

The AHASH Mode of Operation

John Viega

This document specifies AHASH, a collision resistant cryptographic hash function built using AES, producing a 256-bit output. This function is an instantiation of MDC-2, a construct that was once popular with DES, and is widely conjectured to have a good security margin when used with AES.

Recent attacks against dedicated cryptographic hash functions [1, 6] such as MD5 and SHA-0 have raised some concerns that current constructs of this sort may eventually be shown insecure.

Certainly, the cryptographic community is much more experienced building block ciphers than it is dedicated cryptographic hash functions, and it would be nice to have a construct that could leverage the extensive peer review that AES has received. Additionally, a construct that consists of only a simple wrapper around the block cipher would facilitate compact and inexpensive hardware designs.

This document specifies AHASH, a cryptographic hash function built using AES that maps strings of length up to 2^{128} bits to a 256-bit output. AHASH is an instantiation of MDC-2 [3], a construct typically instantiated with DES. The pairing of MDC-2 with DES is specified in ISO standard 10118-2. We use the name AHASH (AES-based hash) to avoid confusion, since the term MDC-2 often implies use of DES.

Section 1.0 is the normative section of this document, specifying the AHASH construct. Section 2.0 provides a brief rationale for the construct, and Section 3.0 contains an intellectual property statement. Reference code and test vectors are provided in Appendix A and Section B respectively.

1.0 AHASH specification

This section contains the complete definition of AHASH for use with AES with 128-bit keys. In Section 1.1 we define the AHASH operation in terms of an underlying compression function C , which we specify in Section 1.2.

In this section, we represent AES encryption of a 128-bit plaintext P under a key k as $E_k(P)$. Strings are considered an array of bytes. A substring from byte number i to j inclusive is denoted by $S[i:j]$. All indexing begins at 1. String concatenation is denoted by \parallel .

1.1 The hash operation

AHASH takes a single string as an input, up to 2^{128} bits in length, and produces a 256-bit output. Let the length of the input string be L . The input string is padded by adding a single one bit to the end, followed by enough zero bits to make the resulting string a multiple of 128 bits in length. Finally, a 128-bit big endian representation of L is added to the end of the string.

Call the fully padded string S . This string is processed in n 128-bit segments, S_1 through S_n , using the following algorithm:

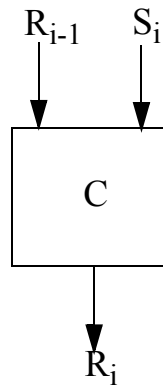
$$R_i = C(R_{i-1}, S_i)$$

R_0 is a fixed value, and is represented by the string consisting of 16 consecutive bytes valued 0x52 (82 in base 10), followed by 16 consecutive bytes valued 0x25 (37 in base 10).

This function is illustrated in Figure 1.

FIGURE 1.

The AHASH operation.



1.2 The compression function C

AHASH is built on a compression function $C(H,P)$, which accepts a 256-bit input H and a 128-bit input P , and produces a 256-bit output R . C has the following definition:

$$k_0 = H[1:16]$$

$$k_1 = H[17:32]$$

$$X = E_{k_0}(P) \oplus P$$

$$Y = E_{k_1}(P) \oplus P$$

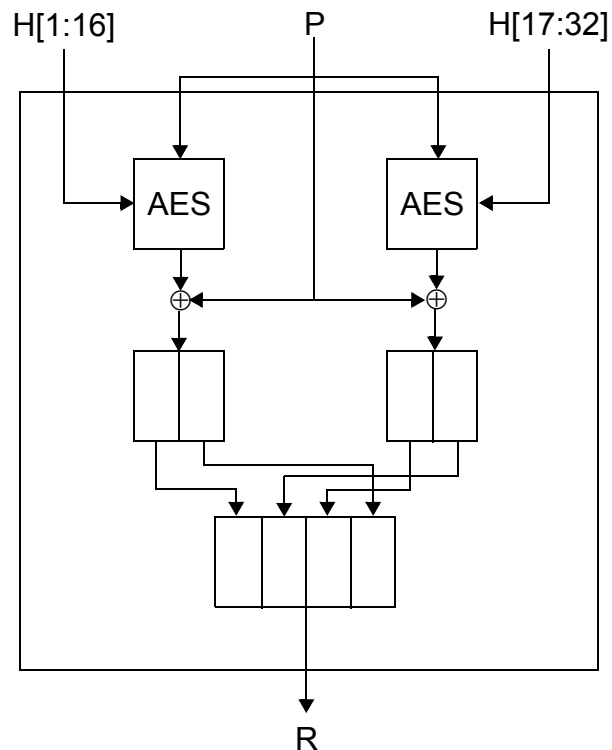
$$R = X[1:8] \parallel Y[9:16] \parallel Y[1:8] \parallel X[9:16]$$

The result of the function C is the value of R .

The compression function is illustrated in Figure 2.

FIGURE 2.

The AHASH compression function, C



2.0 Analysis

AHASH is a straightforward implementation of the MDC-2 algorithm, instantiated with AES. MDC-2 itself uses two separate instances of the Matyas-Meyer-Oseas (MMO) compression function [5], runs them in parallel, then reorders the output. The underlying MMO function is provably secure in the ideal cipher model [2], which makes a strong assumption that the underlying block cipher acts like a random function. The assumption that a block cipher is ideal is a stronger assumption than the PRP assumption (that the block cipher is indistinguishable from a random permutation).

To date, there is no similar proof of security for the entire MDC-2 construct. However, in the ideal cipher model, the lower bound for MDC-2 security is clearly no worse than that of MMO. Unfortunately, MMO on a cipher with 128-bit blocks does not provide adequate long-term security against collision (birthday) attacks.

For an underlying block cipher with block size n , the best known pre-image attack on MDC-2 requires $O\left(2^{\frac{3n}{2}}\right)$ effort [4], and the best known collision attack is a birthday attack, requiring $O(2^n)$ work.

If AES were an ideal cipher, the security of the base MMO would be birthday bound, not bound by AES key strength. We conjecture that, if AES in MMO mode were to be broken, then AES would have damaging related key attacks, in which case it would almost certainly fail in any hashing mode of operation.

If the MDC-2 construct itself is shown to have significantly lower security bounds than the best known attacks, and the resulting strength is not consistent with the best known attacks on AES, then it would be wise to avoid using MDC-2 with any strength AES key.

Note that we do not believe AES is likely to be subject to practical related key attacks, nor do we believe that AHASH itself will be subject to a feasible attack. However, if one of these failures occurs, using larger AES keys is unlikely to be wise risk mitigation. Therefore, while AHASH variants with larger key sizes are certainly possible, we see no compelling reason to specify them.

3.0 Intellectual Property

AHASH is believed to be free of intellectual property restrictions. The underlying construct is MDC-2, which was covered by patent in the United States [3]. However, that patent expired on August 28, 2004.

References

- [1] E. Biham, R. Chen, “Near-Collisions of SHA-0”, *Advances in Cryptology - Crypto ‘04*, LNCS 3152, Springer-Verlag (2004), to appear.
- [2] J. Black, P. Rogaway, T. Shrimpton, “Black-box Analysis of the block-cipher-based hash-function constructions from PGV”, *Advances in Cryptology - Crypto ‘02*, LNCS 2442, Springer-Verlag (2002), pp. 320-355.
- [3] B. Brachtel, D. Coppersmith, M. Hyden, S. Matayas, C. Meyer, J. Oseas, S. Pilpel, M. Schilling, “Data Authentication Using Modification Detection Codes Based on a Public One Way Encryption Function,” U.S. Patent Number 4,908,861, March 13, 1990.
- [4] X. Lai, J. Massey, “Hash functions based on block ciphers,” *Advances in Cryptology - Eurocrypt ‘92*, LNCS 658, Springer-Verlag (1993), pp. 55-70.
- [5] S. Matyas, C. Meyer, J. Oseas, “Generating strong one-way functions with cryptographic algorithm”, *IBM Technical Disclosure Bulletin*, 27 (1985), pp. 5658-5659.
- [6] X. Wang, D. Feng, X. Lai, H. Yu, “Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD”, *Cryptology ePrint Archive*, Report 2004/199, <http://eprint.iacr.org/2004/199/>.

Appendix A Reference code

This code relies on the optimized AES implementation distributed from Vincent Rijmen's AES page (<http://www.esat.kuleuven.ac.be/~rijmen/rijndael/>).

```
#include "rijndael-alg-fst.h"
#include <stdlib.h> /* for size_t definition. */
#include <string.h> /* for memset definition. */

#define BLOCK_SZ 16

#define AES_ENC_SETUP(sched, key) rijndaelKeySetupEnc(sched, key, 128)
#define AES_ENC(sched, pt, ct) rijndaelEncrypt(sched, 10, pt, ct)
typedef u32 AES_T[4*(MAXNR+1)];

static void ahash_compress(char *, char *);

void ahash(char *str, size_t len, char out[32]) {
    size_t i, n, lo;
    char leftovers[16] = {0}, *p;

    memset(out, 0x52, 16);
    memset(out + 16, 0x25, 16);

    n = len / 16;
    lo = len % 16;
    p = str;

    for (i=0;i<n;i++) {
        ahash_compress(out, p);
        p += 16;
    }
    for(i = 0; i<lo; i++) {
        leftovers[i] = p[i];
    }

    leftovers[i] = 0x80;
    ahash_compress(out, leftovers);
    memset(leftovers, 0, 16);

    /* We don't worry about sizeof(size_t) or endianness issues, as
     * this works until the day that size_t is greater than 128 bits.
     */
    i = 16;
    while (i-- && len) {
        leftovers[i] = len & 0xff;
        len /= 256;
    }
    ahash_compress(out, leftovers);
}

static void ahash_compress(char *h, char *p) {
    AES_T sched0, sched1;
    int i;
    char tmp;

    AES_ENC_SETUP(sched0, h);
    AES_ENC_SETUP(sched1, h+16);
    AES_ENC(sched0, p, h);
    AES_ENC(sched1, p, h+16);
    for (i=0;i<16;i++) {
        h[i] ^= p[i];
        h[i+16] ^= p[i];
    }
    for (i=8;i<16;i++) {
        tmp = h[i];
        h[i] = h[i+16];
        h[i+16] = tmp;
    }
}
```

Test vectors

H₁ : 5a21ee5b84a8446b05e0393525fcfd8e
d8e9550e2cec0f1d0d0fed9410fd4068
P₂ : 101112131415161718191a1b1c1d1e1f
H₂ : 8ff184935ceb6a872d22a73450b6ebe8
217dafe37d7913cae22d4086b7015e69
P₃ : 00000000000000000000000000000001f
Result : 2543861f780f4f605c81fb6b959a0918
fb06f2301cfba713edbd7820a8f159c5

Test case 5

Input : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
3031323334
H₀ : 52525252525252525252525252525252
25252525252525252525252525252525
P₁ : 000102030405060708090a0b0c0d0e0f
H₁ : 5a21ee5b84a8446b05e0393525fcfd8e
d8e9550e2cec0f1d0d0fed9410fd4068
P₂ : 101112131415161718191a1b1c1d1e1f
H₂ : 153642c1826230816a0a46e103fb05ad
67691b1925054ae67202145cc09b91d0
P₃ : 202122232425262728292a2b2c2d2e2f
H₃ : 17a111e89849c528f7b5462f5c62ce08
3bd56752d775d64ee69842a5cccf560e
P₄ : 30313233348000000000000000000000
H₄ : e807ea32c42c29bcd76b65ba9e60eb48
8a55cbad64d16feeb2dd71fc7446606b
P₅ : 000000000000000000000000000000035
Result : 22d7b528fffac96ef9120b97f310f847
f68d5fef912a1bd7ef6ee02db75be30d

Test case 6

Input : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
303132333435363738393a3b3c3d3e3f
404142434445464748494a4b4c4d4e4f
505152535455565758595a5b5c5d5e5f
60616263
H₀ : 52525252525252525252525252525252
25252525252525252525252525252525
P₁ : 000102030405060708090a0b0c0d0e0f
H₁ : 5a21ee5b84a8446b05e0393525fcfd8e
d8e9550e2cec0f1d0d0fed9410fd4068
P₂ : 101112131415161718191a1b1c1d1e1f
H₂ : 153642c1826230816a0a46e103fb05ad
67691b1925054ae67202145cc09b91d0
P₃ : 202122232425262728292a2b2c2d2e2f
H₃ : 17a111e89849c528f7b5462f5c62ce08
3bd56752d775d64ee69842a5cccf560e
P₄ : 303132333435363738393a3b3c3d3e3f
H₄ : 031080fc1838906f1d483616a55514d5
0b17bf4045ec4dec457c321d58d13426

Test vectors

P₅ : 404142434445464748494a4b4c4d4e4f
H₅ : 8e67bb0e70160744cd8b5bd4f0cd5a6c
1e118e9ef5268138e455a01138dcbd82
P₆ : 505152535455565758595a5b5c5d5e5f
H₆ : 4bae5b8f9938b9d15d3b536dfcd33e80
fa87614969bcfea20fe3e1a961989441
P₇ : 606162638000000000000000000000
H₇ : efb34affaacb460758bb5dc5938c8351
29a406de8fb9581ab8e75d771c98c944
P₈ : 00000000000000000000000000000064
Result : 07ea267b1d5561ff5a8fb104293253f9
02569143ca48ef7fbee4109ca07e75c

Test case 7

Input : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
303132333435363738393a3b3c3d3e3f
404142434445464748494a4b4c4d4e4f
505152535455565758595a5b5c5d5e5f
606162636465666768696a6b6c6d6e6f
707172737475767778797a7b7c7d7e7f
H₀ : 525252525252525252525252525252
252525252525252525252525252525
P₁ : 000102030405060708090a0b0c0d0e0f
H₁ : 5a21ee5b84a8446b05e0393525fcfd8e
d8e9550e2cec0f1d0d0fed9410fd4068
P₂ : 101112131415161718191a1b1c1d1e1f
H₂ : 153642c1826230816a0a46e103fb05ad
67691b1925054ae67202145cc09b91d0
P₃ : 202122232425262728292a2b2c2d2e2f
H₃ : 17a111e89849c528f7b5462f5c62ce08
3bd56752d775d64ee69842a5ccccf560e
P₄ : 303132333435363738393a3b3c3d3e3f
H₄ : 031080fc1838906f1d483616a55514d5
0b17bf4045ec4dec457c321d58d13426
P₅ : 404142434445464748494a4b4c4d4e4f
H₅ : 8e67bb0e70160744cd8b5bd4f0cd5a6c
1e118e9ef5268138e455a01138dcbd82
P₆ : 505152535455565758595a5b5c5d5e5f
H₆ : 4bae5b8f9938b9d15d3b536dfcd33e80
fa87614969bcfea20fe3e1a961989441
P₇ : 606162636465666768696a6b6c6d6e6f
H₇ : ce190fb756ce8a23a552681800307488
91198548a84a5b99e3a28608400fff05
P₈ : 707172737475767778797a7b7c7d7e7f
H₈ : 8c252a6617a3e343b173cfc9865954d8
dd0d799108ed916e42efa0c0b6e8c5d4
P₉ : 800000000000000000000000000000
H₉ : f47ace8848511db84dbccb5a3a0bcbfd
95159a0f27d1c4fb9e5020330cd6aa87
P₁₀ : 00000000000000000000000000000080
Result : 48c06cdb3810905f2a20d049094c0b67
47382a220487113f35ec7f83fa883166