Message Authentication on 64-bit Architectures

Ted Krovetz

Department of Computer Science
California State University, Sacramento CA 95819 USA

Abstract. This paper introduces VMAC, a message authentication algorithm (MAC) optimized for high performance in software on 64-bit architectures. On the Athlon 64 processor, VMAC authenticates 2KB cache-resident messages at a cost of about 0.5 CPU cycles per message byte (cpb) — significantly faster than other recent MAC schemes such as UMAC (1.0 cpb) and Poly1305 (3.1 cpb). VMAC is a MAC in the Wegman-Carter style, employing a "universal" hash function VHASH, which is fully developed in this paper. VHASH employs a three-stage hashing strategy, and each stage is developed with the goal of optimal performance in 64-bit environments.

Keywords: Message authentication, universal hashing, architectural optimization.

I personally believe there are two main architectures out
there: Power and x86-64 [both of which are 64-bit architectures].
— 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, especially in domains where high performance is desirable, are designed with optimizations tailored for particular architectures. 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 software-optimized message authentication algorithms (or MACs) are all designed to run well on 32-bit architectures [5, 6, 9]. 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 64-bit processors. As an example consider UMAC, which

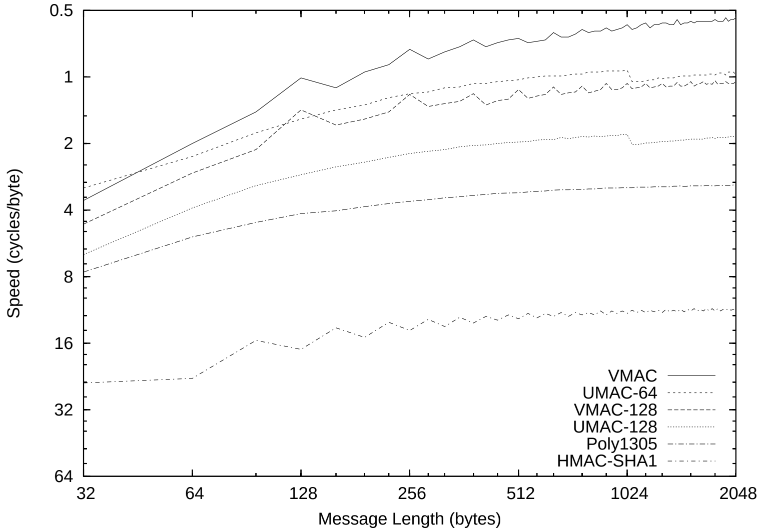

Fig. 1. *Efficiency on the Athlon 64 processor of the hash functions underlying recent MACs, measured in CPU cycles per byte of message hashed. VMAC and UMAC-64 produce 64-bits while the others produce 128 (or 160 for SHA-1).*

was designed specifically for optimal performance on 32-bit architectures [5]. On an Athlon 64 processor, UMAC achieves peak speeds of about 1.1 CPU cycles per byte of message authenticated (cpb) when it is restricted to using 32-bit operands, but improves only slightly to 1.0 cpb when allowed full use of 64-bit operands.

This paper presents VMAC, the fastest reported MAC on contemporary 64-bit processors, achieving peak speeds of 0.5 cpb on the Athlon 64 (and as low as 0.3 cpb if one allows larger internal keys, see Figure 4). This compares favorably with other MACs. Figure 1 shows the performance of the internal hash functions of recent high-speed MAC's.[1] (Hash speeds determine a MAC's relative speed because all modern MACs hash their messages first.) Although the main goal of the VMAC design effort is high speed on 64-bit processors, VMAC is careful to avoid some of the perceived deficiencies of recent high-speed MACs, particularly by limiting use of internal hash key (VMAC uses 160 bytes) and avoidance of data-dependant side-channel attacks. VMAC has the desirable properties of being provably secure, parallelizable, and patent-free.

[1] Timings in this paper are generated using `gcc` 4.0 with optimization level `-O4` (or `-fast` if available) and appropriate `-march` or `-mcpu` settings. The bulk of performance claims are based on an AMD Athlon 64 "Manchester" (family 15, model 43, stepping 1, L2 cache 512K). Other architectures are reported in Section 3.6. All data 16-byte aligned and in cache. UMAC and Poly1305 timings are made from code obtained at their author's websites. SHA timings are as reported by OpenSSL.

MESSAGE AUTHENTICATION. Message authentication is used when parties wish to communicate with assurance that received messages come from the claimed sender without alteration. All of the fastest MACs follow principles developed by Wegman and Carter [3, 5–7, 9, 14]. 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 usually need only be secure against forgery during the lifetime of the session. If an adversary cannot forge with a probability of more than about $1/2^{60}$ per attempt, then the MAC is likely suitable for most communications where attackers are not allowed an excessive number of forgery attempts. VMAC and UMAC are flexible in their security levels. They are able to produce 64-bit tags with forgery probabilities close to $1/2^{60}$ twice as fast as versions of VMAC and UMAC which produce 128-bit tags (which have forgery probabilities closer to $1/2^{100}$). Most other MAC schemes, including the ones used for comparison in this paper, are not designed for such flexibility. As a result, when making speed comparisons with VMAC's 64-bit tags, other schemes, producing longer tags, are unavoidable disadvantaged.

The key to speed in a Wegman-Carter MAC is the hash function used. 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, so this paper focuses on the hash function used in VMAC, known as VHASH. 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 suitable for Wegman-Carter message authentication (and peak speeds reported for Intel Pentium 4 processors by their authors) are hash127 (around 4.4 cpb), hash1305 (3.4 cpb), Badger (2.2 cpb) and UMAC (1.0 cpb). The speeds of all of these favorably compare with popular non-Wegman-Carter MACs such as HMAC-SHA1 and CBC-AES-MAC, both of which require more than 10 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 domain A and 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. A stronger notion bounds an adversary's ability to guess differences between hash outputs. A hash-function family H is *ε-almost delta universal* (ε-AΔU) if the probability is no more than ε that $h(m) - h(m') = d$ for any two distinct inputs m, m' and any chosen constant d when hashed by a randomly selected member h of H. There are stronger notions of universal hashing defined by Wegman, Carter and Stinson [7, 13, 14], but ε-AΔU is adequate for message authentication, and is achieved by VHASH.

2 Three-Stage Hashing

VHASH uses a three-stage hashing strategy where each hash stage is made of a discrete hash function with a particular purpose. The first stage rapidly compresses by a fixed ratio the message to be hashed, thus reducing the data to be processed by later (slower) stages. The second stage hashes the newly compressed message to a fixed length, and the third stage distills the security of the second-stage output into a smaller number of bits. In this section, we investigate appropriate primitive hash functions for each stage and develop them for 64-bit architectures. In the next section, VHASH is assembled from the functions described here and analyzed for the purposes of message authentication. The first MAC to employ a three-stage strategy was UMAC [5, 11].

2.1 Stage 1 – Acceleration

The goal of the first stage is to act as an accelerant for later hash stages by compressing, at a constant ratio, long inputs into shorter ones at very high speed. VHASH uses the NH hash family for this purpose, breaking the VHASH input into b-bit blocks (the final block may be shorter) and using NH to hash each into 128 bits. The hashed blocks are then concatenated into a string shorter than the original VHASH input. When b is set at 1,024 bits (as we later recommend), the compression is 8:1 for messages whose length is a multiple of 1,024.

NH was originally designed as a parameterized hash function [5]. 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 $M_1, M_2, \ldots, M_\ell$ and $K_1, K_2, \ldots K_n$ where $\ell = |M|/w$. Then, each block is interpreted as a w-bit unsigned binary integer $m_1, m_2, \ldots, m_\ell$ and $k_1, k_2, \ldots k_n$. Finally, the hash result is computed as

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

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 any multiple of $2w$ bits in length (but still no longer than nw bits). In the context of VHASH, $w = 64$ and $nw = b$ is suggested 1,024.

Characteristics. The chief advantage of NH is extreme speed. Every operation is done naturally and efficiently on contemporary processors if w is chosen appropriately. On 64-bit processors with good support for multiplying 64-bit quantities into a 128-bit result, defining $w = 64$ results in very high speeds.

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})-AU 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})-AU guarantee when $w = 32$. Each NH hashing of the message would require $2j$ multiplications and produce a hash value with a (2^{-32})-AU guarantee. This would have to be done twice, under separate keys, to achieve the (2^{-64})-AU 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 passes with $w = 32$ takes about 2 cpb while a single pass with $w = 64$ requires only around 0.5 cpb on the Athlon 64.

One of the design goals for VHASH is achieving a balance between performance and internal key requirement. As can be seen in Figure 4, increasing the NH key length in VHASH increases VHASH performance for long messages greatly at first, but performance increases drop-off at around 128–256 bytes. We recommend 128 bytes for the NH hash key for applications which are not extremely memory constrained. This choice harnesses most of the potential speed gains of NH with fairly low key requirement.

2.2 Stage 2 – Fix Length

The first stage produces an output proportional in length to the original input, which means that to achieve a fixed length, further hashing is necessary. Recent research into various polynomial-based hash functions have yielded hash functions appropriate for the task with good speed and universality guarantees [1, 3, 11, 12]. Section 3 will address domain reconciliation necessary between stage-one outputs and stage-two inputs.

A simple and efficient method to hash a string M is to fix prime number p and break M into fixed-length blocks $M_1, M_2, M_3, \ldots, M_\ell$ in such a way that when the blocks are interpreted as unsigned integers $m_1, m_2, m_3, \ldots, m_\ell$, 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

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

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

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

Because $M \neq M'$, at least one of the coefficients 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 a hash family $H = \{h_k \,|\, 0 \leq k < p\}$, then H is an ε-AΔU hash family for $\varepsilon = \ell/p$.

Characteristics. With care, polynomial hashing can be made to perform well. Horner's Rule suggests rephrasing $h_k(M)$ as $((\cdots((m_1k + m_2)k + m_3)k \cdots)k + m_\ell)k \bmod p$, which allows $h_k(M)$ to be computed as a sequence of ℓ multiplications and additions modulo p [10]. 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 $0 \leq y < 2^{61}$, 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 $0 \leq x, z < 2^{64}$, 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 with 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 $|A| \leq p$, the probability of randomly choosing one of them increases to at most $\ell/|A|$. 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 $A = \{y2^{64} + z \mid 0 \leq y < 2^{62}, 0 \leq z < 2^{63}\}$ eliminates all but one possible carry beyond 128-bits when computing jk on a 64-bit architecture for any $0 \leq j < 2^{128}$.

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 cpb on the Athlon 64 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 Athlon 64. 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 (within the modulus).

2.3 Stage 3 – Distillation

When NH is defined for $w = 64$ and the Polynomial hash is defined over prime modulus $p = 2^{127} - 1$, as is the case in VHASH, the resulting universality guarantee of the first two stages composed can be no better than (2^{-64})-AΔU (more on this in Section 3) and yet the output requires 127 bits. To reduce the disparity between the number of bits needed for the hash output and the universality guarantee, one final hash is used to hash the fixed length stage-two output into fewer bits.

Another well known provably universal hashing function is the inner product over a prime modulus [8]. Again, let p be a prime and let M be broken into fixed-length blocks $M_1, M_2, M_3, \ldots, M_\ell$ in such a way that when the blocks are interpreted as unsigned integers $m_1, m_2, m_3, \ldots, m_\ell$, each is less than p. Then, choosing a vector $\mathbf{k} = (k_1, k_2, \ldots, k_\ell)$ with $0 \leq k_i < p$ for all $1 \leq i \leq \ell$ defines the hash output as

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

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

$$h_{\mathbf{k}}(M) - h_{\mathbf{k}}(M') = (m_1 - m'_1)k_1 + (m_2 - m'_2)k_2 + \cdots + (m_\ell - m'_\ell)k_\ell \bmod p = d$$

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

```
$\mathsf{VHASH}[b](M, K, k, k_1, k_2)$
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.      $\ell_i = |M_i|$ for each $1 \le i \le n$
4.      Let $M_n = M_n || 0^j$ where $j \ge 0$ is the smallest integer so
            that $|M_n| + j$ $\mathbf{mod}$ $128 = 0$
5.      Byte-reverse each 64-bit word in $M_i$ for each $1 \le i \le n$
6.      $a_i = (\mathsf{NH}[b/64, 64](K, M_i) \ \mathbf{mod}\ 2^{126}) + (\ell_i \ \mathbf{mod}\ b)2^{64}$ for each $1 \le i \le n$
7.      $p = k^{n+1} + a_1 k^n + a_2 k^{n-1} + \cdots + a_n k^1 \ \mathbf{mod}\ (2^{127} - 1)$
8.      $p_1 = (p \ \mathbf{div}\ 2^{64}) \ \mathbf{mod}\ 2^{60}$
9.      $p_2 = p \ \mathbf{mod}\ 2^{60}$
10.     $h = p_1 k_1 + p_2 k_2 \ \mathbf{mod}\ (2^{61} - 1)$
```

Fig. 2. *The hash family VHASH is ε-AΔU, when K, k, k_1, k_2 are chosen randomly from their domains, where $\varepsilon = 2^{-59.9} + (\ell/b)2^{-107}$.*

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 $jk \pmod{p}$ can be efficiently computed as $(jk \ \mathbf{div}\ 2^{64})2^3 + (jk \ \mathbf{mod}\ 2^{64})$ because $2^{64} = 2^3 \pmod{p}$. This is exactly the computation done by VHASH in its third stage.

3 VHASH Definition

With these component hash functions as building blocks and the three-stage hash function as a model, a hash function suitable for authenticating arbitrary messages and optimized for 64-bit architectures can be presented. For any b which is a positive multiple of 128, Figure 2 specifies the hash family $\mathsf{VHASH}[b]$ where choosing a random function h from the family is achieved by choosing K, k, k_1 and k_2 uniformly at random from their domains and letting $h(\cdot) = \mathsf{VHASH}(\cdot, K, k, k_1, k_2)$.

Theorem 1. *Let b be any positive multiple of 128 and let ℓ be any positive integer, then* $\mathsf{VHASH}[b]$ *is ε-A$\varDelta$U over all binary strings up to length ℓ bits where $\varepsilon = 2^{-59.9} + (\ell/b)2^{-107}$.*

The theorem will be proven over a sequence of lemmas later in this section. With this result, VHASH can easily be embedded in a Wegman-Carter MAC which we call VMAC. Here we summarize its construction. (It will be fully specified in a separate document.) Let $p = 2^{61} - 1$ and N be some nonce space. Then for functions $f : N \to \mathbb{Z}_p$ and $h \in \mathsf{VHASH}[b]$, tag generation under VMAC is defined as $\mathsf{VMACTagGen}_{f,h}(m, n) = h(m) + f(n) \bmod p$ for message m and nonce n. We define the security of a nonce-based MAC scheme, such as VMAC, that uses tag-generation function $\mathsf{TagGen}(m, n)$ as follows. Assume an adversary knows any sequence of triples $(m_1, n_1, t_1) \dots (m_q, n_q, t_q)$ where each n_i is unique and $t_i = \mathsf{TagGen}(m_i, n_i)$ for each $1 \le i \le q$. The MAC scheme is α-secure if the adversary cannot produce (m, n, t) with probability exceeding α where $(m, n) \ne (m_i, n_i)$ for any i and $t = \mathsf{TagGen}(m, n)$. The following theorem follows from the theory of Wegman-Carter MACs.

Proposition 2. *Let ℓ be a positive integer, $p = 2^{61} - 1$ and N be some non-empty set. Let b be a positive multiple of 128. Let $\mathsf{VMACTagGen}_{f,h}(m, n) = h(m) + f(n) \bmod p$ for randomly chosen functions $f : N \to \mathbb{Z}_p$ and $h \in \mathsf{VHASH}[b]$. Then, $\mathsf{VMACTagGen}_{f,h}$ is a $(2^{-59.9} + (\ell/b)2^{-107})$-secure over messages upto ℓ bits in length.*

3.1 VHASH Analysis

The hash functions seen 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 equal length. These must be extended to provide universality guarantees over all lengths. Each stage of VHASH will now be analyzed for universality guarantee and interface.

3.2 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-A$\varDelta$U hash function, the resulting hash function is still A$\varDelta$U but with a reduced universality guarantee.

```
$\mathsf{VHASH\text{-}128}[b](M, K_1, K_2, k)$
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 $0 \dots 2^{127} - 2$, inclusive
Algorithm:
1. $n = \max(\lceil |M|/b \rceil, 1)$
2. Let $M_1, M_2, \dots, M_n$ be strings so that $M_1||M_2|| \cdots ||M_n = M$ and
     $|M_i| = b$ for $1 \le i < n$.
3. $\ell_i = |M_i|$ for each $1 \le i \le n$
4. Let $M_n = M_n||0^j$ where $j \ge 0$ is the smallest integer so
     that $|M_n| + j$ mod $128 = 0$
5. Byte-reverse each 64-bit word in $M_i$ for each $1 \le i \le n$
6. $a_i = (\mathsf{NH}[b/64, 64](K_1, M_i)$ mod $2^{126}) + (\ell_i$ mod $b)2^{64}$ for each $1 \le i \le n$
7. $b_i = \mathsf{NH}[b/64, 64](K_2, M_i)$ mod $2^{126}$ for each $1 \le i \le n$
8. $h = k^{2n+1} + a_1 k^{2n} + b_1 k^{2n-1} + a_2 k^{2n-2} + b_2 k^{2n-3} + \cdots$
     $+ a_n k^2 + b_n k^1$ mod $(2^{127} - 1)$
```

Fig. 3. *The hash family VHASH-128 is ε-AΔU, when K_1, K_2, k_1, k_2 are chosen randomly from their domains, where $\varepsilon = (\ell/b)2^{-118}$.*

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 $\{0, 1, 2, \dots, n-1\}$. When s is a string, $|s|$ is its bitlength.

Lemma 3. *Let $H = \{h : A \to \mathbb{Z}_n\}$ be an ε-AΔU hash family (where the operation is addition modulo n) and $H_i = \{\mathrm{Zero}_i \circ h \mid h \in H\}$, 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-*AΔU*, we know that $\Pr[h(a) - h(b) = d] \le \varepsilon$ 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 $h' = \mathrm{Zero}_i \circ h$ for some $0 \le i < \lg n$. 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.

When $x_i = y_i$, 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 $\Pr[h(a) - h(b) = d'$ and $x_i = y_i] = \Pr[h(a) - h(b) = d'] \cdot \Pr[x_i = y_i \mid h(a) - h(b) = d'] \le \varepsilon \cdot 1$. 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-$A\Delta U$. □

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 $h = \mathrm{Zero}_0 \circ f_I$), then $h(s||0)$ and $h(s||1)$ always collide for every s.

3.3 Stage 1 – NH

The goal of the first hashing phase (Lines 1–6 of Figure 2) 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). Letting b be any positive multiple of 128, Lines 1–6 of Figure 2 defines 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. Lines 1–6 work as follows. Given string M, 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 length b and M_n is whatever is left over (Lines 1–2). If M was the empty string, then 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, and so not in the domain of NH which is only defined for inputs with length divisible by $2w$. Appending the fewest number of zero bits needed to make it so will bring M_n into the domain of NH (Line 4). The blocks are then each hashed independently by NH, the two most significant bits of the results are zeroed (Line 6), and the result has the modulo-b pre-zero-padding length of its corresponding block added. Finally,. The n resulting values form a vector which is the hash function's output.

Lemma 4. *Let b be any positive multiple of 128. Lines 1–6 of Figure 2 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 2. Let M_m and M'_n be zero extended to the nearest multiple of 128 bits, if needed, as described in Line 4. The byte-reversal of Line 5 has no effect on whether $M_i = M'_i$ for any i. 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 $2w = 128$ bits in length (which both M_i and M'_i are guaranteed to be), the probability that $(\mathrm{NH}(K, M_m)$ **mod** $2^{126}) -$

$(\mathrm{NH}(K, M'_n) \ \mathbf{mod}\ 2^{126}) = 0$ is no more than $9/2^{64}$. The factor of nine comes from the $\mathbf{mod}\ 2^{126}$, which has the affect of zeroing the top two bits of the NH output. Lemma 3 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 $\ell_m \neq \ell'_n$ because one string was a proper prefix of the other before padding. There is thus zero probability that $(\mathrm{NH}(K, M_m) \ \mathbf{mod}\ 2^{126}) + (\ell_m \ \mathbf{mod}\ b)2^{64} = (\mathrm{NH}(K, M'_n) \ \mathbf{mod}\ 2^{126}) + (\ell'_n \ \mathbf{mod}\ b)2^{64}$ 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 $9/2^{64}$. □

3.4 Stage 2 – Polynomial

The goal of the second hashing phase (Lines 7–9 in Figure 2) 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 utilize Lemma 3 by zeroing seven bits and then breaking in two the result 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.

Lemma 5. *Let $n \geq 0$ be an integer. Lines 7–9 of Figure 2 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 2 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 3 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, $(3^7(n+1))/2^{120} < n/2^{107}$. □

3.5 Stage 3 – Inner-Product

Line 10 of Figure 2 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 $(k_1, k_2) \in \mathbb{Z}_{2^{61}-1} \times \mathbb{Z}_{2^{61}-1}$. The output from the second hashing phase is a pair of values less than 2^{60}, so no adjustment is needed. The following proposition needs no further proof.

Proposition 6. *Line 10 of Figure 2 defines a* $(1/(2^{61}-1))$*-AΔU hash family over* $\mathbb{Z}_{2^{61}-1} \times \mathbb{Z}_{2^{61}-1}$.

Putting it Together. Lines 1–10 of Figure 2 define VHASH as the composition of three universal hash functions. The properties of composed hash functions are well known [4, 13]. If H_1 is an ε_1-AU family of hash functions with codomain A, and H_2 is an ε_2-AU family of hash functions with domain B where $A \subseteq B$, then $H = \{h_2 \circ h_1 \,|\, h_1 \in H_1, h_2 \in H_2\}$ is $(\varepsilon_1 + \varepsilon_2)$-AU. If H_2 is ε_2-AΔU, then H is $(\varepsilon_1 + \varepsilon_2)$-A$\Delta$U. This leads immediately to the result of Theorem 1.

If an application needs collision probabilities less than those of VHASH, then VHASH could be applied to given messages twice, using a different key each time. Alternatively, Figure 3 gives a hash function VHASH-128 based on the same principles as VHASH, but producing 128-bit outputs without the need for significantly more internal key than VHASH. Although no proof of correctness is given here, the arguments mirror those of VHASH. VHASH-128 is $(\ell/b)2^{-118}$-AΔU.

3.6 VHASH Performance

The performance of VHASH is influenced by many factors, the most important being how efficiently the host architecture multiplies 64-bit and adds 128-bit quantities. The Athlon 64 and recently released Intel Core 2 architectures — both 64-bit and designed for high “performance-per-watt” — are very efficient in these operations and so perform at the level described in this paper. Architectures which do not support fast 64-bit multiplication and multi-precision addition do not execute VHASH as quickly. Consider multiplication of 64-bit operands into a 128-bit result. On the Athlon 64 this can be done using a single instruction with a latency of five cycles, and VHASH hashes at a peak of 0.5 cpb. Intel’s 64-bit NetBurst architecture (eg, “Nacona”) can also perform the multiplication in a single instruction, but has a latency of 12 cycles, resulting in a VHASH peak of 1.4 cpb. The PowerPC 970 requires two instructions to complete a 64-bit multiplication, with a total latency of 13 cycles, and VHASH peaks at 1.0 cpb. The PowerPC version is faster than the NetBurst version due to NetBurst’s horrible multiprecision addition latencies which also impact performance, but to a lesser extent than multiplication.

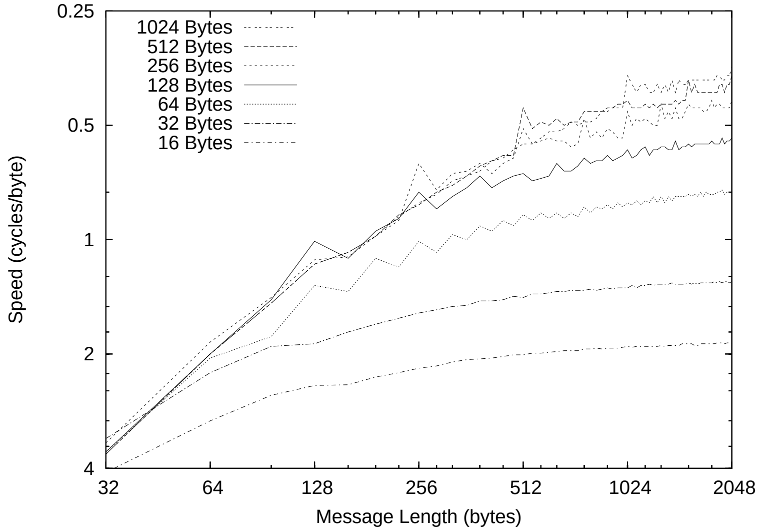

Fig. 4. *Performance measured in Athlon 64 cycles per byte of message hashed for various NH key lengths and message lengths. Gains diminish greatly beyond 128 bytes.*

VHASH slows down significantly on 32-bit architectures. Computing a 64-bit multiplication on a 32-bit architecture using the primary-school multiprecision multiplication algorithm requires four 32-bit multiplications and several multiprecision additions to produce a 128-bit result. On the Motorola PowerPC 7450, which has a six cycle latency per 32-bit multiplication, VHASH peaks at 5.0 cpb. On a 32-bit Intel NetBurst architecture, 32-bit multiplication latency is 11 cycles, but use of SSE vector instructions allows for a peak speed of 6.4 cpb. Clearly, VHASH benefits from architectures which multiply 64-bit registers fast, enabling exceptional VHASH performance.

Other hash functions are somewhat less variable across the mentioned architectures. UHASH and SHA1 were designed for 32-bit architectures, and so moving from 64-bit to 32-bit architectures has no inherent disadvantage. Poly1305 is multiplication based, but uses a processor's floating point unit and so is less affected by changes to general purpose register width. For all three hash functions, it is the efficiency of the processor implementation which will have the greatest impact. For example, Poly1305 has peak performance on a 64-bit PowerPC 970 of 6.6 cpb and 7.3 cpb on a slightly less efficient 32-bit PowerPC 7410. On the 64-bit Athlon 64 Poly1305 peaks at 3.1 cpb while it peaks on the much less efficient 32-bit Intel NetBurst at 5.2 cpb.

Other factors having significant affect on VHASH performance are the size of the message being hashed and the length b of the key used in the first (NH) stage of hashing. Figure 4 shows how these parameters affect performance on the Athlon 64 as measured in cycles per byte. Hashing overhead is amortized over all bytes being hashed, so as message lengths increase, overhead contributes less. Also, increasing the length of the Stage 1 NH key reduces the amount of data hashed by Stages 2 and 3. Since Stage 1 is much faster than the later stages, increasing the NH key length improves performance on longer messages.

Acknowledgements

The author wishes to thank the anonymous reviewers of SAC 2006 and FSE 2006 for their helpful reviews, especially in urging a closer look at performance on 32-bit architectures. Also, Phil Rogaway's comments and Joe Olivas's assitance in gathering timing data were timely and quite helpful. Thanks!

References

1. Afanassiev V, Gehrmann C, Smeets B. Fast message authentication using efficient polynomial evaluation. In *Fast Software Encryption – FSE 1997*. Springer-Verlag, 1997; 190–204.
2. Bernstein D. Stronger security bounds for Wegman-Carter-Shoup authenticators. In *Advances in Cryptology – EUROCRYPT 2005*. Springer-Verlag, 2005; 164–180.
3. Bernstein D. The Poly1305-AES message-authentication code. In *Fast Software Encryption – FSE 2005*. Springer-Verlag, 2005; 32–49.
4. Bierbrauer J, Johansson T, Kabatianskii G, Smeets B. On families of hash functions via geometric codes and concatenation. In *Advances in Cryptology – CRYPTO 1993*. Springer-Verlag, 1993; 331–342.
5. Black J, Halevi S, Krawczyk H, Krovetz T, Rogaway P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO 1999*. Springer-Verlag, 1999; 216–233.
6. Boesgaard M, Christensen T, Zenner E. Badger – A fast and provably secure MAC. In *Applied Cryptography and Network Security – ACNS 2005*. Springer-Verlag, 2005; 176–191.
7. Carter L, Wegman M. Universal classes of hash functions. *J. of Computer and System Sciences* 1981; **22**:265–279.
8. Cormen T, Leiserson C, Rivest R, Stein C. *Introduction to algorithms*. MIT Press, 2001. Section 11.3.3.
9. Halevi S, Krawczyk H. MMH: Software message authentication in the Gbit/second rates. In *Fast Software Encryption – FSE 1997*. Springer-Verlag, 1997; 172–189.
10. Knuth D. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd ed.* Addison-Wesley, 1998; 486–489.
11. 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.
12. Shoup V. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology – CRYPTO 1996*. Springer-Verlag, 1996; 313-328.
13. Stinson D. Universal hashing and authentication codes. *Designs, Codes and Cryptography* 1994; **4**:369–380.
14. 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.