Algorithm Specification

Introduction

This paper specifies the Maraca keyed hash algorithm, explains its design decisions and constants, and does some cryptanalysis of it.

Maraca runs in 5.3 cycles/byte in software, and up to 42 bytes/cycle in hardware. Maraca take about 28200 cycles in software to hash the empty string. Its requires 6688 of memory bytes, though it could be done in under 512 bytes with random access to an insecure sliding window of 7K of user data. Avalanche takes 7 permutations, but a new block is accepted every 3 permutations. Every block is used four times. The block usage pattern guarantees that any self-cancelling delta passes through at least 30 permutations. Its security when the result is truncated to n bits, n at most 512, is expected to be as good as any hash reporting n bits can be. The result can be up to 1024 bits, though that still only guarantees 512 bits of security. The hash was named Maraca because it was required to have a name, and it consists of many little pieces being shaken up and rattled around.

Maraca has a 1024-bit internal state divided up into 64 chunks of 16 bits each. A 1024-bit permutation applies a nonlinear 8-bit permutation to the even and odd bits of every 16-bit chunk, then shuffles bits. Below is a display of the percentage of the time each state bit is changed when bit 0 of chunk 0 is changed and the 1024-bit permutation is reversed three times. It shows 64 16-bit chunks (both chunks and bits within chunks are left to right, top to bottom). It shows the correlation between the even (odd) bits within a chunk due to the 8-bit permutation. A similar number of bits would be affected if the 1024-bit permutation were run forward three times, but the affected bits would show no visible pattern. Exchanging bits between chunks is the last thing done forward, but the first thing done in reverse, so in reverse the chunk structure is still visible.

Percentage of bitflips per bit from a 1-bit input delta after three 1024-bit permutations in reverse:

```
 5  0  3  0   0  0  0  0   0  0 29  0   0  0  0  0   0  0  0  9   0  0  0 35   7 23  4 28   0  6  0  4
 4  0  4  0   0  0  0  0  29  0 35  0   0  0  0  0   0  3  0  9   0 35  0 42   6 30  6 39   0  5  0  6
 4  0  4  0   0  0  0  0  29  0 47  0   0  0  0  0   0  3  0  3   0 35  0 56   6 30  6 38   0  5  0  6
 3  0  5  0   0  0  0  0  12  0 18  0   0  0  0  0   0  0  0  0   0 14  0 21   5 27  7 35   0  5  0  7

 0  0  0  0   0 16  0 14   0  0  0  0   0  0  0  0   0  0 28  0   0  0  0  0  19 14 19 14   0  8  0  8
 0  0  0  0   0 14  0 14   0  0  0  0   0  0  0  0   9  0 28  0   0  0  0  0  16 12 19 14   0  7  0  8
 0  0  0  0   0 18  0 16   0  0  0  0   0  0  0  0   9  0  9  0   0  0  0  0  21 16 19 14   0  9  0  8
 0  0  0  0   0 20  0 16   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0  24 18 19 14   0 10  0  8

 0  0  0  0   0  0  0  0  21  0 19  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
 0  0  0  0   0  0  0  0  26  0 24  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
 0  0  0  0   0  0  0  0  26  0 16  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
 0  0  0  0   0  0  0  0  21  0 26  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0

 0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0 14  0  8   0  0  0  0
 0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0 12  0 13   0  0  0  0
 0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0 12  0 12   0  0  0  0
 0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  9  0 14   0  0  0  0

 0  0 12  0   9  0 12  0   0  0  0  0   0  0  0  0   0 12  0  7   8  0  6  0   0  0  0  0   5 23  4 30
12  0 14  0   9  0 12  0   0  0  0  0   0  0  0  0   0 10  0 11   7  0  9  0   0  0  0  0   6 31  5 30
12  0 19  0   9  0 13  0   0  0  0  0   0  0  0  0   0 10  0  9   7  0  8  0   0  0  0  0   6 32  6 36
 5  0  7  0   8  0 10  0   0  0  0  0   0  0  0  0   0  9  0 11   8  0 10  0   0  0  0  0   5 32  7 28

 0 21  0 18   0  0  0  0   0  8 56  7   0  0  9  0   0  0  0  0   6  0  6  0   0  0  0  0   0  0  0  0
 0 19  0 24   0  0  0  0  19 10 56  9   3  0  9  0   0  0  0  0   5  0  6  0   0  0  0  0   0  0  0  0
 0 20  0 21   0  0  0  0  19 10 19  6   3  0  3  0   0  0  0  0   7  0  6  0   0  0  0  0   0  0  0  0
 0 23  0 26   0  0  0  0   0  8  0 10   0  0  0  0   0  0  0  0   8  0  6  0   0  0  0  0   0  0  0  0

23 19 20 19   5  0  5  0   3  0  3  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
29 16 26 19   7  0  6  0   3  0  3  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
29 21 17 19   7  0  4  0   4  0  3  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
23 23 29 19   5  0  7  0   4  0  3  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0

 8  0  7  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  6  0  5   0  0  0  0
 7  0  7  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  7  0  6   0  0  0  0
 9  0  8  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  7  0  6   0  0  0  0
10  0  8  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0   0  6  0  8   0  0  0  0
```

Specification

The keyed hash **Maraca(key,message)** is built in several layers:

- The key, which is between 0 and 1024 bits, and is a multiple of 128 bits
- The modified message, which is a function of the key, key length, original message, and original message length
- The modified message is a multiple of 1024 bits in length, and is chopped into 1024-bit data blocks
- A 1024-bit internal state consists of 64 16-bit chunks
- A 1024-bit permutation consists of:
    - applying an 8-bit permutation to the even and odd bits of all 64 chunks
    - breaking symmetry by XORing constants to the bottom 6 bits of all chunks
    - rearranging the 16 bits of chunks the same way in all 64 chunks
    - barrelshifting each of the 16 bits among the 64 chunks

- Each block is used four times, at irregular intervals over the course of 139 1024-bit permutations, by being XORed to the internal state.
- 30 permutations are applied after the final use of the final block.
- The final result is the final 1024-bit internal state. If fewer than 1024 bits of result are needed, the result is truncated.

Data structure

Any implementation of Maraca must store at least:

- A 1024-bit internal state
- The total length of the data seen, mod 1024
- A key, which is up to 1024 bits in length
- The length of the key (which is required to be a multiple of 128 bits)
- The current data block, which may be up to 1024 bits in length
- 47 1024-bit accumulators

The 47 1024-bit accumulators can be omitted if random access into a 7K sliding window of the original message is used instead. The pattern of accesses would not need to be secret. The requirement of a 7K sliding window would disallow messages from being added in pieces, unless the end of the old piece is always available when the start of the new piece is being hashed. The 47 accumulators allow data blocks to be read just once, in order.

Since Maraca only makes use of the length mod 1024 bits, messages of unlimited length can be hashed. Any implementation would need to track which data blocks have been hashed so far, so implementations will need to store the full length, or at least the full length of the current piece.

The reference and optimized implementations given actually store

- the 1024-bit internal state (hash)
- a 1024-bit key (key)
- a 1024-bit aligned buffer (buf) storing the current (possibly partial) data block
- room for 47 1024 bit accumulators (abuf)
- the optimized version has a ring buffer of 64 pointers to the accumulators (a)
- an 8-byte length (length)
- an int (running) to represent a boolean saying whether Update() and Final() may be called
- an int (hashbitlen) giving the requested message digest size
- an int (keybitlen) saying how big the key is
- an 8-byte length (offset) pointing into the ring buffer of accumulators

The optimized version uses 6688 bytes of memory. The reference version uses 6424 bytes of memory.

Producing the modified message from the key and message

Internally, Maraca(K,M) hashes a modified message M with a key K. The modified message is a multiple of 1024 bits in length, and is split into 1024-bit data blocks which are hashed consecutively.

Given an original message M with A bits, and a key K with B bits, the modified message is roughly K followed by M followed by K followed by lengths, with some padding. B is required to be a multiple of 128 bits in length. Let D be the number of padding bits needed to rightpad M to a whole byte. Let E be 1 + 2D + 16B. Then the modified message is K, followed by M, followed by D padding bits of value 0, followed by K again, followed by the 2-byte little-endian representation of E, followed by as many 0-valued bits as needed

to complete the last 1024-bit block.

```
M: original message
K: key
A = length(M) in bits;
B = length(K) in bits;
C = A mod 8;          // B mod 8 is required to be 0
if (C == 0) {
  D = 0;
} else {
  D = 8 - C;
}
E = 1 + 2*D + 16*B;    // E is a little-endian 16-bit quantity
F = D bits of value 0;
G = B + A + D + 16 + B;
H = G mod 1024;
if (H == 0) {
  I = 0;
} else {
  I = 1024 - H;
}
J = I bits of value 0;
Z = K || M || F || K || E || J;   // Z is the modified message
```

This is the only place the total original message length is used, and only the total original message length mod 1024 is needed. For short messages with short keys, the modified message may fit in one data block. It is possible for the final use of the key to be split across two data blocks, and it is possible for the lengths E and the final padding J to be the only thing in the final data block.

The construction of E allows keys to be any number of bits in 0..1024. The only reason for the restriction that keys must be a multiple of 128 bit is that it allows software to read the modified message in 16-byte chunks if the original message was aligned to a 16-byte boundary. Some padding or shifting would need to be specified to deal with keylengths that are not multiples of 8 bits.

The block schedule

The modified message is a multiple of 1024 bits. If the modified message has n*1024 bits, it is split into blocks numbered 0..n-1. Every block is XORed to the internal state four times, using a schedule of staggered offsets, interleaved with uses of other blocks. The j=1..4th use of every block is left-rotated by 128*j(j-1)/2 bits.

The initial vector for the 1024-bit internal state is all zero.

In the optimized and reference implementations, the ith data block is fetched before the 3ith permutation, then the first use (rotated by 0) is XORed directly to the internal state. (Like data blocks, permutations are numbered starting at 0.) The i+46th accumulator is assigned to the ith data block left-rotated by 6*128 bits. The (i+41-6((i+2)%4))th accumulator is XORed to the ith data block left-rotated by 3*128 bits. The (i+21-6(i%4))th accumulator is XORed to the ith data block left-rotated by 1*128 bits. Before the 3i+1th permutation, the ith accumulator is XORed to the internal state.

The code below, like the reference implementation, splits the 1024-bit state into 16 64-bit (8-byte) values. The optimized implementation splits it into 8 128-bit values. The code below assumes the message was modified to handle the key and lengths by preprocessing, but the reference and optimized implementation modify the original message in flight.

```
/* xor something to the internal state */
```

```
static void do_combine( const u8 *thing, u8 *hash)
{
  int k;
  for (k=0; k<16; ++k)
    hash[k] ^= thing[k];
}

/* add the ith block to its second, third, fourth accumulators */
void accumulate( u8 a[47][16], const u8 *block, int i)
{
  u8 *fourth = a[(i + 46) % 47];                /* 4th accumulator */
  u8 *third  = a[(i + 41 - 6*((i+2)%4)) % 47];  /* 3rd accumulator */
  u8 *second = a[(i + 21 - 6*(i%4)) % 47];      /* 2nd accumulator  */
  int k;

  for (k=0; k<16; ++k)              /* for every piece of a data block */
  {
    u8 val = block[k];
    fourth[(k+12) % 16]  = val;     /* left-rotated by 6*128 bits */
    third [(k+ 6) % 16] ^= val;     /* left-rotated by 3*128 bits */
    second[(k+ 2) % 16] ^= val;     /* left-rotated by 1*128 bits */
  }
}

/* use the ith block; do permutations 3i through 3i+2 */
static void one_combine( u8 a[47][16], u8 i, u8 *hash, const u8 *block)
{
  accumulate( a, block, i);         /* xor ith block to accumulators */
  do_combine( block, hash);         /* xor ith block to the state */
  perm( hash);
  do_combine( a[i % 47], hash);     /* xor ith accumulator to the state */
  perm( hash);
  perm( hash);
}

/* produce a 1024-bit hash value for a modified message m */
static void hash_modified_message( const u8 *m, u8 blocks, u8 hash[16])
{
  u8   abuf[47][16];
  u8   zero[16];
  u8 **a;
  u8   i;

  memset(abuf, 0, sizeof(abuf));    /* set all accumulators to all 0 */
  memset(hash, 0 sizeof(hash));     /* initial vector of all 0 */
  memset(zero, 0 sizeof(zero));     /* zero buffer is all 0 */

  for (i=0; i<blocks; ++i)          /* use all the blocks */
    one_combine(a, i, hash, m[i]);

  for (i=blocks; i<blocks+46; ++i)  /* use up the accumulators */
    one_combine(a, i, hash, zero);

  for (i=2; i<30; ++i)              /* 30 strengthening permutations */
    perm(hash);
}
```

An alternative implementation would not use accumulators, but instead would read the appropriate data blocks and XOR them to the internal state. Before the 3i+1th permutation, the i-46th block should be left-rotated by 6*128 bits and XORed to the internal state, the i-23-6(i%4)th block should be left-rotated by 3*128 bits and XORed to the internal state, and the i-3-6((i+2)%4)th block should be left-rotated by 1*128

bits and XORed to the internal state.

Nothing needs to be XORed when the pattern calls for uses of data blocks before the first or after the last block. Or equivalently, as in the code above and the reference implementation, an all-zero block could be used for nonexistent blocks.

After the final use of the final block, 30 more permutations are applied to the internal state to strengthen the message digest. The result of the hash is this final internal state, truncated to the desired length.

The 1024-bit permutation

The 1024-bit internal state consists of 64 16-bit chunks. In the reference implementation, the 64 16-bit chunks are stored as 16 64-bit (8 byte) values. The ith value holds the ith bit of all the chunks. The jth bit of all the values forms the jth chunk. In a 1024-bit data block, the ith bit becomes the (i/64)th bit of the (i%64)th chunk.

The 8-bit permutation

Mixing a 16-bit chunk uses the 8-bit permutation twice, on its even and odd bits. The C function eight(x,y) below calculates the 8-bit permutation on the even values of x for all 64 chunks in parallel and put the results in the even values of y. eight(&x[1],&y[1]) would handle the odd bits. ~m is m with all bits flipped, (m ^ n) is the XOR of m and n, (m | n) is the OR of m and n, and (m & n) is the AND of m and n. The type u8 is an unsigned 64-bit quantity.

```
typedef  unsigned long long  u8;

#define xor(a,b) (a ^ b)
#define or(a,b) (a | b)
#define and(a,b) (a & b)
#define ant(a,b) (~a & b)
#define not(a) ~a
#define read(x,i) x[i*2]
#define write(y,i,a) y[i*2] = a

/* apply the 8-bit permutation to the even bits of all 64 chunks */
static void eight( u8 *x, u8 *y)
{
  u8 q,r;
  u8 a0 = read(x,0);
  u8 a1 = read(x,1);
  u8 a2 = read(x,2);
  u8 a3 = read(x,3);
  u8 a4 = read(x,4);
  u8 a5 = read(x,5);
  u8 a6 = read(x,6);
  u8 a7 = read(x,7);

  q = xor(a4,a5);
  r = xor(a7,a0);
  write(y,0,xor(q,r));

  q = xor(a5,a1);
  r = xor(a3,a2);
  write(y,1,xor(q,r));

  q = xor(a3,a4);
  r = xor(a5,a1);
  write(y,2,xor(q,r));
```

```
  q = ant(xor(a0,a3),xor(a6,a4));
  r = ant(xor(a4,a6),or(and(a0,a1),ant(a0,a3)));
  write(y,3,or(q,r));

  q = ant(xor(a6,a7),xor(a5,a2));
  r = and(or(not(or(a2,a5)),and(a4,a5)),xor(a7,a6));
  write(y,4,or(q,r));

  q = and(xor(a5,a0),xor(a4,a2));
  r = ant(xor(a2,a4),or(and(a0,a5),ant(a0,a1)));
  write(y,5,or(q,r));

  q = and(xor(a4,a2),xor(a7,a6));
  r = ant(xor(a2,a4),or(ant(a7,a0),and(a6,a7)));
  write(y,6,or(q,r));

  q = and(or(not(or(a6,a7)),ant(and(a6,a7),a0)),xor(a4,a2));
  r = ant(xor(a2,a4),or(ant(and(a0,a2),a6),ant(a0,a7)));
  write(y,7,or(q,r));

}
```

The same 8-bit permutation could be calculated using NAND trees in hardware, given bits named a..h:

```
      ((((a e) ((a a) (e e))) ((f (h h)) (h (f f))))
       (((a (e e)) (e (a a))) ((f h) ((f f) (h h)))))

      ((((b c) ((b b) (c c))) ((d (f f)) (f (d d))))
       (((b (c c)) (c (b b))) ((d f) ((d d) (f f)))))

      ((((b d) ((b b) (d d))) ((e (f f)) (f (e e))))
       (((b (d d)) (d (b b))) ((e f) ((e e) (f f)))))

      ((((a d) ((a a) (d d))) ((e (g g)) (g (e e))))
       (((a b) (d (a a))) ((e g) ((e e) (g g)))))

      ((((c (f f)) (f (c c))) ((g h) ((g g) (h h))))
       (((e f) ((c c) (f f))) ((g (h h)) (h (g g)))))

      ((((a (f f)) (f (a a))) ((c (e e)) (e (c c))))
       (((a f) (b (a a))) ((c e) ((c c) (e e)))))

      ((((a (h h)) (g h)) ((c e) ((c c) (e e))))
       (((c (e e)) (e (c c))) ((g (h h)) (h (g g)))))

      ((((a (g h)) ((g g) (h h))) ((c (e e)) (e (c c))))
       (((c e) ((c c) (e e))) ((g (a c)) (h (a a)))))
```

The same 8-bit permutation could be calculated by lookup in a 256-element table with 8-bit values:

```
static int eight_tab[256] =
{
    0, 65, 38, 79,146,179,148,189, 14, 71, 40, 73,156,181,154,187,
  141,164,139,162, 31, 86, 57, 80,131,170,133,172, 17, 88, 55, 94,
   23,118, 49,120,165,132,163,138, 25,112, 63,126,171,130,173,140,
  186,147,188,149,  8, 97, 46,103,180,157,178,155,  6,111, 32,105,
  152,209,190,215, 74,227, 76,229,150,223,176,217, 68,237, 66,235,
   85,244, 83,250,135, 70,161, 72, 91,242, 93,252,137, 64,175, 78,
  143,230,169,224,109,196,107,194,129,232,167,238, 99,202,101,204,
  114,211,116,221,144,113,182,127,124,213,122,219,158,119,184,121,
```

```
  145, 16,183, 30, 67,226, 69,236,159, 22,185, 24, 77,228, 75,234,
   92,245, 90,243,142,  7,168,  1, 82,251, 84,253,128,  9,166, 15,
  134, 39,160, 41,100,197, 98,203,136, 33,174, 47,106,195,108,205,
  123,210,125,212,153, 48,191, 54,117,220,115,218,151, 62,177, 56,
  201,192,239,198, 27, 50, 29, 52,199,206,225,200, 21, 60, 19, 58,
    4, 37,  2, 43,214, 87,240, 89, 10, 35, 12, 45,216, 81,254, 95,
  222,247,248,241, 44,  5, 42,  3,208,249,246,255, 34, 11, 36, 13,
   51, 18, 53, 28,193, 96,231,110, 61, 20, 59, 26,207,102,233,104
}
```

The section "How the 8-bit permutation was chosen" explains the origin of this 8-bit permutation. The optimized implementation uses logic (xor, etc) to compute the 8-bit permutation. The reference implementation uses a lookup table.

The 1024-bit permutation

The 1024-bit permutation first applies the 8-bit permutation to the even bits and odd bits of every 16-bit chunk. Then it XORs these constants to the 64-bit values representing the bottom 6 bits:

```
0: 0x18f8aa72369b75c2LL;
1: 0x337b824aab77201fLL;
2: 0x60bd51315e37b49cLL;
3: 0x82ed31eb138e02efLL;
4: 0x5fe101ed66fc3130LL;
5: 0x1019906dca58dffbLL;
```

"Symmetry-breaking constants" explains why these are needed and how these constants were generated.

The 1024-bit permutation then left-barrelshifts each of the 16 values by these amounts:

```
static int shift[16] = {14,55,61,35,30,60,32,49,16,4,42,5,26,11,47,17};
```

"Exchanging bits between chunks" explains how those constants were chosen. Then it shuffles the 16 values using this mapping:

```
static int map[LEN] = {2,12,8,3,1,15,11,0,14,5,13,7,9,4,10,6};
for (k=0; k<16; ++k) new[map[k]] = old[k];
```

"The 16-bit and 8-bit permutations" explains how the shift constants were chosen.

Putting it all together,

```
static int shift[16] = {14,55,61,35,30,60,32,49,16,4,42,5,26,11,47,17};
static int map[16] = {2,12,8,3,1,15,11,0,14,5,13,7,9,4,10,6};

static void perm(u8 *x)
{
  int i;
  u8 y[16];

  /* do 128 8-bit permutations */
  eight( x, y);           /* permute the eight even bits */
  eight( &x[1], &y[1]);   /* permute the eight odd bits */

  /* break symmetry among the 64 16-bit permutations */
  y[0] ^= 0x18f8aa72369b75c2LL;
  y[1] ^= 0x337b824aab77201fLL;
  y[2] ^= 0x60bd51315e37b49cLL;
  y[3] ^= 0x82ed31eb138e02efLL;
```

```
  y[4] ^= 0x5fe101ed66fc3130LL;
  y[5] ^= 0x1019906dca58dffbLL;

  /* rotate the output bits among the 64 chunks */
  for (i=0; i<16; ++i)
  {
    y[i] = (y[i] << shift[i]) | (y[i] >> (64-shift[i]));
  }

  /* shuffle the 16 output bits within each chunk */
  for (i=0; i<16; ++i)
  {
    x[map[i]] = y[i];
  }
}
```

Efficiency: Space and Speed

Maraca would be slower if it used a smaller internal state, so it always internally produces a 1024 bit hash. All sizes of message digests have the same speed, since they all produce a 1024 bit hash then truncate it to the desired length.

Maraca does not have a set up cost. It does do a minimum of 169 permutations, even for the empty string, with every additional 128-byte block requiring an additional 3 permutations.

The 32-bit optimized version and 64-bit optimized version are the same. It uses 128-bit SSE intrinsics for almost everything except control logic. The reference version uses 64-bit operations for most things, but rearranges bits so it can do 8-bit lookups for the 8-bit permutation. The reference version is over 150 times slower than the optimized version.

In software, the optimized implementation of Maraca runs in 5.3 cycles/byte on a Intel dual E5320 1.86Ghz quad-core Dell machine with 2GB RAM running 64-bit Windows Vista compiled with Visual Studio 2008 cl /O2. It is still 5.3 cycles/byte per copy if 8 copies of it are run simultaneously. It runs in 5.5 cycles/byte on a Intel 6300 1.86GHz dual-core machine with 3GB RAM running 32-bit Windows XP compiled with gcc -O3 -msse2 (gcc version 3.4.4 from Cygwin). It was timed on a 2^{30} byte input (the same 64KB message added 16K times) with GetTickCount() from windows.h . The optimized implementation probably runs between 5.3 and 5.5 cycles per byte on both reference platforms.

8-bit machines need to do 16 operations per SSE operation used, so they are probably 16 times slower than the SSE implementation, that would be 90 cycles/byte. A more serious limitation is that Maraca uses nearly 7K of memory. This will prevent it from running on some very small platforms. If a platform has random access into a 7K sliding window of the input, it could refetch blocks when needed rather than fetching them once and caching them. A secure implementation of Maraca could then use under 512 bytes of secret memory (128 bytes of state, up to 128 bytes of key, miscellaneous registers). The fetch pattern is deterministic so the extra fetches need not reveal anything about the internal state.

The 8-bit permutation was counted as requiring 182 NAND gates, though it would be less if shared subexpressions were taken into account. There are 128 8-bit permutations, so a hardware implementation is estimated to require around 23296 gates. One implementation is to do a 1024-bit permutation, XOR a block, then store the results in flipflops. That's an estimate of 23K+(3+2)K=28K gates, not counting memory.

In hardware constructed purely out of NAND gates, Maraca requires 5 gate delays per permutation for the 8-bit permutations, plus 6 gate delays every three permutations to do XOR a data block and an accumulator. More gate delays, and some waiting, would be required whenever clocked flipflops store results. Breaking

symmetry does not require extra gates delays, nor does scrambling the bits within chunks or rotating bits among the chunks. That totals 3*5+6=21 gate delays per 1024-bit block. Three way or domino logic would be faster. Hardware could do a single permutation (5 gate delays) and xor (3 gate delays) and store the results in flipflops (1 gate delay?) in a cycle (12 gates delays?). A new 128-byte block could be hashed every three cycles. This is probably faster than memory can be fed to it.

The optimized implementation takes about 28200 cycles to hash the empty string in software. This was timed on the Intel dual E5320 1.86Ghz quad-core Dell machine with 2GB RAM running 64-bit Windows Vista compiled with Visual Studio 2008 cl /O2 by hashing the empty string 2^{20} times and timing it with GetTickCount() from windows.h . Hashing the empty string in hardware would require a minimum of 47*21+28*5=1127 gate delays. Each block hashed in hardware takes at least 21 more gate delays.

Origin of constants

Maraca's known possible weaknesses are described in this paper. Maraca contains no intentional weaknesses or trapdoors.

Maraca was designed to be a SHA-3 candidate that was as fast as possible in hardware. Most of its constants were determined empirically, by hillclimbing trying to increase various measures. It is possible to verify they meet the measures described.

It is not possible give a way to rederive these exact constants. They were found by several months of computing, depending on when hillclimbing algorithms were tweaked and how, the seeds to pseudorandom number generators, the pseudorandom number generator algorithms used, the weightings of the competing criteria, and the timing of various power outages which ended some of the searches. The constants in Maraca contain less entropy than did the search that led to Maraca.

State size

Maraca was chosen to have a 1024-bit state. The biggest message digests SHA-3 requires are 512 bits. There are attacks that are blocked if the internal state is at least twice the size of the message digest. For example the long message attack described in the section on randomized hashing shows Maraca provides at most 512 bits of security.

A 1024-bit state also allows 128 8-bit permutations to be calculated in parallel using SSE instructions, making it faster in software than a similar design would be with a 512-bit state. Hardware is faster with a 1024-bit state than a 512-bit state, too.

A larger state would not have had further gains in software speed, and would have exceeded a Xeon's L1 data cache size. There were requests for SHA-3 to fit in small environments, for example under 512 bytes or under 4096 bytes. 512 bytes is not feasible with a design like this if blocks must be cached. 4096 bytes was plausible, but wasn't achieved.

The 16-bit and 8-bit permutation

Cryptography also often uses lookup tables to accomplish 8-bit permutations. Permutations in lookup tables can be much more thorough than the one used by Maraca. Looking up values in a 256-element table is hard to parallelize, so these use an 8-bit input to look up 64-bit or 128-bit values, perhaps representing many different 8-bit permutations in parallel. Maraca did not explore this path, largely because Whirlpool is already exploring it.

Another path is to keep a large internal state which is mixed a few bytes at a time, and perhaps sampled in a data-dependent way. For example, an adaptation of RC4 to hashing. These cannot be parallelized, so their hardware implementation is slower than the path taken by Maraca.

Cryptography often uses permutations built out of Feistel networks, A = A ^ f(B). Most of the mixing in a Feistel network comes from the XOR and all of the nonlinearity comes from f(B). Maraca uses a nonlinear 16-bit permutation that is not a Feistel network. It does mixing and nonlinearity simultaneously. A comparable Feistel network would have been 6 NANDs deep instead of 5. It would have eight input bits affect only themselves. Maraca's 16-bit permutation has every input bit used by at least three output bits, and there is no input delta (other than all zeros) that maps to any output delta with probability over 3/4.

Balanced boolean functions

How can all 5-NAND-deep permutations be constructed, rather than just Feistel networks? Every output bit of any n-bit permutation will be a balanced boolean function of n inputs, that is, it will produce 2^{n-1} zeros and 2^{n-1} ones. Any m output bits are also balanced, covering 0..(2^m-1) 2^{n-m} times. Given all balanced boolean functions with n inputs, any n-bit permutation could be constructed by the hillclimbing procedure of choosing functions for one output bit at a time and assuring that the first m bits always form an m-bit balanced function.

Hardware is often constructed out of NAND gates. A NAND (not-and) maps two input values (0,0), (0,1), (1,0), or (1,1) to 1, 1, 1, 0 respectively. A tree of NAND gates can be constructed to calculate any boolean function. The speed of hardware calculating a function is proportional to the depth of that tree. The speed of software calculating that function is proportional to the number of nodes in that tree. Software may go faster than that by sharing common subexpressions or by using multigate operations. Trees that were 5 NAND gates deep were explored for Maraca.

If changing all the inputs to a function by some delta causes an output bit to change with probability p, define the **effect** of that delta on that output bit as min(p, 1-p). Increasing the effect of deltas was used in many of the hillclimbing criteria.

Maraca first chose a large pool of good balanced boolean functions that were 5 NANDs deep, then constructed multibit permutations out of them. The criteria used to qualify a balanced boolean function as "good" was **bit-avalanche**, defined as the sum of the effects of the n 1-bit input deltas. Functions with bit-avalanche greater than 3.125 were allowed into the pool. Every balanced boolean function with n input bits has bit-avalanche between 1 and n. Balanced boolean functions with n inputs have an average bit-avalanche of n/2.

Is XOR optimal for bit-avalanche for a given NAND gate depth? It is optimal for trees of depth 1,3,5. The XOR of n input bits has bit-avalanche of n. The XOR of 2^n input bits can be calculated by a NAND tree of depth 2n+1. So can its complement. If X, Y are boolean functions and x, y are their complements, ((X NAND y) NAND (x NAND Y)) is (X XOR Y), and ((X NAND Y) NAND (x NAND y)) is its complement. For 4 input bits, XOR has bit-avalanche of 4 while the closest runner up that is 5 NAND gates deep (which looks very much like XOR) has bit-avalanche of 3.625. Still, no proof was found that XOR is optimal. A tree of depth 2n+1 can use up to 2^{2n+1} input bits. The average boolean function of 2^{2n+1} input bits has bit-avalanche of 2^{2n}, much greater than XOR's 2^n, but these functions generally require deeper trees.

A 5 gate deep tree is the NAND of two half trees up to 4 gates deep. If two half trees produce 1 with probabilities p and q, their NAND most likely produces 0 with probability pq. The half trees that produce balanced functions will produce 1 with average probability sqrt(1/2). The bit-avalanche of the full tree is at

most the sum of the bit-avalanches of its half trees. The maximum bit-avalanche for 4-deep trees was 2.25, achieved by some four, five, and six variable fucntions. No 4-deep tree with 7 variables or more had bit-avalanche over 2.125.

The space of 5 gate deep trees was too large to enumerate. Instead, all unbalanced boolean functions that were 4 NAND gates deep with 5 inputs or less were collected which had bit-avalanche over 1.5 and which produced 1 with probability between 0.625 and 0.75. Their full cross product was explored and 5-nand-deep balanced functions with bit-avalanche over 2.875 were accumulated. Although there were 6-variable balanced functions with average output deltas up to 3.125, no acceptable 8-bit or 16-bit permutations were ever found containing anything but 4-variable or 5-variable functions. (Perhaps there was a flaw in the search?) In the end only 4 and 5 variable balanced functions with bit-avalanche of 3.125 or above were used to generate permutations, with extra copies of a^b^c^d and ~(a^b^c^d), because that was found to produced good permutations faster.

There are standard ways of measuring the nonlinearity of balanced boolean functions. Unfortunately these criteria were not considered while selecting balanced boolean functions for Maraca.

16-bit permutations

16-bit permutations can be constructed by building balanced functions one bit at a time, replacing a bit or extending the function by a bit when possible. However, it is much faster to build an 8-bit permutation this way, then apply it to both the even bits and the odd bits. Unrestricted 16-bit permutations can then be explored by replacing one boolean function at a time. (Are most 16-bit permutations built of 5 NAND deep boolean balanced functions reachable this way? Are all?)

16-bit permutations built of 5 NAND deep balanced boolean functions can be judged by whether they achieve avalanche after being applied 3 times, then by what their minimum effect is out of all 1-bit input deltas and all output bits. Minimum effects as high as 0.42 were seen. There appears to be a ceiling at 0.25 that is hard to pass, and another at 0.375.

Some of the best 16-bit permutations were seen early on, when the permutation was nearly two copies of a single 8-bit permutation. This was probably because the 8-bit space is the square root the size of the 16-bit space, so exceptionally good 8-bit permutations were easier to come by, and two copies of a very good 8-bit permutation was itself very good. Having two exact copies of the same 8-bit permutation also has the advantage in software that there are 128 identical permutations to compute instead of 64. SSE instructions can be performed on 128 bits at a time.

So, Maraca restricted its 16-bit permutations to two copies of an 8-bit permutation, mainly so SSE could be used to double its speed in software. After the two copies of the 8-bit permutation are applied, the 16 bits are reordered.

8-bit permutations

256 deltas can be used against 8-bit permutations. The all-zero delta always maps to itself. Maraca required its 8-bit permutation to have no other input delta that maps to any output delta with probability 1. Maraca also required that every input bit be used in calculating at least 3 output bits.

Given an 8-bit permutation to apply to the even and odd bits of a 16 bit chunk, and a reordering of the 16 bits, the minimum effect after 3 applications of the 16-bit permutation can be measured. Several thousand such 16-bit permutations were generated with minimum avalanche of .25, where the 8-bit permutation had no delta other than 0 had probability 1, and where every input bit to the 8-bit permutation was used by at least 3

output bits. C code was generated for these 8-bit permutations, and they were timed. The fastest ones had the hillclimbing for their 16-bit shuffle redone more thoroughly, and the best of those were tested as a component in the rest of Maraca. The final 16-bit permutation chosen has minimum effect of .39 for all one-bit deltas after three applications.

The 8-bit permutation used by Maraca has three deltas that map to another delta with probability 3/4: 0x0a->0x08, 0x08->0x0e, and 0x2a->0x3f. Due to the bit scrambling between permutations, 0x0a->0x08->0x0e cannot be chained. In the 8-bit permutation, the average output delta has probability 36/256 given its input delta. That means given two arbitrary internal states, given their delta before one permutation, their delta after one permutation has average probability 2^{-362}. Some 5-gate-deep permutations had an average output delta probability of 18/256, but they took twice as long in software. Average output delta probability seems highly correlated with speed in software. Maximum nonzero output delta probability wasn't so correlated, but few candidate 8-bit permutations had no nonzero output delta of 3/4 probability or above, and none had no nonzero output delta of probability 9/16 or above.

The symmetry-breaking constants

If all 64 16-bit chunks used the same 16-bit permutation, and all 64 chunks in the internal state were initialized with the same 16-bit value, then any number of 1024-bit permutations would map to states where all 64 16-bit chunks held the same 16-bit value. To avoid this, six symmetry-breaking constants are XORed to break this symmetry quickly. Every 16-bit chunks is given its own pattern for flipping the bottom 6 bits immediately after the 8-bit permutations have been applied, causing every 16-bit chunk to map values differently. It is not clear that the chunks need to be this assymmetric; it may have been sufficient for just one chunk to always produce different results from the others.

The six 64-bit constants used to break symmetry were generated by the program below. It was a coincidence (due to optimizing for speed in software) that 3 bits of the 8-bit permutation are full XORs, so could be complemented in hardware at no cost. The only requirements on the symmetry breaking constants were that the ith bits of the six constants form a distinct value for all i in 0..63, and that the permutation be fairly irregular. Rearranging these constants had little effect on any other measures, so this program was not carefully tuned. It starts by assigning the ith group of 6 bits to the value i XOR 42, then applying a nonlinear permutation an excessive number of times. 42 was used to avoid mapping the 0th bits to the value 0. It is not clear if that is needed.

```
// C code to generate the constants used for symmetry breaking

typedef  unsigned long long  u8;

int main()
{
  int i;
  u8  z[6];
  for (i=0; i<6; ++i) {
    z[i] = 0;
  }
  for (i=0; i<64; ++i) {
    int x = i^42;
    int j;
    for (j=0; j<100; ++j) {
      x = (x+(x<<3))&63;
      x = (x^(x>>2))&63;
    }
    for (j=0; j<6; ++j) {
      if (x & (1<<j)) {
        z[j] ^= (((u8)1)<<i);
```

```
      }
    }
  }
  for (i=0; i<6; ++i) {
    printf("  y[%d] ^= 0x%.16llxLL;\n", i, z[i]);
  }
}
```

Exchanging bits between chunks

The 64 16-bit chunks must interact, otherwise the 1024-bit permutation would be a set of 64 isolated 16-bit permutations. The fastest way to do this interaction in hardware is by fixed wires that shuffle the bits among the chunks, rather than XORing bits or doing any other logic to them. The fastest way to shuffle the bits among the chunks in software is by doing 64-bit barrelshifts on the 16 values representing bits 0..15 for the 64 chunks.

Maraca barrelshifts all of the 16 values representing bits 0..15 by a different amount, exchanging bits between chunks. The amounts were required to be all different. None was allowed to be zero. If the values are i+8j, then all i in 0..7 and j in 0..7 were required to be represented exactly twice. Duplicates were avoided because this would allow a 2-bit delta from the 8-bit permutation to be fed into a single 8-bit permutation in the next 1024-bit permutation. Zero was avoided because it might allow a fixed point.

Starting from a random set of shift constants of the form i+8j where i in 0..7 and j in 0..7 were all represented twice, hillclimbing was used to increase the sum of the minimum effect of any input bit on any output bit after seven 1024-bit permutations forward and the minimum effect of six 1024-bit permutations in reverse. Solutions containing 0 or containing duplicates were discarded. For deltas with one bit set, the constants chosen give Maraca a minimum effect of .37 after seven permutations, and a minimum effect of .38 after six permutations in reverse.

If the internal state starts out nearly all zero or nearly all one, one-bit inputs require 8 1024-bit permutations (minimum effect .12), or 7 1024-bit permutations in reverse (minimum effect .41), to have nonzero effect on every output bit. This may be relevant for the first use of the first block, but there is no known tractable way to cause Maraca to have an internal state of nearly all zeros or nearly all ones after that.

Analysis of the block schedule

Maraca uses data blocks that are 1024 bits, the same size as the internal state. The ith block (starting at 0) is XORed to the internal state before permutation 3i, 3(i+21-6*(i%4))+1, 3(i+41-6*((i+2)%4))+1, and 3(i+46)+1. The block is rotated by 0, 1*128, 3*128, 6*128 bits for the first, second, third, fourth use respectively. The hash starts with the first use of the first block and ends 30 permutations after the last use of the last block. Blocks before the 0th block or after the last block can be left out, or equivalently, assumed to be all zero. This guarantees all deltas pass through at least 30 permutations.

Merkle-Damgard hashes use a block, then do some number of consecutive permutations, then use the next block. Some number of permutations achieves avalanche with a minimum effect of at least 10%. An unconfirmed source said DES used 3x avalanche. Measurements say Whirlpool does 5x avalanche. Maraca was arbitrarily required to take twice forward+backward avalanche, or roughly 4x avalanche. With nearly-zero states, forward avalanche takes 8 permutations and backward takes 7, so Maraca was required to apply at least 2(7+8)=30 permutations to every delta. The proper minimum requirement is probably between 20 and 50 permutations.

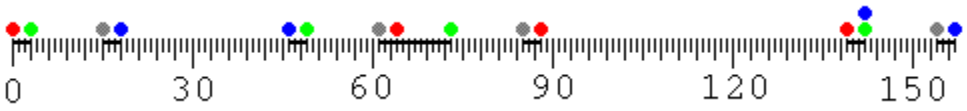
Originally, Maraca was going to be a Merkle-Damgard hash so those 30 permutations were to be consecutive.

Maraca still requires deltas to pass through 30 permutations, but it doesn't require those permutations to be consecutive. Even if permutations that a delta passes through are not required to be consecutive, they *might* be consecutive. So the nonconsecutive limit should be at least the consecutive limit: 30 permutations.

Although there are 139 permutations between the first and final use of every block, which easily exceeds the 30 permutations required, a delta could be introduced by the first use and cancelled by the second, then introduced by the third and cancelled by the fourth. That spans at least 43 permutations. And what about the distance between the second and third permutation? And what about deltas spanning two or more blocks?

This can be described as a game. Draw a line representing the course of the hash. Mark it at unit intervals with a permutation between every mark. A second preimage attack will hash two messages containing a delta, trying to get both to hash to the same thing. Every mark XORs zero, one, or more blocks containing deltas. Draw line segments connecting marks that XOR deltas. A line segment may start or stop or continue through a mark using one delta. A mark using two or more deltas can also be treated as a zero-length segment (this represents the deltas being the same so the XOR cancels them out). No matter what set of blocks is chosen, the line segments must always span at least 30 permutations. If a schedule of block uses passes this game, then any delta is required to pass through 30 permutations. Those permutations might not be consecutive.

Example: Deltas in blocks 0,1,5,6 require 30 permutations in Maraca



The first use of every block is at least one permutation away from any other use of any block, so at least one permutation is required per delta-containing block. The first and last use are at least three permutations away from any other use of any block. So a delta in 26 or more blocks will always have line segments spanning at least 30 permutations.

A program was written to brute-force play that game (http://burtleburtle.net/bob/crypto/maraca/spacing.c), requiring 30 permutations. If the spacing between uses of a block is i, i+3(X-6*(i%4))+1, i+3(Y-6*((i+2)%4))+1, i+3*Z+1 permutations, then X=21, Y=41, Z=46 is the first solution that passes for 9 delta-containing blocks. It passes for at least 15 delta-containing blocks in messages up to length 40, up to 10 with length 60, up to 8 with length 80, and up to 7 with length 100. That's as far as the search has been run. The probability of failing with n+1 delta-containing blocks is empirically less than the probability of failing with n, so it is likely this does pass all the way up to 30 delta-containing blocks in messages of any length.

Adjusting the number of permutations required by deltas requires rerunning that brute-force program for finding block scheduling patterns. These pattern spaces were not fully explored. The table below gives a number of patterns that have been found and their guarantees. The patterns say to use the ith block before the listed permutations, with the jth use (j in 1..u) left rotated by 128*j(j-1)/2 bits. These patterns were tested for at least up to 10 out of 40 blocks, and up to 7 out of 80 blocks. **d** is the guaranteed number of permutations a delta must pass through, **p** is the number of permutations between new blocks, **u** is the number of uses of each data block, and **b** is the number of accumulator buffers needed. All of these require d additional permutations after the last use of the last block to strengthen the output.

d	p	u	b	pattern
1	1	1	1	i
1	2	2	1	2i, 2i+1

3	2	2	2	2i, 2(i+1)+1
4	2	2	3	2i, 2(i+2)+1
5	2	2	7	2i, 2(i+6-4(i%2))+1
6	2	2	12	2i, 2(i+11-4(i%2))+1
6	2	3	6	2i, 2(i+4-4(i%2))+1, 2(i+6)+1
7	2	2	20	2i, 2(i+19-4(i%4))+1
7	2	3	8	2i, 2(i+5-4(i%2))+1, 2(i+7)+1
8	2	2	27	2i, 2(i+26-4(i%4))+1
8	2	3	9	2i, 2(i+6-4(i%2))+1, 2(i+8)+1
9	2	3	10	2i, 2(i+5-4(i%2))+1, 2(i+9)+1
10	2	3	11	2i, 2(i+6-4(i%2))+1, 2(i+10)+1
11	2	3	20	2i, 2(i+2+4(i%4))+1, 2(i+19)+1
12	2	3	21	2i, 2(i+4+4(i%4))+1, 2(i+20)+1
13	2	3	28	2i, 2(i+14-4(i%4))+1, 2(i+27-4(i%2))+1
14	2	3	33	2i, 2(i+14-4(i%4))+1, 2(i+32-4((i+2)%4))+1
15	2	4	21	2i, 2(i+3)+1, 2(i+13-8(i%2))+1, 2(i+20)+1
15	2	4	26	2i, 2(i+22-6(i%4))+1, 2(i+23-6((i+2)%4))+1, 2(i+25)+1
16	2	3	40	2i, 2(i+14-4(i%4))+1, 2(i+39-8((i+1)%2))+1
16	2	4	24	2i, 2(i+14-8(i%2))+1, 2(i+4)+1, 2(i+23)+1
17	2	4	29	2i, 2(i+14-4(3i%4))+1, 2(i+22-8(i%2))+1, 2(i+28)+1
18	2	4	32	2i, 2(i+10-8(i%2))+1, 2(i+27-4(3i%4))+1, 2(i+31)+1
19	2	4	34	2i, 2(i+11-8(i%2))+1, 2(i+27-4(i%4))+1, 2(i+33)+1
20	2	4	37	2i, 2(i+17-4(3i%4))+1, 2(i+32-4(i%4))+1, 2(i+36)+1
20	2	4	39	2i, 2(i+21-6(3i%4))+1, 2(i+34-4((i+1)%4))+1, 2(i+38)+1
21	2	4	42	2i, 2(i+11-8(i%2))+1, 2(i+35-4(3i%4))+1, 2(i+41)+1
21	2	4	43	2i, 2(i+19-12(i%2))+1, 2(i+38-4((3i+1)%4))+1, 2(i+42)+1
22	2	4	46	2i, 2(i+11-8(i%2))+1, 2(i+29-4(3i%4))+1, 2(i+45)+1
23	2	4	48	2i, 2(i+13-8(i%2))+1, 2(i+39-4(i%4))+1, 2(i+47)+1
23	2	4	49	2i, 2(i+19-12(i%2))+1, 2(i+45-4((3i+2)%4))+1, 2(i+48)+1
24	2	4	53	2i, 2(i+19-12(i%2))+1, 2(i+45-4((i+1)%4))+1, 2(i+52)+1
24	2	4	53	2i, 2(i+16-4(3i%4))+1, 2(i+41-8(i%2))+1, 2(i+52)+1
25	2	4	61	2i, 2(i+22-6(i%4))+1, 2(i+48-8(i%2))+1, 2(i+61)+1
26	2	4	62	2i, 2(i+22-6(i%4))+1, 2(i+49-8(i%2))+1, 2(i+62)+1
27	2	4	63	2i, 2(i+20-16(i%2))+1, 2(i+51-6(3i%4))+1, 2(i+63)+1
30	3	4	47	3i, 3(i+21-6(i%4))+1, 3(i+41-6((i+2)%4))+1, 3(i+46)+1

If a larger d is needed, multiplying the positions of all uses by q will give a guarantee of d*q. For example using the ith data block before permutation 4i, 4(i+17-4(3i%4))+2, 4(i+32-4(i%4))+2, 4(i+36)+2 would guarantee that every delta must pass through at least 2*20=40 permutations.

The rotate of the jth use by 128*j(j-1)/2 bits causes set bits in a delta to stay in the same 16-bit chunk, and

even in the same 8-bit permutation, but it changes which of the 8 bits in the permutation is maps to. The 8 bits of the 8-bit permutation are all calculated by asymmetrically different balanced binary functions. So the fact that a delta is good for one use doesn't suggest the delta will be good for the other three uses, which will be rotated by different amounts. The formula f(j) = j(j-1)/2 mod 8 for j in 1..8 has the nice property that all f(j)-f(j+1) are different, and all f(j) are different as well.

Maraca is expected to have outputs that are not only different for similar inputs, the outputs are expected to not resemble one another. If n permutations are required to guarantee a delta does not map to another delta with probability over 2^{512}, any block uses within n permutations of the end of the hash do not affect the result enough. Omitting even the last use of the last block would reduce Maraca's guarantee from 30 permutations to 24 permutations. To guarantee that all block uses have sufficient effect on the output, a guarantee of n requires that n additional permutations be applied to the internal state after the last use of the last block.

The output strengthening is deterministic, and could be rolled back to the last use of the last block. Any block uses in the 7 preceding permutations are likely to not have fully achieved avalanche, even if they were not carefully chosen. An attacker could hash two messages, roll back their strengthening to the intermediate state after the last use of the last block, and see this. It is not clear that this attack is useful, because it is a resemblance of two internal states, not two results. This attack could have been prevented by XORing the key to the final result, or XORing the first block to the final result. This was not done because it would introduce more problems of its own.

The block schedule interleaves the uses of blocks. This help speed, by allowing a new block to be combined every 3 permutations, rather than every 30 permutations or every 139 permutations. It also blocks some attacks. For example differential analysis, related-key boomerang distinguishers, and length extension attacks assume there is an internal state after all uses of blocks 0..i but before all uses of blocks i+1 on. That is not the case in Maraca.

Analysis of the key and length usage

Maraca uses the original message, key, message length, and key length to form a modified message of roughly key || message || key || lengths plus some padding. Maraca divides the modified message into blocks and hashes them all the same way. The section "Producing the modified message from the key and message" gives the exact specification.

The length and key are hashed the same way as the rest of the data, except that the key is hashed twice, once at the beginning and once near the end of the message. No additional attacks are known that are enabled by the key appearing twice. The key appears at the beginning and near the end of the modified message. If the key is not known, the hash cannot be rolled forward from the beginning. If the key is not known, the 30 permutations of strengthening can be rolled back, but the last block (unless the length field is in its own last block) cannot be rolled back. If the length field is in its own last block, the last use of that block can be rolled back, but the next-to-last block contains the key, so that cannot be rolled back. Even when the last block contains just the length and its last use can be rolled back, there are three other uses of that block that cannot be reached without knowing the key.

Only the length mod 1024 is added to the modified message. This allows messages of unlimited length to be hashed. Given a message, could another message that is q*1024 bits longer be produced that maps to the same modified message? No. The final block of a message contains the 2-byte length field, which contains a 1. The following nonexistant blocks are essentially all zero. If a message is q blocks longer, one of those following nonexistant all-zero blocks has been replaced by the new block containing the length, which contains a 1, so is different. If the original message had L blocks, the internal states of these two messages will differ from their 3L+3 to 3L+166th permutation (and on through the 3L+169th hash for the longer

message). This is quite a bit more computational difference than the 30 permutations required to securely diffuse deltas.

If two messages differ in length by less than 1024 bit, that length difference is recorded in the 2-byte length field and hashed the same way as any other data. If the message differs only in the number of bits in the final byte, and the diffing bits are all zero, the only difference will be in the 2-byte length field. The length field comes after the final use of the key, and may be in its own block. In that case it may be possible to undo the internal state to before the last use of the length field. The previous 3 uses cannot be undone without knowing the key. Maraca, minus the final 30 permutations and the last use of the last block, only guarantees that deltas pass through 24 permutations, for example a delta in blocks (2,3,7) need only pass through 24 permutations.

The length and both instances of the key, like all data, are added in a non-bijection, over the course of 139 permutations, interleaved with uses of other data blocks. Maraca does not have a mode where the hash of one message is used as the initial vector of the next: Maraca's initial vector is always all-zero. If the hashes of multiple blocks are concatenated then hashed to form a tree-structured hash, each hash ended with 30 permutations of result strengthening, so there is no linear use of the shared key that might cancel out due to uses of these multiple hash values.

The key and length are not used while hashing blocks in the middle of the message. They add negligible overhead to the hashing of large messages. Attacks on the middle of the hash do not have to worry about the computation being dependent on the key or the length. Computations in the middle are only affected by the internal state and 46 data blocks before and the 46 data blocks after that point.

Cryptanalysis

Known attacks

Differential cryptanalysis introduces a delta in one block. The mixing ending at the next use of the next block should give greater than expected probability of some other delta, which that next block will cancel out. As-is, this attack does not work on Maraca because there are three other uses of each of these blocks to account for. These three other uses are rotated by different multiples of 128 bits, producing deltas that may not be as useful as the original.

Rotations of multiples of 128 bits would not be a problem if all the deltas are symmetric under 128-bit rotations. Any delta that is symmetric under 128-bit rotation is either 0 or 255 for each 8-bit permutation. There are 4 16-bit deltas that each chunk could have matching that description (0xffff, 0xaaaa, 0x5555, 0x0000), but the only one that can map to anything in that set is 0x0000, which of course maps to itself. Symmetric deltas that return to symmetry after multiple permutations have not been ruled out.

Characteristics

Characteristics are a type of delta useful for differential cryptanalysis. These are deltas that map to themselves with greater than expected probability after some number of rounds.

A program (http://burtleburtle.net/bob/crypto/findingc.html) used multiple trials and linear algebra to try to find characteristics for the 1024-bit permutation. 2^{16} seeds were tried, and any result was tried 2^{16} times. The same was done for 1..6 applications of the permutation. No self-referential differentials were found this way.

If a delta was all 0 or all 1 for all 64 bits in each value, it might take advantage of self-preserving deltas for the 16-bit chunks. The symmetry-breaking XORs do not alter the input delta to output delta after one permutation, so any delta that is self-preserving for one 16-bit chunk will be self-preserving for all of them.

There are 239 such self-preserving deltas. The strongest (other than 0x0000) is 0xc864, which has probability 10/256 per 16-bit chunk, so probability (10/256)^^64 = 2^{-299} after one 1024-bit permutation. This won't be distinguishable from noise after four permutations.

Other uses of a block introducing a characteristic would be rotated in three other ways for the three other uses, introducing deltas that are probably not characteristics.

A trial cryptanalysis of Maraca

Maraca guarantees 30 permutations per delta instead of 30 consecutive permutations per delta. One way to take advantage of that is to find a delta involving many blocks and many short sequences of permutations. The goal would be to introduce a delta at the start of each sequence and cancel it out a few permutations later. A "gap" is a sequence of permutations during which a delta is not cancelled out.

Passing 30 permutations with 20 blocks containing deltas may be easier than passing 30 permutations with 2 blocks containing deltas. The problem is less overdetermined with 20 blocks. How much easier is not clear. The strongest delta after 1 permutation has probability about 3/4, so if 60 blocks could be exactly chosen to bracket 30 gaps of 1 permutations each, Maraca could be broken in $(3/4)^{30}$ trials, which is about 13 bits of security total. This can't be done because the first use of the first block and last use of the last block need to be adjacent to gaps of at least three consecutive permutations. More important, every block is used four times. The first use of every block is on its own, and the other three uses are XORed to two other blocks. For a set of delta-containing blocks to cancel out after every gap, the deltas must satisfy a system of equations, with more equations than blocks.

A promising pattern for cryptanalysis is a delta using blocks 1,2,6,7,25,29,30. This 7-block pattern requires passing through 32 permutations, with no sequence longer than 3 permutations before some block is XORed. The block usages and gaps are at permutations 3..6, 18..21, 31..34, 46..49, 73..75, 87..90, 118..121, 127..130..133, 142..142, 145..145, 157..157..160, 214..214, and 226..229. This has three relations where two blocks rotated by some amount must cancel out (before permutations 142, 145, and 214), leaving 4 blocks worth of deltas to choose. There is also one required inequality between two rotated blocks, before permutation 157. The 127..130..133 sequence allows the block usage before permutation 130 to modify the delta.

Let B2 be the delta in block 2, B2+k mean the delta in block 2 rotated left by k*128 bits, and fff(B2) mean an output delta after three permutations have been applied to a pair of initial states with delta B2. Algebra shows that breaking Maraca with this pattern requires B2+6=fff(B2+3), and also B2+3=fff(B2-3), and several other constraints. Solving these systems of equations would be aided by tables of deltas D where D=fff(D+k) for k=0..7. Other tables for n consecutive permutations would be useful too, for example the gap 73..75 will probably need the D=ff(D+k) table.

All settings of the bits in one 8-bit permutation were tested in each of the 128 8-bit permutations, forward three 1024-bit permutations, forward two and back one, and forward one and back two, and back three, all for 2^{12} trials. These are promising because their low weight encourages high probability deltas. This covers all one-bit deltas, and also all deltas consisting of one 8-bit group set to all ones.

All 65535 nonzero setting of the bits in two 8-bit permutations were tested for all choices of 2 out of the 128 8-bit permutations. 2^3 trials were done for each.

Deltas of all 2^{16}-1 combinations of 16 64-bit values set to (u8)0's or ~(u8)0's were tested, except all zeros, for 2^{16} trials each. If the delta in a 16-bit chunk maps to itself, it will map to itself in all chunks, and the

barrelshifts of the 64-bit values would be a no-op.

All choices of one through six of the 128 8-bit permutations were chosen and set to 255 (all ones), for just three permutations forward. These are all symmetric under rotation by multiples of 128 bits. One trial for 6 groups, 2^6 trials for 5 groups, 2^{10} trials for 4 groups, and 2^{16} trials for three groups and two groups were tried.

All two-bit deltas were tested forward three permutations, back one permutation and forward two, and back two and forward one. All were tested for 2^{12} trials.

The 8-bit permutation only maps a few input deltas to output deltas with probability 1/2 or greater: 2,8,10,20,28,32,34,42,54,62. All choices of three 8-bit permutations were tested with all combinations of these deltas for 2^3 trials each.

No deltas were found that mapped to themselved rotated by a multiple of 128 bits after three permutations. Only a few days of singlethreaded compute time was spent in the search, but this would have caught the most obvious low hanging fruit. High probability deltas satisfying this requirement may exist. There are too many possible high probability deltas to enumerate. "High probability" could be as low as 2^{-51} and still allow a 10-gap pattern with 3 permutations each to do better than brute force for a 512-bit hash. It is possible, though unlikely, that no such high probability deltas exist. Even if some do exist, it is not enough to satisfy just one such requirement, they must satisfy several such requirements simultaneously.

Uses of Maraca

Maraca is a keyed hash. The key and result can be up to 1024 bits, but it is only intended to provide 512 bits of security.

Using Maraca with a truncated result

If the 1024 bit result is truncated to n bits, n up to 512, it is expected to provide the minimum of the security of 1024-bit Maraca and the security that can be provided by any hash of that many bits.

No Maraca-specific attacks are known to be enabled by shorter results, for example partially matching results that become fully matching on truncation, because the 30 strengthening permutations at the end are expected to fully diffuse all bits from all uses of all blocks to all output bits equally.

Shorter keys do cause the key to not occupy its own block. For very short keys and messages, the whole modified message may fit in one 1024-bit block. Even so, an unknown key prevents the last use of the last (non-length) block from being rolled back, and prevents the calculation from being redone forward at all (since the first thing done is XORing first use of the first block). Since the first 128 bits of each block map to the low bits of all 128 8-bit permutations, all keys over 128 bits change the results of all 128 nonlinear 8-bit permutations at the beginning. All 128 8-bit permutations are affected by the final use of key at the end too.

The result truncated to 224 bits is expected to require 2^{224} blocks to be hashed for a first preimage attack and 2^{112} blocks to be hashed for a second preimage attack. The result truncated to 256 bits is expected to require 2^{256} blocks to be hashed for a first preimage attack and 2^{128} blocks to be hashed for a second preimage attack. The result truncated to 384 bits is expected to require 2^{384} blocks to be hashed for a first preimage attack and 2^{192} blocks to be hashed for a second preimage attack. The result truncated to 512 bits is expected to require 2^{512} blocks to be hashed for a first preimage attack and 2^{256} blocks to be hashed for a second preimage attack.

The long message attacks described in "randomized hashing" give no advantage over brute force when results are truncated to 512 bits or less.

Using Maraca as a keyed hash

Maraca(K,M) is a keyed hash, taking a key K and a message M. The key is required to be a multiple of 128 bits in length. If no key is needed, provide an empty key with a length of zero bits. It produces a 1024 bit output. If only n bits of output are needed (n less than 1024), truncate the output to n bits. Any predetermined n bits may serve as the output with no loss of security, since the final 30 permutations make sure all information from the final block uses is well spread out.

Given a key and an n-bit output, it is expected that on average 2^n blocks must be hashed to find a message which will produce that that output. Given a (key,message) pair, it is expected that on average $2^{n/2}$ blocks must be hashed to find another (key,message) pair that will hash to the same thing as the first pair.

Using Maraca as a PRF

A PRF(s,x) is pseudorandom function with a seed s and an input x.

Maraca can be used as a PRF(s,x) by using s as the key and x as the message: Maraca(s,x). This requires s to be a multiple of 128 bits. Maraca produces a 1024 bit output. If a larger output is required, an 8-byte little endian counter could be appended to x, and the outputs of Maraca with consecutive counter values (starting with 0) could be appended until the desired length is reached. If fewer bits are required, the result can be truncated.

This is a use of Maraca as a keyed hash, so it has no less security than Maraca when used as a keyed hash. Maraca as a keyed hash is designed to provide up to 512 bits of security, despite producing up to 1024 bits of output. Maraca is expected to require at least 2^{512} blocks to be hashed to find an (s,x) that produces a given 1024 bit output, and at least 2^{256} blocks to be hashed to find (s1,x1), (s2,x2) that produce the same 1024 bit output.

Maraca can also be used with HMAC(s,x) as a PRF(s,x).

Using Maraca with HMAC

An HMAC(k,m) hashes a message m using a key k and a cryptographic hash function. It assumes message and key lengths are measured in bytes. Maraca has 128 byte blocks and produces a 128 byte output. Paraphrasing the Wikipedia description of HMAC,

```
HMAC(key, message)
{
    opad = 0x5c repeated 128 times;
    ipad = 0x36 repeated 128 times;
    empty = a byte array of length 0;

    if (length(key) > 128) {
        key = Maraca(empty, key);
    }

    for (i=0; i<length(key); ++i) {
        ipad[i] = ipad[i] XOR key[i]
        opad[i] = opad[i] XOR key[i]
    }
```

```
    partial = Maraca(empty, ipad || message);
    return Maraca(empty, opad || Maraca(empty, partial));
}
```

Maraca with a 1024-bit result is a bijection of its initial state, and a bijection of the state immediately after the final use of every block. However Maraca is not a bijection for individual blocks, it resembles a random mapping instead. If a random mapping is repeatedly applied to its own result, initially every application loses about half a bit of entropy. Since HMAC applied Maraca to the message twice instead of once, it is expected to produce about 1023 bits of entropy per input block instead of about 1023.5, which helps second preimage attacks slightly. This has no effect on security if the 1024 bit result is truncated by at least two bit. Maraca is only intended to provide 512 bits of security anyhow.

No other attacks are known that are easier when using Maraca as an HMAC than when using Maraca directly. Maraca is not a Merkle-Damgard hash, so some attacks that might normally be tried on an HMAC cannot be used in their original form against HMAC on Maraca. For example there is no state that is after all uses of the first i blocks but before the first use of block i+1, so length extension does not work.

Using Maraca with randomized hashing

Randomized hashing is a method of modifying a message M with a random seed r. Let r_len be 16 bits holding the binary representation of the length of r, with least significant bit last. Let r_extended by r appended to itself enough times to be longer than M, then truncated to the length of M. Then the modified message M' = r || r_extended XOR M || r_len. Given M', r and M can be deduced. Maraca(empty, M') is the randomized hash of M with seed r.

Given an n bit hash value V (n no more than 512), Maraca guarantees n bits of security against finding an M2' such that Maraca(empty, M2') is equal to V. If M1,r1 are known such that M1' is the modified message produced by randomized hashing for (M1,r1) and Maraca(empty, M1')=V, does that make it easier to find (M2,r2) which randomized hashing modify to M2' where Maraca(empty, M2') is V but M2' is different from M1'?

Kelsey and Schneier showed a long message attack on an n-bit state that requires hashing $2^{n/2}$ blocks, which can be modified to attack Maraca, as follows. An attacker could choose a message M1 consisting of 2^{512} identical blocks. This will produce 2^{512} 1024-bit internal states. There will probably be a collision among them. A modified message, M2, could be constructed by appending the tail after the second instance of the repeated state to the head up to and including the first instance. Although the second, third, fourth uses of blocks from the head will be used while hashing the tail, all blocks in both the head and the tail have the same value (except the very last block, which contains the lengths). So the extra uses of head blocks will not change the result when hashing the tail. M2 will have the same hash value as M1. This attack requires 2^{512} work. It can be avoided by expecting only 512 bits of security out of Maraca for randomized hashing (SHA-3 expects no more than 512 bits of security). It could also be avoided by limiting the maximum message size (SHA-3 limits it to 2^{64}-1 bits). Merkle-Damgold hashes that report their entire internal state are subject to the same attack, but the attack on them does not require all blocks in M1 to have the same value. This attack is why Maraca guarantees 512 bits of security, not 1024.

Suppose Maraca's 1024-bit output is truncated to n bits, and (M1, r1) are modified to M1' and Maraca(empty, M1') = V. Randomized hashing of 2^n randomly chosen (M2,r2) pairs is likely to find another pair mapping to V. If n is 512 or less, no known attack is more efficient than this.

Advantages and Limitations

Maraca's advantages are

- It is fast in hardware. Hashing an additional 128 bytes requires logic 21 gates deep.
- It is fast in software for large inputs (5.3 cycles/byte).
- It can hash inputs of unbounded size.
- It can produce hashes up to 1024 bits in length, though it is only intended to provide 512 bits of security.
- If random access to a 7K sliding window of the input is available, it requires under 512 bytes of working memory.
- Blocks are used multiple times.
- There is no internal state after all uses of blocks 0..i but before all uses of blocks i+1 onward.

Maraca's limitations are

- It is slow for small inputs (about 28200 cycles for the empty string). It is likely in practice that some other algorithm would be used for small inputs.
- If each input byte is seen only once, it requires 6688 bytes of memory.
- Its block scheduling method, and its goal of requiring deltas to pass through at least 30 permutations, is new.
- Its block scheduling method has not been proven to satisfy its goal of requiring deltas to pass through at least 30 permutations.
- It is hard to analyze because its state is large and its permutation is weak.

Summary

Maraca is a secure keyed hash function submitted to the NIST SHA-3 hash competition. It is designed to be as secure as any hash can be that reports 512 bits or less. Internally it always produces a 1024 bit result which is truncated to the desired length. It runs in 5.3 cycles/byte in software, and up to 42 bytes/cycle in hardware. Maraca takes about 28200 cycles in software to hash the empty string. It requires 6688 bytes of memory. Avalanche takes 7 permutations, but a new block is accepted every 3 permutations. Every block is used four times, interleaved with uses of other blocks. The block usage pattern guarantees that any self-cancelling delta passes through at least 30 permutations. Its many constants were chosen empirically, mostly by hillclimbing.

References

- "Finding Characteristics of Block Ciphers", Bob Jenkins 1996, http://burtleburtle.net/bob/crypto/findingc.html
- "Active Boolean Function Nonlinearity Measurement in JavaScript", Terry Ritter, 1998, http://www.ciphersbyritter.com/JAVASCRP/NONLMEAS.HTM
- "How to Protect DES against Exhaustive Key Search (an analysis of DESX)", Joe Kilian and Phillip Rogaway, 2001, Journal of Cryptology, vol 14
- "Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV", J Black, P Rogaway, T Shrimpton, Advances in Cryptology, CRYPTO '02, LNCS 2442
- "The Whirlpool Hashing Function", Paulo S.L.M. Barreto and Vincent Rijmen, 2003, http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html
- "On the Security of Encryption Modes of MD4, MD5, and HAVAL", Jongsung Kim, Alex Biryukov, Bart Preneel, and Sangjin Lee. 2005, Information and Communications Security, proceedings vol:3783
- "Second Preimages on n-bit Hash Functions for Much Less than 2^n Work", J Kelsey, B Schneier,

EUROCRYPT 2005 Proceedings
- "EnRUPT: The Simpler The Better", Sean O'Niel 2008, http://www.enrupt.com/index.php/2008/07/03/irrupt64_for_sha3
- FIPS 198, "The Keyed-Hash Message Authentication Code (HMAC)", 2002
- "spacing.c", Bob Jenkins 2008, http://burtleburtle.net/bob/crypto/maraca/spacing.c
- SP 800-106, "Randomized Hashing for Digital Signatures", 2008
- "Work in progress: Maraca, a submission for SHA-3", Bob Jenkins 2008, http://burtleburtle.net/bob/crypto/maraca/nisthash.html