Tiger: A Fast New Hash Function

Ross Anderson[1] and Eli Biham[2]

[1] Cambridge University, England; email `rja14@cl.cam.ac.uk`
[2] Technion, Haifa, Israel; email `biham@cs.technion.ac.il`

Abstract. Among those cryptographic hash function which are not based on block ciphers, MD4 and Snefru seemed initially quite attractive for applications requiring fast software hashing. However collisions for Snefru were found in 1990, and recently a collision of MD4 was also found. This casts doubt on how long these functions' variants, such as RIPE-MD, MD5, SHA, SHA1 and Snefru-8, will remain unbroken. Furthermore, all these functions were designed for 32-bit processors, and cannot be implemented efficiently on the new generation of 64-bit processors such as the DEC Alpha. We therefore present a new hash function which we believe to be secure; it is designed to run quickly on 64-bit processors, without being too slow on existing machines.

1 Motivation and Design Requirements

Cryptographic hash functions are very important for cryptographic protocols. When used with signature schemes, their role is to reduce the amount of data which must be signed [Pre93] and to break up any properties such as multiplicative homomorphism which might be exploited by an opponent [And93]. In short, they need to be both efficient and secure; and in most commercial applications, they need to run quickly in software on all the common hardware platforms.

Some hash functions are based on feedforward modes of block ciphers [Pre93], but the main contenders have been the functions based on MD4 [Riv90], which include MD5 [Riv92], RIPE-MD [RACE95], SHA [NIST92] and SHA-1 [NIST95]. Another family was Snefru, and its derivative Snefru-8 [Mer90].

However, collisions for Snefru were found in 1990 [BS91] [BS93], and recently a collision of MD4 has also been found [Dob95]. These attacks cast doubt on the security of the other members of these families. One may only speculate at how long each function will remain unbroken; however it seems prudent to start work now on replacements.

From the performance point of view, all the functions mentioned above were designed for 32-bit processors. The next generation of processors has 64-bit words, and includes the DEC Alpha series as well as forthcoming processors from Intel, HP and IBM. It seems reasonable to assume that, with the exception of microcontrollers used in embedded applications, the majority of systems will use 64-bit processors within five years or so. However, on such processors, the above families of hash functions cannot be implemented efficiently.

For example, the MD family uses many 32-bit rotations and additions, so a 64-bit register can only handle one 32-bit value at a time, which decreases the potential speed by a factor of about two. Moreover, the Alpha architecture does not have any rotation operations, whether 64-bit or 32-bit.

From these considerations, we believe that a next generation hash function:

- should be secure. At the very least it must be one-way, collision-free and multiplication-free;
- should run quickly on 64-bit processors, and yet not run too slowly on the already fielded 32-bit machines such as Intel's 80486;
- should, insofar as possible, be usable as a drop-in replacement for MD4, MD5, SHA and SHA-1.

2 Our Proposal

In this paper we propose a new hash function, which is called Tiger, as it is strong and fast: as fast as SHA-1 on 32-bit processors, and about three times faster on 64-bit (DEC Alpha) processors. It is also expected to be faster than SHA-1 on 16-bit processors, since SHA-1 is optimized for 32-bit machines, while our proposal is designed to work adequately on many word sizes.

Its main operation is table lookup into four S-boxes, each from eight bits to 64 bits. On 32-bit machines this can be implemented as a pair of table lookups, with the offset computation done only once. The other operations are 64-bit additions and subtractions, 64-bit multiplication by small constants (5, 7 and 9), 64-bit shifts and logical operations such as XOR and NOT. All these operations are at most twice as slow on 32-bit machines, with the exception of the shifts and the multiplications by small constants which are four or five times slower (Alpha processors have special instructions which multiply by constants of the form 4 ± 1 and 8 ± 1).

For drop-in compatibility, we adopt the outer structure of the MD4 family: the message is padded by a single '1' bit followed by a string of '0's and finally the message length as a 64-bit word. The result is divided into n 512-bit blocks.

The size of the hash value, and of the intermediate state, is three words, or 192 bits. This value was chosen for the following reasons:

1. Since we use 64-bit words, the size should be a multiple of 64;
2. To be compatible with applications using SHA-1, the hash size should be at least 160 bits;
3. All the successful shortcut attacks on existing hash functions attack the intermediate state, rather than the final hash value. The attacker typically chooses two colliding values for an intermediate block, and this propagates to a collision of the full function. However, these attacks would not work if the intermediate hash values were larger.

Tiger with the full 192 bits of output in use may be called Tiger/192, and we recommend its use in all new applications. When replacing other functions in existing applications, we suggest two shorter variants:

1. Tiger/160: the hash value is the first 160 bits of the result of Tiger/192, and is used for compatibility with SHA and SHA-1;
2. Tiger/128: the hash value is the first 128 bits of the result of Tiger/192, and is used for compatibility with MD4, MD5, RIPE-MD, the Snefru variants and some hash functions based on block ciphers.

We conjecture that all the three variants of Tiger are collision-free, in that collisions for Tiger/N cannot be found with substantially less effort than $O(2^{N/2})$. We also believe that they are one-way and multiplication-free [And93].

The efficiency of this function is partially based on the potential parallelism in its design. In the MD and Snefru families, each operation depends directly on the result of the previous operation, and thus RISC processors cannot be used efficiently due to pipeline stalls. In each round of Tiger, the eight table lookup operations can be done in parallel, so compilers can make best use of pipelining. The design also allows efficient hardware implementation.

The memory size required by Tiger is only slightly more than the size of the four S boxes. If this can be accommodated within the cache of the processor, the computation runs about twice as fast (measured on DEC Alpha). The size of the four S boxes is $4 \cdot 256 \cdot 8 = 8096 = 8$ Kbytes, which is about the size of the cache on most machines. If eight S boxes were used, 16 Kbytes would be required, which is twice as the size of the cache on Alpha.

3 Specification

In Tiger all the computations are on 64-bit words, in little-endian/2-complement representation. We use three 64-bit registers called a, b, and c as the intermediate hash values. These registers are initialized to h_0 which is:

```
a = 0x0123456789ABCDEF
b = 0xFEDCBA9876543210
c = 0xF096A5B4C3B2E187
```

Each successive 512-bit message block is divided into eight 64-bit words x0, x1, ..., x7, and the following computation is performed to update h_i to h_{i+1}.

This computation consists of three passes, and between each of them there is a *key schedule* — an invertible transformation of the input data which prevents an attacker forcing sparse inputs in all three rounds. Finally there is a feedforward stage in which the new values of a, b, and c are combined with their initial values to give h_{i+1}:

```
save_abc
pass(a,b,c,5)
key_schedule
pass(c,a,b,7)
key_schedule
pass(b,c,a,9)
feedforward
```

where

1. `save_abc` saves the value of h_i

   ```
   aa = a ;
   bb = b ;
   cc = c ;
   ```

2. `pass(a,b,c,mul)` is

   ```
   round(a,b,c,x0,mul);
   round(b,c,a,x1,mul);
   round(c,a,b,x2,mul);
   round(a,b,c,x3,mul);
   round(b,c,a,x4,mul);
   round(c,a,b,x5,mul);
   round(a,b,c,x6,mul);
   round(b,c,a,x7,mul);
   ```

   where `round(a,b,c,x,mul)` is

   ```
   c ^= x ;
   a -= t1[c_0] ^ t2[c_2] ^ t3[c_4] ^ t4[c_6] ;
   b += t4[c_1] ^ t3[c_3] ^ t2[c_5] ^ t1[c_7] ;
   b *= mul;
   ```

   and where `c_i` is the ith byte of c ($0 \leq i \leq 7$). Note that we use the notation of the C programming language, where `^` denotes the XOR operator, and the notation `X op= Y` means `X = X op Y`, for any operator `op`. The S boxes t1 to t4 would take ten pages to publish here, so they will be published electronically along with the full source code, and made available from the authors' world wide web home pages.

3. `key_schedule` is

```
x0 -= x7 ^ 0xA5A5A5A5A5A5A5A5;
x1 ^= x0;
x2 += x1;
x3 -= x2 ^ ((~x1)<<19);
x4 ^= x3;
x5 += x4;
x6 -= x5 ^ ((~x4)>>23);
x7 ^= x6;
x0 += x7;
x1 -= x0 ^ ((~x7)<<19);
x2 ^= x1;
x3 += x2;
x4 -= x3 ^ ((~x2)>>23);
x5 ^= x4;
x6 += x5;
x7 -= x6 ^ 0x0123456789ABCDEF;
```

   where `<<` and `>>` are logical (rather than arithmetic) shift left and shift right operators.

4. `feedforward` is

```
a ^= aa ;
b -= bb ;
c += cc ;
```

   The resultant registers a, b, c are the 192 bits of the (intermediate) hash value h_{i+1}.

Figure 1 describes the compression function. In this figure the black area denotes the affected registers, where the slanted lines point to the affecting bytes in the white area. The variables y0, y1, ..., y7, and z0, z1, ..., z7 denote the values of x0, x1, ..., x7 in the second and the third passes, respectively. Finally, the last intermediate value h_n is taken as the output of Tiger/192.

4 Security

1. The nonlinearity comes mostly from S-boxes from 8 bits to 64 bits. This is much better than merely combining additions and XORs (i.e., using the carry bits), and it affects all the output bits, not just neighboring bits.
2. There is a strong avalanche, in that each message bit affects all the three registers after three rounds — much faster than in any other hash function. The avalanche in 64-bit words (and 64-bit S boxes) is much faster than when shorter words are used.

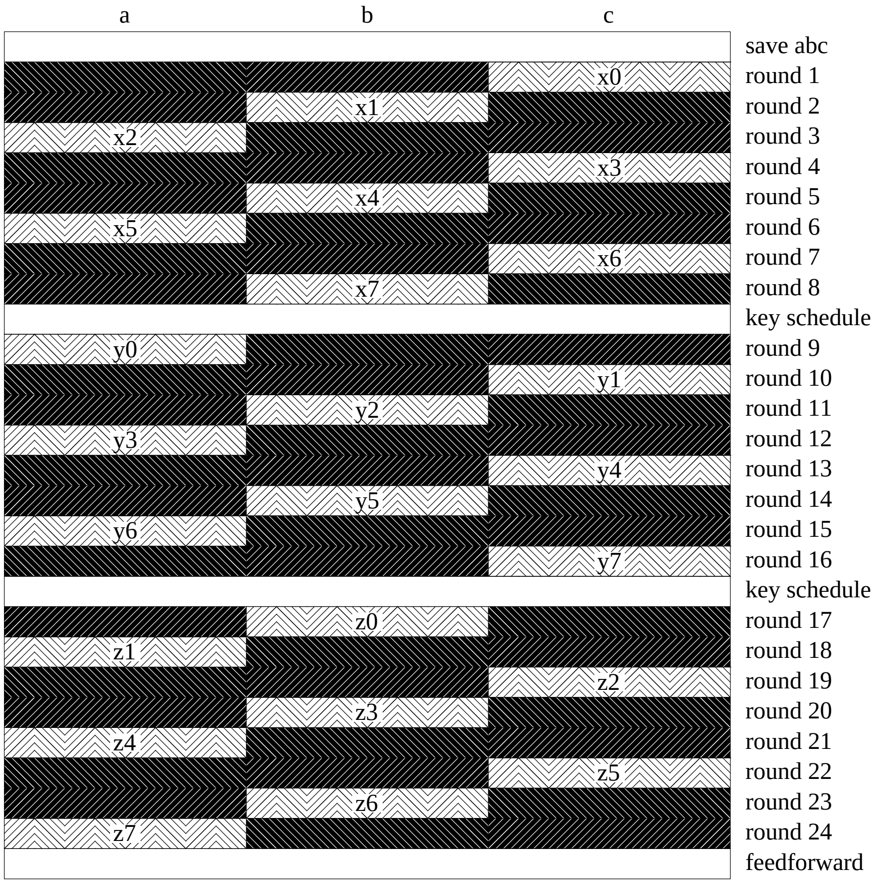

Fig. 1. Outline of the compression function of Tiger

3. As remarked above, all shortcut attacks on MD*/Snefru target one of the intermediate blocks. Increasing the intermediate value to 192 bits helps thwart these attacks.
4. The key schedule ensures that changing a small number of bits in a message affects many bits during the various passes. Together with the strong avalanche, it helps Tiger to resist attacks similar to Dobbertin's differential attack on MD4 (where changing certain bits in the message affects at most two bits in many rounds, and then these small differences can be made to cancel out in the last pass).
5. The multiplication of the register `b` in each round also contributes to the resistance to such attacks, since it ensures that bits which were used as inputs to S boxes in the previous rounds are mixed into other S boxes as well, and

to the same S boxes with a different input difference. This multiplication also prevents related-key [B94] attacks on the hash function, since the constant differs in each round.
6. The feedforward prevents meet-in-the-middle birthday attacks that find preimages of the hash function (although their complexity would be 2^{96} anyway).

5 Summary

In this paper we have put forward a new hash function, called Tiger, which is designed to be both fast and secure. Its core is three rounds, each of which uses eight lookups into 8-to-64-bit S-boxes to provide a strong nonlinear avalanche plus a number of register operations to increase diffusion and make differential attacks harder.

It can be implemented efficiently on 32-bit and 64-bit machines. On the former it is as fast as SHA1, but unlike SHA1, it can use the full power of 64-bit machines, on which it is about 2.5 times faster than SHA1. It can also be implemented on 16-bit machines, on which it should still be faster than SHA1.

It outputs 192-bit hash values. For compatibility with existing hash functions, we suggest that its output can be truncated to 160 or 128 bits if required for compatibility with existing applications. We believe that even these shortened variants are more secure than existing functions of the same output length; however if the ultra-cautious wish to add extra passes to Tiger, then they are welcome to do so, and we suggest a multiplicative constant of 9 in all the extra passes. We call these variants Tiger#, or Tiger#/N, where # is the number of passes, and N is the number of bits in the hash value.

As usual when suggesting a new cryptographic primitive, we urge people to study the strength of Tiger; we will appreciate attacks, analysis and any other comments. More information on the current status of Tiger, an updated copy of this paper, and reference implementations, will be available at the authors' home pages at the URLs: `http://www.cs.technion.ac.il/~biham/` and `http://www.cl.cam.ac.uk/users/rja14/`.

References

[And93] RJ Anderson, "The Classification of Hash Functions", in *'Codes and Ciphers', proceedings of Fourth IMA Conference on Cryptography and Coding*, pp 83–93

[BS91] E Biham, A Shamir, "Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer (extended abstract)", in *Advances in Cryptology — CRYPTO '91*, pp 156–171

[BS93] E Biham, A Shamir, *'Differential Cryptanalysis of the Data Encryption Standard'* (Springer 1993)

[B94] E Biham, "New Types of Cryptanalytic Attacks Using Related Keys" in *Journal of Cryptology v 7 no 4 (1994) pp 229–246*

[Dob95] H Dobertin, "MD4 is not collision-free" *preprint, September 1995*

[Mer90] RC Merkle, "A Fast Software One-Way Hash Function" in *Journal of Cryptology v 3 no 1 (1990) pp 43–58*

[NIST92] National Institute of Standards and Technology, *'Secure Hash Standard'*, FIPS 186, US Department of Commerce, January 1992

[NIST95] National Institute of Standards and Technology, *'Secure Hash Standard'*, FIPS 186-1, US Department of Commerce, April 1995

[Pre93] B Preneel, *'Analysis and Design of Cryptographic Hash Functions'*, PhD Thesis, Catholic University of Leuven 1993.

[Riv90] RL Rivest, "The MD4 message-digest algorithm", in *Advances in Cryptology — CRYPTO '90*, Springer LNCS v 537 pp 303–311; also Internet RFC 1320, April 1992

[Riv92] RL Rivest, "The MD5 message-digest algorithm", Internet RFC 1321, April 1992

[RACE95] *'Integrity Primitives for Secure Information Systems'*, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040, Springer LNCS v 1007, 1995.

Appendix — Source for the Compression Function of Tiger

```
word64 t1[256] = {...};
word64 t2[256] = {...};
word64 t3[256] = {...};
word64 t4[256] = {...};

TIGER_compression_function (state, block)
word64 state[3];
unsigned word64 block[8];
{
  word64 a = state[0], b = state[1], c = state[2];
  word64 x0=block[0], x1=block[1], x2=block[2], x3=block[3],
         x4=block[4], x5=block[5], x6=block[6], x7=block[7];
  word64 aa, bb, cc;

#define save_abc aa = a; bb = b; cc = c;

#define round(a,b,c,x,mul) \
      c ^= x; \
      a -= t1[((c)>>(0*8))&0xFF] ^ t2[((c)>>(2*8))&0xFF] ^ \
           t3[((c)>>(4*8))&0xFF] ^ t4[((c)>>(6*8))&0xFF] ; \
      b += t4[((c)>>(1*8))&0xFF] ^ t3[((c)>>(3*8))&0xFF] ^ \
           t2[((c)>>(5*8))&0xFF] ^ t1[((c)>>(7*8))&0xFF] ; \
      b *= mul;
```

```
#define pass(a,b,c,mul) \
      round(a,b,c,x0,mul) \
      round(b,c,a,x1,mul) \
      round(c,a,b,x2,mul) \
      round(a,b,c,x3,mul) \
      round(b,c,a,x4,mul) \
      round(c,a,b,x5,mul) \
      round(a,b,c,x6,mul) \
      round(b,c,a,x7,mul)

#define key_schedule \
      x0 -= x7 ^ 0xA5A5A5A5A5A5A5A5; \
      x1 ^= x0; \
      x2 += x1; \
      x3 -= x2 ^ ((~x1)<<19); \
      x4 ^= x3; \
      x5 += x4; \
      x6 -= x5 ^ ((~x4)>>23); \
      x7 ^= x6; \
      x0 += x7; \
      x1 -= x0 ^ ((~x7)<<19); \
      x2 ^= x1; \
      x3 += x2; \
      x4 -= x3 ^ ((~x2)>>23); \
      x5 ^= x4; \
      x6 += x5; \
      x7 -= x6 ^ 0x0123456789ABCDEF;

#define feedforward a ^= aa; b -= bb; c += cc;

#define compress \
      save_abc \
      pass(a,b,c,5) \
      key_schedule \
      pass(c,a,b,7) \
      key_schedule \
      pass(b,c,a,9) \
      feedforward

  compress;

  state[0] = a; state[1] = b; state[2] = c;
}
```

This article was processed using the LaTeX macro package with LLNCS style