Collision Attack on the Waterfall Hash Function

Scott Fluhrer[1]

[1] Cisco Systems sfluhrer@cisco.com

December 16, 2008

Abstract. We give a method that appears to be able to find colliding messages for the Waterfall hash function with approximately $O(2^{70})$ work for all hash sizes. If correct, this would show that the Waterfall hash function does not meet the required collision resistance.

Keywords: hash function, Waterfall, SHA-3, collision

1 Description of Waterfall

Waterfall is a hash function designed by Bob Hattersley, and which was submitted to the SHA-3 competition. In this hash function design, it parses the input as a series of 32-bit words. Each word is input and updates three nonlinear shift registers ("streams" in the waterfall documentation) and two entropy pools. After the words (and a possible final partial word) has been input, the shift registers and the entropy pools (and the total bitcount) is used to create the final hash.
With the recommended algorithm parameters, the three streams are of 16 words, 7 words and 6 words, and both entropy pools are 32 words long, for a total of 93 words (2,976 bits) of internal state. Here is how each input word is processed; each word updates each of the three streams as follows (for s := 1, 2, 3):

$$\begin{aligned} previ &:= i_s \\ i_s &:= (i_s + 1) \bmod StreamSize_s \\ Stream_s[i_s] &:= Xbox[\ Stream_s[i_s] \oplus Stream_s[previ] \oplus Input \oplus Constant_s\ ] \end{aligned}$$

Here:

- Input is the input word
- $Stream_s$ is the actual stream array
- i_s is the per-stream active pointer
- Xbox is a nonlinear 32-bit permutation
- $Constant_s$ is a constant that is different for each pool.

Once all three streams have been updated, then the two pools are updated from the 7 word and the 6 word stream by xoring in the updated word in the stream into two separate 32 word circular buffers as follows:

$Pool_2[j] := Pool_2[j] \oplus Stream_2[i_2]$
$Pool_3[j] := Pool_3[j] \oplus Stream_3[i_3]$
$j := (j + 1) \bmod 32$

Here:

- $Pool_2$, $Pool_3$ are the two pool arrays
- $Stream_2[i_2]$, $Stream_3[i_3]$ are the words from streams 2, 3 that were just updated
- j_s is the active pointer (identical for both pools)
- Xbox is a nonlinear 32-bit permutation
- $Constant_s$ is a constant that is different for each pool.

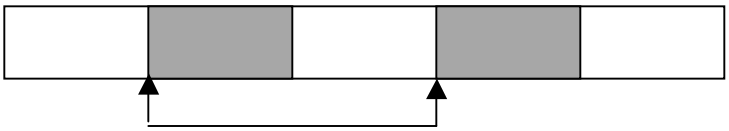
Now, the states of the three streams and the two pools (and the total bit count) are the only state used to generate the final hash once the message has been processed. The details of computing the final hash are somewhat complex, however, that is irrelevant. If we can find two messages of equal length that generate a collision in those 93 words, we will have a final hash collision. That will be our approach.

2 Overview of Collision

We will generate our collision using a differential. Our general approach would be to generate a few (one though four, depending on where in the differential we are) words at a time, and verify those words successfully propagate the differential before continuing. Because the search involved with finding each set of words ("section") is never more than an $O(2^{64})$ effort, and that we only occasionally need to backtrack over a section that we've already generated, this limits the total work to a small multiple of the work for any one section.

Our strategy can be broken up into two phases. In the first phase, we generate a difference within the streams, and then apply other differences that will eventually cancel out those differences. Figuring out how such a differential would work is straightforward; the Waterfall specification gives such an example in table 4.2.12 (although we'll be using a different one listed below); it turns out that generating two messages that produce such a differential is not impractical (more on the details of this later).
On the other hand, while such a differential cancels out all differences within the streams, it does leave differences within the pools. The solution of this is the second phase; here, we generate the exact same differential again later in the message, with the bitwise differences within streams 2 and 3 exactly the same (the actual values of the differences within stream 1 can differ from the previous phase, as that does not propagate to a pool). Because the pools are updated by a simple xor, applying the same differential to the pools cancels the original differential. Note that we cannot just reuse the same message bytes as we generated for the first occurrence of the differential (because the exact message words we choose depend on the stream state, and those will be different at the start of

phase 2). Therefore we will need to search for the differential again (and this time with additional constraints on the differentials within streams 2 and 3).
Hence, an overall schematic of the differential would be (where the shaded version are where message differences appear):



The exact contents of the common regions, and the exact spacing between the differentials (as long as the distance is a multiple of the pool size) would appear to be irrelevant for the difficulty in constructing the differential. We will depend on the exact contents of the initial and middle common regions, as they do affect the state of the streams at the start of the differentials, and the exact values of the differentials we generate do depend on that.

3 Details of the Differential

Here is the actual differential we will use:

Step	Input	Stream1	Stream2	Stream3
0	0	0000000000000000	0000000	000000
1	1	0000000000000000	0000000	000000
2	1	0000000000000001	0000001	000001
3	1	0000000000000011	0000011	000011
4	0	0000000000000110	0000110	000110
5	1	0000000000001100	0001100	001100
6	1	0000000000011001	0011001	011001
7	0	0000000000110010	0110010	110010
8	0	0000000001100100	1100100	100101
9	0	0000000011001000	1001001	001010
10	0	0000000110010000	0010010	010100
11	0	0000001100100000	0100100	101000
12	0	0000011001000000	1001000	010001
13	0	0000110010000000	0010001	100011
14	0	0001100100000000	0100011	000110
15	0	0011001000000000	1000111	001100
16	0	0110010000000000	0001110	011000
17	0	1100100000000000	0011100	110000
18	0	1001000000000001	0111000	100001
19	0	0010000000000010	1110000	000010

20	0	0100000000000100	1100001	000100
21	1	1000000000001000	1000011	001000
22	0	0000000000010000	0000110	010001
23	0	0000000000100000	0001100	100011
24	0	0000000001000000	0011000	000110
25	0	0000000010000000	0110000	001100
26	0	0000000100000000	1100000	011000
27	0	0000001000000000	1000001	110000
28	0	0000010000000000	0000010	100001
29	0	0000100000000000	0000100	000010
30	0	0001000000000000	0001000	000100
31	0	0010000000000000	0010000	001000
32	0	0100000000000000	0100000	010000
33	1	1000000000000000	1000000	100000
34		0000000000000000	0000000	000000

In the above table, 0 means that there is no differential in the corresponding word, and 1 means that there is a nonzero differential. There are some words that must have the same differential (so that they will cancel out at times); these are not listed here, but these will be listed a constraints in the differential analysis.
This differential is longer than the one listed in Waterfall paper (which is only 30 words long). We selected this differential because each piece of it can be constructed with search of no more than O(2**64) operations expected.
Also note that we start off the differential with an input of 0 (no difference), and with the corresponding trivial differential for one word in the streams. We do this because when constructing the initial part of the differential, we'll need to specify the value of this word; as it is part of what we'll need to specify, we list it as part of the differential.
Our differential causes some differences within the pools as well. Because we will be careful to repeat (and hence, undo) those differences during phase 2, we don't need to analyze those differences.

4 Constructing the Differential

Our goal for each phase is, given the state of the waterfall streams at the start of the differential, to create two messages strings (we'll call them 'A' and 'B') that have the differential characteristics listed above. In addition, during the second phase, we'll also need to contain the differentials in streams 2 and 3 to what the differentials were in the first phase.
To find such message strings A and B, we break up the search into 'stages'. For each stage, we take a number of steps (the actual number of steps will vary, depending on the local complexity of the differential), and search for the message values for both A and B for those steps. When the input differential is listed as 1, we search for two separate values for that input world; when the input differential is listed as 0, we search for a common value.

Here is how the search can be handled for each stage: you start at the first step of the stage, and initialize both the A version of the streams and the B version of the streams to be the hash state immediately prior to when the stage input occurs. Then, we scan through possible values of the first input word, and when we find one that meets all the constraints of the first word, we can then step to the second word, and start searching there. If we find a failure in the second word (there is no such input words that meet those constraints, we go back to the first word and look for another input that would meet the constraints there). In all cases, there is at least O(1) expected input words that meet all the constraints.
Some notes:

- On occasion, there will be no input words that meet all the constraints of a step. When this happens, we'll need to step back to the previous step. As this happens only occasionally, this adds only a constant factor to the running time.
- When we compute a step with an input differential (and hence the A input and the B input are different for this word), then there will generally be a constraint applied to the step as well. In this case, we can pick one of the streams that have a constraint, iterate through the possible A inputs, and compute the corresponding B input as:

  $$B := Stream_B[i_s] \oplus Stream_B[previ] \oplus Constant_s \oplus Xbox^{-1}[\ Delta \oplus Xbox\ [Stream_A[i_s] \oplus Stream_A[previ] \oplus A \oplus Constant_s\ ]\ ]$$

  where Delta is the target differential for that particular stream. Then, we can iterate through the other constraints on this step in the usual manner.
  If the constraint is that there be a 0 differential, then this computation can be simplified to:

  $$B := Stream_B[i_s] \oplus Stream_B[previ] \oplus Stream_A[i_s] \oplus Stream_A[previ] \oplus A$$

  In both cases, this allows us to scan through all possible settings of this step in only $O(2^{32})$ iterations; the time estimates we give below assume that we do this.

- Some of the constraints will be listed as "this differential must be the same as it was during phase 1". This means that, during phase 2, we need to constrain the differential here to the exact same value (xor) as we used during the corresponding step during the first phase. This constraint does not apply to phase 1; we can use any nonzero value here (and we'll need to repeat it during phase 2).

4.1 Stage 1: Steps 0-3

In these steps, the following constraints apply:

- Step 1: the stream2 differential will need to be the same as it was during phase 1
- Step 1: the stream3 differential will need to be the same as it was during phase 1
- Step 2: the stream2 differential will need to be the same as it was during phase 1

- Step 2: the stream3 differential will need to be the same as it was during phase 1
- Step 3: the stream1 differential will need to be 0
- Step 3: the stream2 differential will need to be 0
- Step 3: the stream3 differential will need to be 0

This stage takes an expected $O(2^{64})$ time during phase 2, and less during phase 1.

4.2 Stage 2: Steps 4-6

In these steps, the following constraints apply:

- Step 5: the stream2 differential will need to be the same as it was during phase 1
- Step 5: the stream3 differential will need to be the same as it was during phase 1
- Step 6: the stream1 differential will need to be 0
- Step 6: the stream2 differential will need to be 0
- Step 6: the stream3 differential will need to be 0

This stage takes an expected $O(2^{64})$ time during phase 2, and less during phase 1.

4.3 Stage 3: Step 7

In this step, the following constraint apply:

- Step 7: the stream3 differential will need to be the same as generated for stream3 during step 2.

This stage takes an expected $O(2^{32})$ time during both phases.

4.4 Stage 4: Step 8

In this step, the following constraint applies:

- Step 8: the stream2 differential will need to be the same as generated for stream2 during step 2.

This stage takes an expected $O(2^{32})$ time during both phases.

4.5 Stage 5: Steps 9

In this step, there are no constraints. The values for the input can be arbitrarily chosen
This stage takes an expected O(1) time during both phases.

4.6 Stage 6: Steps 10-12

In these steps, the following constraints apply:

- Step 11: the stream3 differential will need to be the same as it was during phase 1
- Step 12: the stream2 differential will need to be the same as it was during phase 1
- Step 12: the stream3 differential will need to be the same as generated for stream3 during step 7

This stage takes an expected $O(2^{64})$ time during phase 2, less during phase 1.

4.7 Stage 7: Step 13

In this step, the following constraint applies:

- Step 13: the stream2 differential will need to be the same as it was during phase 1

This stage takes an expected $O(2^{32})$ time during phase 2, less during phase 1

4.8 Stage 8: Step 14

In this step, the following constraint applies:

- Step 14: the stream2 differential will need to be the same as generated for stream2 during step 8.

This stage takes an expected $O(2^{32})$ time during both phases.

4.9 Stage 9: Step 15

In this step, there are no constraint. The values for the input can be arbitrarily chosen
This stage takes an expected O(1) time during both phases.

4.10 Stage 10: Steps 16, 17

In these steps, the following constraint applies:

- Step 17: the stream1 differential will need to be the same as generated for stream1 during step 2
- Step 17: the stream3 differential will need to be the same as generated for stream3 during step 12.

This stage takes an expected $O(2^{64})$ time during both phases.

4.11 Stage 11: Steps 18-21

In these steps, the following constraints apply:

- Step 19: the stream2 differential will need to be the same as it was during phase 1
- Step 20: the stream2 differential will need to be the same as generated for stream2 during step 14.
- Step 21: the stream1 differential will need to be 0
- Step 21: the stream2 differential will need to be 0
- Step 21: the stream3 differential will need to be the same as it was during phase 1

This stage takes an expected $O(2^{64})$ time for the second phase, less during phase 1.

4.12 Stage 12: Step 22

In this step, the following constraint applies:

- Step 22: the stream3 differential will need to be the same as generated for stream3 during step 17

This stage takes an expected $O(2^{32})$ time for both phases.

4.13 Stage 13: Steps 23-25

In these steps, there are no constraints. The values for the input can be arbitrarily chosen
This stage takes an expected O(1) time during both phases.

4.14 Stage 14: Step 26

In this step, the following constraint applies:

- Step 26: the stream2 differential will need to be the same as generated for stream2 during step 20

This stage takes an expected $O(2^{32})$ time for both phases.

4.15 Stage 15: Step 27

In this step, the following constraint applies:

- Step 27: the stream3 differential will need to be the same as generated for stream3 during step 22

This stage takes an expected $O(2^{32})$ time for both phases.

4.16 Stage 16: Steps 28-30

In these steps, there are no constraints. The values for the input can be arbitrarily chosen
This stage takes an expected O(1) time during both phases.

4.17 Stage 17: Steps 31-33

In these steps, the following constraint applies:

- Step 33: the stream1 differential will need to be 0
- Step 33: the stream2 differential will need to be 0
- Step 33: the stream3 differential will need to be 0

This stage takes an expected $O(2^{64})$ time for both phases.

In this stage, the processing is slightly different. We'll step through the 2^{64} possible choices for input words 31 and 32; and see which one leaves a common value for

$$Stream_B[i_s] \oplus Stream_B[previ] \oplus Stream_A[i_s] \oplus Stream_A[previ]$$

for all three streams. Once we find such a value, we can then choose appropriate values for input word 33.

If we examine all those steps, no step takes more than an expected $O(2^{64})$ steps. If we consider the two phases, and add in some budget for the occasional backtrack to a previous stage, we come up with a total cost of about $O(2^{70})$ steps.

References

1. Bob Hattersley, Waterfall Hash, Algorithm Specification and Analysis, 15 October 2008, http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/Waterfall.zip