Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: April 29, 2017

A. Cope
Google

October 26, 2016

Hash-Encrypt-Hash, a block cipher mode of operation
draft-cope-heh-00

Abstract

This memo describes a block cipher mode of operation known as Hash-Encrypt-Hash (HEH).

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 29, 2017.

Copyright Notice

Table of Contents

1. Introduction

This memo describes the implementation of the Hash Encrypt Hash (HEH) block cipher mode of operation as both an encryption algorithm and an AEAD. The primary benefit of HEH is that it extends the the strong pseudorandom permutation property of block ciphers to arbitrary-length messages. This means that if any bit of the plaintext is flipped, each bit in the ciphertext will flip with 50% probability. No block cipher mode of operation that is currently in widespread use has this property. Additionally, HEH is more resistant to misuse than commonly-used block cipher modes of operation. For example, if nonces are reused, CTR fails catastrophically, and CBC will leak common prefixes of the underlying block size. HEH has neither of those problems.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Notation

blk_key - key for the underlying block cipher.

block - 16 bytes.

buffer[i] - block i of buffer. Defined for 0 <= i < N.

buffer[N+] - bytes 16 * N until the end of buffer. The unpadded partial block.

EMPTY - buffer of length 0.

GF(2^128) - The Galois field of 2^128 elements, as defined in section 4.1.

msg - shorthand for message, a buffer that is an input to a function.

N - FLOOR(msg_length / 16), number of full blocks of msg.

out_msg - buffer that is a transformation of msg. out_msg_length = msg_length unless otherwise explicitly specified

prf_key - pseudo-random function key.

tau_key - 16 byte key used to compute the hash.

XOR - bitwise exclusive-or.

XXXX_length - length of XXXX in bytes.

* - Multiplication in GF(2^128) as defined in section 4.2.

+ - Addition in GF(2^128) as defined in section 4.3.

0^i - buffer of i zero bytes.

|| - concatenation.

3. Overview

3.1. Key size

All implementations MUST support a key size of 48 bytes. For a 48-byte key, the first 16 bytes correspond to tau_key. The second 16 bytes correspond to prf_key. The final 16 bytes correspond to blk_key. Implementations MAY also support key sizes of 64 and 80 bytes, in which case tau_key corresponds to the first 16-byte chunk. The remainder of the key is split in half, with the first half corresponding to prf_key and the second half corresponding to blk_key.

3.2. Block cipher

HEH MUST use a block cipher with a block size of 128 bits.

3.3. Nonce and AAD

HEH SHOULD support a 16-byte nonce. Support for other nonce lengths between 0 and 2^32-1 (inclusive) bytes is OPTIONAL. Support for additional authenticated data (AAD) and support for varying AAD lengths between 0 and 2^32-1 (inclusive) bytes is OPTIONAL. Security implications are discussed in section 7.1

4. GF(2^128) math

4.1. GF(2^128)

GF(2^128) is the Galois field of 2^128 elements defined by the irreducible polynomial x^128 + x^7 + x^2 + x + 1.

Elements in the field are converted to and from 128-bit strings by taking the least-significant bit of the first byte to be the coefficient of x^0, the most-significant bit of the first byte to the the coefficient of x^7, and so on, until the most-significant bit of the last byte is the coefficient of x^127 [AES-GCM-SIV].

Examples:
10000111 || 0^15 = x^7 + x^2 + x + 1
0^15 || 00000001 = x^120.
0^15 || 10000000 = x^127.

4.2. Multiplication in GF(2^128)

```
Input
   Two 128 bit elements X, Y

Output
   128 bit element X * Y
```

Multiplication is defined on 128 bit blocks by converting them to polynomials as described above, and then computing the resulting product modulo x^128 + x^7 + x^2 + x + 1.

4.3. Addition in GF(2^128)

```
Input
   Two 128 bit elements X, Y

Output
   128 bit element X + Y
```

For any two 128 bit elements X, Y in the Galois field, X + Y is defined as X XOR Y.

The operations + and XOR are interchangeable within this document. For consistency we use + on 128 bit strings and XOR if the arguments are not 128 bits long.

5. Algorithm

When appropriate, we will explain the output as both a mathematical formula and in pseudo-code. This information is redundant, and it exists to provide additional clarity. Implementations need not implement the exact algorithm specified by the pseudocode, so long as the output matches what the pseudocode would produce.

5.1. generate_betas

To generate the beta_keys needed by HEH_hash, we take the CMAC as defined in [CMAC] of the nonce, AAD, nonce_length, AAD_length and plaintext_length. We use CMAC because it is a pseudorandom function on variable length inputs.

```
   Input
      prf_key, nonce, AAD, plaintext_length

   Output
      beta1_key = CMAC(key = prf_key, message = pad_16(nonce) ||
                       pad_16(AAD) || pad_16(nonce_length ||
                       AAD_length || plaintext_length))
      beta2_key = x * beta1_key
      return beta1_key, beta2_key
```

Where pad_16(X) = X right-padded with 0's up to a multiple of 16 bytes. If X is already a multiple of 16 bytes (including if X is 0 bytes), this is a no-op.

The following MUST be true in order to generate conformant ciphertext:

- nonce_length, AAD_length, and plaintext_length MUST be 4 bytes long.
- nonce_length, AAD_length, and plaintext_length MUST be stored in little-endian format.
- The input to CMAC MUST be padded with 0x00 bytes up to a multiple of 16 bytes.
- CMAC MUST use the same block cipher that is used in CTS_2ECB_encrypt.
- CMAC MUST be implemented as described in [CMAC]. In particular, if CMAC is being reimplemented for HEH, be advised that there is a multiply-by-x substep of CMAC that uses a different finite field representation than the one described in section 4.

5.2. poly_hash

Poly_hash treats each block of msg as a coefficient to a polynomial in GF(2^128), and evaluates that polynomial at tau_key to create a hash. Poly_hash is called as a subroutine of HEH_hash so that any minor change to msg will result in every block being changed in HEH_hash with high probability. Note that the coefficients of m_{N-1} and m_N are flipped. This is done to simplify the implementation of HEH_hash_inv.

```
Input
   msg, tau_key

Output
   k^N * m_0 + ... + k^2 * m_{N-2} + k * m_N + m_{N-1}
   Where k = tau_key,
   m_i = msg[i], for i = 0 to N-1,
   m_N = msg[N+] padded up to 16 bytes with a 0x01 byte
      followed by 0x00 bytes.  When msg_length is a multiple of
      16, m_N is composed entirely of padding, i.e. 0x0100...00.

pseudo-code:
   p = 0^16
   For i = 0 to N - 2
      p *= tau_key
      p += msg[i]
   p *= tau_key
   p += m_N // as defined above
   p *= tau_key
   p += msg[N-1]
   return p
```

5.3. HEH_hash

The Hash step in Hash-Encrypt-Hash. HEH_hash is an invertible hash function used to ensure any change to the msg will result in every full block being modified with high probability.

```
      Input
         msg, beta_key, tau_key

      Output
         out_msg = (m_0 + R, ..., m_{N-2} + R, R, m_N) +
                     (xb, x^2b, ..., x^{N-1}b, b, 0)
            where m_i = msg[i] for i = 0 to N-1,
            m_N = msg[N+],
            R = out_msg of poly_hash,
            b = beta_key,
            x is the element x in GF(2^128).

      pseudo-code:
         R = poly_hash(msg, tau_key)
         e = beta_key * x
         For i = 0 to N-2
            out_msg[i] = msg[i] + R + e
            e = e * x
         out_msg[N-1] = R + beta_key
         out_msg[N+] = msg[N+]
         return out_msg
```

5.4. HEH_hash_inv

Inverse of HEH_hash

```
      Input
         msg, beta_key, tau_key
      Output
         out_msg

      pseudo-code
         R = msg[N-1] + beta_key
         e = beta_key * x
         For i = 0 to N-2
            out_msg[i] = msg[i] + R + e
            e = e * x
         out_msg[N+] = msg[N+]
         out_msg[N-1] = 0^16
         // now all block in out_msg are correct except for
         // out_msg[N-1], which is all zeroes
         R_without_constant_term = poly_hash(out_msg, tau_key)
         out_msg[N-1] = R + R_without_constant_term
         return out_msg
```

5.5. CTS_2ECB_encrypt

The encryption step of Hash-Encrypt-Hash. Uses a modification of CTS-ECB. Because HEH_hash is the identity function on partial blocks, we instead xor the partial block with the final encrypted full block then re-encrypt the final full block. This technique is discussed in [TET].

```
Input
   msg, blk_key
Output
   out_msg

pseudo-code
For i = 0 to N-1
   out_msg[i] = block_cipher_encrypt(blk_key, msg[i])
if msg_length % 16 != 0
   // XOR the partial block with the first k bytes of out_msg[N-1]
   // where k is the number of bytes in the partial block
   out_msg[N+] = msg[N+] XOR out_msg[N-1]
   out_msg[N-1] = block_cipher_encrypt(blk_key, out_msg[N-1])
return out_msg
```

5.6. CTS_2ECB_decrypt

Inverse of CTS_2ECB_encrypt.

```
Input
   msg, blk_key
Output
   out_msg

pseudo-code
For i = 0 to N-1
   out_msg[i] = block_cipher_decrypt(blk_key, msg[i])
if msg_length % 16 != 0
   // XOR the partial block with the first k bytes of out_msg[N-1]
   // where k is the number of bytes in the partial block
   out_msg[N+] = msg[N+] XOR out_msg[N-1]
   out_msg[N-1] = block_cipher_decrypt(blk_key, out_msg[N-1])
return out_msg
```

5.7. HEH_encrypt

Core encryption function of HEH.

```
   Input
      prf_key, blk_key, tau_key, nonce, AAD, msg
   Output
      out_msg

   pseudo-code
      beta1_key, beta2_key = generate_betas(prf_key, nonce, AAD,
                                            msg_length)
      out_msg = HEH_hash(msg, beta1_key, tau_key)
      out_msg = CTS_2ECB_encrypt(out_msg, blk_key)
      out_msg = HEH_hash_inv(out_msg, beta2_key, tau_key)
      return out_msg
```

5.8. HEH_decrypt

Core decryption function of HEH.

```
   Input
      prf_key, blk_key, tau_key, nonce, AAD, msg
   Output
      out_msg

   pseudo-code
      beta1_key, beta2_key = generate_betas(prf_key, nonce, AAD,
                                            msg_length)
      out_msg = HEH_hash(msg, beta2_key, tau_key)
      out_msg = CTS_2ECB_decrypt(out_msg, blk_key)
      out_msg = HEH_hash_inv(out_msg, beta1_key, tau_key)
      return out_msg
```

6. HEH as an AEAD

Because HEH is a strong pseudorandom permutation, it can also provide authentication with minimal modification. Support for authentication is OPTIONAL. To provide authentication, append 16 zero bytes to the end of the plaintext, then encrypt. When decrypting, we can verify authenticity of the message by asserting that the final 16 bytes of the plaintext are the expected zero bytes.

6.1. HEH_AEAD_encrypt

Authenticated encryption function of HEH. Returns ciphertext which is 16 bytes longer than plaintext msg.

```
   Input
      prf_key, blk_key, tau_key, nonce, AAD, msg
   Output
      padded_out_msg

   pseudo-code
      // append a full block of zeros
      padded_msg = msg || 0^16
      return HEH_encrypt(prf_key, blk_key, tau_key, nonce, AAD,
                         padded_msg)
```

6.2. HEH_AEAD_decrypt

Authenticated decryption function of HEH. Returns either plaintext which is 16 bytes shorter than msg or indication of inauthenticity FAIL.

```
   Input
      prf_key, blk_key, tau_key, nonce, AAD, msg,
   Output
      unpadded_out_msg or FAIL

   pseudo-code
      out_msg = HEH_DECRYPT(prf_key, blk_key, tau_key, nonce, AAD,
                            msg)

      // If final block is not all zeros, FAIL
      if out_msg[(out_msg_length - 16):out_msg_length] != 0^16
         return FAIL

      // Drop the zero-block that was added in HEH_AEAD_encrypt
      unpadded_out_msg = out_msg[0:(out_msg_length - 16)]
      return unpadded_out_msg
```

7. Security considerations

The minimum length of the plaintext for HEH is 16 bytes. The maximum length is 2^32 - 1 bytes. When using HEH as an AEAD, this minimum and maximum apply to padded_msg.

7.1. Security implementations of nonce use

If no nonce is used (or, equivalently, if a 'nonce' is re-used for multiple messages) then HEH is a strong pseudorandom permutation. In this case the consumer should be aware that if the same plaintext, nonce, and key combination is used more than once it will result in a ciphertext collision.

If a unique nonce is used for each plaintext and key combination, then HEH is semantically secure. We make no claim that using randomly generated nonces or using longer nonces generates additional security.

7.2. Authentication

As HEH is a strong pseudorandom permutation, [AUTH] shows that authentication can be provided by appending a known authentication code to the plaintext, then encrypting the resulting string.

8. References

8.1. Normative References

[CMAC] National Institute of Standards and Technology, "NIST Special Publication 800-38B", 2005.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

[AES-GCM-SIV] Gueronr, S., Langley, A., and Y. Lindell, "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. draft-gueron-gcmsiv-03", 2016.

[AUTH] Bellare, M. and P. Rogaway, "Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography", 2000.

[HEH] Sarkar, P., "Efficient Tweakable Enciphering Schemes from (Block-Wise) Universal Hash Functions", 2008.

[NIST.500-20.1977] National Institute of Standards and Technology, "Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard", NIST 500-20, November 1977.

[TET] Halevi, S., "Invertible Universal Hashing and the TET Encryption Mode", 2007.

Appendix A. Test Vectors

AES-128 was used as the block cipher for all of the test vectors

```
aes_key =    00000000000000000000000000000000
tau_key =    00000000000000000000000000000000
prf_key =    00000000000000000000000000000000
nonce =      EMPTY
AAD =        EMPTY
plaintext =  00000000000000000000000000000000
ciphertext = 310f55672a44bf35b3320895e90d3f30

aes_key =    000102030405060708090A0B0C0D0E0F
tau_key =    000102030405060708090A0B0C0D0E0F
prf_key =    000102030405060708090A0B0C0D0E0F
nonce =      EMPTY
AAD =        EMPTY
plaintext =  00000000000000000000000000000000
             00000000000000000000000000000000
             00000000000000000000000000000000
             00000000000000000000000000000000
ciphertext = 6e20347c7a0609d04cda4fd26ff3b7d0
             3a2e48b13369671c763c24a010d34bd9
             2e2707fce73d89a92ad6f191d9cc38cc
             c9d8e526885730b4835d6d18c3c55d

aes_key =    000102030405060708090A0B0C0D0E0F
tau_key =    000102030405060708090A0B0C0D0E0F
prf_key =    000102030405060708090A0B0C0D0E0F
nonce =      EMPTY
AAD =        EMPTY
plaintext =  00000000000000000000000000000000
             00000000000000000000000000000000
             00000000000000000000000000000001
             00000000000000000000000000000000
ciphertext = 77a09f9af01bf2341c8550734e771abc
             a41398130c7658d83c075492ece8981d
             d5ee21816802cbff60e87fb9ab2cb771
             d44fabfbf59dacdf46931e49d632c1
```

```
   aes_key =    000102030405060708090A0B0C0D0E0F
   tau_key =    000102030405060708090A0B0C0D0E0F
   prf_key =    000102030405060708090A0B0C0D0E0F
   nonce =      000000000000000000000000000000000
   AAD =        EMPTY
   plaintext =  000000000000000000000000000000000
                000000000000000000000000000000000
                000000000000000000000000000000000
                00000000000000000000000000000000
   ciphertext = fb309047c54eccfdc490a29f7c0363c3
                cbaf2eee6218eb206297e49bf28bf33f
                763baaabf01954dbb4af2ed9a7e09204
                5ae481fc58f2dabf5dc9b147d508b1

   aes_key =    000102030405060708090A0B0C0D0E0F
   tau_key =    000102030405060708090A0B0C0D0E0F
   prf_key =    000102030405060708090A0B0C0D0E0F
   nonce =      000000000000000000000000000000000
   AAD =        EMPTY
   plaintext =  000000000000000000000000000000000
                000000000000000000000000000000000
                000000000000000000000000000000001
                00000000000000000000000000000000
   ciphertext = 9cdfa55083e0a3b50d3583346e6e40d6
                0f81c81a9c4081fbb36eb4bffccac950
                cd33fdb34311e632023d3ec6496ecf58
                3e14156d392a589983afdd223e7f6c

   aes_key =    a8da249b5efa13c2c194bf32ba38a377
   tau_key =    68f82787dc3033fd655b8e512e02ff9d
   prf_key =    21281e64cd9c3388f62c438ff56ff58f
   nonce =      4d4761372b4786f0d647b5c2e8cf8527
   AAD =        EMPTY
   plaintext =  b8ee29e4a5d1e755d0fde722637636e2
                f80cf8fe6576e7cac142f5ca5aa8ac2a
                d6a67479105440abdc90b166416ce3cb
                6119FA19AA99F0265850BD29C49E2436
                4d47
   ciphertext = 9726afc277e930f3912c976c779927e0
                a9b80ee83db1881300c3752a54f07c1e
                66f89d556bda0d2dc318536e1a34e6b7
                ab7576469349ea9927cd15429e25d050
                9f9a
```

```
   aes_key =    000102030405060708090A0B0C0D0E0F
   tau_key =    000102030405060708090A0B0C0D0E0F
   prf_key =    000102030405060708090A0B0C0D0E0F
   nonce =      000102030405060708090A0B0C0D0E0F
   AAD =        000102030405060708090A0B0C0D0E0F
   plaintext =  00000000000000000000000000000000
                00000000000000000000000000000000
   ciphertext = 7f5eac36f1fee71cc79e4046c1d11f94
                cd9219968157de2b3c23c139ff671914

   aes_key =    000102030405060708090A0B0C0D0E0F
   tau_key =    000102030405060708090A0B0C0D0E0F
   prf_key =    000102030405060708090A0B0C0D0E0F
   nonce =      000102030405060708090A0B0C0D0E0F
                000102030405
   AAD =        0102030405060708090A0B0C0D0E0F00
                010203
   plaintext =  00000000000000000000000000000000
                00000000000000000000000000000000
   ciphertext = a4f3f950f6f07b892248655a9bc88262
                87f7f81312a2a6408d0ad2bed078202a

   aes_key =    36DAF975AAE45061AF88079422E5E6A9
   tau_key =    D0A8C8E6B3FDC335C4E98C9BBB1310E4
   prf_key =    AA2610D3A619A8F8A222D3DBFB082D17
   nonce =      4164A1FFAEEF4B23324C47279AFB02E8
   AAD =        948F6D03EA0BDE71A0233AC87753F10E
   plaintext =  6A2EDA8E07C10918507F0B5E4F32053C
                335D179A8F476ED1D08A458C00726F63
                6365BF26A7003F43C0270BBB44EC780E
                6119FA19AA99F0265850BD29C49E2436
                A9
   ciphertext = a962d37c10b43303a522aac165230d67
                2cabebfa385d2c7b21468d0af9cab3a7
                5bb5c1c332e1afd77b1b98697672c36b
                bd05ab6b0f47c759f464689831d3ce9e
                93
```

```
   aes_key =    880D8B115BA55842FF4505C5E45F78F6
   tau_key =    F83B77EE7445C4190B326489ECA17CF8
   prf_key =    9F8BF70E528CC1344300AE428506A937
   nonce =      131D6E569B5CCB6E563D2CED8616E6AC
   AAD =        01BD52F7065A35A07EE70D9A881EDDB4
   plaintext =  00000000000000000000000000000000
                B1E0CC8A07264432823C68B2EF59E592
                D271271029F6364CEEE577D9FDA8E5C4
                131D6E569B5CCB6E563D2CED8616E6AC
                C6
   ciphertext = a8da249b5efa13c2c194bf32ba38a377
                21281e64cd9c3388f62c438ff56ff58f
                68f82787dc3033fd655b8e512e02ff9d
                c4fb5c2937d3c85c5cb1196c3b0e99af
                42

   aes_key =    880D8B115BA55842FF4505C5E45F78F6
   tau_key =    F83B77EE7445C4190B326489ECA17CF8
   prf_key =    9F8BF70E528CC1344300AE428506A937
   nonce =      131D6E569B5CCB6E563D2CED8616E6AC
   AAD =        01BD52F7065A35A07EE70D9A881EDDB4
   plaintext =  01000000000000000000000000000000
                B1E0CC8A07264432823C68B2EF59E592
                D271271029F6364CEEE577D9FDA8E5C4
                131D6E569B5CCB6E563D2CED8616E6AC
                C6
   ciphertext = b8ee29e4a5d1e755d0fde722637636e2
                f80cf8fe6576e7cac142f5ca5aa8ac2a
                d6a67479105440abdc90b166416ce3cb
                4d4761372b4786f0d647b5c2e8cf8527
                4b
```

Author's Address


```
   Alex Cope
   Google
   747 6th St S
   Kirkland, WA  98033
   USA

   Email: alexcope@google.com
```