Semi-free start collision attack on Blender

Xu Liangyu and Li Ji
Sony China Research Laboratory
{Liangyu.Xu, Ji.Li}@sony.com.cn

Abstract. Blender is a cryptographic hash function submitted to NIST's SHA3 competition. We have found a semi-free start collision attack on Blender with trivial complexity. One pair of semi-free start collision messages with zero initial values is presented.

1. Description of Blender

The hash function Blender consists of two procedures: preparing message, and hash computing. Blender has four variants regarding to the bit length of digest (224, 256, 384, 512). The procedures differ just a little among the four variants. The attack approach presented in this paper on different variants is almost the same. So here we just give a brief description of Blender-256 with digest length of 256 bits.

Blender-256 uses eight 32-bit state variables, a0 to a7, eight 32-bit result variables, H0 to H7, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 32-bit intermediate values, T, T1 and T2, and one intermediate integer value r used to hold a rotation factor.

In the preparing message procedure of Blender-256, there are 5 steps.

Step 1: Padding. The message M to be hashed with length of ℓ bits is padded to P with p bytes, where $p = (\ell + 7) >> 3$. If the length of the message M is an exact multiple of 8 bits, no padding is added and the padded message P is identical to the original message M. Otherwise, the complement of the last bit of the message shall be appended repeatedly until the resulting length reaches the next exact multiple of 8 bits. The amount of padding added is at most seven bits.

Step 2: Filling. The fill data F is the padded message P truncated to 13 bytes if necessary, unless the message M has zero length in which case F is 13 bytes of all zeros. The amount of fill data to be appended to the padded message depends on the block size and the message length. For the detail of filling, please refer to the specification of Blender [1].

Step 3: Appending the Message Lengths. After the message has been filled to the appropriate length, the message length as held in the byte array L is appended to the message. The single byte ℓ_ℓ, the length of the length, is then appended to the result to complete the assembled message. The latter should be two 32-bit words short of an exact multiple of the block size.

Step 4: Parsing the Assembled Message. After a message has been assembled as described above, it must be parsed into a number of 32-bit words before the hash computation can begin. The first byte of the message becomes the least significant byte of the first 32-bit word and successive bytes of the message become the progressively higher order bytes within the word. Successive words are defined similarly.

Step 5: Appending the Checksums. The final step in preparing the message is to append two 32-bit checksum words. The first checksum is the complement of the sum modulo 2^{32} of all the 32-bit words in the parsed message. The second checksum is the sum modulo 2^{32} of the complement of all the 32-bit words in the parsed message.

The hash computing procedure includes 2 steps.

Step 1: Initialization

Before hash computation begins, the working variables, a0 to a7, are initialized to the following eight 32-bit words in hex:

a0 = 6a09e667

a1 = bb67ae85

a2 = 3c6ef372

a3 = a54ff53a

a4 = 510e527f

a5 = 9b05688c

a6 = 1f83d9ab

a7 = 5be0cd19

Step 2: Round function

1. Compute the preliminary intermediate values using add-with-carry:

   $[c1,T1] = (a5 \oplus W_t) + (a1 \oplus ROTL^8 (a3)) + c1$

   $[c2,T2] = (a0 \oplus ROTR^8 (W_t)) + (a4 \oplus ROTR^8 (a2)) + c2$

   where, W_t is the t^{th} 32-bit word of the result after preparing message procedure.

2. Compute the rotation factor:

   $r = 8 – (c1 + c2)$

3. Rotate the intermediate values:

   $T1 = ROTL^r (T1)$

   $T2 = ROTR^r (T2)$

4. Compute the next state:

   T = ROTR7 (a0)

   a0 = a1 ⊕ T2

   a1 = a2 ⊕ T1

   a2 = a3 ⊕ T2

   a3 = a4 ⊕ T1

   a4 = a5 ⊕ T2

   a5 = a6 ⊕ T1

   a6 = a7 ⊕ T2

   a7 = T ⊕ T1

5. Update the hash result variables:

   H0 = H0 + a0

   H1 = H1 + a1

   H2 = H2 + a2

   H3 = H3 + a3

   H4 = H4 + a4

   H5 = H5 + a5

   H6 = H6 + a6

   H7 = H7 + a7

After repeating step 2 for each word in the prepared message, the resulting 256-bit message digest of the message M is

H0 || H1 || H2 || H3 || H4 || H5 || H6 || H7

2. Observations

The round function of Blender is depicted in the following Figure 1.

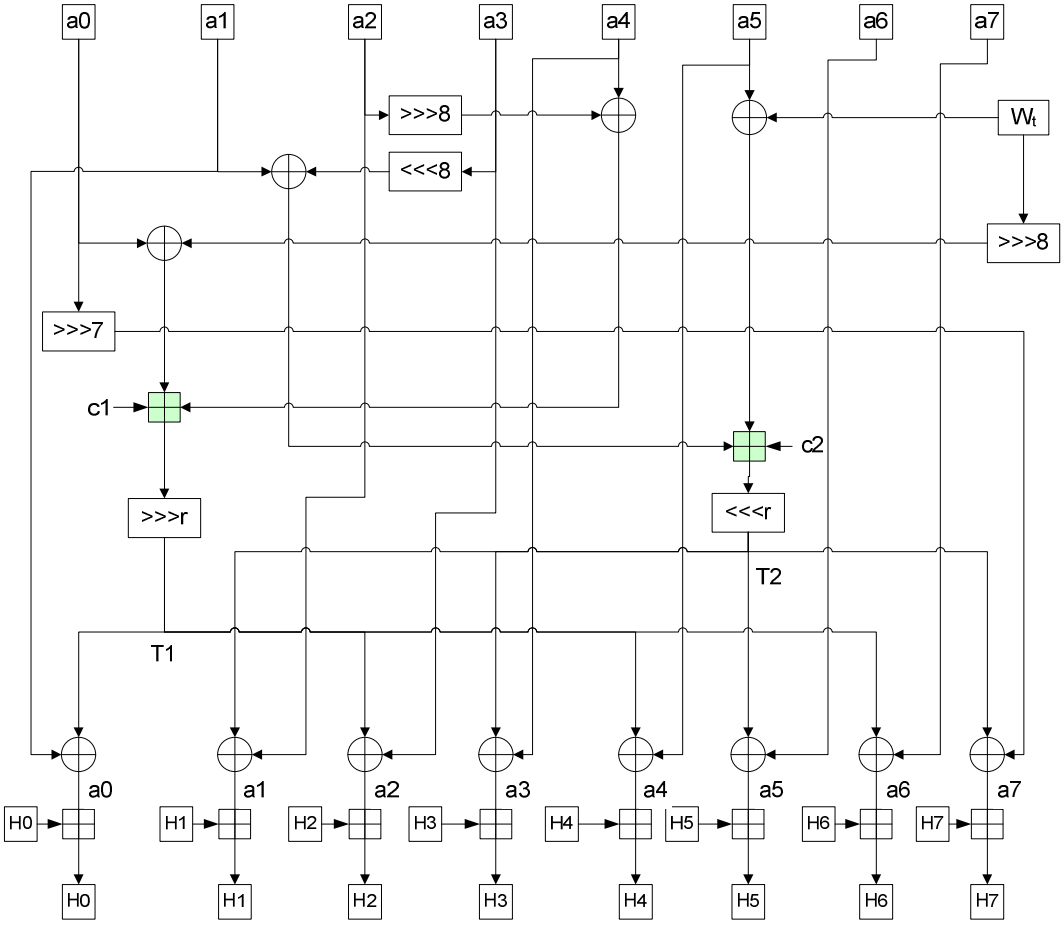

Figure 1 The round function of Blender

Observation 1: Local collision in round function

The round function plays an important role in difference diffusion. But some special differences at a0~a7 will cancel with each other under some conditions in one round. An example of the special differences (xor difference) is △= 0xffffffff at a0~a7 and at message word Wt. If the two add-with-carry and the 8 final modular additions can be seen as xor operations, the differences at output a0~a7 will disappear. We show this differential in the following Figure 2. It should be pointed that, rotations on △= 0xffffffff don’ t change the difference value.

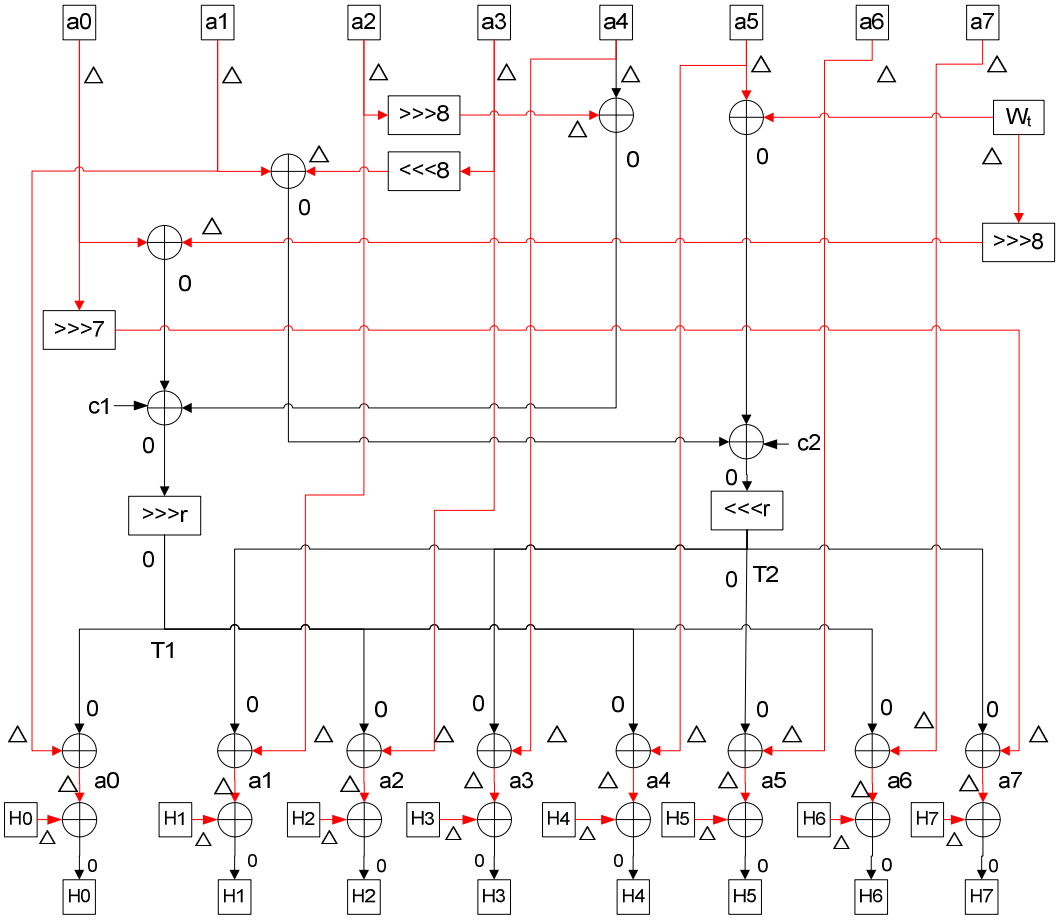

Figure 2 The local collision in round function

Observation 2: Generate the special differences at a0~a7

The initial values a0~a7 have no difference, and set the difference at W_{t-1} to △=0xffffffff. Again, if the two add-with-carry and the 8 final modular additions can be seen as xor operations, the differences at output a0~a7 are all △= 0xffffffff. This is just the input differences at a0~a7 in observation 1. The Figure 2 shows the procedure of generating the special differences at a0~a7.

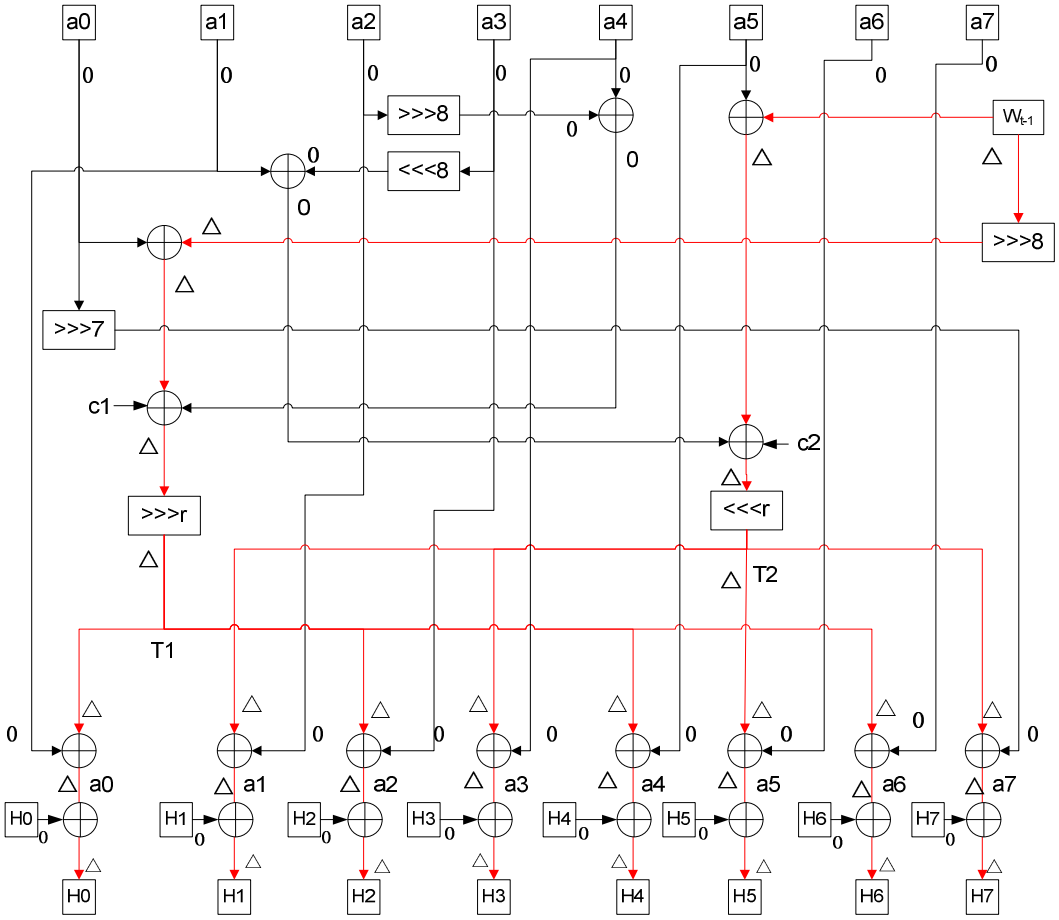

Figure 3 Generate differences △ at all output a0~a7

3. Semi-free start collision attack on Blender

From the above two observations, we can easily construct such a pair of message M1 and M2, each of which has 2 32-bit words. M1 ⊕ M2 = 0xffffffff ffffffff. For example, M1 = 0x00000000 ffffffff, M2 = 0xffffffff 00000000. If all the additions can be seen as xor operations, M1 and M2 will lead to a two-round collision. Now we pay attention to the step 2 and step 5 in the preparing message procedure. In step 2, filling data with length of 13 bytes truncated from message will be appended. We must make sure there is no difference on the filling data. So we insert 4 additional zero message words (16 bytes > 13 bytes) before M1 and M2. And checking the step 5, we make sure that the appended checksums are also the same. So the two messages (0x00000000 00000000 00000000 00000000 00000000 ffffffff and 0x00000000 00000000 00000000 00000000 ffffffff 00000000) will collide under some conditions.

Now let's discuss these conditions. We have just mentioned that the precondition of observation 1 and observation 2 is that the two add-with-carry and the 8 final modular additions can be seen as xor operations in each round. A modular addition has the some effect when there is no carry occurring at each bit of the addition. If we can make sure that one operand in the modular addition with two operands is always zero, then carry will never occur. In fact, if we set the initial values of a0~a7 in Figure 2 to zeros in round t-1, M1 and M2 mentioned above will collide with probability of 1 within two rounds. And the 4 zero words inserted before M1 and M2 will not

change the initial zero values if we set the initial values to zeros. Now we come to the semi-free start collision, i.e., we set the initial values to zeros (the real initial values for a0~a7 are presented in **Step 1. Initialization**), and then we use two messages (0x00000000 00000000 00000000 00000000 00000000 ffffffff and 0x00000000 00000000 00000000 00000000 ffffffff 00000000) to construct a collision for whole hash function Blender-256. The attack is so-called semi-free start collision.

4. Discussion on semi-free start collision and collision attack on Blender.

In fact, the semi-free start collision message pair can be longer than the pair we presented above. Because, at first, we can insert zero word after any message word without change the state variables. And the second, we can repeat the pair 0x00000000 ffffffff and 0xffffffff 00000000 after the pair respectively for any times.

We can construct collision attack from semi-free start collision. 2^{256} pre-computations can be done to search a message block which results in a zero state variables a0~a7. And appending the semi-free start collision message pair, we can get a collision message pair.

5. Conclusion

We showed that Blender is not semi-free start collision resistant. As Blender utilizes the same initial values as SHA2, the differential presented above will never generate a real collision (or collide with trivial probability). So we don't announce Blender is fully broken. We recommend the authors of Blender to use some random constants in modular additions in the round function to avoid such kind of attack.

Reference

[1] Colin Bradbury, specification of Blender.
http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/Blender.zip

Appendix

The two semi-free start collision messages for Blender-256.

The initial values:

a0=a1=a2=a3=a4=a5=a6=a7=0x00000000

Message 1:

0x00000000 00000000 00000000 00000000 00000000 ffffffff

Message 2:

0x00000000 00000000 00000000 00000000 ffffffff 00000000

Semi-free start collision hash digest:

f50b433f415f9700f50b433f415f9700f50b433f415f9700750b42fe04206f79