IPSec Working Group
INTERNET-DRAFT
Expires April 2001

T. Krovetz, Intel
J. Black, UNR
S. Halevi, IBM
A. Hevia, U.C. San Diego
H. Krawczyk, Technion
P. Rogaway, U.C. Davis

October 2000

UMAC: Message Authentication Code using Universal Hashing
<draft-krovetz-umac-00.txt>

Status of this Memo

Abstract

This specification describes how to generate an authentication tag (also called a "MAC") using the UMAC message authentication code. UMAC is designed to be very fast to compute, in software, on contemporary processors. Measured speeds are as low as 1.0 cycles per byte. The heart of UMAC is a universal hash function, UHASH, which relies on addition and multiplication of 16-bit, 32-bit, or 64-bit numbers, operations well-supported by contemporary machines.

To generate the authentication tag on a given message, UHASH 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 "cryptographic" use is a block cipher, AES, to generate the pseudorandom pads and internal key material.

Table of Contents

1 Introduction

This specification describes how to generate an authentication tag (also called a "MAC") using the UMAC message authentication code. Typically the authentication tag will be transmitted along with a message and a nonce to allow the receiver of the message to verify the message's authenticity. Generation and verification of the authentication tag depends on the message, the nonce, and on a secret key (typically, shared by sender and receiver).

UMAC is designed to be very fast to compute, in software, on contemporary processors. The heart of UMAC is a universal hash function, UHASH, which relies on addition and multiplication of 16-bit, 32-bit, and 64-bit numbers. These operations are supported well by contemporary machines.

For many applications, especially ones with short-lived authentication needs, sufficient speed is already obtained by algorithms such as HMAC-SHA1 [2, 9] or the CBC-MAC of a block cipher [1, 8]. But for the most speed-demanding applications, UMAC may be a better choice: An optimized implementation of UMAC can achieve peak performance which is about an order of magnitude faster than what can be achieved with HMAC or CBC-MAC. Moreover, UMAC offers a tradeoff between forging probability and speed (it is possible to trade forging probability for speed). UMAC has been designed so that computing the prefix of a tag can be done faster than computing the entire tag. This feature allows for a receiver to verify the authenticity of a message to various levels of assurance depending on its needs and resources. Finally, UMAC enjoys better analytical security properties than many other constructions.

Closely associated to this specification are the papers [3, 4, 10, 11]. See those papers for descriptions of the ideas which underlie this algorithm, for performance data, and for proofs of the correctness and maximal forging probability of UMAC.

The UMAC algorithms described in the papers [3, 4] are "parameterized". This means that various low-level choices, like the endian convention and the underlying cryptographic primitive, have not been fixed. One must choose values for these parameters before the authentication tag generated by UMAC (for a given message, key, and nonce) becomes fully-defined. In this document we provide two collections of parameter settings, and have named the sets UMAC16 and UMAC32. The parameter sets have been chosen based on experimentation and provide good performance on a wide variety of processors. UMAC16 is designed to excel on processors which provide small-scale SIMD parallelism of the type found in Intel's MMX and Motorola's AltiVec instruction sets, while UMAC32 is designed to do well on processors

with good 32- and 64- bit support. UMAC32 may take advantage of SIMD parallelism in future processors.

UMAC has been designed to allow implementations which accommodate "on-line" authentication. This means that pieces of the message may be presented to UMAC at different times (but in correct order) and an on-line implementation will be able to process the message correctly without the need to buffer more than a few dozen bytes of the message. For simplicity, the algorithms in this specification are presented as if the entire message being authenticated were available at once.

The ideas which underlie UMAC go back to Wegman and Carter [12]. The sender and receiver share a secret key (the MAC key) which determines:

* The key for a "universal hash function". This hash function is "non-cryptographic", in the sense that it does not need to have any cryptographic "hardness" property. Rather, it needs to satisfy some combinatorial property, which can be proven to hold without relying on unproven hardness assumptions. The concept of a universal hash function (family) is due to [5].

* The key for a pseudorandom function. This is where one needs a cryptographic hardness assumption. The pseudorandom function may be obtained (for example) from a block cipher or cryptographic hash function. The concept of a pseudorandom function (family) is due to [6].

To authenticate a message, Msg, one first applies the universal hash function, resulting in a string which is typically much shorter than the original message. The pseudorandom function is applied to a nonce, and the result is used in the manner of a Vernam cipher: the authentication tag is the xor of the output from the hash function and the output from the pseudorandom function. Thus, an authentication tag is generated as

```
     AuthTag = f(Nonce) xor h(Msg).
```

Here f is the pseudorandom function shared between the sender and the receiver, and h is a universal hash function shared by the sender and the receiver. In UMAC, a shared key is used to key the pseudorandom function f, and then f is used for both tag generation and internally to generate all of the bits needed by the universal hash function. For a general discussion of the speed and assurance advantages of this approach see, for example, [3, 7].

The universal hash function that we use is called UHASH. It combines

several software-optimized algorithms into a multi-layered structure. The algorithm is moderately complex. Some of this complexity comes from extensive speed optimizations.

For the pseudorandom function we use the block cipher of the Advanced Encryption Standard (AES). (At the time of this working draft, the AES definition process is still in progress. Here AES refers to the final blok cipher defined by this process.) Any block cipher with the same block-length (128 bits) and key-length (128 bits) could trivially be substituted in place of what we call AES. With slightly more effort one can define UMAC using a pseudorandom function other than a block cipher.

One unusual feature of UMAC is that authentication-tag generation depends on a nonce (in addition to depending on the message and key) It is imperative that the nonce not be reused when generating authentication tags under the same key. Thus the nonce will normally be implemented by a counter, though any other way to achieve a non-repeating value (or almost certainly non-repeating value) is acceptable.

This document specifies the procedure for generating the authentication tag from the message, key and nonce. The exact way in which the message, nonce and authentication tag are transmitted between sender and receiver is not specified here. It is the responsibility of the particular applications using UMAC to specify how the message, nonce and tag are transmitted. For example, an application may choose to send the three values concatenated by some encoding scheme while others may choose not to transmit the nonce at all if it is known to both parties (e.g., when the nonce is a shared state used to detect replay of messages), or to send only part of the bits of the nonce.

Section 8 discusses security considerations that are important for the proper understanding and use of UMAC.

To the authors' knowledge no ideas utilized in UMAC have been or will be patented. To the best of the authors' knowledge, it should be possible to use UMAC immediately, without any intellectual property concerns.

Public-domain reference code for UMAC is available from the UMAC homepage: http://www.cs.ucdavis.edu/~rogaway/umac/ Other information, like timing data and papers, are distributed from the same URL.

1.1 Organization

The rest of this document is organized as follows: In Section 2 parameters of the named parameter sets UMAC16 and UMAC32 are described. In Section 3 we introduce the basic notations used throughout the rest of the document. Section 4 describes the methods used for generating the Vernam pad and the pseudorandom strings needed internally for hashing. In Sections 5 and 6 the universal hash function is described. Finally, in Section 7 we describe how all these components fit together in the UMAC construction. Some readers may prefer to read sections 4-7 backwards, in order to get a top-down description. Section 8 describes some security considerations in the use of UMAC.

2 Named parameter sets: UMAC16 and UMAC32

As described in [3, 4], a concrete instantiation of UMAC requires the setting of many parameters. We have chosen two sets of values for all of these parameters which allow for good performance on a wide variety of processors. For maximum performance we offer UMAC16 which is designed to exploit the vector-parallel instructions on the Intel MMX and Motorola AltiVec instruction sets. For good performance on processors which support 32- and 64-bit quantities well, we offer UMAC32.

2.1 Named parameters

Throughout the algorithms described in this document, we have integrated most of the parameters required for a concrete UMAC instantiation as unnamed numeric constants. However, we have named six parameters and assign them the following values depending on whether one wishes to use UMAC16 or UMAC32.

	UMAC16	UMAC32
WORD-LEN	2	4
UMAC-OUTPUT-LEN	8	8
L1-KEY-LEN	1024	1024
UMAC-KEY-LEN	16	16
ENDIAN-FAVORITE	LITTLE	LITTLE
L1-OPERATIONS-SIGN	SIGNED	UNSIGNED

Here we give a brief explanation of the role each named parameter plays.

WORD-LEN:	Specifies the size in bytes of a "word". UMAC will be significantly faster in execution if the executing machine supports well certain operations on datatypes of this size. Note that WORD-LEN is not necessarily the native wordsize of the target machine (and on some machines a smaller value turns out to be preferable).
UMAC-OUTPUT-LEN:	Specifies the length of the authentication tag generated by UMAC, in bytes.
L1-KEY-LEN:	Specifies the "block length," in bytes, on which the hash-function initially operates. This much storage (and then some) will be needed in the run-time environment for UMAC's internal keys.
UMAC-KEY-LEN:	Specifies the length in bytes of the user-supplied UMAC key.
ENDIAN-FAVORITE:	Specifies which endian-orientation will be followed in the reading of data to be hashed. This need not be equal to the native endianess of any specific machine running UMAC.
L1-OPERATIONS-SIGN:	Specifies whether the strings manipulated in the hash-function are to be initially considered as signed or unsigned integers.

2.2 Alternative instantiations

Although this document only specifies two named parameter sets, the named parameters could be altered to suit specific authentication needs which are not adequately served by either UMAC16 or UMAC32. Below, we list alternatives that are supported by this specification for each of the named parameters.

```
  WORD-LEN           ::= 2 | 4
  UMAC-OUTPUT-LEN    ::= 1 | 2 | ... | 31 | 32
  L1-KEY-LEN         ::= 32 | 64 | 128 | 256 | ... | 2^28
  UMAC-KEY-LEN       ::= 16 | 32
  ENDIAN-FAVORITE    ::= BIG | LITTLE
  L1-OPERATIONS-SIGN ::= SIGNED | UNSIGNED
```

Roughly speaking, doubling UMAC-OUTPUT-LEN approximately doubles execution time and squares (ie. decreases) the probability of MAC

forgery. Setting ENDIAN-FAVORITE to BIG causes UMAC to perform better on big-endian processors rather than little-endian processors. Setting L1-OPERATIONS-SIGN to UNSIGNED slightly increases UMAC security at the expense of complicating implementations on systems which do not support unsigned integers well. This effectively disallows the use of Intel's MMX instructions which only support signed integers. Finally, increasing L1-KEY-LEN tends to speed tag generation on large messages, but requires more memory for processing and could potentially slow the processor by overflowing its cache.

2.3 Naming convention

A concise shorthand may be used to specify an instance of UMAC. The word "UMAC" followed by up to six parameters specifies unambiguously an instance of UMAC. If only a prefix of the six parameters are written, it is implicitly specified that those missing parameters take on default values listed below. The format of the shorthand is "UMAC-w/l/n/k/s/e", and the meaning of the letters (and their defaults) is as follows:

```
    w = WORD-LEN            (4)
    l = UMAC-OUTPUT-LEN     (8)
    n = L1-KEY-LEN          (1024)
    k = UMAC-KEY-LEN        (16)
    s = L1-OPERATIONS-SIGN  (UNSIGNED)
    e = ENDIAN-FAVORITE     (LITTLE)
```

Some examples

```
    UMAC-4/8/1024/16/UNSIGNED/LITTLE  (Same as named set "UMAC32" )
    UMAC-2/8/1024/16/SIGNED/LITTLE    (Same as named set "UMAC16" )
    UMAC-4/12                         ("UMAC32" with 96-bit output)
    UMAC-2/8/4096                     ("UMAC16" with 4K L1-key and)
                                      (unsigned L1-OPERATIONS     )
```

3 Notation and basic operations

The specification of UMAC involves the manipulation of both strings and numbers. String variables are denoted with initial capitals (upper-case), whereas numeric variables are denoted in all lower-case. Global parameters are denoted in all capital letters. Simple functions, like those for string-length and string-xor, are written with all lower-case, while the algorithms of UMAC are named in all upper-case.

Whenever a variable is followed by an underscore ("_"), the

underscore is intended to denote a subscript, with the subscripted expression needing to be evaluated to resolve the meaning of the variable. For example, if i=2, then M_{2 * i} refers to the variable M_4.

We now define some basic operations for manipulating strings and numbers, and for converting between the two.

3.1 Operations on strings

In this specification, we view the messages to be hashed (as well as the keys used for hashing) as strings of bytes. A "byte" is an 8-bit string. The algorithms have been designed so that they are easily extendable to allow arbitrary bit-strings, if necessary. We use the following notation to manipulate these strings.

length(S):	The length of string S in bytes.
zeroes(n):	The string made of n zero-bytes.
S xor T:	The string which is the bitwise exclusive-or of S and T. Strings S and T must have the same length.
S and T:	The string which is the bitwise conjunction of S and T. Strings S and T must 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-bytes to the nearest non-zero multiple of n bytes. Formally, zeropad(S,n) = S || zeroes(i), where i is the smallest nonnegative integer such that S || zeroes(i) is non-empty and n divides length(S)+i.

3.1.1 ENDIAN-SWAP: Adjusting endian orientation

This routine is used to make the data input to UMAC conform to the ENDIAN-FAVORITE global parameter.

3.1.1.1 Discussion

The most time consuming portion of many UMAC computations involves the reading of key and message data from memory. Because big- and little-endian computers will read these bytes differently, specifying a particular endian-orientation for UMAC could have significant performance ramifications. If necessary, the key-bytes can be preprocessed once during key setup to eliminate the need for their reorientation during performance-critical tag generation. But, message data presumably cannot be preprocessed. Any reorientation needed for each message must be done during tag generation, introducing a significant penalty to computers whose native endian-orientation is opposite to that specified for UMAC. Therefore, UMAC defines a parameter, ENDIAN-FAVORITE, which allows UMAC to be specified to favor big- or little-endian memory conventions. If the parameter is set to favor little-endian computers, then we specify the reversal of the bytes of every word in the input message using the following support function. By reversing the data in the specification, an implementation on a little-endian machine would in fact do nothing but read the input data using native-endian word loads. The loads would automatically reverse the bytes within each word, fulfilling the requirements of the specification. Any other endian reorientation needed to comply with the specification requires an insignificant amount of time during each tag calculation.

3.1.1.2 Interface

```
Function Name:
  ENDIAN-SWAP
Input:
  S, string with length divisible by WORD-LEN bytes.
Output:
  T, string S with each word endian-reversed.
```

3.1.1.3 Algorithm

Compute T using the following algorithm.

```
  //
  // Break S into word-size chunks
  //
  n = length(S) / WORD-LEN
  Let S_1, S_2, ..., S_n be strings of length WORD-LEN 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_{WORD-LEN}  be bytes
     so that W_1 || W_2 || ... || W_{WORD-LEN} = S_i
  SReversed_i = W_{WORD-LEN} || W_{WORD-LEN - 1} || ... || W_1
  T = T || SReversed_i

Return T
```

3.2 Operations on integers

In this specification, we generally use standard notation for 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 integer i-th power.
lg a:	The base-2 logarithm of integer a.
floor(x):	The largest integer less than or equal to x.
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:

x	prime(x) [Decimal]	prime(x) [Hexadecimal]
19	2^19 - 1	0x0007FFFF
32	2^32 - 5	0xFFFFFFFB
36	2^36 - 5	0x0000000F FFFFFFFB
64	2^64 - 59	0xFFFFFFFF FFFFFFC5
128	2^128 - 159	0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFF61

3.3 String-Integer conversion operations

We convert between strings and integers 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). Here n must be between 1 and the bit-length of S.
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. If no such string exists then uint2str(n,i) is undefined.
str2sint(S):	The integer whose binary representation in two's-complement is the string S. More formally, if S is t bits long then str2sint(S) = -2^{t-1} * bit(S,1) + 2^{t-2} * bit(S,2) + ... + 2^{1} * bit(S,t-1) + bit(S,t).
sint2str(n,i):	The i-byte string S such that str2sint(S) = n. If no such string exists then sint2str(n,i) is undefined.

3.4 Mathematical operations on strings

One of the primary operations in the universal hashing part of UMAC is repeated application of addition and multiplication on strings. We use "+_n" and "*_n" to denote the string operations which are equivalent to addition and multiplication modulo 2^n, respectively. These operations correspond exactly with the addition and multiplication operations which are performed efficiently on registers by modern computers. So, when n is 16, 32 or 64, these operations can be preformed by computers very quickly.

There are two interpretations of the operators depending on whether the strings are interpreted as signed or unsigned integers. The global parameter L1-OPERATIONS-SIGN determines which interpretation is made.

If strings S and T are interpreted as signed integers (that is, L1-OPERATIONS-SIGN == SIGNED) then

```
  "S *_n T" as uint2str(str2sint(S) * str2sint(T) mod 2^n, n/8), and

  "S +_n T" as uint2str(str2sint(S) + str2sint(T) mod 2^n, n/8).
```

If strings S and T are interpreted as unsigned integers (that is, L1-OPERATIONS-SIGN == UNSIGNED) then we define

```
  "S *_n T" as uint2str(str2uint(S) * str2uint(T) mod 2^n, n/8), and

  "S +_n T" as uint2str(str2uint(S) + str2uint(T) mod 2^n, n/8).
```

In any case, the number n must be divisible by 8. In this document we use S *_16 T, S *_32 T, S *_64 T, S +_32 T and S +_64 T, corresponding to multiplication of 2, 4 and 8 byte numbers, and the addition of 4 and 8 byte numbers.

4 Key and pad derivation functions

UMAC, as described in this document, requires either a 16- or 32-byte key which is used with a key-derivation function (KDF) to produce pseudorandom bits needed within the universal hash function.

4.1 KDF: Key derivation function

Stretching the user-supplied key into pseudorandom bits used internally by UMAC is done with a key-derivation function (KDF). In this section we define a KDF which is efficiently instantiated with a block cipher. The Advanced Encryption Standard (AES) is used in output-feedback mode to produce the required bits. If UMAC-KEY-LEN is 16, then the 128-bit key/128-bit block-length variant of AES is used, and if UMAC-KEY-LEN is 32, then the 256-bit key/128-bit block-length variant is used. The KDF requires an "index" parameter. Using the same key, but different indices, generates different pseudorandom outputs.

4.1.1 Interface

```
Function Name:
  KDF
Input:
  K, string of length UMAC-KEY-LEN bytes  // key to AES
  index, a non-negative integer less than 256.
  numbytes, a positive integer.
Output:
  Y, string of length numbytes bytes.
```

4.1.2 Algorithm

Compute Y using the following algorithm.

```
//
// Calculate number of AES iterations, set indexed starting point
//
n = ceil(numbytes / 16)
T = zeroes(15) || uint2str(index, 1)
Y = <empty string>

//
// Build Y using AES in a feedback mode
//
for i = 1 to n do
  T = AES(K, T)
  Y = Y || T

Y = Y[1..numbytes]

Return Y
```

4.2 PDF: Pad-derivation function

The Wegman-Carter MAC scheme used in UMAC requires the exclusive-or of a pseudorandom string with the output from the universal hash function. The pseudorandom string is obtained by applying a pad-derivation function (PDF) to a nonce which, for security reasons, must change with each authentication-tag computation. Nonces may be any number of bytes from 1 to 16, but all nonces in a single authentication session must be of equal length. In this section we define a PDF which is efficiently instantiated with a block cipher. Again we use AES with either 16- or 32-bytes keys depending on the value of UMAC-KEY-LEN.

4.2.1 Discussion

The PDF output is exclusive-or'd with the result of the universal hash function. AES, however, may provide more or fewer bits per invocation than are needed for this purpose. For example, UMAC-OUTPUT-LEN is normally 8 bytes and AES produces an output of 16 bytes. It would save processing time if half of the AES output bits could be used to generate one tag, and then the second half of the same AES output could be used for the tag of the next message. For this reason, we include an optimization which allows the use of different substrings of the same AES output. This optimization is

effective only when nonces are sequential. We do so by using the low bits of the nonce as an index into the AES output, which is generated using the higher bits of the nonce which are not used for indexing. This speeds message authentication by reducing the average time spent by AES for each authentication. Note that if a counter-variable is used to exploit this optimization, and the variable is stored in memory, then the variable must be treated as big-endian. If UMAC-OUTPUT-LEN is larger than 16, then two AES invocations are required to produce a sufficient number of bits.

4.2.2 Interface

```
Function Name:
  PDF
Input:
  K, string of length UMAC-KEY-LEN bytes   // key for AES
  Nonce, string of length 1 to 16 bytes.
Output:
  Y, string of length UMAC-OUTPUT-LEN bytes.
```

4.2.3 Algorithm

Compute Y using the following algorithm.

```
   //
   // Make Nonce 16 bytes by prepending zeroes
   //
   Nonce = Nonce || zeroes(16 - length(Nonce))

   //
   // If one AES invocation is enough for more than one
   // PDF invocation.
   //
   if (UMAC-OUTPUT-LEN <= 8) then

      //
      // Compute number of index bits needed
      //
      i = floor(16 / UMAC-OUTPUT-LEN)
      numlowbits = floor(lg(i))

      //
      // Extract index bits and zero low bits of Nonce
      //
      nlowbitsnum = str2uint(Nonce) mod 2^numlowbits
      Nonce = Nonce xor uint2str(nlowbitsnum, 16)
```

```
         //
         // Generate subkey, AES and extract indexed substring
         //
         K' = KDF(K, 128, UMAC-KEY-LEN)
         T = AES(K', Nonce)
         Y = T[ nlowbitsnum      * UMAC-OUTPUT-LEN + 1 ..
               (nlowbitsnum + 1) * UMAC-OUTPUT-LEN]

      else

         //
         // Repeated AES calls to build length
         //
         K_1 = KDF(K, 128, UMAC-KEY-LEN)
         K_2 = KDF(K, 129, UMAC-KEY-LEN)
         if (UMAC-OUTPUT-LEN <= 16)
            Y = AES(K_1, Nonce)
         else
            Y = AES(K_1, Nonce) || AES(K_2, Nonce)
         Y = Y[1..UMAC-OUTPUT-LEN]

      Return Y
```

5 UHASH-32: Universal hash function for a 32-bit word size

UHASH is a keyed hash function, which takes as input a string of arbitrary length, and produces as output a string of fixed length (such as 8 bytes). The actual output length depends on the parameter UMAC-OUTPUT-LEN.

UHASH has been shown to be epsilon-ASU ("Almost Strongly Universal"), where epsilon is a small (parameter-dependent) real number. Informally, saying that a keyed hash function is epsilon-ASU means that for any two distinct fixed input strings, the two outputs of the hash function with a random key "look almost like a pair of random strings". The number epsilon measures how non-random the output strings may be. For details, see [3, 4, 11].

UHASH has been designed to be fast by exploiting several architectural features of modern commodity processors. It was specifically designed for use in UMAC. But UHASH is useful beyond that domain, and can be easily adopted for other purposes.

UHASH does its work in three layers. First, a hash function called NH [3] is used to compress input messages into strings which are typically many times smaller than the input message. Second, the compressed message is hashed with an optimized "polynomial hash

function" into a fixed-length 16-byte string. Finally, the 16-byte string is hashed using an "inner-product hash" into a string of length WORD-LEN bytes. These three layers are repeated (with a modified key) until the outputs total UMAC-OUTPUT-LEN bytes.

Note: Because the repetitions of the three-layer scheme are independent (aside from sharing some internal key), it follows that each "word" of the final output can be computed independently. Hence, to compute a prefix of a UMAC tag, one can simply repeat the three-layer scheme fewer times. Thus, computing a prefix of the tag can be done significantly faster than computing the whole tag.

5.1 NH-32: NH hashing with a 32-bit word size

The first of the three hash-layers that UHASH uses is the NH hash function [3]. More than any other part of UHASH, NH is designed to be fast on modern processors, because it is where the bulk of the UHASH work is done. The NH universal hash function hashes an input string M using a key K by considering M and K to be arrays of integers, each WORD-LEN bytes in length, and performing a sequence of arithmetic operations on them. See [3] for definitions, proofs and rationale relating to NH.

The NH-32 algorithm is designed to perform well on processors which support well multiplications of 32-bit operands into 64-bit results. NH-32 is also designed to exploit the recent trend of including instructions for small-scale vector parallelism in uniprocessor CPUs. Intel's Streaming SIMD 2 instruction set is a good example of this trend. It supports an instruction, which multiplies two pairs of 32-bit operands into two 64-bit results, which can be used by UHASH-32 for accelerated hashing. To accommodate this parallelism, NH-32 accesses data-words in pairs which are 4 words (16 bytes) apart.

5.1.1 Interface

```
Function Name:
  NH-32
Input:
  K, string of length L1-KEY-LEN bytes.
  M, string with length divisible by 32 bytes.
Output:
  Y, string of length 8 bytes.
```

5.1.2 Algorithm

Compute Y using the following algorithm.

```
//
// Break M and K into 4-byte chunks
//
t = length(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

Return Y
```

5.2 L1-HASH-32: First-layer hash

To limit the length of key required in the first layer of hashing, L1-HASH-32 breaks the input message into chunks no longer than L1-KEY-LEN and NH hashes each with a key of the same length.

5.2.1 Discussion

The NH hash function requires a key which is just as long as the message being hashed. To limit the amount of key used in the NH hashing layer, we use a key of fixed length (defined by the parameter L1-KEY-LEN), and process the message in chunks of this length (or less). The L1-HASH-32 algorithm takes an input message and breaks it into chunks of L1-KEY-LEN bytes (except the last chuck, which may be shorter and may need to be zero-padded to an appropriate length). Each chunk is hashed with NH-32, and the outputs from all the NH invocations are annotated with some length information and concatenated to produce the final L1-HASH-32 result.

If ENDIAN-FAVORITE is LITTLE, then each word in the input message is required to be endian reversed.

5.2.2 Interface

```
   Function Name:
     L1-HASH-32
   Input:
     K, string of length L1-KEY-LEN bytes.
     M, string of length less than 2^64 bytes.
   Output:
     Y, string of length (8 * ceil(length(M)/L1-KEY-LEN)) bytes.
```

5.2.3 Algorithm

Compute Y using the following algorithm.

```
     //
     // Break M into L1-KEY-LEN byte chunks (final chunk may be shorter)
     //
     t = ceil(length(M) / L1-KEY-LEN)
     Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || .. ||
        M_t, and length(M_i) = L1-KEY-LEN 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(L1-KEY-LEN * 8, 8)
     Y = <empty string>
     for i = 1 to t-1 do
       if (ENDIAN-FAVORITE == LITTLE) then  // See endian discussion
          ENDIAN-SWAP(M_i)                  // in section 3.1.1
       Y = Y || (NH-32(K, M_i) +_64 Len)

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

     return Y
```

5.3 POLY: Polynomial hash

The output from L1-HASH is a string which is shorter than, but still proportional to, that of its input. The POLY hash algorithm takes an arbitrary message and hashes it to a fixed length.

5.3.1 Discussion

Polynomial hashing treats an input message as a sequence of coefficients of a polynomial, and the hash-key is the point at which this polynomial is evaluated. The security guarantee assured by polynomial hashing degrades linearly in the length of the message being hashed. If two messages of n words are hashed, then the probability they collide when hashed by POLY with a prime modulus of p is no more than n / p. For more information on the polynomial hashing schemes used in UMAC see [10].

The parameter 'wordbits' specifies the prime modulus used in the polynomial as well as the granularity (length of words) in which the input message should be broken. Because some strings of length wordbits are greater than prime(wordbits), a mechanism is needed to fix words which are not in the range 0 .. prime(wordbits) - 1. To this end, any word larger than 'maxwordrange' is split into two words guaranteed to be in range, and each is hashed by the polynomial hash.

5.3.2 Interface

```
Function Name:
  POLY
Input:
  wordbits, positive integer divisible by 8.
  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.
```

5.3.3 Algorithm

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

     //
     // For each input word, compare it with maxwordrange.  If larger
     // then hash the words 'marker' and (m - offset), both in range.
     //
     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

     Return y
```

5.4 L2-HASH-32: Second-layer hash

Because L1-HASH may produce quite long strings, and POLY's security guarantee degrades linearly, a scheme is required to allow long strings while ensuring that the collision probability never grows beyond a certain pre-set bound. This is accomplished by dynamically increasing the prime modulus used in the polynomial hashing as the collision probability bound is approached.

5.4.1 Discussion

The probability of two n-word messages hashing to the same result when polynomially hashed with prime modulus p is as much as (n / p). To maintain a limit on the maximum collision probability, a scheme is needed to disallow (n / p) growing too large. The scheme used here hashes a number of words n_1 under modulus p_1 until (n_1 / p_1) reaches a critical point. The result of the hash-so-far is prepended to the remaining message needing to be hashed, and the hashing continues, but under a prime modulus p_2 which is substantially larger than p_1. Hashing continues for n_2 more words until (n_2 /

p_2) also reaches a critical point, at which time a new larger prime p_3 could be used.

Because polynomial hashing under a small prime modulus is often faster than hashing under a large one, this dynamic ramping-up of the polynomial's modulus provides a hash function which is faster on short messages, but still accommodates long ones.

The keys used for polynomial hashing are restricted to particular subsets to allow for faster implementations on 32-bit architectures. The restrictions allow an implementor to disregard some potential arithmetic carries during computation.

For more information see [10].

5.4.2 Interface

```
Function Name:
  L2-HASH-32
Input:
  K, string of length 24 bytes.
  M, string of length less than 2^64 bytes.
Output:
  Y, string of length 16 bytes.
```

5.4.3 Algorithm

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 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 (length(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 .. length(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)

      Y = uint2str(y, 16)

      Return Y
```

5.5 L3-HASH-32: Third-layer hash

The output from L2-HASH-32 is 16 bytes long. This final hash function hashes the 16-byte string to a fixed length of 4 bytes using a simple inner-product hash with affine translation. A 36-bit prime modulus is used to improve security.

5.5.1 Interface

```
   Function Name:
     L3-HASH-32
   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.
```

5.5.2 Algorithm

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)

     //
     // 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
```

5.6 UHASH-32: Three-layer universal hash

The hash functions L1-HASH, L2-HASH and L3-HASH are used together in a straightforward manner. 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 L1-KEY-LEN bytes, then L2-HASH is skipped as an optimization. Because L3-HASH outputs a string whose length is only WORD-LEN bytes long, multiple iterations of this three-layer hash are used, with different keys each time, until UMAC-OUTPUT-LEN have been generated. To reduce memory requirements, L1-HASH and L3-HASH both reuse most of their key-material between iterations.

5.6.1 Interface

```
   Function Name:
     UHASH-32
   Input:
     K, string of length UMAC-KEY-LEN bytes.
     M, string of length less than 2^64 bytes.
   Output:
     Y, string of length UMAC-OUTPUT-LEN bytes.
```

5.6.2 Algorithm

Compute Y using the following algorithm.

```
     //
     // Calculate iterations needed to make UMAC-OUTPUT-LEN bytes
     //
     streams = ceil(UMAC-OUTPUT-LEN / WORD-LEN)

     //
```

```
     // Define total key needed for all iterations using KDF.
     // L1Key and L3Key1 both reuse most key between iterations.
     //
     L1Key  = KDF(K, 0, L1-KEY-LEN + (streams - 1) * 16)
     L2Key  = KDF(K, 1, streams * 24)
     L3Key1 = KDF(K, 2, streams * 64)
     L3Key2 = KDF(K, 3, streams * 4)

     //
     // For each iteration, extract key and three-layer hash.
     // If length(M) <= L1-KEY-LEN, then skip L2-HASH.
     //
     Y = <empty string>
     for i = 1 to streams do
       L1Key_i  = L1Key [(i-1) * 16 + 1 .. (i-1) * 16 + L1-KEY-LEN]
       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-32(L1Key_i, M)
       if (length(M) <= L1-KEY-LEN) then
         B = zeroes(8) || A
       else
         B = L2-HASH-32(L2Key_i, A)
       C = L3-HASH-32(L3Key1_i, L3Key2_i, B)
       Y = Y || C
     Y = Y[1 .. UMAC-OUTPUT-LEN]

     Return Y
```

6 UHASH-16: Universal hash function for a 16-bit word size

See Section 5 (UHASH-32) for general discussion of the UHASH algorithm. Each sub-section of Section 6 will note only differences between UHASH-32 and UHASH-16.

6.1 NH-16: NH hashing with a 16-bit word size

The NH-16 algorithm is designed to exploit the recent trend of including instructions for small-scale vector parallelism in uniprocessor CPUs. Intel's MMX and Mororola's AltiVec instruction sets are good examples of this trend. Both support single-instruction multiply-add instructions on vectors of 16-bit words which can be used by UHASH-16 for accelerated hashing. To accommodate this parallelism, NH-16 accesses data-words in pairs which are 8 words (16 bytes) apart.

6.1.1 Interface

```
   Function Name:
     NH-16
   Input:
     K, string of length L1-KEY-LEN bytes.
     M, string with length divisible by 32 bytes.
   Output:
     Y, string of length 4 bytes.
```

6.1.2 Algorithm

Compute Y using the following algorithm.

```
     //
     // Break M and K into 2-byte chunks
     //
     t = length(M) / 2
     Let M_1, M_2, ..., M_t be 2-byte strings
       so that M = M_1 || M_2 || .. || M_t.
     Let K_1, K_2, ..., K_t be 2-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 8 apart to accommodate vector-parallelism.
     //
     Y = zeroes(4)
     i = 1
     while (i < t) do
       Y = Y +_32 ((M_{i+0} +_16 K_{i+0}) *_32 (M_{i+ 8} +_16 K_{i+ 8}))
       Y = Y +_32 ((M_{i+1} +_16 K_{i+1}) *_32 (M_{i+ 9} +_16 K_{i+ 9}))
       Y = Y +_32 ((M_{i+2} +_16 K_{i+2}) *_32 (M_{i+10} +_16 K_{i+10}))
       Y = Y +_32 ((M_{i+3} +_16 K_{i+3}) *_32 (M_{i+11} +_16 K_{i+11}))
       Y = Y +_32 ((M_{i+4} +_16 K_{i+4}) *_32 (M_{i+12} +_16 K_{i+12}))
       Y = Y +_32 ((M_{i+5} +_16 K_{i+5}) *_32 (M_{i+13} +_16 K_{i+13}))
       Y = Y +_32 ((M_{i+6} +_16 K_{i+6}) *_32 (M_{i+14} +_16 K_{i+14}))
       Y = Y +_32 ((M_{i+7} +_16 K_{i+7}) *_32 (M_{i+15} +_16 K_{i+15}))
       i = i + 16

     Return Y
```

6.2 L1-HASH-16: First-layer hash

To limit the length of key required in the first layer of hashing,
L1-HASH-16 breaks the input message into chunks no longer than

L1-KEY-LEN bytes and NH hashes each with a key of that same length.

6.2.1 Interface

```
Function Name:
  L1-HASH-16
Input:
  K, string of length L1-KEY-LEN bytes.
  M, string of length less than 2^64 bytes.
Output:
  Y, string of length (4 * ceil(length(M)/L1-KEY-LEN)) bytes.
```

6.2.2 Algorithm

Compute Y using the following algorithm.

```
  //
  // Break M into L1-KEY-LEN byte chunks (final chunk may be shorter)
  //
  t = ceil(length(M) / L1-KEY-LEN)
  Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || .. ||
    M_t, and length(M_i) = L1-KEY-LEN 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(L1-KEY-LEN * 8, 4)
  Y = <empty string>
  for i = 1 to t-1 do
    if (ENDIAN-FAVORITE == LITTLE) then  // See endian discussion
       ENDIAN-SWAP(M_i)                  // in section 3.1.1
    Y = Y || (NH-16(K, M_i) +_32 Len)

  //
  // For the last chunk: pad to 32-byte boundary, endian-adjust,
  // NH hash and add bit-length.  Concatenate the result to Y.
  //
  Len = uint2str(length(M_t) * 8, 4)
  M_t = zeropad(M_t, 32)
  if (ENDIAN-FAVORITE == LITTLE) then
     ENDIAN-SWAP(M_t)
  Y = Y || (NH-16(K, M_t) +_32 Len)

  return Y
```

6.3 L2-HASH-16: Second-layer hash

L2-HASH-16 differs from L2-HASH-32 by beginning the ramped hash with a smaller prime modulus. See Section 5.3 for the definition of POLY.

6.3.1 Interface

```
Function Name:
  L2-HASH-16
Input:
  K, string of length 28 bytes.
  M, string of length less than 2^64 bytes.
Output:
  Y, string of length 16 bytes.
```

6.3.2 Algorithm

Compute Y using the following algorithm.

```
  //
  //  Extract keys and restrict to special key-sets
  //
  Mask32  = uint2str(0x1fffffff, 4)
  Mask64  = uint2str(0x01ffffff01ffffff, 8)
  Mask128 = uint2str(0x01ffffff01ffffff01ffffff01ffffff, 16)
  k_32    = str2uint(K[1..4]   and Mask32)
  k64    = str2uint(K[5..12]  and Mask64)
  k128   = str2uint(K[13..28] and Mask128)

  //
  // If M no more than 2^11 bytes, hash under 32-bit prime.
  // Otherwise, hash under increasingly long primes.
  //
  if (length(M) <= 2^11) then         // 2^9 32-bit words

     y = POLY(32, 2^32 - 6, k_32, M)

  else if (length(M) <= 2^33) then    // 2^31 32-bit words

     M_1 = M[1 .. 2^11]
     M_2 = M[2^11 + 1 .. length(M)]
     M_2 = zeropad(M_2 || uint2str(0x80,1), 8)
     y = POLY(32, 2^32 - 6,    k_32, M_1)
     y = POLY(64, 2^64 - 2^32, k64, uint2str(y, 8) || M_2)

  else
```

```
        M_1 = M[1 .. 2^11]
        M_2 = M[2^11 + 1 .. 2^33]
        M_3 = M[2^33 + 1 .. length(M)]
        M_3 = zeropad(M || uint2str(0x80,1), 16)
        y = POLY(32,  2^32 - 6,     k_32,  M_1)
        y = POLY(64,  2^64 - 2^32,  k64,  uint2str(y,  8) || M_2)
        y = POLY(128, 2^128 - 2^96, k128, uint2str(y, 16) || M_3)

     Y = uint2str(y, 16)

     Return Y
```

6.4 L3-HASH-16: Third-layer hash

The L3-HASH-16 algorithm differs from L3-HASH-32 by hashing under a 19-bit prime modulus (instead of a 36-bit prime modulus) and then returning a 2-byte result (instead of a 4-byte result).

6.4.1 Interface

```
   Function Name:
     L3-HASH-16
   Input:
     K1, string of length 32 bytes.
     K2, a string of length 2 bytes.
     M, a string of length 16 bytes.
   Output:
     Y, a string of length 2 bytes.
```

6.4.2 Algorithm

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) * 4 + 1 .. i * 4]
       m_i = str2uint(M_i)
       k_i = str2uint(K_i) mod prime(19)

     //
```

```
      // Inner-product hash, extract last 32 bits and affine-translate
      //
      y = (m_1 * k_1 + ... + m_8 * k_8) mod prime(19)
      y = y mod 2^16
      Y = uint2str(y, 2)
      Y = Y xor K2

      Return Y
```

6.5 UHASH-16: Three-layer universal hash

The algorithm UHASH-16 differs from UHASH-32 only in the size of its keys generated, and in that it refers to the 16-bit variants of the three-layer hash functions.

6.5.1 Interface

```
   Function Name:
     UHASH-16
   Input:
     K, string of length UMAC-KEY-LEN bytes.
     M, string of length less than 2^64 bytes.
   Output:
     Y, string of length UMAC-OUTPUT-LEN bytes.
```

6.5.2 Algorithm

Compute Y using the following algorithm.

```
      //
      // Calculate iterations needed to make UMAC-OUTPUT-LEN bytes
      //
      streams = ceil(UMAC-OUTPUT-LEN / WORD-LEN)

      //
      // Define total key needed for all iterations using KDF.
      // L1Key and L3Key1 both reuse most key between iterations.
      //
      L1Key  = KDF(K, 0, L1-KEY-LEN + (streams - 1) * 16)
      L2Key  = KDF(K, 1, streams * 28)
      L3Key1 = KDF(K, 2, streams * 32)
      L3Key2 = KDF(K, 3, streams * 2)

      //
      // For each iteration, extract key and three-layer hash.
```

```
     // If length(M) <= L1-KEY-LEN, then skip L2-HASH.
     //
     Y = <empty string>
     for i = 1 to streams do
       L1Key_i  = L1Key [(i-1) * 16 + 1 .. (i-1) * 16 + L1-KEY-LEN]
       L2Key_i  = L2Key [(i-1) * 28 + 1 .. i * 28]
       L3Key1_i = L3Key1[(i-1) * 32 + 1 .. i * 32]
       L3Key2_i = L3Key2[(i-1) * 2 + 1  .. i * 2]

       A = L1-HASH-16(L1Key_i, M)
       if (length(M) <= L1-KEY-LEN) then
         B = zeroes(12) || A
       else
         B = L2-HASH-16(L2Key_i, A)
       C = L3-HASH-16(L3Key1_i, L3Key2_i, B)
       Y = Y || C
     Y = Y[1 .. UMAC-OUTPUT-LEN]

     Return Y
```

7 UMAC tag generation

Tag generation for UMAC proceeds as follows. Use UHASH to hash the message and apply the PDF to the nonce to produce a string to xor with the UHASH output. The resulting string is the authentication-tag.

7.1 Interface

```
   Function Name:
     UMAC
   Input:
     K, string of length UMAC-KEY-LEN bytes.
     M, string of length less than 2^64 bytes.
     Nonce, string of length 1 to 16 bytes.
   Output:
     AuthTag, string of length UMAC-OUTPUT-LEN bytes.
```

7.2 Algorithm

Compute AuthTag using the following algorithm.

```
     if (WORD-LEN == 2) then
        HashedMessage = UHASH-16(K, M)
     else
        HashedMessage = UHASH-32(K, M)
```

```
     Pad                = PDF(K, Nonce)
     AuthTag            = Pad xor HashedMessage

     Return AuthTag
```

8 Security considerations

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

8.1 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 it is assumed that both operations are implemented using the Advanced Encryption Standard (AES). However, the full design and specification allow for the replacement of these components. Indeed, it is straightforward to use other block ciphers or other cryptographic objects, such as 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 any "cryptographic assumptions": its strength is specified by a purely mathematical property stated in terms of collision probability, and this property is proven in an absolute sense. In this way, the strength of UHASH is guaranteed regardless of future advances in cryptanalysis.

The analysis of UMAC [3, 4] 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 which forges with probability significantly exceeding the established collision probability will give rise to an attack of comparable complexity which breaks the AES, in the sense of distinguishing AES from a family of random permutations. This design approach essentially obviates the need for cryptanalysis on UMAC: cryptanalytic efforts might as well focus on AES, the results imply.

8.2 Tag lengths and forging probability

A MAC algorithm is used between two parties that share a secret MAC key, K. Messages transmitted between these parties are accompanied by authentication tags computed using K and a given nonce. Breaking

the MAC means that the attacker is able to generate, on its own, 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.

UMAC is designed to make this guessing strategy the best possible attack against UMAC as long as the attacker does not invest the computational effort needed to break the underlying cipher, e.g. AES, used to produce the one time pads used in UMAC computation. More precisely, under the assumed strength of this cipher UMAC provides for close-to-optimal security with regards to forgery probability as represented in the next table.

UHASH-OUTPUT-LEN (bytes)	Forging probability using a random tag	Approximate actual forging probability in UMAC (using a clever tag)
2	2^-16	2^-15
4	2^-32	2^-30
8	2^-64	2^-60
16	2^-128	2^-120

Recall that the parameter UHASH-OUTPUT-LEN specifies the length of the UMAC authentication tag. The above table states, for example, for the case of an 8-byte tag that the ideal forging probability would be 2^-64 while UMAC would withstand an actual forging probability of 2^-60. Note that under this tag length (which is the default length in UMAC) the probability of forging a message is well under the chance that a randomly guessed DES key is correct. DES is now widely seen as vulnerable, but the problem has never been that some extraordinarily lucky attacker might, in a single guess, find the right key. Instead, the problem is that large amounts of computation can be thrown at the problem until, off-line, the attacker finds the right key.

With UMAC, off-line computation aimed at exceeding the forging probability is hopeless, regardless of tag length, as long as the underlying cipher is not broken. The only way to forge is to interact with the entity that verifies the MAC and to try a huge amount 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 - as long as AES maintains its believed security there is no such attack for UMAC. Instead, a 1 / 2^60 forging probability means that if an attacker could try out 2^60 MACs, then the attacker would probably get one right. But the architecture of any real system should render this infeasible. One can certainly imagine an attacker having a high bandwidth channel (e.g., 1 Gbit/second or more) over which it can continually present attempted forgeries, the attacker being signaled when a correct tag is found, but realizing such a scenario in a real system would represent a major lapse in the security architecture.

It should be pointed out that once an attempted forgery is successful, it is entirely possible that all subsequent messages under this key may be forged, too. This is important to understanding in gauging the severity of a successful forgery.

In conclusion, the default 64-bit tags seem appropriate for most security architectures and applications. In cases where when the consequences of an authentication failure are not extremely severe, and when the system architecture is designed to conscientiously limit the number of forgery attempts before a session is torn down, 32-bit authentication tags may be adequate. For the paranoid, or if an attacker is allowed a fantastic number of forgery tests, 96- or 128-bits may be utilized.

8.3 Selective-assurance authentication

We have already remarked about the flexibility built into UMAC to use authentication tags of various lengths: shorter tags are faster to compute and one needs to transmit fewer bits, but the forging probability is higher. There is an additional degree of flexibility built into the design of UMAC: even if the sender generates and transmits a tag of 8 bytes, say, a receiver may elect to verify only the first 4 bytes of the tag, and computing that 4-byte prefix by the receiver will be substantially faster than computing what the full 8-byte tag would be. Indeed when WORD-LEN is 2 one can more quickly check the 2-byte prefix of the tag than the 4-byte prefix of the tag, one can more quickly check the 4-byte prefix of the tag than the

6-byte prefix of the tag, and so forth. When WORD-LEN is 4 one can more quickly check the 4-byte prefix of the tag than an entire 8-byte tag, and so forth. This type of flexibility allows different parties who receive a MAC (as in a multicast setting) to authenticate the transmission to the extent deemed necessary and to the extent consistent with any computational limits of the receiver.

In a scenario where receivers are allowed to verify short prefixes of longer tags, it is even more important that conservative policies are followed when a bad tag is presented to the receiver. Because short prefixes are easier to forge than are long ones, an attacker may attempt to forge short prefixes and then leverage information gained from these attacks to forge longer tags. If the attacker can learn which short tags are good and which are bad, the attacker may be able to learn enough to allow longer forgeries.

One salient feature of the security-performance trade-off offered by UMAC is its usability in contexts where performance is severely constrained. In such cases, using a mild-security authentication tag can be of significant value especially if the alternative would be not to use authentication at all (a possible such scenario could be the high-speed transmission of real-time multimedia streams). Another potential scenario where short and fast-to-compute tags can be useful is for fast detection of data forgery intended as a denial of service attack. In this case, even a moderate increase in the attacker's difficulty to produce forgeries may suffice to make the attack worthless for the attacker. Moreover, being able to detect just a portion of attempted forgeries may be enough to identify the attack.

8.4 Nonce considerations

The function UMAC (Section 7) requires a nonce with length in the range 1 to 16 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.

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 exemplify some of the

possibilities:

1. The nonce is an eight-byte [resp., four-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 [resp., 2^32] messages within a session.

2. The nonce is a 16-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 16-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. The application will then specify how to construct the nonce from this already-existing field.

Note that if the nonce starts at zero and is incremented with each message then an attacker can easily ascertain the number of messages which have been sent during a session. If this is information which one wishes to deny the attacker then one might have the sender initialize the nonce to a random value, rather than to zero. Inspecting the current nonce will no longer reveal to the attacker the number of messages which have been sent during this session. This is a computationally cheaper approach than enciphering the nonce.

8.5 Guarding against replay attacks

A replay attack entails the attacker repeating a message, nonce, and authentication tag. In systems, replay attacks may be quite damaging, and many applications will want to guard against them. 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 and tag 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.

9 Acknowledgments

Thanks are due to David Balenson and David Carman of NAI Labs, who suggested the advantages of allowing a receiver to verify authentication tags to various forgery probabilities. Thanks are also due to David McGrew and Scott Fluhrer of Cisco Systems for encouraging us to improve UMAC performance on short messages.

Phillip Rogaway, John Black, and Ted Krovetz were supported in this work under Rogaway's NSF CAREER Award CCR-962540, and under MICRO grants 97-150, 98-129, and 99-103 funded by RSA Data Security, Inc., and ORINCON Corporation. Much of Rogaway's work was carried out during two sabbatical visits to Chiang Mai University. Special thanks to Prof. Darunee Smawatakul for helping to arrange these visits.

10 References

[1] ANSI X9.9, "American National Standard for Financial Institution Message Authentication (Wholesale)", American Bankers Association, 1981. Revised 1986.

[2] M. Bellare, R. Canetti, and H. Krawczyk, "Keyed hash functions and message authentication", Advances in Cryptology - CRYPTO '96, LNCS vol. 1109, pp. 1-15. Full version available from http://www.research.ibm.com/security/keyed-md5.html/

[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. Full version available from http://www.cs.ucdavis.edu/~rogaway/umac

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "The UMAC message authentication code", work in progress, 2000.

To be available from http://www.cs.ucdavis.edu/~rogaway/umac

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

[6] O. Goldreich, S. Goldwasser and S. Micali, "How to construct random functions", Journal of the ACM, 33, No. 4 (1986), pp. 210-217.

[7] S. Halevi and H. Krawczyk, "MMH: Software message authentication in the Gbit/second rates", Fast Software Encryption, LNCS Vol. 1267, Springer-Verlag, pp. 172-189, 1997.

[8] ISO/IEC 9797-1, "Information technology - Security techniques - Data integrity mechanism using a cryptographic check function employing a block cipher algorithm", International Organization for Standardization, 1999.

[9] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication", RFC-2104, February 1997.

[10] T. Krovetz, and P. Rogaway, "Fast universal hashing with small keys and no preprocessing", work in progress, 2000. To be available from http://www.cs.ucdavis.edu/~rogaway/umac

[11] T. Krovetz, and P. Rogaway, "Variationally universal hashing", work in progress, 2000. To be available from http://www.cs.ucdavis.edu/~rogaway/umac

[12] 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.

11 Author contact information

Authors' Addresses

John Black
Department of Computer Science
University of Nevada
Reno NV 89557
USA

EMail: jrb@cs.unr.edu

Shai Halevi

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

EMail: shaih@watson.ibm.com

Alejandro Hevia
Department of Computer Science & Engineering
University of California at San Diego
La Jolla CA 92093
USA

EMail: ahevia@cs.ucsd.edu

Hugo Krawczyk
Deprtment of Electrical Engineering
Technion
Haifa 32000
ISRAEL

EMail: hugo@ee.technion.ac.il

Ted Krovetz
Intel Corporation
1900 Prairie City Road
Folsom CA 95630
USA

EMail: tdk@acm.org

Phillip Rogaway
Department of Computer Science
University of California
Davis CA 95616
USA

EMail: rogaway@cs.ucdavis.edu

A Suggested application programming interface (API)

```
/* umac.h */

typedef struct UMAC_CTX *umac_ctx_t;

umac_ctx_t umac_alloc(char key[]);
  /* Dynamically allocate UMAC_CTX struct */
```

```
       /* initialize variables and generate   */
       /* subkeys for default parameters.     */

     int umac_free(umac_ctx_t ctx);
       /* Deallocate the context structure.   */

     int umac_set_params(umac_ctx_t ctx, void *params);
       /* After default initialization,      */
       /* optionally set parameters to       */
       /* different values and reset for     */
       /* new message.                       */

     int umac_update(umac_ctx_t ctx, char *input, long len);
       /* Incorporate len bytes pointed to by  */
       /* input into context ctx.             */

     int umac_final(umac_ctx_t ctx, char tag[], char nonce[]);
       /* Incorporate nonce value and return  */
       /* tag.  Reset ctx for next message.   */

     int umac(umac_ctx_t ctx, char *input,  long len,
              char tag[], char nonce[]);
       /* All-in-one (non-incremental)        */
       /* implementation of the functions     */
       /* umac_update() and umac_final().     */
```

Each routine returns zero if unsuccessful.

B Reference code and test vectors

See the UMAC World Wide Web homepage for reference code and test vectors.

http://www.cs.ucdavis.edu/~rogaway/umac/