Practical Attacks on NESHA-256

Orr Dunkelman[1] and Tor E. Bjørstad[2]

[1] École Normale Supérieure, INRIA, CNRS, Paris, France.
<orr.dunkelman@ens.fr>
[2] The Selmer Center, Dept. of Informatics, University of Bergen, Norway.
<tor.bjorstad@ii.uib.no>

Abstract. NESHA-256 is a cryptographic hash function designed by Esmaeili et al. and presented at WCC '09. We show that NESHA-256 is highly insecure.

1 Introduction

Cryptographic hashing has been a vital research area in recent years, after significant breakthroughs were made in attacking the most common algorithms, MD5 and SHA-1 [7, 8]. As a consequence of these advances, and worry that the current hash standard, SHA-2, may not be as secure as anticipated, NIST launched a public competition to establish a new hash standard, called SHA-3 [5]. The competition attracted 64 submissions, and is scheduled to conclude in 2012.
The hash algorithm NESHA-256 was proposed in [6] and presented at WCC '09. It was intended to be submitted as a candidate to the NIST competition, but this did for various reasons not happen. The algorithm is a traditional narrow-pipe Merkle-Damgård construction, using a custom block cipher in the well-known Davies-Meyer mode.

2 The NESHA-256 Compression Function

The NESHA-256 algorithm is a word-based design. Its internal state consists of eight 32-bit words, denoted $A, \ldots, H$; the message to be hashed is processed in 512-bit chunks, $M_0, \ldots, M_{15}$. The message schedule of NESHA-256 produces 16 expanded words, $\hat{M}_1, \ldots, \hat{M}_{15}$ from the message using modular addition and exclusive or, with each expanded word depending on four message words.
In the compression function, the message, expanded message, and chaining value is processed by four independent branches, whose outputs are combined to form the new chaining value. The branches differ only by the order in which the message and expanded message words are used. Each branch consists of four rounds, which again consists of a nonlinear layer based on T-functions [3] and modular addition, and a word-wise diffusion layer based on the pseudo-Hadamard transform [4].

A high level overview of NESHA-256 compression function can be found in Fig. 1, and a diagram showing part of the nonlinear layer is given in Fig. 2. The T-functions f and g are defined as follows:

$$f(x) = x + ((x * x)|7), \tag{1}$$

$$g(x) = x * x + 3 * x + \texttt{0xbf597fc7}. \tag{2}$$

For further information about the algorithm we refer to the NESHA-256 specification [6].

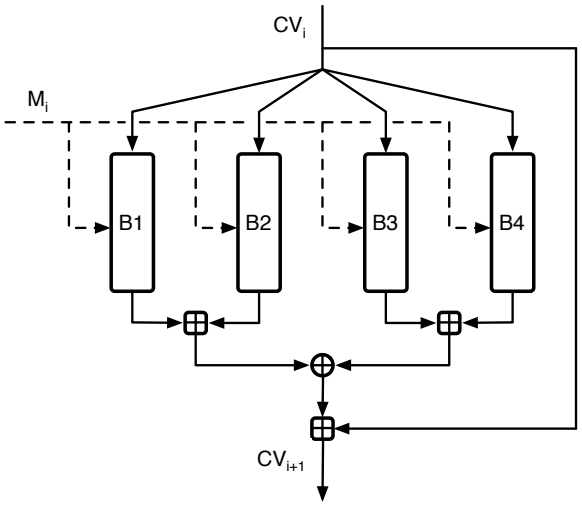

Fig. 1. High level view of NESHA-256 compression function

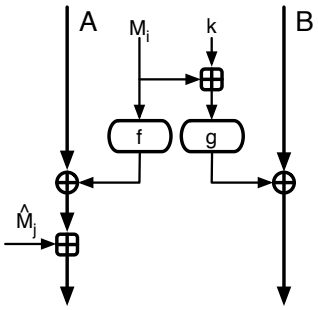

Fig. 2. Detail from the non-linear layer in the NESHA-256 round function

3 Attacking NESHA-256

Whereas the pseudo-Hadamard transform layer provides good wordwise diffusion, the bit-level properties of NESHA-256 are a different story. The only GF(2)-nonlinear components in the compression function are modular additions, and the T-functions f and g. A salient feature of all these functions is that bit differences propagate only to the left, from low order to higher order bits. Furthermore, the compression function contains no bitwise permutations, rotations or other devices to diffuse differences from the higher order bits into the rest of the state.

An immediate consequence of this is that the compression function is linear with respect to the high order bit of every word in the state and message. Since we have more degrees of freedom in the message than constraints from the chaining value, it is easy to find differences that lead to collisions. A particularly simple one is found by inserting the XOR difference (δ = `0x80000000`) in *every* message word. This leads to a zero difference in each of the 16 expanded message words, and the difference in the four branches of the compression function becomes zero after every other round. There are many other input differences that work, since there are 2^{16} possible assignments to the high bits in the message, and only 2^8 possible outcomes.

As the above differential is completely independent of the choice of message, the approach can also be used directly to find (several) second preimages: given a fixed message block M, we immediately know that $M' = M_0 \oplus \delta, \ldots, M_{15} \oplus \delta$ is a second preimage of the compression function. Using a C-implementation of the compression function obtained from the NESHA-256 team [1,6][1], it appears that there are 511 non-zero high bit differences that yield the same hash output.

Another interesting idea is to exploit the poor bitwise diffusion of NESHA-256 to obtain preimages for the compression function. The plan is to use a bit-slicing approach, somewhat similar to the technique used in [2]. First, we exhaustively search the 16 low order bits of the message block[2] to find an input slice such that the 8 low order bits of the computed chaining value take the desired value. Having obtained a correct configuration for the first slice, the search can be repeated, building the preimage incrementally from right to left over the remaining 31 slices. Under normal circumstances, this procedure can be *expected* to yield a preimage in roughly $2^5 \cdot 2^8 = 2^{13}$ compression function calls.

Unfortunately, the procedure does not work as expected, when tested using the reference implementation of `NESHA_Compression_Function()` from [1,6]. Looking more closely at the implementation, we observe some odd behaviour:

[1] Source code is given in the appendix of the WCC pre-proceedings version of [6], but currently not in the ePrint version of same; an electronic version of this implementation was obtained via [1].

[2] The last 65 bits of the final block are used by the MD-padding, so to find preimages of the hash function itself there are only 14 degrees of freedom. Under normal circumstances, this is still plenty, since we only need to match an 8-bit constraint.

- For the low order 16-bit message slice, we obtain only 32 possible configurations for the corresponding 8-bit output slice, each occuring exactly 2048 times. For a random function, we would expect that all 256 slices appear with a roughly uniform distribution.
- The possible configurations of the low-order bit slices appears to be a partition; starting from the low-order slice of some IV, it is not possible to reach any of the other 224 configurations *at all*.
- For the other bit slices, we obtain 128 possible configurations of the 8-bit output slice, each occuring 512 times. This is also highly nonrandom.
- The output of our obtained implementation does *not* agree with the test vectors given in [6].

It is currently unclear to us whether it is the implementation or the test vectors that are incorrect. In case the implementation is correct, it would from the above seem that it is impossible to obtain arbitrary preimages for the hash function itself [3]. If the implementation we are using contains errors, there is little reason to believe that our attack will not work as expected on a patched version, thus obtaining arbitrary preimages for the full hash function after only about 2^{13} compression function calls. Either way, it seems that the algorithm is highly flawed.

Pseudo-preimages are of course readily computable by our bit-slicing approach, using the freedom in both message block and the IV to obtain up to 24 degrees of freedom in each slice. Experimentally, the complexity of this lies around 2^{20} compression function calls. Finding "near-preimages" is also simple, by choosing a message that yields a low-weight error (0 when possible) at each step.

4 Conclusion

We believe that NESHA-256 is highly insecure. Our analysis is quite superficial, yet reveals severe weaknesses in the algorithm as specified. We believe future research efforts should be directed towards the currently unbroken candidate algorithms in the NIST Hash Function Competition, rather than attempting to repair or patch NESHA-256.

References

1. M. R. S. Abyaneh and M. M. Hassanzadeh. Personal communication, May 2009.
2. C. de Cannière and C. Rechberger. Preimages for Reduced SHA-0 and SHA-1. In *Proceedings of CRYPTO '08*, volume 5157 of *LNCS*, pages 179–202, 2008.
3. A. Klimov and A. Shamir. Cryptographic Applications of T-functions. In *Proceedings of SAC '03*, volume 3006 of *LNCS*, pages 248–261, 2004.

[3] On the other hand, a preimage of any *valid* hash output can still be found with our method. As the algorithm does not hash onto the entire output space, the efficiency of brute force search will also be greater than expected.

4. H. Lipmaa. On Differential Properties of Pseudo-Hadamard Transform and Related Mappings. In *Proceedings of INDOCRYPT '02*, volume 2551 of *LNCS*, pages 48–61, 2002.
5. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available online as `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf` (July 2009).
6. Y. E. Salehani, S. A. H. A. E. Tabatabaei, M. R. S. Abyaneh, and M. M. Hassanzadeh. NESHA-256, NEw 256-bit Secure Hash Algorithm. In *Pre-proceedings of WCC '09*, 2009. First published in the Cryptology ePrint Archive as `http://eprint.iacr.org/2009/033`.
7. X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Proceedings of CRYPTO '05*, volume 3621 of *LNCS*, pages 17–36, 2005.
8. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of EUROCRYPT '05*, volume 3494 of *LNCS*, pages 19–35, 2005.

A Appendix

The examples below give a second preimage and a pseudo-preimage for the NESHA-256 compression function, as well as a near-preimage of zero.

Collision / second preimage (1 comp. function call)

IV `0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,`
`0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19.`

M0 `0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000.`

M1 `0x80000000, 0x80000000, 0x80000000, 0x80000000,`
`0x80000000, 0x80000000, 0x80000000, 0x80000000,`
`0x80000000, 0x80000000, 0x80000000, 0x80000000,`
`0x80000000, 0x80000000, 0x80000000, 0x80000000.`

H0 `0xbb6f7a73, 0x629d3473, 0xf5c1b822, 0x8628e9dc,`
`0x65e358f7, 0xd5532c5a, 0xceeb5265, 0x389e294b.`

H1 `0xbb6f7a73, 0x629d3473, 0xf5c1b822, 0x8628e9dc,`
`0x65e358f7, 0xd5532c5a, 0xceeb5265, 0x389e294b.`

Pseudo-preimage of 0 (853000 comp. function calls)

IV2 `0x5442ce30, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000.`

M2 `0xa912939a, 0xb0d36eb4, 0x3a05a47a, 0x872eeed2,`
`0x89a5ce62, 0xcb4ba1cc, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0xda532ad8, 0x00000000, 0x00000000, 0x00000000.`

H2 `0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000.`

Near-preimage of 0 (892 comp. function calls, weight 24)

IV3 `0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,`
`0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19.`

M3 `0x835424be, 0x7fcead27, 0xb3817d8e, 0x65ab5dc0,`
`0x8198dc9c, 0x03d060d0, 0x00000001, 0x00000000,`
`0x00000000, 0x00000000, 0x00000000, 0x00000000,`
`0x00000000, 0x01000000, 0x00000000, 0x000001bf.`

H3 `0x00040f08, 0x00000000, 0x08200000, 0x00000000,`
`0x00000000, 0x40000000, 0x804aa0f4, 0x20010003.`