Message Authentication on 64-bit Architectures

Ted Krovetz
Department of Computer Science
California State University
Sacramento CA 95819 USA
tdk@acm.org

Abstract

This paper takes UMAC — a message authentication algorithm (MAC) optimized for performance on 32-bit architectures — as its starting point, and adapts its strategies for optimum performance on 64-bit architectures. The resulting MAC, called UMAC8, achieves per message forgery probabilities of about 2^{-60} and 2^{-120} for tags of length 64 and 128 bits. The UMAC strategies are discussed at length and adapted for 64-bit environments, but are also modified to address several UMAC shortcomings, particularly key-agility and susceptibility to timing attacks. UMAC achieved peak throughput rates, when generating 64-bit tags, of 1.0 CPU cycle per byte of message authenticated, while UMAC8 achieves 0.5 cycles per byte.

I personally believe there are two main architectures out there: Power and x86-64.
— Linus Torvalds, 2005.

1 Introduction

Over the years, as design and manufacturing techniques have improved, and demand for memory addressability has increased, register lengths have become longer. The recent adoption of 64-bit register architectures for mainstream processors from IBM, Intel and Advanced Micro Devices is a natural evolution in this process. It is reasonable to believe that, just as 32-bit processors did before them, 64-bit processors will become dominant not only in servers but also in desktops and laptops.

Many algorithms are designed with optimizations tailored for particular architectures. This is especially true in domains, such as cryptography, where high performance is desirable. Changing architectures while keeping the same designs can easily lead to suboptimal performance. This is the case with high-speed message authentication and the move from 32-bit to 64-bit architectures. The fastest reported message authentication schemes (or MACs) are all designed to run well on 32-bit architectures [3, 4, 7]. While these MACs generally work equally well on both 32- and 64-bit architectures — because the newer architectures support older instructions at full speed — they are not designed to take advantage of new capabilities found in the 64-bit processors.

This paper takes as a starting point the fastest reported MAC, called UMAC [3]. UMAC is designed specifically for optimal performance on 32-bit architectures where it can authenticate long cache-resident messages on the Pentium 4 at a rate of 2 CPU cycles per byte (cpb) of message without the use of special-purpose hardware.[1] The strategies used by UMAC to gain high-performance on 32-bit architectures are

[1]A faster version operates at about 1.0 cpb using Intel's SSE architecture, but for maximum applicability, this paper will only give results of implementations that use instructions widely available on multiple architectures.

analyzed for applicability on 64-bit architectures. A modified version of UMAC, called UMAC8, achieving peak performance of well 0.5 cpb on the Athlon64 is then described.

Although the main goal of this design effort is high speed on 64-bit processors, some effort if afforded other properties. UMAC has been criticized for attaining its high speed at the expense of too much (about 1KB) internal key material allocated per authentication session. This is not a problem for a single peer-to-peer connection, but a busy server might see thrashing in its cache if it is handling many sessions at once. UMAC8 reduces the reliance on key material stored in memory to 160 bytes per session. It is also known that UMAC may be susceptible to a timing attack focussed on some data-dependent computation in its use of polynomial hashing if the polynomial hash is implemented without care. UMAC8 changes the polynomial hashing strategy to avoid such a possibility. UMAC8 retains many desirable properties of UMAC by being provably secure, highly parallelizable, and both patent- and copyright-free.

1.1 Message Authentication and Universal Hashing

Message authentication is used when two parties wish to communicate and have some assurance that each received message comes from the purported sender and has not been altered along the way. All of the fastest MACs follow principles introduced by Wegman and Carter [1, 3, 4, 5, 7, 11]. The basic Wegman-Carter message authentication paradigm is for the sender first to hash the message with a hash function known only to himself and the receiver. The sender then applies some cryptographic function (usually encryption) to the resulting hash value, which produces a message tag that is sent along with the message to the receiver. The receiver can then repeat the process, verifying that the received tag is valid for the received message. In a correctly designed MAC, only those knowing the secret hash function and cryptographic keys have a reasonable chance of creating a valid tag for any new message. If, however, an adversary is able to produce a valid tag for a new message without knowing the hash function and cryptographic keys, then a forgery has occurred. Due to their ephemeral nature, communication sessions need only be secure against forgery during the lifetime of the session. If an adversary cannot forge with a probability of more than $1/2^{60}$, then the MAC is likely suitable for most communications where attackers are not allowed an excessive number of forgery attempts.

The key to speed in a Wegman-Carter MAC is the hash function used. Because authentication speeds are determined by the sum of the (length-dependent) time it takes to hash the message being authenticated plus the (constant) time it takes to cryptographically produce the tag, this paper focuses on the hash functions used in UMAC and UMAC8, known respectively as UHASH and UHASH8. This is reasonable because any speed improvements in the cryptographic part of a Wegman-Carter MAC could be applied equally to all such schemes, so improvements relative to other Wegman-Carter MACs will come almost entirely from improvements in hashing.

Notable recent examples of fast hash functions (and their peak speeds) suitable for Wegman-Carter message authentication are hash127 (around 4.4 cpb), hash1305 (3.4 cpb), Badger (2.2 cpb) and UMAC (2.0 cpb). The speeds of each of these favorably compare with popular established non-Wegman-Carter MACs such as HMAC-SHA1 and CBC-AES-MAC, both of which require considerably more than ten cpb.

UNIVERSAL HASHING. The hash function used in a Wegman-Carter MAC must be chosen from a *universal* hash function family. A hash-function family H is a collection of hash functions, each $h \in H$ having some common domain A and some common, finite, codomain B. A hash-function family H is *ε-almost universal* (ε-AU) if the probability is no more than ε that any two distinct inputs m, m' hash to the same output when hashed by a randomly selected member of H. A small value for ε indicates that an adversary is unlikely to be able to choose a pair of inputs that hash to the same output, as long as the hash function is chosen randomly. Although we are often interested in bounding collision probabilities, a stronger notion is useful in cryptography because it bounds an adversary's ability to produce hash outputs that differ by a known constant. A hash-function family H is *ε-almost delta universal* (ε-AΔU) if the probability is no more than

ε that any two distinct inputs m, m' differ by any chosen constant d when hashed by a randomly selected member of H. There are stronger notions of universal hashing defined by Wegman, Carter and Stinson [5, 10, 11], but ε-AΔU is adequate for message authentication, and is achieved by UHASH8.

2 UHASH8 Building Blocks

UHASH, the universal hash function on which UHASH8 is modeled, is in essence a composition of three separate hash functions, each with a particular purpose. A high-level overview of UHASH is given here, and more details can be found in the original UMAC papers [3, 9].

NH. First, the message to be hashed by UHASH is broken into 1KB blocks and each is hashed by NH into 64-bit strings, which are then concatenated. This hash phase reduces long messages into strings up to 128 times smaller. In UHASH, NH is tailored for 32-bit architectures and is extremely fast on systems supporting 32-bit multiplication well. Due to the output length's dependence on the input length, this stage's output is not suitable for message authentication without further processing.

Polynomial. The string produced by the NH phase is broken into 64-bit blocks, each of which is interpreted as a coefficient in a polynomial modulo a 64- or 128-bit prime (depending on the original message length). The output of this hash stage is the result of evaluating the polynomial for a particular value. At this point, the original UHASH input has been hashed to a string of no more than 128 bits. This function is several times slower than NH, so NH acts as an accelerator by reducing the input length to this stage.

Inner-Product. The 64- or 128-bit output of the polynomial hash is broken into 16-bit blocks which are then hashed by an inner-product hash, producing 32 bits. The sole purpose of this step is to distill the universality guarantee of the previous stage's output into a smaller number of bits.

The end result of these three steps is a 32-bit output from a composed hash function that is roughly 2^{-30}-AΔU when each stage's key material is chosen randomly. To form a hash that is 2^{-60}-AΔU, UHASH does these steps twice on each message, each time under a different key, concatenating the resulting hashes into a 64-bit result.

We now look more closely at these three stages of hashing in UHASH and adapt their strategies for high-speed to 64-bit environments.

2.1 NH

NH is a parameterized hash function. Given positive integer parameters n and w and a key K of length nw bits, then NH can hash any string M that is a multiple of $2w$ bits in length but not longer than nw bits. First M and K are broken into w-bit blocks $M_1, M_2, \ldots, M_l$ and $K_1, K_2, \ldots K_n$ where $l = |M|/w$. Then, each block is interpreted as a w-bit unsigned binary integer $m_1, m_2, \ldots, m_l$ and $k_1, k_2, \ldots k_n$. Finally, the hash result is computed as

$$\mathrm{NH}[n,w](K,M) = \sum_{i=1}^{l/2} ((m_{2i-1} + k_{2i-1} \mathbf{\ mod\ } 2^w) \times (m_{2i} + k_{2i} \mathbf{\ mod\ } 2^w)) \mathbf{\ mod\ } 2^{2w}.$$

NH is a hash family, and choosing a random function from the hash family is done by choosing a random nw-bit key K. NH is known to be (2^{-w})-AΔU over messages of the same length (ie, M and M' are distinct, but $|M| = |M'|$), and small modifications to the original proof show that NH is (2^{-w})-AΔU over messages that are a multiple of $2w$ bits in length (but still no longer than nw bits) [3].

CHARACTERISTICS. The chief advantage of NH is extreme speed. Because all key and message blocks are defined as w-bit quantities and all arithmetic is done modulo 2^w and 2^{2w}, every operation is done naturally and efficiently on contemporary processors if w is chosen appropriately. On 32-bit processors with good support for multiplying 32-bit quantities into a 64-bit result, defining $w = 32$ results in high speed. The same is true for $w = 64$ on 64-bit processors if there is good support for multiplying 64-bit quantities into a 128-bit result.

On a 64-bit architecture, NH performance when $w = 64$ is about four times better than when $w = 32$. If one's goal is a (2^{-64})-AΔU guarantee over messages of length $128j$ bits, then NH$[n,w]$ achieves this goal using j multiplications when $w = 64$, but requires $4j$ multiplications when $w = 32$. To see this, consider how one would achieve a (2^{-64})-AΔU guarantee when $w = 32$. Each NH hashing of the message would require $2j$ multiplications and produce a hash value with a (2^{-32})-AΔU guarantee. This would have to be done twice, under separate keys, to achieve the (2^{-64})-AΔU goal, whereas only j 64-bit multiplications are needed to achieve the same guarantee on a 64-bit architecture. This is borne out experimentally. Two parallel passes on a Pentium 4 with $w = 32$ takes about 2 cpb, while a single pass on an AMD Athlon64 with $w = 64$ requires only around 0.5 cpb. Admittedly this comparison is based on different platforms, but they are similar enough to be suggestive of the magnitude of speedup one might expect. Direct Athlon64-based comparisons are given later in Section 3.6.

If UHASH8 were identical to UHASH except for the use of $w = 64$ in NH, the result would be a hash function that runs nearly four times faster for long cache-resident messages on 64-bit architectures. This would be a notable result, but another goal of the UHASH8 design is to reduce the amount of required key from 1KB in NH to something closer to 128 bytes in UHASH8. Reducing the key-length used by NH (ie, lowering n), however, increases hashing overhead and the length of inputs passed on to the polynomial hashing phase, and so increases the time to hash the same length data. A balance must be struck between NH hashing speed and key-length.

2.2 Polynomial Hashing

A simple and efficient method to hash a binary string M is to fix prime number p and break M into fixed-length blocks $M_1, M_2, M_3, \ldots, M_l$ in such a way that when the blocks are interpreted as unsigned integers $m_1, m_2, m_3, \ldots, m_l$, each is less than p (for example, by making each block $\lfloor \log_2 p \rfloor$ bits). Then, choosing an integer key $0 \le k < p$ defines the hash output as

$$h_k(M) = m_1 k^l + m_2 k^{l-1} + \cdots + m_l k^1 \bmod p\,.$$

Two different messages M, M' of the same block length l differ by constant d when hashed by this function if

$$h_k(M) - h_k(M') = (m_1 - m'_1)k^l + (m_2 - m'_2)k^{l-1} + \cdots + (m_l - m'_l)k^1 \bmod p = d\,.$$

Because $M \neq M'$, at least one of the terms in this polynomial is non-zero. This being a polynomial of degree at most l, there are at most l values for k which cause $h_k(M) - h_k(M') - d \pmod p$ to evaluate to zero. If we define this hash family as $H = \{h_k \,|\, 0 \le k < p\}$, then H is an ε-AΔU hash family for $\varepsilon = l/p$.

CHARACTERISTICS. With care, polynomial hashing can be made to perform well. Horner's Rule suggests rephrasing $h_k(M)$ as $((\cdots((m_1 k + m_2)k + m_3)k \cdots)k + m_l)k \bmod p$, which allows $h_k(M)$ to be computed as a sequence of l multiplications and additions modulo p [8]. Those multiplications and additions modulo p can be made efficient by choosing a convenient p and restricting the choice of k to a convenient set.

By choosing p to be of the form $p = 2^a - b$ for some small b, reductions modulo p can be done efficiently in a lazy manner. Each time a value c becomes at least 2^a, it can be rewritten as the (modulo p) equivalent $c - 2^a + b$. For example $p = 2^{61} - 1$ is prime. This means that, in a 64-bit register, a value c greater than p but less than 2^{64} can be reduced by computing $c = (c \textbf{ div } 2^{61}) + (c \textbf{ mod } 2^{61})$. This equality simply recognizes that $c = x2^{61} + y$ for some x and $0 \le y < 2^{61}$, and replaces 2^{61} with the equivalent (modulo p) value 1. The **div** and **mod** operations extract x and y, and can be computed efficiently using bitwise operations. This process is "lazy" for two reasons. First, numbers can be allowed to get as large as desired before performing a reduction as long as values do not exceed the register's capacity. Second, a reduction to the range $0,\ldots,p-1$ is not necessary until a final result is needed. So, when this method is followed to perform an intermediate reduction, the result need not be in the range $0,\ldots,p-1$. This puts off expensive range checks until the very end of the polynomial hash. Particularly useful primes on a 64-bit architecture are $2^{127}-1$ and $2^{61}-1$.

Another source of inefficiencies is register carries during addition. Whenever a number is too large to be represented in a single CPU register, the number is generally split into multiple registers, and arithmetic on the larger number is accomplished by some sequence of smaller operations. For example, if we rewrite 128-bit values j and k as $j = w2^{64} + x$ and $k = y2^{64} + z$ where $0 \le x, z < 2^{64}$, then $jk = wy2^{128} + (wz + xy)2^{64} + xz$. This means that to compute jk, we can put the top 64-bits of j and k into 64-bit registers w and y, and their low 64-bits into x and z. The result jk is then assembled by appropriately multiplying, shifting and adding wy, wz, xy and xz.

Consider the case where a polynomial is being evaluated modulo prime $p = 2^{127}-1$ using Horner's Rule and lazy modulo reduction whenever an intermediate value exceeds 128-bits. Each step in the Horner's Rule evaluation is a multiplication and addition of the form $jk + m \bmod p$, with $k, m < p$ and $j < 2^{128}$. As just seen, say that j and k are 128-values spread into registers w, x, y and z so that $jk = wy2^{128} + (wz + xy)2^{64} + xz \pmod p$. Because $2^{128} = 2 \pmod p$, this can be rewritten $jk = ((wz + xy) \textbf{ mod } 2^{64})2^{64} + (2(((wz + xy) \textbf{ div } 2^{64}) + wy) + xz) \pmod p$. If j and k are unrestricted, then every addition could result in a carry beyond 128-bits. These carries must be accumulated and dealt with, which could be inefficient. Ideally this computation of jk would involve no carries beyond 128-bits, allowing a more efficient computation.

Eliminating carries can be done by restricting k. The polynomial hash described in this section is (l/p)-AΔU when hashing l-block messages and choosing k from $0,\ldots,p-1$. This is due to the fact that there are at most l values in the range $0,\ldots,p-1$ that cause $h_k(M) - h_k(M') - d \pmod p$ to evaluate to zero. If k is chosen from some subset $A \subseteq \{0,\ldots,p-1\}$ instead, there would still be at most l values that cause $h_k(M) - h_k(M') - d \pmod p$ to evaluate to zero, but because $|A| \le p$, the probability of randomly choosing one of them increases to at most $l/|A|$. This means A can be chosen judiciously to exclude keys which cause excessive carries. In the case of evaluating polynomials modulo $p = 2^{127}-1$, restricting k to elements of $A = \{y2^{64} + z \mid 0 \le y < 2^{62}, 0 \le z < 2^{63}\}$ eliminates all but one possible carry beyond 128-bits when computing jk on a 64-bit architecture for any $0 \le j < 2^{128}$.

Experimentally, we have found that long sequences of cache-resident message blocks, each already less than $2^{127}-1$, can be hashed at a rate of 1.7 Athlon64 cpb when k is chosen as described to avoid excessive carries. When hashing sequences of values less than $2^{61}-1$ over modulus $2^{61}-1$, allowing k to be any value less than $2^{61}-1$, messages can be hashed at 1.3 cpb on the Athlon64. It should be noted that hashing an arbitrary string would not be nearly as fast due to the need of breaking the string into appropriate blocks.

The original UHASH use of polynomial hashing in its second hash stage is a balancing act. The goal of UMAC's entire three-stage composed hash is to be as close as possible to (2^{-32})-AΔU. This means that no stage can have significantly worse (and should strive to do significantly better) than this level. For the polynomial stage of UHASH to avoid exceeding (2^{-32})-AΔU, it dynamically alters the prime modulus to a larger number as the input length increases. This allows short strings to be hashed using a (more efficient) small modulus, but still allows for longer strings to be hashed (using a less efficient large modulus). On 64-bit architectures, fixing on a larger modulus is adequately efficient so as not to need dynamic moduli.

2.3 Inner Product Hashing

Another well known provably universal hashing paradigm is the inner product over a prime modulus [6]. Again, let p be a prime and let M be broken into fixed-length blocks $M_1, M_2, M_3, \ldots, M_l$ in such a way that when the blocks are interpreted as unsigned integers $m_1, m_2, m_3, \ldots, m_l$, each is less than p. Then, choosing a vector $\mathbf{k} = (k_1, k_2, \ldots, k_l)$ with $0 \leq k_i < p$ for all $1 \leq i \leq l$ defines the hash output as

$$h_{\mathbf{k}}(M) = m_1 k_1 + m_2 k_3 + \cdots m_l k_l \bmod p\,.$$

For any two different messages M, M' of the same block length l and integer $0 \leq d < p$, when $\mathbf{k}$ is chosen at random, the probability that

$$h_{\mathbf{k}}(M) - h_{\mathbf{k}}(M') = (m_1 - m_1')k_1 + (m_2 - m_2')k_2 + \cdots + (m_l - m_l')k_l \bmod p = d$$

is exactly $1/p$. It follows that inner product hashing over a prime modulus forms an ε-AΔU hash family for $\varepsilon = 1/p$.

CHARACTERISTICS. Inner-product hashing requires at least as much key as message being hashed. This makes it unsuitable for long messages. But, for short messages, implementations can be efficient using strategies already discussed for polynomial hashing. In particular, lazy modular reduction and choosing a prime modulus of the form $p = 2^a - b$ where b is small, results in good performance. For example, when $p = 2^{61} - 1$, $j < 2^{64}$ and $k < p$, the product jk (mod p) can be efficiently computed as $(jk \ \mathbf{div}\ 2^{64})2^3 + (jk \ \mathbf{mod}\ 2^{64})$ because $2^{64} = 2^3$ (mod p).

3 UHASH8 Algorithm

With these hash functions as building blocks and UHASH as a model, a hash function suitable for authenticating arbitrary messages and optimized for 64-bit architectures can be presented. The hash functions presented so far have interfaces that are incompatible with one another without some adaptation. For example, NH produces outputs with values up to $2^{128} - 1$, whereas the polynomial hash only accepts sequences of values less than $2^{127} - 1$. Similarly, the polynomial hash produces a value less than $2^{127} - 1$, but the inner-product expects a sequence of values less than $2^{61} - 1$. To address these problems, a lemma is introduced which allows out-of-range values to be brought into range with a manageable increase to the probabilities involved. Length issues must also be resolved. As presented, each hash function seen so far has a universality guarantee when hashing messages of the equal length. These must be extended to provide universality guarantees over all lengths. Figure 1 specifies the finished hash function UHASH8. Each of the interfaces between component hash families and the extensions for arbitrary lengths will now be addressed separately.

3.1 First: A Lemma

The primary tool used to fix the problem that one hash function produces values that are outside of the domain of a second hash function is the following lemma which says that if we routinely zero any fixed bit-position of the outputs of an ε-AΔU hash function, the resulting hash function is still AΔU but with a reduced universality guarantee.

DEFINITIONS. When x is a non-negative integer, let x_i be 1 if the binary representation of x has a 1 in the position of weight 2^i and 0 otherwise. Let $\mathrm{Zero}_i(x)$ be the function that returns x if $x_i = 0$ and returns $x - 2^i$ if $x_i = 1$ (ie, it returns x with the 2^i position zeroed). $\mathbb{Z}_n$ is the set $\{0, 1, 2, \ldots, n-1\}$.

Lemma 1 *Let $H = \{h : A \to \mathbb{Z}_n\}$ be an ε-AΔU hash family (where the operation is addition modulo n) and $H_i = \{\mathrm{Zero}_i \circ h \,|\, h \in H\}$, then H_i is (3ε)-AΔU for every i.*

Proof: Let $a \neq b$ be elements of A, and d and d' be elements of $\mathbb{Z}_n$. Because H is ε-$A\Delta U$, we know that $\Pr[h(a) - h(b) = d] \leq \varepsilon$ when h is chosen randomly from H, but what is the probability $\Pr[h'(a) - h'(b) = d']$ when h' is chosen randomly from H_i?

Let h be chosen randomly, and let $h' = \text{Zero}_i \circ h$ for some $0 \leq i < \lg n$. Define $x = h(a)$ and $y = h(b)$. There are four possible combinations for the values of x_i and y_i: (x_i, y_i) could equal $(0,0)$, $(0,1)$, $(1,0)$ or $(1,1)$. We look at each case.

If $x_i = y_i$, then $h'(a) - h'(b) = d'$ if and only if $h(a) - h(b) = d'$. Using conditional probability we can bound the likelihood of this scenario as $\Pr[h(a) - h(b) = d' \text{ and } x_i = y_i] = \Pr[h(a) - h(b) = d'] \cdot \Pr[x_i = y_i \,|\, h(a) - h(b) = d'] \leq \varepsilon \cdot 1$. Similarly, if (x_i, y_i) is $(0,1)$ or $(1,0)$ then $h'(a) - h'(b) = d'$ if and only if $h(a) - h(b)$ is $d' + 2^i$ or $d' - 2^i$, respectively, each of which is similarly bounded by $\varepsilon \cdot 1$. These three cases being the only ones in which $h'(a) - h'(b) = d'$, H' must be 3ε-$A\Delta U$. $\diamondsuit$

Note that a similar result is not possible for ε-AU hash families. Zeroing a bit of an ε-AU hash family can eliminate all guarantees. The identity function $f_I(x) = x$ is 0-AU, but if you zero the last bit of the output (ie, define $h = \text{Zero}_0 \circ f_I$), then $h(s||0)$ and $h(s||1)$ always collide for every s.

3.2 First Hash Phase – NH

The goal of the NH hashing phase (Lines 1–6 of Figure 1) is to hash arbitrary messages into much shorter representations (albeit proportional in length to their originals) in such a way that two distinct arbitrary-length messages have a low probability of hashing to the same result (so that inputs to the next hash phase are unlikely to be the same). As described in Section 2, NH is parameterized on word size w and number of words n, and can hash strings that are a multiple of $2w$ bits in length but no longer than $b = nw$ bits. Letting $w = 64$ (for optimization on 64-bit architectures) and n any positive even integer (ie, NH with these parameters accepts any string of length a multiple 128 bits but not longer than $b = nw$ bits), then lines 1–6 of Figure 1 define a hash family utilizing NH. The domain of the hash family is binary strings of any length. The codomain is vectors of integers from $\mathbb{Z}_{2^{126}}$. Randomly choosing a function from the hash family is achieved by choosing a random b-bit string K. To this end, Lines 1–6 work as follows. Given string M and parameter b (which must be a positive multiple of 128), break M into $n = \lceil |M|/b \rceil$ blocks $M_1, M_2, \ldots, M_n$ so that each of the first $n-1$ blocks is of length b and M_n is whatever is left over (Lines 1–2). If M was the empty string, then for convenience n is set to 1. Each of the blocks $M_1, \ldots, M_{n-1}$ is guaranteed to be in the domain of NH. Block M_n may not be a multiple of 128, so appending the fewest number of zeros needed to make it so will bring M_n also into the domain of NH (Lines 4–5). The blocks can then each be hashed independently by NH. Each NH result then has the (pre-zero-padding) length of each block added, and the two most significant bits of the result zeroed (Line 6). These n resulting values form a vector which is the hash function's output.

Lemma 2 *Let b be any positive multiple of 128. Lines 1–6 of Figure 1 define a $(9/2^{64})$-AU hash family over binary strings of arbitrary length.*

Proof: Let b be a positive multiple of 128, K be a uniformly distributed b-bit string, and $M \neq M'$ arbitrary binary strings. Let $M = M_1, \ldots, M_m$ and $M' = M'_1, \ldots, M'_n$ be broken into blocks and let l_i and l'_i represent the length of M_i and M'_i as described in Lines 1–3 of Figure 1. Let M_m and M'_n be zero extended to the nearest multiple of 128 bits, if needed, as described in Lines 4–5. what is the probability that identical vectors are produced by evaluating Line 6 on $M_1, \ldots, M_m$ and $M'_1, \ldots, M'_n$? If $n \neq m$, the probability of collision is zero because the vectors produced will be different lengths. There are two other cases to examine.

If $n = m$ and $M_i \neq M'_i$ for some $1 \leq i \leq n$, then, because NH is 2^{-64}-AΔU over strings that are a multiple of $2w$ bits in length (which both M_i and M'_i are guaranteed to be), the probability that $(\text{NH}(K, M_m) \textbf{ mod } 2^{126}) - (\text{NH}(K, M'_n) \textbf{ mod } 2^{126}) = l'_n 2^{64} - l_m 2^{64}$ is no more than $9/2^{64}$. The factor of nine comes from the **mod** 2^{126},

```
UHASH8[b](M,K,k,k_1,k_2)
Inputs:
    $M$, a string of any length
    $K$, a string of length $b$ bits, where $b = 128i$ for some integer $i > 1$
    $k$, an element of $\{w2^{96} + x2^{64} + y2^{32} + z \mid w,x,y,z \in \mathbb{Z}_{2^{30}}\}$
    $k_1, k_2$, integers in the range $0 \ldots 2^{61} - 2$, inclusive
Output:
    $h$, an integer in the range $0 \ldots 2^{61} - 2$, inclusive
Algorithm:
1.  $n = \max(\lceil |M|/b \rceil, 1)$
2.  Let $M_1, M_2, \ldots, M_n$ be strings so that $M_1||M_2||\cdots||M_n = M$ and
        $|M_i| = b$ for $1 \le i < n$.
3.  $l_i = |M_i|$ for each $1 \le i \le n$
4.  Let $j \ge 0$ be the smallest integer so that $|M_n| + j \textbf{ mod } 128 = 0$
5.  $M_n = M_n||0^j$
6.  $a_i = (\mathsf{NH}[b/64, 64](K, M_i) \textbf{ mod } 2^{126}) + l_i 2^{64}$ for each $1 \le i \le n$
7.  $p = k^{n+1} + a_1 k^n + a_2 k^{n-1} + \cdots + a_n k^1 \textbf{ mod } (2^{127} - 1)$
8.  $p_1 = (p \textbf{ div } 2^{64}) \textbf{ mod } 2^{60}$
9.  $p_2 = p \textbf{ mod } 2^{60}$
10. $h = p_1 k_1 + p_2 k_2 \textbf{ mod } (2^{61} - 1)$
```

Figure 1: *The hash family UHASH8 is ε-AΔU, when K, k, k_1, k_2 are chosen randomly from their domains, where $\varepsilon \approx 2^{-60} + (l/b)2^{-107}$.*

which has the affect of zeroing the top two bits of the NH output. Lemma 1 says that this causes up to a factor of nine degradation.

There is one more situation to consider: when one string is a proper prefix of the other before zero-padding, but the two strings are identical afterward. In this case, $M_m = M'_n$ because the strings are the same after padding but $l_m \neq l'_n$ because one string was a proper prefix of the other before padding. There is thus zero probability that $(\text{NH}(K, M_m) \textbf{ mod } 2^{126}) + l_m 2^{64} = (\text{NH}(K, M'_n) \textbf{ mod } 2^{126}) + l'_n 2^{64}$ because the NH hashes are guaranteed to give the same result, but two different lengths are added.

In every case, the probability that the vectors output are identical when hashing M and M' under key K and parameter b is no more than $9/2^{64}$. $\diamondsuit$

3.3 Second Hash Phase – Polynomial

The goal of the second hashing phase (Lines 7–9 in Figure 1) is to take the unbounded-length output of the first NH hash phase and hash it to a short fixed-length string in such a way that if two inputs to this stage differ then the probability that the outputs collide is low. Lines 7–9 define a universal hash family. The domain of the hash family is vectors of integers from $\mathbb{Z}_{2^{127}-1}$. The codomain is ordered pairs from $\mathbb{Z}_{2^{60}} \times \mathbb{Z}_{2^{60}}$. Choosing a random function from the hash family is done by choosing a random element $k \in \{w2^{96} + x2^{64} + y2^{32} + z \mid w,x,y,z \in \mathbb{Z}_{2^{30}}\}$. Line 7 is a simple polynomial evaluation hash modulo $2^{127} - 1$. Lines 8–9 apply Lemma 1 by zeroing seven bits to produce an output in the domain of the third hash phase. Since the first NH phase outputs sequences of values less than 2^{126}, those outputs are suitable without modification for hashing by the polynomial hash.

```
UHASH8-128[$b$]($M, K_1, K_2, k$)
Inputs:
   $M$, a string of any length
   $K_1, K_2$, strings of length $b$ bits, where $b = 128i$ for some integer $i > 1$
   $k$, an element of $\{w2^{96} + x2^{64} + y2^{32} + z \mid w,x,y,z \in \mathbb{Z}_{2^{30}}\}$
Output:
   $h$, an integer in the range $0 \ldots 2^{127} - 2$, inclusive
Algorithm:
1. $n = \max(\lceil |M|/b \rceil, 1)$
   Let $M_1, M_2, \ldots, M_n$ be strings so that $M_1 || M_2 || \cdots || M_n = M$ and
      $|M_i| = b$ for $1 \le i < n$.
   $l_i = |M_i|$ for each $1 \le i \le n$
   Let $j \ge 0$ be the smallest integer so that $|M_n| + j \textbf{ mod } 128 = 0$
   $M_n = M_n || 0^j$
   $a_i = \mathsf{NH}[b/64, 64](K_1, M_i) \textbf{ mod } 2^{126}$ for each $1 \le i \le n$
   $b_i = (\mathsf{NH}[b/64, 64](K_2, M_i) \textbf{ mod } 2^{126}) + l_i 2^{64}$ for each $1 \le i \le n$
   $h = k^{2n+1} + a_1 k^{2n} + b_1 k^{2n-1} + a_2 k^{2n-2} + b_2 k^{2n-3} + \cdots$
      $+ a_n k^2 + b_n k^1 \textbf{ mod } (2^{127} - 1)$
```

Figure 2: *The hash family UHASH8-128 is ε-AΔU, when K_1, K_2, k_1, k_2 are chosen randomly from their domains, where $\varepsilon \approx (l/b)2^{-119}$.*

Lemma 3 *Let $n \ge 0$ be an arbitrary integer. Lines 7–9 of Figure 1 define a $(n/2^{107})$-AU hash family over vectors of length up to n of values less than $2^{127} - 1$.*

Proof: It is known that the polynomial hash of Section 2 is universal over vectors of the same length. So, to allow vectors of varying length, let n be an integer no less than the length of the longest vector to be hashed. Then, to hash vector $m_1, m_2, \ldots, m_j$ with the polynomial hash of Section 2, first prepend $n - j$ zeros and a one to the vector, resulting in a vector $0, 0, \ldots, 0, 1, m_1, \ldots, m_j$ of length $n + 1$ elements. This preprocessing assures that all vectors hashed by the polynomial are the same length, and it assures that any pair of vectors that are different before preprocessing are also different after preprocessing. This preprocessing step extends the basic polynomial hash of Section 2 to vectors up to length n, but maintains a $((n+1)/2^{120})$-AΔU guarantee when key k is chosen from $\{w2^{96} + x2^{64} + y2^{32} + z \mid w,x,y,z \in \mathbb{Z}_{2^{30}}\}$. Notice that Line 7 of Figure 1 produces the same result as would the preprocessed polynomial hash just described. This is because the prepended zeros have no computational effect but are used only as a conceptual device to make all vectors equal length. Thus the hash on Line 7 is also $((n+1)/2^{120})$-AΔU. Lemma 1 tells us that zeroing seven bits as in Lines 8–9, degrades the universality guarantee by up to a factor of 3^7. To simplify the guarantee, $(3^7(n+1))/2^{120}$ can be bounded from above by $n/2^{107}$. $\diamondsuit$

3.4 Third Hash Phase – Inner-Product

Line 10 of Figure 1 is a straightforward application of the inner-product hash from Section 2. It is a hash family with domain $\mathbb{Z}_{2^{61}-1} \times \mathbb{Z}_{2^{61}-1}$ and codomain $\mathbb{Z}_{2^{61}-1}$. Choosing a random function from the hash family is done by choosing a random $(k_1, k_2) \in \mathbb{Z}_{2^{61}-1} \times \mathbb{Z}_{2^{61}-1}$. Because the output from the second hashing phase is a pair of values less than 2^{60}, no adjustment is needed. The following proposition needs no further proof.

	Message Length			
NH Key	40 Bytes	576 Bytes	1500 Bytes	4 KB
16 Bytes	4.1	2.3	2.2	2.2
32 Bytes	3.2	1.2	1.1	1.0
64 Bytes	2.5	0.7	0.7	0.7
128 Bytes	2.6	0.6	0.5	0.5
256 Bytes	3.1	0.7	0.5	0.4
512 Bytes	3.1	0.7	0.5	0.4
1024 Bytes	3.2	0.7	0.6	0.4
UHASH	4.8	1.5	1.2	1.0
Poly1305	8.9	3.3	3.2	3.1

Figure 3: **UHASH8 Performance.** *Eficiency of the UHASH8 hash function over various NH key sizes and message lengths, measured in Athlon64 CPU cycles per byte of message hashed. All data is resident in cache. Authentication with UHASH8 requires 64-bits of block-cipher output, so at least 125 additional cycles per message authenticated is needed. The original UHASH and Poly1305, also run on the Athlon64, are listed for comparison. Note that Poly1305 does not have a 64-bit option, so 128-bit output timings are given for it.*

	Message Length			
NH Key	40 Bytes	576 Bytes	1500 Bytes	4 KB
128 Bytes	4.0	1.3	1.1	1.1

Figure 4: **UHASH8-128 Hash Performance.** *Efficiency of the UHASH8-128 hash function over 128 byte NH key size and various message lengths, measured in Athlon64 CPU cycles per byte of message hashed. All data is resident in cache. Authentication requires an additional block-cipher invocation (at least 250 cycles per message).*

Proposition 1 *Line 10 of Figure 1 defines a $(1/(2^{61}-1))$-AΔU hash family over $\mathbb{Z}_{2^{61}-1} \times \mathbb{Z}_{2^{61}-1}$.*

3.5 Putting it Together – UHASH8

Lines 1–10 of Figure 1 define UHASH8 as the composition of three compatible universal hash functions. The properties of composed hash functions is well known [2, 10]. In particular, if H_1 is an ε_1-AU family of hash functions, all with common domain A, and H_2 is an ε_2-AU family of hash functions, all with common codomain $B \subseteq A$, then $H = \{h_1 \circ h_2 \,|\, h_1 \in H_1, h_2 \in H_2\}$ is $(\varepsilon_1 + \varepsilon_2)$-AU. If H_1 is ε_1-AΔU, then H is $(\varepsilon_1 + \varepsilon_2)$-A$\Delta$U. This leads immediately to the following result.

Theorem 1 *Let b be any positive multiple of 128. Figure 1 defines a hash family that is ε-AΔU over all binary strings up to length l bits where $\varepsilon = (9/2^{64}) + (\lceil l/b \rceil / 2^{107}) + (1/(2^{61}-1)) \approx 2^{-60} + (l/b)2^{-107}$.*

Figure 2 gives a version of UHASH8 producing 128-bit outputs. Although no proof of correctness is given here, the arguments mirror those of UHASH8.

3.6 UHASH8 Performance

Network communications on the Internet is dominated by just a few message lengths corresponding to common sizes seen in the TCP protocol. A few of the most common of these are 40, 576 and 1500 bytes.

Figures 3 and 4 show UHASH8 and UHASH8-128 hashing performance on these message lengths. Every message length is hashed for several different NH key lengths to see how NH key length affects performance. As expected, on long messages longer NH key lengths indicate higher performance because overhead is reduced and the number of hashes beyond NH is reduced.

For comparison, two high-speed hash functions not designed for 64-bit architectures are listed, the original UHASH and Poly1305 [1, 3]. Source code for these other hash functions was taken from the author's websites.

References

[1] Bernstein D. The Poly1305-AES message-authentication code. To appear in *Proceedings of the 12th Workshop on Fast Software Encryption*. Springer-Verlag, 2005.

[2] Bierbrauer J, Johansson T, Kabatianskii G, Smeets B. On families of hash functions via geometric codes and concatenation. In *Advances in Cryptology – CRYPTO '93*. Springer-Verlag, 1993; 331–342.

[3] Black J, Halevi S, Krawczyk H, Krovetz T, Rogaway P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99*. Springer-Verlag, 1999; 216–233.

[4] Boesgaard M, Christensen T, Zenner E. Badger – A fast and provably secure MAC. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005*. Springer-Verlag, 2005; 176–191.

[5] Carter L, Wegman M. Universal classes of hash functions. *J. of Computer and System Sciences* 1981; **22**:265–279.

[6] Cormen T, Leiserson C, Rivest R, Stein C.. *Introduction to algorithms*. MIT Press, 2001. Section 11.3.3.

[7] Halevi S, Krawczyk H. MMH: Software message authentication in the Gbit/second rates. In *Proceedings of the 4th Workshop on Fast Software Encryption*. Springer-Verlag, 1997; 172–189.

[8] Knuth D. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd ed*. Addison-Wesley, 1998; 486–489.

[9] Krovetz T, Rogaway P. Fast universal hashing with small keys and no preprocessing: The PolyR construction. In *Information Security and Cryptology – ICICS 2000*. Springer-Verlag, 2000; 73–89.

[10] Stinson D. Universal hashing and authentication codes. *Designs, Codes and Cryptography* 1994; **4**:369–380.

[11] Wegman M, Carter L. New hash functions and their use in authentication and set equality. *J. of Computer and System Sciences* 1979; **18**:143–154.