Network Working Group
Request for Comments: 4418
Category: Informational

T. Krovetz, Ed.
CSU Sacramento

March 2006

UMAC: Message Authentication Code using Universal Hashing

Status of This Memo

Copyright Notice

Abstract

This specification describes how to generate an authentication tag using the UMAC message authentication algorithm. UMAC is designed to be very fast to compute in software on contemporary uniprocessors. Measured speeds are as low as one cycle per byte. UMAC relies on addition of 32-bit and 64-bit numbers and multiplication of 32-bit numbers, operations well-supported by contemporary machines.

To generate the authentication tag on a given message, a "universal" hash function is applied to the message and key to produce a short, fixed-length hash value, and this hash value is then xor'ed with a key-derived pseudorandom pad. UMAC enjoys a rigorous security analysis, and its only internal "cryptographic" component is a block cipher used to generate the pseudorandom pads and internal key material.

Table of Contents

1. Introduction

UMAC is a message authentication code (MAC) algorithm designed for high performance. It is backed by a rigorous formal analysis, and there are no intellectual property claims made by any of the authors to any ideas used in its design.

UMAC is a MAC in the style of Wegman and Carter [4, 7]. A fast "universal" hash function is used to hash an input message M into a short string. This short string is then masked by xor'ing with a pseudorandom pad, resulting in the UMAC tag. Security depends on the sender and receiver sharing a randomly-chosen secret hash function and pseudorandom pad. This is achieved by using keyed hash function H and pseudorandom function F. A tag is generated by performing the computation

```
Tag = H_K1(M) xor F_K2(Nonce)
```

where K1 and K2 are secret random keys shared by sender and receiver, and Nonce is a value that changes with each generated tag. The receiver needs to know which nonce was used by the sender, so some method of synchronizing nonces needs to be used. This can be done by explicitly sending the nonce along with the message and tag, or agreeing upon the use of some other non-repeating value such as a sequence number. The nonce need not be kept secret, but care needs to be taken to ensure that, over the lifetime of a UMAC key, a different nonce is used with each message.

UMAC uses a keyed function, called UHASH (also specified in this document), as the keyed hash function H and uses a pseudorandom function F whose default implementation uses the Advanced Encryption Standard (AES) algorithm. UMAC is designed to produce 32-, 64-, 96-, or 128-bit tags, depending on the desired security level. The theory of Wegman-Carter MACs and the analysis of UMAC show that if one "instantiates" UMAC with truly random keys and pads then the probability that an attacker (even a computationally unbounded one) produces a correct tag for any message of its choosing is no more than 1/2^30, 1/2^60, 1/2^90, or 1/2^120 if the tags output by UMAC are of length 32, 64, 96, or 128 bits, respectively (here the symbol ^ represents exponentiation). When an attacker makes N forgery attempts, the probability of getting one or more tags right increases linearly to at most N/2^30, N/2^60, N/2^90, or N/2^120. In a real implementation of UMAC, using AES to produce keys and pads, the forgery probabilities listed above increase by a small amount related to the security of AES. As long as AES is secure, this small additive term is insignificant for any practical attack. See Section 6.2 for more details. Analysis relevant to UMAC security is in [3, 6].

UMAC performs best in environments where 32-bit quantities are efficiently multiplied into 64-bit results. In producing 64-bit tags on an Intel Pentium 4 using SSE2 instructions, which do two of these multiplications in parallel, UMAC processes messages at a peak rate of about one CPU cycle per byte, with the peak being achieved on messages of around four kilobytes and longer. On the Pentium III, without the use of SSE parallelism, UMAC achieves a peak of two cycles per byte. On shorter messages, UMAC still performs well: around four cycles per byte on 256-byte messages and under two cycles per byte on 1500-byte messages. The time to produce a 32-bit tag is a little more than half that needed to produce a 64-bit tag, while 96- and 128-bit tags take one-and-a-half and twice as long, respectively.

Optimized source code, performance data, errata, and papers concerning UMAC can be found at http://www.cs.ucdavis.edu/~rogaway/umac/.

2. Notation and Basic Operations

The specification of UMAC involves the manipulation of both strings and numbers. String variables are denoted with an initial uppercase letter, whereas numeric variables are denoted in all lowercase. The algorithms of UMAC are denoted in all uppercase letters. Simple functions, like those for string-length and string-xor, are written in all lowercase.

Whenever a variable is followed by an underscore ("_"), the underscore is intended to denote a subscript, with the subscripted expression evaluated to resolve the meaning of the variable. For example, if i=2, then M_{2 * i} refers to the variable M_4.

2.1. Operations on strings

Messages to be hashed are viewed as strings of bits that get zero-padded to an appropriate byte length. Once the message is padded, all strings are viewed as strings of bytes. A "byte" is an 8-bit string. The following notation is used to manipulate these strings.

bytelength(S):	The length of string S in bytes.
bitlength(S):	The length of string S in bits.
zeroes(n):	The string made of n zero-bytes.
S xor T:	The string that is the bitwise exclusive-or of S and T. Strings S and T always have the same length.

S and T: The string that is the bitwise conjunction of S and T. Strings S and T always have the same length.

S[i]: The i-th byte of the string S (indices begin at 1).

S[i...j]: The substring of S consisting of bytes i through j.

S || T: The string S concatenated with string T.

zeropad(S,n): The string S, padded with zero-bits to the nearest positive multiple of n bytes. Formally, zeropad(S,n) = S || T, where T is the shortest string of zero-bits (possibly empty) so that S || T is non-empty and 8n divides bitlength(S || T).

2.2. Operations on Integers

Standard notation is used for most mathematical operations, such as "*" for multiplication, "+" for addition and "mod" for modular reduction. Some less standard notations are defined here.

a^i: The integer a raised to the i-th power.

ceil(x): The smallest integer greater than or equal to x.

prime(n): The largest prime number less than 2^n.

The prime numbers used in UMAC are:

n	prime(n) [Decimal]	prime(n) [Hexadecimal]
36	2^36 - 5	0x0000000F FFFFFFFB
64	2^64 - 59	0xFFFFFFFF FFFFFFC5
128	2^128 - 159	0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFF61

2.3. String-Integer Conversion Operations

Conversion between strings and integers is done using the following functions. Each function treats initial bits as more significant than later ones.

bit(S,n): Returns the integer 1 if the n-th bit of the string S is 1, otherwise returns the integer 0 (indices begin at 1).

str2uint(S): The non-negative integer whose binary representation is the string S. More formally, if S is t bits long then str2uint(S) = 2^{t-1} * bit(S,1) + 2^{t-2} * bit(S,2) + ... + 2^{1} * bit(S,t-1) + bit(S,t).

uint2str(n,i): The i-byte string S such that str2uint(S) = n.

2.4. Mathematical Operations on Strings

One of the primary operations in UMAC is repeated application of addition and multiplication on strings. The operations "+_32", "+_64", and "*_64" are defined

"S +_32 T" as uint2str(str2uint(S) + str2uint(T) mod 2^32, 4),
"S +_64 T" as uint2str(str2uint(S) + str2uint(T) mod 2^64, 8), and
"S *_64 T" as uint2str(str2uint(S) * str2uint(T) mod 2^64, 8).

These operations correspond well with the addition and multiplication operations that are performed efficiently by modern computers.

2.5. ENDIAN-SWAP: Adjusting Endian Orientation

Message data is read little-endian to speed tag generation on little-endian computers.

2.5.1. ENDIAN-SWAP Algorithm

Input:
S, string with length divisible by 4 bytes.
Output:
T, string S with each 4-byte word endian-reversed.

Compute T using the following algorithm.

```
//
// Break S into 4-byte chunks
//
```

```
n = bytelength(S) / 4
Let S_1, S_2, ..., S_n be strings of length 4 bytes
   so that S_1 || S_2 || ... || S_n = S.

//
// Byte-reverse each chunk, and build-up T
//
T = <empty string>
for i = 1 to n do
  Let W_1, W_2, W_3, W_4  be bytes
     so that W_1 || W_2 || W_3 || W_4 = S_i
  SReversed_i = W_4 || W_3 || W_2 || W_1
  T = T || SReversed_i
end for

Return T
```

3. Key- and Pad-Derivation Functions

Pseudorandom bits are needed internally by UHASH and at the time of tag generation. The functions listed in this section use a block cipher to generate these bits.

3.1. Block Cipher Choice

UMAC uses the services of a block cipher. The selection of a block cipher defines the following constants and functions.

BLOCKLEN	The length, in bytes, of the plaintext block on which the block cipher operates.
KEYLEN	The block cipher’s key length, in bytes.
ENCIPHER(K,P)	The application of the block cipher on P (a string of BLOCKLEN bytes) using key K (a string of KEYLEN bytes).

As an example, if AES is used with 16-byte keys, then BLOCKLEN would equal 16 (because AES employs 16-byte blocks), KEYLEN would equal 16, and ENCIPHER would refer to the AES function.

Unless specified otherwise, AES with 128-bit keys shall be assumed to be the chosen block cipher for UMAC. Only if explicitly specified otherwise, and agreed to by communicating parties, shall some other block cipher be used. In any case, BLOCKLEN must be at least 16 and a power of two.

AES is defined in another document [1].

3.2. KDF: Key-Derivation Function

The key-derivation function generates pseudorandom bits used to key the hash functions.

3.2.1. KDF Algorithm

```
Input:
  K, string of length KEYLEN bytes.
  index, a non-negative integer less than 2^64.
  numbytes, a non-negative integer less than 2^64.
Output:
  Y, string of length numbytes bytes.
```

Compute Y using the following algorithm.

```
  //
  // Calculate number of block cipher iterations
  //
  n = ceil(numbytes / BLOCKLEN)
  Y = <empty string>

  //
  // Build Y using block cipher in a counter mode
  //
  for i = 1 to n do
    T = uint2str(index, BLOCKLEN-8) || uint2str(i, 8)
    T = ENCIPHER(K, T)
    Y = Y || T
  end for

  Y = Y[1...numbytes]

  Return Y
```

3.3. PDF: Pad-Derivation Function

This function takes a key and a nonce and returns a pseudorandom pad for use in tag generation. A pad of length 4, 8, 12, or 16 bytes can be generated. Notice that pads generated using nonces that differ only in their last bit (when generating 8-byte pads) or last two bits (when generating 4-byte pads) are derived from the same block cipher encryption. This allows caching and sharing a single block cipher invocation for sequential nonces.

3.3.1. PDF Algorithm

```
   Input:
     K, string of length KEYLEN bytes.
     Nonce, string of length 1 to BLOCKLEN bytes.
     taglen, the integer 4, 8, 12 or 16.
   Output:
     Y, string of length taglen bytes.

   Compute Y using the following algorithm.

      //
      // Extract and zero low bit(s) of Nonce if needed
      //
      if (taglen = 4 or taglen = 8)
        index = str2uint(Nonce) mod (BLOCKLEN/taglen)
        Nonce = Nonce xor uint2str(index, bytelength(Nonce))
      end if

      //
      // Make Nonce BLOCKLEN bytes by appending zeroes if needed
      //
      Nonce = Nonce || zeroes(BLOCKLEN - bytelength(Nonce))

      //
      // Generate subkey, encipher and extract indexed substring
      //
      K' = KDF(K, 0, KEYLEN)
      T = ENCIPHER(K', Nonce)
      if (taglen = 4 or taglen = 8)
        Y = T[1 + (index*taglen) ... taglen + (index*taglen)]
      else
        Y = T[1...taglen]
      end if

      Return Y
```

4. UMAC Tag Generation

Tag generation for UMAC proceeds by using UHASH (defined in the next section) to hash the message, applying the PDF to the nonce, and computing the xor of the resulting strings. The length of the pad and hash can be either 4, 8, 12, or 16 bytes.

4.1. UMAC Algorithm

```
Input:
  K, string of length KEYLEN bytes.
  M, string of length less than 2^67 bits.
  Nonce, string of length 1 to BLOCKLEN bytes.
  taglen, the integer 4, 8, 12 or 16.
Output:
  Tag, string of length taglen bytes.
```

Compute Tag using the following algorithm.

```
  HashedMessage = UHASH(K, M, taglen)
  Pad           = PDF(K, Nonce, taglen)
  Tag           = Pad xor HashedMessage

  Return Tag
```

4.2. UMAC-32, UMAC-64, UMAC-96, and UMAC-128

The preceding UMAC definition has a parameter "taglen", which specifies the length of tag generated by the algorithm. The following aliases define names that make tag length explicit in the name.

```
  UMAC-32(K, M, Nonce) = UMAC(K, M, Nonce, 4)
  UMAC-64(K, M, Nonce) = UMAC(K, M, Nonce, 8)
  UMAC-96(K, M, Nonce) = UMAC(K, M, Nonce, 12)
  UMAC-128(K, M, Nonce) = UMAC(K, M, Nonce, 16)
```

5. UHASH: Universal Hash Function

UHASH is a keyed hash function, which takes as input a string of arbitrary length, and produces a 4-, 8-, 12-, or 16-byte output. UHASH does its work in three stages, or layers. A message is first hashed by L1-HASH, its output is then hashed by L2-HASH, whose output is then hashed by L3-HASH. If the message being hashed is no longer than 1024 bytes, then L2-HASH is skipped as an optimization. Because L3-HASH outputs a string whose length is only four bytes long, multiple iterations of this three-layer hash are used if a total hash-output longer than four bytes is requested. To reduce memory

use, L1-HASH reuses most of its key material between iterations. A significant amount of internal key is required for UHASH, but it remains constant so long as UMAC's key is unchanged. It is the implementer's choice whether to generate the internal keys each time a message is hashed, or to cache them between messages.

Please note that UHASH has certain combinatoric properties making it suitable for Wegman-Carter message authentication. UHASH is not a cryptographic hash function and is not a suitable general replacement for functions like SHA-1.

UHASH is presented here in a top-down manner. First, UHASH is described, then each of its component hashes is presented.

5.1. UHASH Algorithm

```
Input:
  K, string of length KEYLEN bytes.
  M, string of length less than 2^67 bits.
  taglen, the integer 4, 8, 12 or 16.
Output:
  Y, string of length taglen bytes.

Compute Y using the following algorithm.

  //
  // One internal iteration per 4 bytes of output
  //
  iters = taglen / 4

  //
  // Define total key needed for all iterations using KDF.
  // L1Key reuses most key material between iterations.
  //
  L1Key  = KDF(K, 1, 1024 + (iters - 1) * 16)
  L2Key  = KDF(K, 2, iters * 24)
  L3Key1 = KDF(K, 3, iters * 64)
  L3Key2 = KDF(K, 4, iters * 4)

  //
  // For each iteration, extract key and do three-layer hash.
  // If bytelength(M) <= 1024, then skip L2-HASH.
  //
  Y = <empty string>
  for i = 1 to iters do
    L1Key_i  = L1Key [(i-1) * 16 + 1 ... (i-1) * 16 + 1024]
    L2Key_i  = L2Key [(i-1) * 24 + 1 ... i * 24]
    L3Key1_i = L3Key1[(i-1) * 64 + 1 ... i * 64]
```

```
      L3Key2_i = L3Key2[(i-1) * 4  + 1 ... i * 4]

      A = L1-HASH(L1Key_i, M)
      if (bitlength(M) <= bitlength(L1Key_i)) then
        B = zeroes(8) || A
      else
        B = L2-HASH(L2Key_i, A)
      end if
      C = L3-HASH(L3Key1_i, L3Key2_i, B)
      Y = Y || C
    end for

    Return Y
```

5.2. L1-HASH: First-Layer Hash

The first-layer hash breaks the message into 1024-byte chunks and hashes each with a function called NH. Concatenating the results forms a string, which is up to 128 times shorter than the original.

5.2.1. L1-HASH Algorithm

```
Input:
  K, string of length 1024 bytes.
  M, string of length less than 2^67 bits.
Output:
  Y, string of length (8 * ceil(bitlength(M)/8192)) bytes.
```

Compute Y using the following algorithm.

```
  //
  // Break M into 1024 byte chunks (final chunk may be shorter)
  //
  t = max(ceil(bitlength(M)/8192), 1)
  Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || ... ||
     M_t, and bytelength(M_i) = 1024 for all 0 < i < t.

  //
  // For each chunk, except the last: endian-adjust, NH hash
  // and add bit-length.  Use results to build Y.
  //
  Len = uint2str(1024 * 8, 8)
  Y = <empty string>
  for i = 1 to t-1 do
    ENDIAN-SWAP(M_i)
    Y = Y || (NH(K, M_i) +_64 Len)
  end for
```

```
    //
    // For the last chunk: pad to 32-byte boundary, endian-adjust,
    // NH hash and add bit-length.  Concatenate the result to Y.
    //
    Len = uint2str(bitlength(M_t), 8)
    M_t = zeropad(M_t, 32)
    ENDIAN-SWAP(M_t)
    Y = Y || (NH(K, M_t) +_64 Len)

    return Y
```

5.2.2. NH Algorithm

Because this routine is applied directly to every bit of input data, optimized implementation of it yields great benefit.

```
Input:
  K, string of length 1024 bytes.
  M, string with length divisible by 32 bytes.
Output:
  Y, string of length 8 bytes.
```

Compute Y using the following algorithm.

```
  //
  // Break M and K into 4-byte chunks
  //
  t = bytelength(M) / 4
  Let M_1, M_2, ..., M_t be 4-byte strings
    so that M = M_1 || M_2 || ... || M_t.
  Let K_1, K_2, ..., K_t be 4-byte strings
    so that K_1 || K_2 || ... || K_t  is a prefix of K.

  //
  // Perform NH hash on the chunks, pairing words for multiplication
  // which are 4 apart to accommodate vector-parallelism.
  //
  Y = zeroes(8)
  i = 1
  while (i < t) do
    Y = Y +_64 ((M_{i+0} +_32 K_{i+0}) *_64 (M_{i+4} +_32 K_{i+4}))
    Y = Y +_64 ((M_{i+1} +_32 K_{i+1}) *_64 (M_{i+5} +_32 K_{i+5}))
    Y = Y +_64 ((M_{i+2} +_32 K_{i+2}) *_64 (M_{i+6} +_32 K_{i+6}))
    Y = Y +_64 ((M_{i+3} +_32 K_{i+3}) *_64 (M_{i+7} +_32 K_{i+7}))
    i = i + 8
  end while

  Return Y
```

5.3. L2-HASH: Second-Layer Hash

The second-layer rehashes the L1-HASH output using a polynomial hash called POLY. If the L1-HASH output is long, then POLY is called once on a prefix of the L1-HASH output and called using different settings on the remainder. (This two-step hashing of the L1-HASH output is needed only if the message length is greater than 16 megabytes.) Careful implementation of POLY is necessary to avoid a possible timing attack (see Section 6.6 for more information).

5.3.1. L2-HASH Algorithm

```
Input:
  K, string of length 24 bytes.
  M, string of length less than 2^64 bytes.
Output:
  Y, string of length 16 bytes.
```

Compute y using the following algorithm.

```
  //
  //  Extract keys and restrict to special key-sets
  //
  Mask64  = uint2str(0x01ffffff01ffffff, 8)
  Mask128 = uint2str(0x01ffffff01ffffff01ffffff01ffffff, 16)
  k64    = str2uint(K[1...8]  and Mask64)
  k128   = str2uint(K[9...24] and Mask128)

  //
  // If M is no more than 2^17 bytes, hash under 64-bit prime,
  // otherwise, hash first 2^17 bytes under 64-bit prime and
  // remainder under 128-bit prime.
  //
  if (bytelength(M) <= 2^17) then             // 2^14 64-bit words

     //
     // View M as an array of 64-bit words, and use POLY modulo
     // prime(64) (and with bound 2^64 - 2^32) to hash it.
     //
     y = POLY(64, 2^64 - 2^32,  k64, M)
  else
     M_1 = M[1...2^17]
     M_2 = M[2^17 + 1 ... bytelength(M)]
     M_2 = zeropad(M_2 || uint2str(0x80,1), 16)
     y = POLY(64, 2^64 - 2^32, k64, M_1)
     y = POLY(128, 2^128 - 2^96, k128, uint2str(y, 16) || M_2)
   end if
```

```
     Y = uint2str(y, 16)

     Return Y
```

5.3.2. POLY Algorithm

```
   Input:
     wordbits, the integer 64 or 128.
     maxwordrange, positive integer less than 2^wordbits.
     k, integer in the range 0 ... prime(wordbits) - 1.
     M, string with length divisible by (wordbits / 8) bytes.
   Output:
     y, integer in the range 0 ... prime(wordbits) - 1.

   Compute y using the following algorithm.

     //
     // Define constants used for fixing out-of-range words
     //
     wordbytes = wordbits / 8
     p = prime(wordbits)
     offset = 2^wordbits - p
     marker = p - 1

     //
     // Break M into chunks of length wordbytes bytes
     //
     n = bytelength(M) / wordbytes
     Let M_1, M_2, ..., M_n be strings of length wordbytes bytes
       so that M = M_1 || M_2 || ... || M_n

     //
     // Each input word m is compared with maxwordrange.  If not smaller
     // then 'marker' and (m - offset), both in range, are hashed.
     //
     y = 1
     for i = 1 to n do
       m = str2uint(M_i)
       if (m >= maxwordrange) then
         y = (k * y + marker) mod p
         y = (k * y + (m - offset)) mod p
       else
         y = (k * y + m) mod p
       end if
     end for

     Return y
```

5.4. L3-HASH: Third-Layer Hash

The output from L2-HASH is 16 bytes long. This final hash function hashes the 16-byte string to a fixed length of 4 bytes.

5.4.1. L3-HASH Algorithm

```
Input:
  K1, string of length 64 bytes.
  K2, string of length 4 bytes.
  M, string of length 16 bytes.
Output:
  Y, string of length 4 bytes.

Compute Y using the following algorithm.

  y = 0

  //
  // Break M and K1 into 8 chunks and convert to integers
  //
  for i = 1 to 8 do
    M_i = M [(i - 1) * 2 + 1 ... i * 2]
    K_i = K1[(i - 1) * 8 + 1 ... i * 8]
    m_i = str2uint(M_i)
    k_i = str2uint(K_i) mod prime(36)
  end for

  //
  // Inner-product hash, extract last 32 bits and affine-translate
  //
  y = (m_1 * k_1 + ... + m_8 * k_8) mod prime(36)
  y = y mod 2^32
  Y = uint2str(y, 4)
  Y = Y xor K2

  Return Y
```

6. Security Considerations

As a message authentication code specification, this entire document is about security. Here we describe some security considerations important for the proper understanding and use of UMAC.

6.1. Resistance to Cryptanalysis

The strength of UMAC depends on the strength of its underlying cryptographic functions: the key-derivation function (KDF) and the pad-derivation function (PDF). In this specification, both operations are implemented using a block cipher, by default the Advanced Encryption Standard (AES). However, the design of UMAC allows for the replacement of these components. Indeed, it is possible to use other block ciphers or other cryptographic objects, such as (properly keyed) SHA-1 or HMAC for the realization of the KDF or PDF.

The core of the UMAC design, the UHASH function, does not depend on cryptographic assumptions: its strength is specified by a purely mathematical property stated in terms of collision probability, and this property is proven unconditionally [3, 6]. This means the strength of UHASH is guaranteed regardless of advances in cryptanalysis.

The analysis of UMAC [3, 6] shows this scheme to have provable security, in the sense of modern cryptography, by way of tight reductions. What this means is that an adversarial attack on UMAC that forges with probability that significantly exceeds the established collision probability of UHASH will give rise to an attack of comparable complexity. This attack will break the block cipher, in the sense of distinguishing the block cipher from a family of random permutations. This design approach essentially obviates the need for cryptanalysis on UMAC: cryptanalytic efforts might as well focus on the block cipher, the results imply.

6.2. Tag Lengths and Forging Probability

A MAC algorithm is used to authenticate messages between two parties that share a secret MAC key K. An authentication tag is computed for a message using K and, in some MAC algorithms such as UMAC, a nonce. Messages transmitted between parties are accompanied by their tag and, possibly, nonce. Breaking the MAC means that the attacker is able to generate, on its own, with no knowledge of the key K, a new message M (i.e., one not previously transmitted between the legitimate parties) and to compute on M a correct authentication tag under the key K. This is called a forgery. Note that if the authentication tag is specified to be of length t, then the attacker

can trivially break the MAC with probability 1/2^t. For this, the attacker can just generate any message of its choice and try a random tag; obviously, the tag is correct with probability 1/2^t. By repeated guesses, the attacker can increase linearly its probability of success.

In the case of UMAC-64, for example, the above guessing-attack strategy is close to optimal. An adversary can correctly guess an 8-byte UMAC tag with probability 1/2^64 by simply guessing a random value. The results of [3, 6] show that no attack strategy can produce a correct tag with probability better than 1/2^60 if UMAC were to use a random function in its work rather than AES. Another result [2], when combined with [3, 6], shows that so long as AES is secure as a pseudorandom permutation, it can be used instead of a random function without significantly increasing the 1/2^60 forging probability, assuming that no more than 2^64 messages are authenticated. Likewise, 32-, 96-, and 128-bit tags cannot be forged with more than 1/2^30, 1/2^90, and 1/2^120 probability plus the probability of a successful attack against AES as a pseudorandom permutation.

AES has undergone extensive study and is assumed to be very secure as a pseudorandom permutation. If we assume that no attacker with feasible computational power can distinguish randomly-keyed AES from a randomly-chosen permutation with probability delta (more precisely, delta is a function of the computational resources of the attacker and of its ability to sample the function), then we obtain that no such attacker can forge UMAC with probability greater than 1/2^30, 1/^60, 1/2^90, or 1/2^120, plus 3*delta. Over N forgery attempts, forgery occurs with probability no more than N/2^30, N/^60, N/2^90, or N/2^120, plus 3*delta. The value delta may exceed 1/2^30, 1/2^60, 1/2^90, or 1/2^120, in which case the probability of UMAC forging is dominated by a term representing the security of AES.

With UMAC, off-line computation aimed at exceeding the forging probability is hopeless as long as the underlying cipher is not broken. An attacker attempting to forge UMAC tags will need to interact with the entity that verifies message tags and try a large number of forgeries before one is likely to succeed. The system architecture will determine the extent to which this is possible. In a well-architected system, there should not be any high-bandwidth capability for presenting forged MACs and determining if they are valid. In particular, the number of authentication failures at the verifying party should be limited. If a large number of such attempts are detected, the session key in use should be dropped and the event be recorded in an audit log.

Let us reemphasize: a forging probability of 1/2^60 does not mean that there is an attack that runs in 2^60 time; to the contrary, as long as the block cipher in use is not broken there is no such attack for UMAC. Instead, a 1/2^60 forging probability means that if an attacker could have N forgery attempts, then the attacker would have no more than N/2^60 probability of getting one or more of them right.

It should be pointed out that once an attempted forgery is successful, it is possible, in principle, that subsequent messages under this key may be easily forged. This is important to understand in gauging the severity of a successful forgery, even though no such attack on UMAC is known to date.

In conclusion, 64-bit tags seem appropriate for many security architectures and commercial applications. If one wants a more conservative option, at a cost of about 50% or 100% more computation, UMAC can produce 96- or 128-bit tags that have basic collision probabilities of at most 1/2^90 and 1/2^120. If one needs less security, with the benefit of about 50% less computation, UMAC can produce 32-bit tags. In this case, under the same assumptions as before, one cannot forge a message with probability better than 1/2^30. Special care must be taken when using 32-bit tags because 1/2^30 forgery probability is considered fairly high. Still, high-speed low-security authentication can be applied usefully on low-value data or rapidly-changing key environments.

6.3. Nonce Considerations

UMAC requires a nonce with length in the range 1 to BLOCKLEN bytes. All nonces in an authentication session must be equal in length. For secure operation, no nonce value should be repeated within the life of a single UMAC session key. There is no guarantee of message authenticity when a nonce is repeated, and so messages accompanied by a repeated nonce should be considered inauthentic.

To authenticate messages over a duplex channel (where two parties send messages to each other), a different key could be used for each direction. If the same key is used in both directions, then it is crucial that all nonces be distinct. For example, one party can use even nonces while the other party uses odd ones. The receiving party must verify that the sender is using a nonce of the correct form.

This specification does not indicate how nonce values are created, updated, or communicated between the entity producing a tag and the entity verifying a tag. The following are possibilities:

1. The nonce is an 8-byte unsigned number, Counter, which is initialized to zero, which is incremented by one following the generation of each authentication tag, and which is always communicated along with the message and the authentication tag. An error occurs at the sender if there is an attempt to authenticate more than 2^64 messages within a session.

2. The nonce is a BLOCKLEN-byte unsigned number, Counter, which is initialized to zero and which is incremented by one following the generation of each authentication tag. The Counter is not explicitly communicated between the sender and receiver. Instead, the two are assumed to communicate over a reliable transport, and each maintains its own counter so as to keep track of what the current nonce value is.

3. The nonce is a BLOCKLEN-byte random value. (Because repetitions in a random n-bit value are expected at around 2^(n/2) trials, the number of messages to be communicated in a session using n-bit nonces should not be allowed to approach 2^(n/2).)

We emphasize that the value of the nonce need not be kept secret.

When UMAC is used within a higher-level protocol, there may already be a field, such as a sequence number, which can be co-opted so as to specify the nonce needed by UMAC [5]. The application will then specify how to construct the nonce from this already-existing field.

6.4. Replay Attacks

A replay attack entails the attacker repeating a message, nonce, and authentication tag. In many applications, replay attacks may be quite damaging and must be prevented. In UMAC, this would normally be done at the receiver by having the receiver check that no nonce value is used twice. On a reliable connection, when the nonce is a counter, this is trivial. On an unreliable connection, when the nonce is a counter, one would normally cache some window of recent nonces. Out-of-order message delivery in excess of what the window allows will result in rejecting otherwise valid authentication tags. We emphasize that it is up to the receiver when a given (message, nonce, tag) triple will be deemed authentic. Certainly, the tag should be valid for the message and nonce, as determined by UMAC, but the message may still be deemed inauthentic because the nonce is detected to be a replay.

6.5. Tag-Prefix Verification

UMAC's definition makes it possible to implement tag-prefix verification; for example, a receiver might verify only the 32-bit prefix of a 64-bit tag if its computational load is high. Or a receiver might reject out-of-hand a 64-bit tag whose 32-bit prefix is incorrect. Such practices are potentially dangerous and can lead to attacks that reduce the security of the session to the length of the verified prefix. A UMAC key (or session) must have an associated and immutable tag length and the implementation should not leak information that would reveal if a given proper prefix of a tag is valid or invalid.

6.6. Side-Channel Attacks

Side-channel attacks have the goal of subverting the security of a cryptographic system by exploiting its implementation characteristics. One common side-channel attack is to measure system response time and derive information regarding conditions met by the data being processed. Such attacks are known as "timing attacks". Discussion of timing and other side-channel attacks is outside of this document's scope. However, we warn that there are places in the UMAC algorithm where timing information could be unintentionally leaked. In particular, the POLY algorithm (Section 5.3.2) tests whether a value m is out of a particular range, and the behavior of the algorithm differs depending on the result. If timing attacks are to be avoided, care should be taken to equalize the computation time in both cases. Timing attacks can also occur for more subtle reasons, including caching effects.

7. Acknowledgements

David McGrew and Scott Fluhrer, of Cisco Systems, played a significant role in improving UMAC by encouraging us to pay more attention to the performance of short messages. Thanks go to Jim Schaad and to those who made helpful suggestions to the CFRG mailing list for improving this document during RFC consideration. Black, Krovetz, and Rogaway have received support for this work under NSF awards 0208842, 0240000, and 9624560, and a gift from Cisco Systems.

Appendix. Test Vectors

Following are some sample UMAC outputs over a collection of input values, using AES with 16-byte keys. Let

```
K  = "abcdefghijklmnop"                  // A 16-byte UMAC key
N  = "bcdefghi"                          // An 8-byte nonce
```

The tags generated by UMAC using key K and nonce N are:

Message	32-bit Tag	64-bit Tag	96-bit Tag
<empty>	113145FB	6E155FAD26900BE1	32FEDB100C79AD58F07FF764
'a' * 3	3B91D102	44B5CB542F220104	185E4FE905CBA7BD85E4C2DC
'a' * 2^10	599B350B	26BF2F5D60118BD9	7A54ABE04AF82D60FB298C3C
'a' * 2^15	58DCF532	27F8EF643B0D118D	7B136BD911E4B734286EF2BE
'a' * 2^20	DB6364D1	A4477E87E9F55853	F8ACFA3AC31CFEEA047F7B11
'a' * 2^25	5109A660	2E2DBC36860A0A5F	72C6388BACE3ACE6FBF062D9
'abc' * 1	ABF3A3A0	D4D7B9F6BD4FBFCF	883C3D4B97A61976FFCF2323
'abc' * 500	ABEB3C8B	D4CF26DDEFD5C01A	8824A260C53C66A36C9260A6

The first column lists a small sample of messages that are strings of repeated ASCII 'a' bytes or 'abc' strings. The remaining columns give in hexadecimal the tags generated when UMAC is called with the corresponding message, nonce N and key K.

When using key K and producing a 64-bit tag, the following relevant keys are generated:

	Iteration 1	Iteration 2
NH (Section 5.2.2)		
K_1	ACD79B4F	C6DFECA2
K_2	6EDA0D0E	964A710D
K_3	1625B603	AD7EDE4D
K_4	84F9FC93	A1D3935E
K_5	C6DFECA2	62EC8672
...		
K_256	0BF0F56C	744C294F
L2-HASH (Section 5.3.1)		
k64	0094B8DD0137BEF8	01036F4D000E7E72
L3-HASH (Section 5.4.1)		
k_5	056533C3A8	0504BF4D4E

k_6	07591E062E	0126E922FF
k_7	0C2D30F89D	030C0399E2
k_8	046786437C	04C1CB8FED
K2	2E79F461	A74C03AA

(Note that k_1 ... k_4 are not listed in this example because they are multiplied by zero in L3-HASH.)

When generating a 64-bit tag on input "'abc' * 500", the following intermediate results are produced:

Iteration 1

L1-HASH	E6096F94EDC45CAC1BEDCD0E7FDAA906
L2-HASH	0000000000000000A6C537D7986FA4AA
L3-HASH	05F86309

Iteration 2

L1-HASH	2665EAD321CFAE79C82F3B90261641E5
L2-HASH	00000000000000001D79EAF247B394BF
L3-HASH	DF9AD858

Concatenating the two L3-HASH results produces a final UHASH result of 05F86309DF9AD858. The pad generated for nonce N is D13745D4304F1842, which when xor'ed with the L3-HASH result yields a tag of D4CF26DDEFD5C01A.

References

Normative References

[1] FIPS-197, "Advanced Encryption Standard (AES)", National Institute of Standards and Technology, 2001.

Informative References

[2] D. Bernstein, "Stronger security bounds for permutations", unpublished manuscript, 2005. This work refines "Stronger security bounds for Wegman-Carter-Shoup authenticators", Advances in Cryptology - EUROCRYPT 2005, LNCS vol. 3494, pp. 164-180, Springer-Verlag, 2005.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and provably secure message authentication", Advances in Cryptology - CRYPTO '99, LNCS vol. 1666, pp. 216-233, Springer-Verlag, 1999.

[4] L. Carter and M. Wegman, "Universal classes of hash functions", Journal of Computer and System Sciences, 18 (1979), pp. 143-154.

[5] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[6] T. Krovetz, "Software-optimized universal hashing and message authentication", UMI Dissertation Services, 2000.

[7] M. Wegman and L. Carter, "New hash functions and their use in authentication and set equality", Journal of Computer and System Sciences, 22 (1981), pp. 265-279.

Authors' Addresses

John Black
Department of Computer Science
University of Colorado
Boulder, CO 80309
USA

EMail: jrblack@cs.colorado.edu

Shai Halevi
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA

EMail: shaih@alum.mit.edu

Alejandro Hevia
Department of Computer Science
University of Chile
Santiago 837-0459
CHILE

EMail: ahevia@dcc.uchile.cl

Hugo Krawczyk
IBM Research
19 Skyline Dr
Hawthorne, NY 10533
USA

EMail: hugo@ee.technion.ac.il

Ted Krovetz (Editor)
Department of Computer Science
California State University
Sacramento, CA 95819
USA

EMail: tdk@acm.org

Phillip Rogaway
Department of Computer Science
University of California
Davis, CA 95616
USA
and
Department of Computer Science
Faculty of Science
Chiang Mai University
Chiang Mai 50200
THAILAND

EMail: rogaway@cs.ucdavis.edu

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).