Observations and Attacks On The SHA-3 Candidate Blender

Craig Newbold
cjnewbold@googlemail.com

Abstract

51 candidates have been accepted as first round candidates in NIST's SHA-3 competition, to decide the new cryptographic hash standard. Many of these submissions have no external cryptanalysis published, so the task begins to analyse their security and eliminate those that have vulnerabilities. In what we believe to be the first published external cryptananalysis of one candidate, Blender, we make observations on its structure, then exploit these features to give a multicollision attack of time complexity around $2^{\frac{n+w}{2}}$, and a first preimage attack of time complexity around $n2^{\frac{n+w}{2}}$. Both attacks have minimal space requirements, so we believe that this constitutes a break of Blender. We then leave possible improvements on these attacks as open problems.

1 Introduction

With SHA-1 broken [2, 3] and SHA-2 under suspicion, NIST announced a competition, similar to the AES process, to find a new hash standard. Of 64 submissions, 51, including Blender [1], met the acceptance criteria. Many of these submissions have no external cryptanalysis published, so the task begins to analyse their security and eliminate those that have vulnerabilities.

In section 2, we give a brief overview of Blender. In section 3, we present what we believe to to be the first published external analysis of Blender, and make observations on Blender's structure and security. In section 4, we present a multicollision attack based on these observations against Blender-n, word size w, with effort less than $2^{\frac{n+w}{2}}$ invocations of Blender, less than the claimed security for all proposed variants of Blender. In section 5, we use a similar technique to mount a preimage attack of complexity around $n2^{\frac{n+w}{2}}$. Finally, we suggest further improvements that may be possible, and leave open questions regarding the round function.

Notation 1 *Throughout this paper, unless noted otherwise, all addition is modulo 2^w, and $\neg$ denotes the bitwise complement.*

2 An Overview of Blender

For our purposes, all messages will be integer numbers of bytes, so some steps are omitted. The input to the main function is a series of words of length w bits ($w =$

32 for Blender-224 and -256, $w = 64$ for Blender-384 and -512). This string is prepared by taking the input message, then appending a message tail consisting of fill bytes to get the completed message to the required length, the length of the message in bits, the length of the length in bytes, and finally 2 checksums (checksum 1 and checksum 2). The fill bytes consist of concatenating the first 13 bytes of the message with themselves repeatedly, until the result is longer than the required length, then truncating the result to the required length. There is a minimum length imposed on the input string, imposed by increasing the number of fill bytes used if nessesscary. The checksums are computed from all previous words (message, length, and fill) as follows. Let $w_1, w_2, ..., w_n$ be the sequence of w bit words up to the checksums. Then:

$checksum1 = \neg(\sum w_i)$

$checksum2 = \sum(\neg w_i)$

Remark 2 *For our analysis, the message passed to Blender will consist of at least 13 arbitrary, constant bytes (enough that the fill bytes consist only of these bytes, and therefore remain constant as we change other parts of the message), followed by whichever blocks we require.*

After the tail is appended to the message to create the input string, a round function is used to repeatly update an 8 word + 2 bit state (the initial state is set equal to that of SHA-2). Each round uses one word of the input string to update the state, then the new state words are added to an accumulator of 8 words, initially set to 0. After the entire string has been used, the output of Blender is the contents of the accumulators.

Remark 3 *For our analysis, we attack only the structure, using no specific features of the round function. We therefore do not describe the round function itself.*

A more detailed specification is provided in the Blender specification [1].

3 Analysis

Our first observation concerns the checksums used in Blender - in particular that, if checksum 1 is equal for two messages of the same length, then checksum 2 is also equal for both messages.

Theorem 4 *Let $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$ be two equal length sequences of 2^w bit words. Then:*

$\neg(\sum x_i) = \neg(\sum y_i)$ *if and only if* $\sum(\neg x_i) = \sum(\neg y_i)$.

Proof. Note that $\neg x = 2^w - 1 - x = -x - 1 \pmod{2^w}$. Now:

$\neg(\sum x_i) = \neg(\sum y_i)$

$\Leftrightarrow \sum x_i = \sum y_i$

$\Leftrightarrow \sum -x_i = \sum -y_i$

$\Leftrightarrow \sum(-x_i - 1) = \sum(-y_i - 1)$

$\Leftrightarrow \sum(\neg x_i) = \sum(\neg y_i)$ $\blacksquare$

We also observe that the size of the main state is little larger than the output size ($n + 2$ bits), so collisions can be found in the main state with effort about $2^{\frac{n}{2}+1}$ by the Birthday Paradox. The security of Blender against Joux multicollisions [6] is therefore entirely reliant on the checksums and accumulator. Furthermore, these are all additive, so if the main state can be controlled, then it may not require massive effort to extend this control to these additive accumulators / checksums. Similar additive accumulators were attacked in [5].

4 Multicollision Attack

Here we present our first contribution - a multicollision attack with effort about $2^{\frac{n+w}{2}}$ for Blender-n. Taking an approach like in [5], we first get Joux multicollisions [6] in the main state, and then change intermediate state values to reach the same value in the accumulator. This can then be repeated to find Joux multicollisions for the entire function.

The basic attack consists of the following steps (the starting state for these steps is that after the arbitrary, constant words have been processed):

1) From the previous state, find some 2 equal length strings of words that collide in the main state.

2) Repeat step 1 until there are over $\frac{n+w}{2}+1$ such pairs, using the output state of the previous step 1 as the input to the next.

3) For each of these possible messages, compute checksum 1 and the accumulator values.

4) Return some pair of these possible messages with equal checksum 1 and the accumulator values.

In step 1, we expect that finding each such pair will take effort about $2^{\frac{n}{2}+1}$ by the Birthday Paradox, as noted above. We repeat this about $\frac{n+w}{2}+1$ times, for total effort about $(n+w)2^{\frac{n}{2}}$. In step 3, we expect an effort of about $2^{\frac{n+w}{2}}$ calculations of checksum 1 and each accumulator value, which should dominate the effort of steps 1 and 2. Again due to the Birthday Paradox, we expect that some pair of these should be equal, and can be returned by step 4. Due to Theorem 4, as both messages are the same length, then if accumulator 1 is equal for both messages, then accumulator 2 is also, so after both messages, all values (main state, checksums, accumulators, padding and length) are equal, so this process can be repeated l times to create a 2^l multicollision.

The total effort is little more than $l2^{\frac{n+w}{2}}$ calculations of checksum 1 and each accumulator value, which is better than the claimed security against multicollisions of 2^n invocations of Blender. Memory requirements are that of a generic Birthday Attack on a state of $n+w$ bits, and could be run essentially memoryless using Floyd‘s cycle finding algorithm.

Remark 5 *This is a very basic and naive implementation - many improvements are possible, such as generating more pairs in step 2, then directly finding multicollisions in step 4. It may also be possible to find collisions in the main state more effectively by exploiting the round function, or to exploit the linear-additive nature of the checksums and accumulators to more efficiently find collisions in the checksum and accumulator values.*

Remark 6 *Also note that as any such colliding pair collides in all state and accumulators, ie adding identical strings to both members of such a colliding pair gives a new colliding pair.*

5 Preimage Attack

We now develop this idea, adapting the techniques of Gauravaram and Kelsey [5] to more effectively control the value of the additive accumulators / checksum, and give our main contribution: a first preimage attack against Blender. We will first create Joux multicollisions in the main state, and then utilise Gauravaram and Kelsey‘s Checksum Control Sequences (CCS) to control the checksums (ensuring that after our message, all subsequent words - fill, length, length of length and checksums - are equal in all cases) and accumulators to reach a desired hash output (remember that the hash output is the final contents of the accumulators). The attack consists of the following steps (the starting state for these steps is that after the arbitrary, constant words have been processed):

1) From the previous state, find some 2 equal length strings of words that collide in the main state.

2) Repeat step 1 until there are over $\frac{n+w}{2}+1$ such pairs, using the output state of the previous step 1 as the input to the next.

3) We now expect to be able to find (within the $2^{\frac{n+w}{2}+1}$ possible message segments) some pair of message segments with equal checksum values, equal values in all but one of the accumulators, and any desired difference in the remaining accumulator.

4) Repeat steps 1-3 n times (starting step 1 from the state of the previous step 3), so that for each accumulator and each 2^x less than 2^w, we have some pair of message segments that can be swapped to change the accumulator value by 2^x, without affecting the main state or checksum. Save each pair of message segments.

5) Compute the hash of the message made up from the concatenation of one member of each pair.

6) Determine the difference in each accumulator from the desired value. Swap message segments as needed to set the accumulator to the desired value. Return this string of message segments.

Step 1 should take effort about $2^{\frac{n}{2}+1}$, which is repeated $\frac{n}{2}+1$ times in step 2. In step 3, we essentially run a modified Birthday Attack collision search, expecting effort about $2^{\frac{n+w}{2}}$ calculations of checksum 1 and each accumulator value. We then repeat all this n times, for a total complexity of about $n2^{\frac{n+w}{2}}$ calculations of checksum 1 and each accumulator value. Compared to this, the complexity of computing one hash, and calculating the required differences (steps 5 and 6) is negligible.

The memory requirements for the Birthday Attack can be removed for minimal extra cost (using a slight varient of Floyd‘s cycle finding algorithm), and we must save only $2n$ message segments of several words each in step 4. The total complexity is therefore better than the claimed (and required) security of 2^n invocations of Blender for a First Preimage Attack, so we believe that Blender is broken.

Remark 7 *Although the discarding of the final main state was intended to make certain attacks more difficult, it was the key feature that permits this preimage attack. Similarly, we exploit the linear-additive nature of the accumulator to control it.*

6 Further Work

We have developed a basic structural multicollision attack, and also a preimage attack, but there are areas highlighted for further work. The round function itself has been completely ignored, yet is relatively simple. We leave it as an open problem whether the round function can be exploited, to more accurately and easily control to main state values. It may also be possible to improve on the work of Gauravaram and Kelsey [5], as there are $\frac{n}{w}$ w bit additive accumulators, rather than one n bit accumulator. Unfortunately the attack complexity is still impractical, so cannot be implemented, but we invite others to review (and possibly improve on) our findings.

References

[1] Colin Bradbury: *BLENDER: A Proposed New Family of Cryptographic Hash Algorithms*, Submission to NIST, 2008. `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`

[2] Christophe De Cannire and Christian Rechberger: *Finding SHA-1 Characteristics: General Results and Applications*, Advances in Cryptology ASIACRYPT 2006.

[3] Martin Cochran: *Notes on the Wang et al. 2^{63} SHA-1 Differential Path*, Cryptology ePrint Archive, 2007. `http://eprint.iacr.org/2007/474`

[4] *NIST: Cryptographic Hash Project*, `http://www.csrc.nist.gov/groups/ST/hash/index.html`

[5] Praveen Gauravaram and John Kelsey: *Cryptanalysis of a class of cryptographic hash functions*, Cryptology ePrint Archive, 2007. `http://eprint.iacr.org/2007/277`

[6] Antoine Joux: *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions*, Crypto 2004, 2004.