following properties:

1. F can be applied to any argument of any size. For notational convenience, F applied to more than one argument is equivalent to F applied to the bitwise concatenation of all its arguments.
2. F produces a fixed-size output. (The output might be 128 or 256 bits.)
3. Given F and x, it is easy to compute $F(x)$.
4. Given F, it is computationally infeasible to find any pair of values x and x' such that $x \neq x'$ and $F(x) = F(x')$.

The phrase "computationally infeasible" used above can be replaced with any one of several more precise definitions—each definition will in turn result in a somewhat different definition of a one-way hash function. Snefru is intended to be a "random" one-way hash function [20], e.g., for all practical purposes it can be modeled by a very large table of random numbers, or by a "demon" who selects a random number whenever we wish to compute some output value. This is discussed further in [18].

This definition is also called a "strong one-way hash function." Weak one-way hash functions are discussed further in [2] and [18] (and a similar definition is used in [3]). Weak one-way hash functions can be very useful, and it is possible to use Snefru as a weak one-way hash function if desired. We will not explain such usage here for three reasons:

(1) in most system applications strong one-way hash functions would probably be used anyway,
(2) weak one-way hash functions are much easier to misuse, and
(3) the performance advantage of weak one-way hash functions is usually modest.

For most uses, strong one-way hash functions provide a simpler and more foolproof tool. In this paper the term "one-way hash function" is used to mean "strong one-way hash function."

A one-way hash function F accepts an arbitrarily large input—however, it is much easier to specify a function that accepts a fixed-size input. We therefore follow a two-step procedure in defining F. First, we define a fixed-size function F_0 which has the same properties as F but which accepts a fixed-size input, and then we use F_0 to construct F. By definition, F_0 has properties 2, 3, and 4 listed above for F; but replaces property 1 (which says that F can have an unlimited input size) with the simpler property that F_0 can accept only a fixed-size input.

A fixed-size one-way hash function F_0 has the following properties:

1. F_0 can be applied to a fixed-size input argument (the input might be 512 bits). The size of the input must be larger than the size of the output. For notational convenience, F_0 applied to more than one argument is equivalent to F_0 applied to the bitwise concatenation of all its arguments.
2. F_0 produces a fixed-size output. (The output might be 128 bits.)
3. Given F_0 and x, it is easy to compute $F_0(x)$.
4. Given F_0, it is computationally infeasible to find any pair x, x' such that $x \neq x'$ and $F_0(x) = F_0(x')$.

We also introduce the auxilliary funtion F_{simple}. This function is used purely as a notational convenience in the proof that follows.

If we view x as an array, then we can define $F(x)$ in terms of F_0 (and F_{simple}) in the following fashion:

Notation: || is the bitwise concatenation operator.

```
FUNCTION F_simple(x[1..n])
BEGIN
result := 0;
FOR i := 1 to n DO
result := F0(result||x[i]);
ENDLOOP;
RETURN(result);
END;

FUNCTION F(x[1..n])
BEGIN
RETURN(F0(F_simple(x)||⟨length of x in bits⟩));
END:
```

(Note that x can be padded with zero bits to ensure that $x[n]$ is of the correct size. In addition, the size of "result" in bits plus the size of $x[i]$ in bits must equal the input size of F_0: 512 bits. If "result" is 128 bits, then each $x[i]$ must be $512 - 128 = 384$ bits. If "result" is 256 bits, then each $x[i]$ must be 512–256 = 256 bits.)

We can show that F must satisfy properties 1–4 if F_0 satisfies properties 2–4, and if F_0 accepts only a fixed-size input. From the definition of F it is obvious that it will accept an input of arbitrary size, so property 1 is satisfied. It is also obvious that F will produce a fixed-size output—which is the size of "result"—so property 2 is satisfied. Property 3 follows because computation of $F(x)$ requires time linear in the size of x (which we actually take as the definition of "easy") and because computation of F_0 is "easy."

The method of construction, and the proof technique, have been given previously [1], [3], [18], [24].

We show that property 4 holds for F in a somewhat roundabout way. First, we note that we have defined the slightly simpler function F_{simple}, which is almost the same as F but omits the final application of F_0. We first prove that a slightly weakened version of property 4 holds for F_{simple}—we call this property 4′. We then show that if F_{simpler} has property 4′, then F must have property 4. Propetry 4′ is:

4′. Finding two inputs x and x' *of the same length* such that $F_{\text{simple}}(x) = F_{\text{simple}}(x')$ is computationally infeasible.

The inductive proof that property 4′ holds for F_{simple} is straightforward: Clearly, if $n = 1$, then property 4′ holds for it holds for F_0. Assume, then, that the property holds for $n - 1$, and we wish to prove it for n. We know that

$$\text{result} := F_0(F_{\text{simple}}(x[1 \ldots n-1]), x[n]).$$

From the fact that property 4 holds for F_0 it follows that neither $F_{\text{simple}}(x[1 \ldots n-1])$ nor $x[n]$ could have been changed—if they had been, we would have two inputs to F_0 that produced the same output. But if $F_{\text{simple}}(x[1 \ldots n-1])$ has not been changed, then $x[1 \ldots n-1]$ has not been changed, by the induction hypothesis.

Q.E.D.

We can now prove that property 4 holds for F in the following fashion:

$$F(x) := F_0(F_{\text{simple}}(x), \langle \text{length of } x \text{ in bits} \rangle).$$

If two different inputs x and x' differ in their length, then the two inputs to the final application of F_0 will be different, and so if $F(x) = F(x')$, then we would have broken F_0. If two different inputs x and x' have the same length, then we know that $F_{\text{simple}}(x) \neq F_{\text{simple}}(x')$ if $x \neq x'$. Again, we would have generated two different inputs to F_0 that map to the same output, thus breaking F_0. Therefore, if F_0 has property 4, then F has property 4.

The implications of this construction are quite simple: in order to design a one-way hash function F we need only design a fixed-size one-way hash function F_0, from which we can then build F. The critical problem, therefore, is how to design F_0.

Although F is provably as secure as F_0, we cannot prove any desirable properties for F_0 at the present time. We must fall back on cryptographic intuition and review by a wide range of people experienced in the field of cryptography. A significant reason for publishing the present proposed design of F_0 is to facilitate and encourage its review by a wide audience. If F_0 resists attack for some reasonable period (a year or two), then our confidence in it as a cryptographic primitive will increase, and its widespread use can then be considered.

In what follows, we define F_0 and present intuitive arguments that it is difficult to break.

Traditionally, one-way hash functions have been designed by treating the input, x, as the key and then encrypting a fixed-size block. We pursue a different approach. In this approach, we treat the input, x, as the input to a block cipher which uses a fixed (all 0) key. We then encrypt x and exclusive or the output of the encryption function with x. That is,

$$F_0(x) \text{ is defined as } E(0, x) \text{ XOR } x,$$

where E(key, plaintext) is a "good" conventional block cipher. We then retain as many bits of the output as we desire.

This general approach has been used before [12], [17] but must still be justified. We prefer this approach to the more traditional approach (in which x is treated as the key to an encryption function) because it is faster. In the traditional approach, it is necessary to mix the key with a fixed plaintext block during the encryption process. This mixing process requires additional steps. In particular, the key must be repeatedly reloaded from memory to be remixed with the block being encrypted. (By way of example, consider that the 56-bit key used in DES is actually expanded internally into a 768-bit key, so that each bit of key material can be mixed into the block being encrypted several times.) On the other hand, if we treat x as the input

to a block cipher, then we can use a fixed key, need do not key mixing, and can still provide excellent security. To show that security is preserved using this method, we appeal to the intuition that a 'good" encryption function appears to be random. That is, any change to the input will produce an unpredictable and apparently random change in the output. $E(0, x)$ is totally unrelated to $E(0, x \text{ XOR } 1)$—changing a single bit produced a "random" change. We presume that there is no effective method of analyzing E and that therefore it must be viewed as a "black box"—it is possible to encrypt or decrypt, but it is not possible to make any prediction about the value that will be produced (unless we have already encrypted or decrypted that particular value).

If E is random then $E(0, x)$ is random—even if x is highly structured. Therefore $E(0, x)$ XOR x is random and cannot be predicted from a knowledge of x. To determine an x' such that $F_0(x) = F_0(x')$, x' must (by definition) satisfy the equation

$$E(0, x) \text{ XOR } x = y = E(0, x') \text{ XOR } x'.$$

However, if we assume that the only operations we can perform for the cryptanalysis are encryption and decryption of a block, i.e., the computation of $E(0, w)$ or $D(0, w)$ (where D stands for Decryption) for any value of w that we choose, then our chances of actually finding x' are little better than chance. We can select some w by any method we desire, and then compute $E(0, w)$—but this will produce a nearly random value which, when XORed with w, will have a very small probability of being equal to y. If we operate in the reverse fashion and compute $D(0, w)$ XOR w it too, for the same reason, will only equal y by chance.

The critical property that we cannot prove is whether E is in fact "random" in the sense needed for the foregoing proof. This question (at present) can only be answered empirically by means of a certificational attack—we have been unable to break this system, and so we hope that it cannot be broken.

We propose a fixed-size one-way hash function, HASH512, which accepts a 512-bit input and produces a 128 or 256-bit output, as desired. We then define the final hash function, HASH, in terms of HASH512. Note that HASH512 corresponds to F_0, while HASH corresponds to F. HASH512 is illustrated in Fig. 1.

If it is indeed the case that this one-way hash function is "random" and no unexpected weaknesses are discovered, then its security will depend on the size of the output value chosen. If the output is 128 bits, then it should require an exhaustive search of some 2^{64} different values of x before finding two that map onto the same

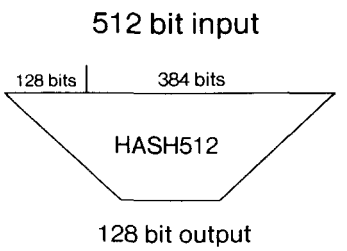

Fig. 1. Block diagram of HASH512 with 128-bit output.

output. If the output is 256 bits, then it should require an exhaustive search of some 2^{128} different values of x before finding two that map onto the same output. While an exhaustive search of 2^{64} possible values provides sufficient security for almost all commercial transactions, some transactions might require higher security. For these transactions, the larger output size is available.

HASH512 accepts a 512-bit input, x. We can provide a definition of HASH512 in terms of a 512-bit block cipher E512 as follows:

$$\text{HASH512}(x) = \text{leading 128 (or leading 256) bits of (E512}(0, x) \text{ XOR } x).$$

If we now specify E512(0, x)—a conventional block cipher that encrypts a 512-bit block and which uses a fixed key—our task is complete.

The block of 512 bits (sixteen 32-bit words) was selected as a compromise between two factors. We can more efficiently hash more data if the block size is large. On the other hand, if the block size is too large it will not fit into the registers on the computer implementing the hash function, so parts of the block will have to be repeatedly loaded and stored. Most modern RISC chips have many registers (more than sixteen 32-bit registers), so on most of these chips all sixteen 32-bit words being hashed can be kept in registers. There will then be no need to load or store parts of the block during computation of the hash function.

We add an additional security parameter,"passes," to both E512 and HASH512. This parameter can be adjusted to suit the security requirements of the particular application. Further study will be required to verify the minimum number of passes that provides adequate security for most commercial applications. Biham crypt-analyzed the two-pass version of Snefru [27], thus demonstrating that more passes are required. A preliminary assessment is that four passes might be sufficient. Users that have requirements for higher security can set this parameter to higher values, as they deem suitable.

The algorithm for E512 is as follows:

```
Function E512(0, x, passes)
x: INT512;  —a 512-bit "integer"
passes;  —a security parameter than can take on the value 2, 3, 4, or even
    higher
BEGIN
blockSize = 512;  —a constant specifying the block size in bits
blockSizeInBytes = blockSize/8;  —the block size in 8-bit bytes, here just 64
    bytes
blockSizeInWords = blockSize/32  —the blocksize in 32-bit words, here just
    16 words
tempBlock, Block: array [0..blockSizeInWords-1] of int32;
StandardSBoxes: ARRAY [1..passes*2] OF ARRAY [0..255] OF int32;
    —Fixed for all time—note that if more passes are needed, more S-boxes
    must be precomputed
rotateSchedule: ARRAY [1..4] := [16, 8, 16, 24];
index: integer;
byteInWord: integer;
```

```
sBoxEntry: int32;
Block := x;  —note that x must be 512 bits or smaller. The trailing bits in
    Block are zero-filled
FOR index := 1 to passes DO
FOR byteInWord := 1 to 4 DO  —for each of the four columns
FOR i := 0 to blockSizeIn Words-1 DO
next := (i + 1) MOD blockSizeInWords;
last := (i − 1) MOD blockSizeInWords;
Pick sboxes in sequence of 1, 1, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, ..., 1, 1, 2, 2, 3, 3, 4, 4,
    3, 3, 4, 4,—etc. Note that using the S-boxes in this sequence prevents
    self-cancelation if the same entry is used twice.
SBoxEntry := standardSBoxes[2*index + ((i/2) MOD 2) − 1] [Block[i].
    bottomByte]:
Block[next] := Block[next] XOR SBoxEntry;
Block[last] := Block[last] XOR SBoxEntry;
ENDLOOP;
    —rotate all the 32-bit words in the block at once by the correct amount
FOR i: INTEGER IN [0...wordCount) DO
Block[i] := RotateRight[Block[i], rotateSchedule[byteInWord]];
ENDLOOP;
ENDLOOP;  —end of byteIn Word going from 1 to 4
ENDLOOP;  —end of index going from 1 to passes
    —flip the Block. The first word and the last word are interchanged, etc.
tempBlock := Block;
For i := 0 to blockSizeIn Words-1 DO
Block[i] := tempBlock[blockSizeInWords-i-1];
ENDLOOP;
RETURN(Block);
END;
```

For efficiency reasons, it is expected that in an actual implementation the inner loops would be unrolled blocksize or 4*blocksize times. This would mean that all shifts would be by fixed amounts (if unrolled 4*blocksize times) and that no array indexing, divisions, or mod computations would actually be performed in the unrolled loop because the values would be known at compile time. The array "Block" would be kept in 16 registers, and reference to individual array elements (because the array indices would be known at compile time) would be to specific registers.

We can define HASH512 in terms of E512 as follows:

```
Function HASH512(x, passes): INT512;
x: INT512;
BEGIN
RETURN(E512(0, x, passes)XOR x);
END:
```

Note that HASH512 returns 512 bits of result. We only use the first 128 or the first 256 bits of the result depending on context. For reasons discussed later, the full 512 bits of output should never be used.

Finally, we define HASH(x, passes) in terms of the fixed-size hash function:

```
Function HASH(x, passes): INT128;
x: ARRAY[0..n − 1] OF INT384;  —this declaration actually defines n
bitCount: INT64;  —by definition, this is the number of valid bits in x
BEGIN
result: INT128;  —a 128-bit "integer"
result := 0;
FOR i := 0 to n − 1 DO result := HASH512(result || x[i], passes);
result := (HASH512(result || ⟨bitCount right justified in a 384-bit field⟩,
    passes));
RETURN(result);
END;
```

The definition of HASH is also illustrated in Fig. 2.

The stronger version of the hash function, which produces a 256-bit output, is defined as

```
Function STRONGERHASH(x, passes): INT256;
x: ARRAY[0..n − 1] OF INT256;  —this declaration actually defines n
bitCount: INT64;  —by definition, this is the number of valid bits in x
BEGIN
result: INT256;  —a 256-bit "integer"
result := 0;
FOR i := 0 to n − 1 DO result := HASH512(result || x[i], passes);
result := HASH512(result || ⟨bitCount right justified in a 256 bit field⟩,
    passes);
RETURN(result);
END;
```

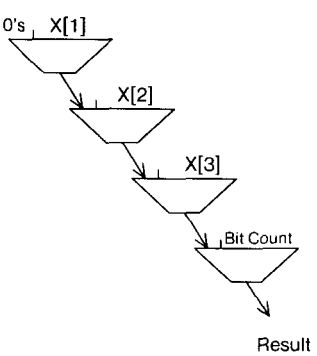

Fig. 2. Block diagram of HASH(x, passes).

Although the above algorithms have been checked for accuracy, the C version (available via anonymous FTP from arisia.xerox.com (13.1.100.206) in directory /pub/hash) should be viewed as defining the actual function. At the time of writing. Snefru Version 2.0 in C is the standard definition. Note that the work of Biham [27] has shown that the number of passes should be greater than two.

3. Making the Standard S-Boxes

We need standard S-boxes in Snefru during the hashing process. We need assurances about how they were generated to avoid any questions or concerns about "trapdoors" or hidden structure. To provide these assurances, the standard S-boxes were generated from a stream of truly random numbers by a simple program. To avoid concerns about "cooked" S-boxes, we adopt the following rules:

1. The program that generates the S-box from a stream of random numbers will be public.
2. The stream of random numbers used as input to the program should be above reproach—it should be selected in such a fashion that it could not reasonably have been tampered with in a fashion that might allow insertion of a trapdoor or other weakness.

The first criteria is met rather simply by publishing the algorithm used to generate the standard S-boxes. The second criteria is met by using the random numbers published in 1955 by the RAND corporation in *A Million Random Digits with* 100,000 *Normal Deviates* (available on magnetic tape for a nominal fee).

Given this approach, insertion of a trapdoor would require

(1) that the publicly known programs that generate the standard S-boxes insert the trapdoor under the very nose of a watching world or that
(2) the trapdoor have been planned and inserted into the random numbers generated by RAND in 1955, over 30 years prior to the design of Snefru (at a time when Snefru's chief designer found successfully negotiating a flight of stairs an absorbing challenge).

4. Summary

We have presented a one-way hash function, Snefru. We hope that Snefru will resist public scrutiny and analysis, but this cannot be taken for granted. A C program implementing Snefru is available from the author for those interested in reviewing the algorithm or in using the algorithm on a preliminary basis.

Acknowledgments

The author would like to thank Dan Greene for his many comments and the many hours of discussion about Snefru. I would also like to thank Dan Swinehart and the many researchers at PARC who have shown both interest and support during the development of this algorithm.

Appendix

The following appendix contains observations that will primarily be useful to those interested in a detailed analysis of the algorithm.

The final "flip" in E512 that reverses the encrypted block has a very specific purpose. Close examination of the algorithm will reveal that without this "flip," the final several table lookups would actually have no impact on the final result. HASH512 throws away the final 256 or 384 bits, so the final computations would be thrown away (would be useless) if the output were not flipped. It seems cryptographically (and otherwise) foolish to compute intermediate values that never influence the final result, and so E512 was "adjusted" so that the values actually used in the final output of HASH512 are influenced by the final computations in E512.

It might be unclear why StandardSBoxes is declared as ARRAY [1..passes*2] of ARRAY [0..255] OF int32. Essentially, we are defining an array of S-boxes, where each individual S-box has 256 entries of 32 bits. Why not use a single S-box, i.e., declare StandardSBox: ARRAY [0..255] OF int32? This would eliminate the rather complicated subscript computations that determine which S-box to use in each table lookup. Unfortunately, if we used the same S-box throughout, we might suffer from cancellation problems. That is, if we should chance to load the same entry from the S-box twice, then when we XORed the same value twice the duplicate values would cancel out. This can be clarified by giving an example of the values of the variables in the inner loop. If we modified the code for the inner loop to use a single S-box, it would appear as follows:

```
next := (i + 1) MOD blockSizeInWords;
last := (i − 1) MOD blockSizeInWords;
SBoxEntry := standardSBox[Block[i].bottomByte];
Block[next] := Block[next] XOR SBoxEntry;
Block[last] := Block[last] XOR SBoxEntry;
```

Let us suppose that the variables in the inner loop have the following values:

$$i = 4,$$

$$\text{next} = 5,$$

$$\text{last} = 3,$$

$$\text{Block}[i].\text{bottomByte} = 31 \text{ (could be any number from 0 to 255, just happens to be 31)},$$

$$\text{SBoxEntry} = 2356664 \text{ (some random 32-bit value)}.$$

And let us suppose that two iterations later these variables take on the values

$$i = 6,$$

$$\text{next} = 7,$$

$$\text{last} = 5,$$

Block[i].bottomByte = 31 (just happens to be 31 again.
About 1 in 256 of this occurring),

SBoxEntry = 2356664
(the *same* random 32-bit value that occured before).

In this case, the first iteration will execute the statement

Block[next] := Block[next] XOR SBoxEntry

which can be given more specifically as

Block[5] := Block[5] XOR 2356664.

When i is increased by two, we will execute the statement

Block[last] := Block[last] XOR SBoxEntry

which can be given more specifically as

Block[5] := Block[5] XOR 2356664.

Putting these two statements together, we have

Block[5] := Block[5] XOR 2356664 XOR 2356664

or

Block[5] := Block[5].

In other words, our two table lookups have canceled each other out. The S-box selection method was chosen specifically to prevent this cancellation from occurring.

It is also unclear why the sixteen 32-bit values in the intermediate block are all rotated at once, instead of being rotated following each table lookup. (This refers to the RotateRight in the almost-inner loop of E512.) The reason for this is somewhat complex, but basically was done to allow efficient implementation of Snefru on machines with differing instruction sets. Snefru is designed so that either a SHIFT or a ROTATE can be used, with some not-too-difficult algorithmic rearrangements. These algorithmic rearrangements, however, cannot be done if each 32-bit word is rotated immediately after the XOR operations in the inner loop of Snefru.

A more complex point is the double use of each entry loaded from the S-box. In the inner loop of Snefru we load a single 32-bit entry from the S-box and XOR this 32-bit entry with both the previous 32-word, and the following 32-bit word. Why not simply XOR with the following 32-bit word?

The reason for doing this is as follows: while the XOR with the previous 32-bit word could be eliminated, it would seriously weaken the hash function. In particular, it would drastically reduce the "circuit depth" of the function. In the following few paragraphs we explain what "circuit depth" is, and why it would be reduced if only a single XOR were used in the inner loop.

For our purposes here, a cryptographic function involves a series of lookups in a series of S-boxes. Each such lookup contributes to the cryptographic complexity of the function. One view of a cryptographic function is just a circuit diagram, in which the "wires" show where the data goes, simple reversible operations (such as

XOR) are used to "mix" data (although they do not contribute any "complexity" to the encryption function), and S-boxes (or table lookups) provide the complexity which is at the heart of the encryption function by taking a set of "input wires," looking up the value encoded on those wires in a ROM, and producing a value on a set of "output wires" corresponding to the results of the table lookup.

According to this view, we can select a single input wire, and a single output wire, and look at the path through the intermediate computations which connects the input to the output. If this path goes through only a few S-boxes, then there cannot have been a great deal of complexity or "mixing" in the encryption function. If, on the other hand, the path has gone through a great many S-boxes, then there has been a great deal of mixing and the encryption function is presumably stronger. We can count the number of S-boxes through which we must go in traveling from input to output, and call this number the "circuit depth."

It would at first seem that if we eliminate the XOR with the previous block in the inner loop that the circuit depth would be unaffected. When we encrypt a 512-bit block, each table lookup is used to produce a 32-bit value which is XORed with the following 32-bit word, and eight bits of this following 32-bit word is then used in turn to provide an index for another lookup operation. That is, each table lookup affects the input to the next table lookup. However, we can view Snefru from two equally valid and symmetrical perspectives: we are either *encrypting* a 512-bit block and XORing the result with the input, or we are *decrypting* a 512-bit block and XORing the result with the "output." While the circuit depth is the same as the number of table lookups if we view the computation as an encryption process. If we look at the hash function as *decrypting* a 512-bit block, the circuit depth drops dramatically.

When we are decrypting a 512-bit block, every time we load a 32-bit value from the S-box we are XORing it not with the *following* 32-bit word, but the *preceding* 32-bit words. This means that we have made no contribution at all to the "circuit depth," because the next eight-bix index will be take from the *preceding* (unchanged) 32-bit word. Therefore, the circuit depth collapses.

To prevent this, we can make the encryption and decryption processes symmetrical. If we XOR the 32-bit word with *both* the preceding *and* the following 32-bit word, then it does not matter whether we are decrypting or encrypting. The circuit depth will be the same, and will equal the number of lookups in either case.

Coppersmith [25] pointed out a very interesting consequence of this double use of the same 32-bit entry. Because a single 32-bit entry is XORed with both the preceding and following word, the overall parity of the 512-bit block is preserved. If the 32-bit entry is of even parity, then the parity of both the following and preceding 32-bit words is preserved when they are XORed with the even parity entry. If the 32-bit entry is of odd parity, then the parity of both the following and preceding 32-bit words is flipped, and therefore the parity of the 512-bit block as a whole is preserved. Further, the odd words and the even words have their parity preserved independently of each other. More generally, Coppersmith observed the following: the 512-bit block can be viewed as a 16 by 32 bit matrix. The 16 bits in a single column remain aligned throughout all the operations in F512, and as a consequence the parity of all thirty-two 16-bit columns is preserved. If we divide

the 16 bits in each column into eight "even-word" bits and eight "odd-word" bits, then the parity is preserved in each of these eight-bit sets as well. Thus, fully 64 bits of parity information is preserved in E512.

The 512-bit block of data used as input to E512 and the 512-bit block of data produced as output are related by the preservation of parity in these 64 disjoint subsets of bits. What cryptographic implications does this preservation have?

First and most obviously, it shows that E512 is unsuited for use as a conventional encryption function (even ignoring the fact that E512 does not specify any method of mixing in key material). Secondly, it means that at least 64 bits (one bit from each subset of the 64 subsets of bits whose parity is preserved, i.e., two consecutive 32-bit words) must be thrown away from the output of E512 if we wish to avoid this problem.

We can more carefully analyze the implications of this observation in the following way: presume, for the moment, that every table lookup in the computation of E512 was in fact random. That is, instead of taking the index from the bottom eight bits of some 32-bit word in the 512-bit block being hashed, instead we imply selected a random eight-bit value, used this as an index into the S-box, and took the resulting 32-bit entry and XORed it with the preceding and following 32-bit words. What patterns would be preserved in the 512-bit output block?

We can view this in the following way. First, we focus on the 16 bits in a single-bit column. Then, we further focus on the bits from the odd words in such a column. (We could equally focus on the bits from the even words, it makes no difference.) These eight bits are modified every time we do a table lookup and two XORs. The XORs modify these eight bits by either flipping or not flipping two adjacent bits. That is, when we do two XORs on two odd words, we either alter two adjacent bits in our set of eight, or we leave all the bits alone. More concisely, we are XORing our eight-bit vector with one of the following eight-bit patterns:

1000 0001
1100 0000
0110 0000
0011 0000
0001 1000
0000 1100
0000 0110
0000 0011

What we note immediately about this set of eight vectors is that *they are not linearly independent*. This is the essence of Coppersmith's observation. However, although they are not linearly independent, we *do* have a matrix of rank 7. That is, this matrix can be used to generate all possible even-parity combinations. This means any subset of seven bits selected from our eight-bit subset of bits will be entirely random. Put another way, throwing away one bit of the eight is sufficient to ensure that the remaining seven bits appear random.

Because HASH throws away fully 384 bits of the output of E512, and STRONGERHASH throws away 256 bits of the output of E512, the specific use of E512 in

these two cases is safe. However, any use of E512 which does not throw out at least two adjacent words would not be safe. Therefore, it would be unwise to use E512 as the basis for the design of an encryption function.

References

1. Secrecy, Authentication, and Public Key Systems, by Ralph C. Merkle, Ph.D. thesis, Stanford University, 1979.
2. A Certified Digital Signature: that antique paper from 1979, *Advances in Cryptology—Crypto* '89, Lecture Notes on Computer Science, Vol. 435, Springer-Verlag, Berlin, pages 218–238.
3. Universal one-way hash functions and their cryptographic applications, by Moni Naor and Moti Yung, *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, Seattle, Washington, May 15–17, 1989, pages 33–43.
4. A high speed manipulation detection code, by Robert R. Jueneman, *Advances in Crytopology—Crypto* '86, Lecture Notes on Computer Science, Vol. 263, Berlin, pages 327–346.
5. Another birthday attack, by Don Coppersmith. *Advances in Cryptology—Crypto* '85, Lecture Notes on Computer Science, Vol. 218, Springer-Verlag, Berlin, pages 14–17.
6. A digital signature based on a conventional encryption function, by Ralph C. Merkle, *Advances in Cryptology—Crypto* '87, Lecture Notes on Computer Science, Vol. 293, Springer-Verlag, Berlin, pages 369–378.
7. *Cryptography and Data Security*, by Dorothy E. R. Denning, Addison-Welsey, Reading, MA, 1982, page 170.
8. On the security of multiple encryption, by Ralph C. Merkle, *Communication of the Association for Computing Machinery*, Vol. 24, No. 7, July 1981, pages 465–467.
9. Results of an initial attempt to cryptanalze the NBS Data Encryption Standard, by Martin Hellman et al., Information System Lab. Report SEL 76-042, Stanford University, 1976.
10. Communication theory of secrecy systems, by C. E. Shannon, *Bell Systems Technical Journal*, Vol. 28, Oct. 1949, pages 656–715.
11. Message authentication, by R. R. Jueneman, S. M. Matyas, and C. H. Meyer, *IEEE Communications Magazine*, Vol. 23, No. 9, September 1985, pages 29–40.
12. Generating strong one-way functions with cryptographic algorithm, by S. M. Matyas, C. H. Meyer, and J. Oseas, *IBM Technical Disclosure Bulletin*, Vol. 27, No. 10A, March 1985, pages 5658–5659
13. Analysis of Jueneman's MDC Scheme, by Don Coppersmith, preliminary version, June 9, 1988. Analysis of the system presented in [4].
14. The Data Encryption Standard: past and future, by M. E. Smid and D. K. Branstad, *Proceedings of the IEEE*, Vol. 76, No. 5, May 1988, pages 550–559.
15. *Defending Secrets, Sharing Data: New Locks and Keys for Electric Information*, U.S. Congress, Office of Technology Assessment, OTA-CIT-310, U.S. Government Printing Office, Washington, October 1987
16. Exhaustive cryptanalysis of the NBS data encryption standard, by Whitfield Diffie and Martin Hellman, *Computer*, June 1977, pages 74–78.
17. *Cryptography: A New Dimension in Data Security*, by Carl H. Meyer and Stephen M. Matyas, Wiley, New York, 1982.
18. One Way Hash Functions and DES, by Ralph C. Merkle, Crypto '89.
19. *Data Encryption Standard (DES)*, Federal Information Processing Standards Publication 46, National Bureau of Standards (U.S.), National Technical Information Service, Springfield, VA, April 1977.
20. Cryptography and Computer Privacy, by H. Feistel, *Scientific American*, Vol. 228, No. 5, May 1973, pages 15–23.
21. Maximum Likelihood Estimation Applied to Cryptanalysis, by Dov Andelman, Ph.D. thesis, Stanford University, 1979.

22. Secure program code with modification detection code, by Carl H. Meyer and Michael Schilling, *Proceedings of the Fifth Worwide Congress on Computers and Communication Security and Protection—Securicom* '88, SEDEP, Paris, pages 111–130.
23. Cryptography—a state of the art review, by Carl H. Meyer, *Proceedings of the Third Annual European Computer Conference*—Comeuro '89, Hamburg, May 8–12, 1989, pages 150–154.
24. Design Principles for Hash Functions, by Ivan Damgaard, Crypto '89.
25. Don Coppersmith, private communication.
26. The MD4 Message Digest Algorithm, by Ron Rivest, Crypto '90.
27. Unpublished cryptanalysis of the 2-pass version of Snefru by Eli Biham.