# UMAC: Fast and Secure Message Authentication

J. Black[1], S. Halevi[2], H. Krawczyk[2,3], T. Krovetz[1], and P. Rogaway[1]

[1] Department of Computer Science, University of California, Davis CA 95616 USA
[2] IBM T.J. Watson Research Center, Yorktown Heights NY 10598 USA
[3] Department of Electrical Engineering, Technion – Israel Institute of Technology, 32000 Haifa ISRAEL

**Abstract.** We describe a message authentication algorithm, UMAC, which can authenticate messages (in software, on contemporary machines) roughly an order of magnitude faster than current practice (e.g., HMAC-SHA1), and about twice as fast as times previously reported for the universal hash-function family MMH. To achieve such speeds, UMAC uses a new universal hash-function family, NH, and a design which allows effective exploitation of SIMD parallelism. The "cryptographic" work of UMAC is done using standard primitives of the user's choice, such as a block cipher or cryptographic hash function; no new heuristic primitives are developed here. Instead, the security of UMAC is rigorously proven, in the sense of giving exact and quantitatively strong results which demonstrate an inability to forge UMAC-authenticated messages assuming an inability to break the underlying cryptographic primitive. Unlike conventional, inherently serial MACs, UMAC is parallelizable, and will have ever-faster implementation speeds as machines offer up increasing amounts of parallelism. We envision UMAC as a practical algorithm for next-generation message authentication.

## 1 Introduction

This paper describes a new message authentication code, UMAC, and the theory that lies behind it. UMAC has been designed with two main goals in mind: **extreme speed** and **provable security**. We aimed to create the fastest MAC ever described, and by a wide margin. (We are speaking of speed with respect to software implementations on contemporary general-purpose computers.) But we insisted that it be demonstrably secure, in the sense of having quantitatively desirable reductions from its underlying cryptographic primitives.

UMAC is certainly fast. On a 350 MHz Pentium II PC, one version of UMAC (where the adversary has $2^{-60}$ chance of forgery) gives a peak performance of 2.9 Gbits/sec (**0.98** cycles/byte). Another version of UMAC (with $2^{-30}$ chance of forgery) achieves peak performance of 5.6 Gbits/sec (**0.51** cycles/byte). For comparison, our SHA-1 implementation runs at **12.6** cycles/byte. (SHA-1 speed upper bounds the speed of HMAC-SHA1 [3], a software-oriented MAC representative of the speeds achieved by current practice.) The previous speed champion among proposed universal hash functions (the main ingredient for making a fast MAC; see below) was MMH [13], which runs at about **1.2** cycles/byte (for $2^{-30}$ chance of forgery) under its originally envisioned implementation.

How has it been possible to achieve these speeds? Interestingly, we have done this with the help of our second goal, provable security. We use the well-known universal-hashing approach to message authentication, introduced by [28], making innovations in its realization. Let us now review this approach and its advantages, and then describe what we have done to make it fly.

### 1.1 Universal-Hashing Approach

Universal Hashing And Authentication. Our starting point is a universal hash-function family [10]. (Indeed the "U" in UMAC is meant to suggest the central role that universal hash-function families play in this MAC.) Remember that a set of hash functions is said to be "$\epsilon$-universal" if for any pair of distinct messages, the probability that they collide (hash to the same value) is at most $\epsilon$. The probability is over the random choice of hash function.

As described in [28], a universal hash-function family can be used to build a MAC. The parties share a secret and randomly chosen hash function from the hash-function family, and a secret encryption key. A message is authenticated by hashing it with the shared hash function and then encrypting the resulting hash. Wegman and Carter showed that when the hash-function family is strongly universal (a similar but stronger property than the one we defined) and the encryption is realized by a one-time pad, the adversary cannot forge with probability better than that obtained by choosing a random string for the MAC.

Why Universal Hashing? As suggested many times before [16, 25, 13], the above approach is a promising one for building a highly-secure and ultra-fast MAC. The reasoning is like this: the speed of a universal-hashing MAC depends on the speed of the hashing step and the speed of the encrypting step. But if the hash function compresses messages well (i.e., its output is short) then the encryption shouldn't take long simply because it is a short string that is being encrypted. On the other hand, since the combinatorial property of the universal hash-function family is mathematically proven (making no cryptographic hardness assumptions), it needs no "over-design" or "safety margin" the way a cryptographic primitive would. Quite the opposite: the hash-function family might as well be the fastest, simplest thing that one can prove universal.

Equally important, the above approach makes for desirable security properties. Since the cryptographic primitive is applied only to the (much shorter) hashed image of the message, we can select a cryptographically conservative design for this step and pay with only a minor impact on speed. And the fact that the underlying cryptographic primitive is used only on short and secret messages eliminates many avenues of attack. *Under this approach security and efficiency are not conflicting requirements—quite the contrary, they go hand in hand.*

Quest For Fast Universal Hashing. At least in principle, the universal-hashing paradigm has reduced the problem of fast message authentication to that of fast universal hashing. Thus there has been much work on the design of fast-to-compute universal hash-function families. Here is a glimpse of some of this work. Krawczyk [16] describes the "cryptographic CRC" which has very fast hardware implementations and reasonably fast software implementations; it needs about 6 cycles/byte, as shown by Shoup [26]. Rogaway's "bucket hashing" [25] was the first hash-function family explicitly targeted for fast software implementation; it runs in about 1.5–2.5 cycles/byte. Halevi and Krawczyk devised MMH [13], which takes advantage of current CPU trends to hash at about 1.5–3 cycles/byte on modern CPUs.

With methods now in hand which hash so very quickly, one may ask if the hash-design phase of making a fast MAC is complete; after all, three cycles/byte may already be fast enough to keep up with high-speed network traffic. But authenticating information at the rate it is generated or transmitted is not the real goal; the goal is to use the smallest possible fraction of the CPU's cycles (so most of the machine's cycles are available for other work), by the simplest possible hash mechanism, and having the best proven bounds.

### 1.2 Our Contributions

Our work represents the next step in the quest for a practical, secure, high-speed MAC. Here we describe the main contributions associated to its design.

New Hash Function Families And Their Analyses. A hash-function family named NH underlies hashing in UMAC. It is a simplification of the MMH and NMH families described in [13]. It works like this: the message $M$ to hash is regarded as a sequence of an even number $\ell$ of integers, $M = (m_1, \ldots, m_\ell)$, where each $m_i \in \{0, \ldots, 2^w - 1\}$ corresponds to a $w$-bit word (e.g., $w = 16$ or $w = 32$). A particular hash function is named by a sequence of $n \geq \ell$ $w$-bit integers $K = (k_1, \ldots, k_n)$. We compute $\mathsf{NH}_K(M)$ as

$$\left( \sum_{i=1}^{\ell/2} ((m_{2i-1} + k_{2i-1}) \bmod 2^w) \cdot ((m_{2i} + k_{2i}) \bmod 2^w) \right) \bmod 2^{2w} . \qquad (1)$$

The novelty of this method is that all the arithmetic is "arithmetic that computers like to do"—no finite fields or non-trivial modular reductions (as used in previous designs for universal hashing) come into the picture.

Despite the non-linearity of this hash function and despite its being defined using two different rings, $Z/2^w$ and $Z/2^{2w}$, not a finite field, we manage to obtain a tight bound on the collision probability: $2^{-w}$ (or $2^{-w+1}$ when using signed integers). Earlier analyses of related hash-function families had to give up a small constant in the analysis [13]. We give up nothing.

After proving our bounds on NH we extend the method using the "Toeplitz construction," a well-known approach to reduce the error probability without much lengthening of the key [18, 16]. Prior to our work the Toeplitz construction was known to work only for *linear* functions over *fields*. Somewhat surprisingly, we prove that it also works for NH. Our proof again achieves a tight bound for the collision probability. We then make further extensions to handle length-issues, finally arriving at the hash-function family actually used in UMAC.

Complete Specification. Previous work on universal-hash-paradigm MACs dealt with fast hashing but did not address in detail how to embed a fast-to-compute hash function into a *concrete, practical, and fully-analyzed* MAC. For some hash-function constructions (e.g., cryptographic CRCs) this step would be straightforward. But for the fastest hash families it is not, since these have some unpleasant characteristics, including specialized domains, long key-lengths, long output-lengths, or good performance only on very long messages. We show how to overcome such difficulties in a practical way which delivers on the promised speed. We provide a complete specification of UMAC, a ready-to-use MAC, in a separate specification document [7]. The technical difficulties mentioned above are not ignored; to the contrary, they are treated with the same care as the hash family itself. Our construction is fully analyzed, beginning-to-end; what is analyzed is exactly what is specified. This has only been possible by co-developing the specification document and the academic paper.

PRF(HASH, Nonce) Construction. Previous work has assumed that one hashes messages down to some fixed length string and then applies some cryptographic primitive. But using universal hashing to reduce a very long message to a fixed-length one can be complex, require long keys, or reduce the quantitative security. Instead, we reduce the length of the message by some pre-set constant factor, concatenate a sender-generated nonce, and then apply a pseudorandom function (PRF). See Section 5 for details.

Experiments. We have been guided by extensive experimentation, through which we have identified the parameters that influence performance. While any reasonable setting of these parameters should out-perform conventional MACs, the fastest version of UMAC for one platform differs from the fastest version for another platform. We have therefore left UMAC parameterized, allowing specific choices to be fine-tuned to the application or platform at hand. Here and in [7] we consider a few reasonable settings for these parameters.

SIMD Exploitation. Unlike conventional, inherently serial MACs, UMAC is parallelizable, and will have ever-faster implementations as machines offer up increasing amounts of parallelism. Our algorithm and specification were specifically designed to exploit the form of parallelism offered by current and emerging SIMD architectures (Single Instruction Multiple Data). These provide some long registers that can, in certain instructions, be treated as vectors of smaller-sized words. For NH to run well we must quickly multiply $w$-bit numbers ($w = 16$ or $w = 32$) into their $2w$-bit product. Many modern machines let us do this particularly well since we can re-appropriate instructions for vector dot-products that were primarily intended for multimedia applications. For now, our fastest implementation of UMAC runs on a Pentium and makes use of its MMX instructions which treat a 64-bit register as a vector of four 16-bit words.

### 1.3 Related Work

MMH Paper. Halevi and Krawczyk investigated fast universal hashing in [13]. Their MMH construction takes advantage of improving CPU support for integer multiplication, particularly the ability to quickly multiply two 32-bit multiplicands into a 64-bit product. Their paper also describes a (formerly-unpublished) hash-function family of Carter and Wegman, $\mathsf{NMH}^*$, and a variation of it, $\mathsf{NMH}$. Our $\mathsf{NH}$ function, as given by formula (1), is a simplification of $\mathsf{NMH}$. The difference between $\mathsf{NH}$ and $\mathsf{NMH}$ is that $\mathsf{NMH}$ uses an additional modular reduction by a prime close to $2^w$, followed by a reduction modulo $2^w$. These changes simplify implementation, increase speed, and lower the collision probability.

Other Work On Universal Hash MACs. Krawczyk describes a "cryptographic CRC" hash-function family. Shoup later studied the software performance of this construction, and gave several related ones [26]. In [16] one finds the Toeplitz construction used in a context similar to ours. An earlier use of the Toeplitz construction, in a different domain, can be found in [18]. A hash-function family specifically targeted for software was Rogaway's "bucket hashing" [25]. Its peak speed is fast but its long output length makes it suitable only for long messages. Nevelsteen and Preneel give a performance study of several universal hash functions proposed for MACs [19]. Patel and Ramzan give an $\mathsf{MMH}$-variant that can be more efficient than $\mathsf{MMH}$ in certain settings [20]. Bernstein reports he has designed and implemented a polynomial-evaluation style hash-function family that runs in 4.5 Pentium cycles/byte [6]. Other recent work about universal hashing for authentication includes [1, 14].

Other Types Of MACs. One popular MAC is the CBC-MAC [2]. The CBC-MAC was analyzed by [5], and a variant of it was later analyzed in [21]. In [7] we make a PRF using a CBC-MAC variant similar to [21].

MACs have been constructed from cryptographic hash-functions. A few such methods are described in [27, 15], and analysis appears in [22, 23]. An increasingly popular MAC of this cryptographic-hash-function type is HMAC, which is described and analyzed in [3, 12]. In one version of UMAC we suggest using HMAC as the underlying PRF. One can view UMAC as an alternative to HMAC, with UMAC being faster but more complex.

Full Version. The full version of this paper is in [8], and there is an associated specification document [7]. Reference code is also available [17].

## 2 Overview of UMAC

Unlike many MACs, our construction is stateful for the sender: when he wants to authenticate some string *Msg* he must provide as input to UMAC (along with *Msg* and the shared key *Key*) a 64-bit string *Nonce*. The sender must not reuse the nonce under the same MAC key. Typically the nonce would be a counter which the sender increments with each transmitted message.

The UMAC algorithm specifies how the message, key, and nonce determine an *authentication tag*. The sender will need to provide the receiver with the message, nonce, and tag. The receiver can then compute what "should be" the tag for this particular message and nonce, and see if it matches the received tag.

UMAC employs a *subkey generation process* in which the shared (convenient-length) key is mapped into UMAC's internal keys. In typical applications subkey generation is done just once, at the beginning of a communication session dur-

***Subkey generation***:
Using a PRG, map *Key* to $K = K_1K_2 \cdots K_{1024}$, with each $K_i$ a 32-bit word, and to $A$, where $|A| = 512$.

***Hashing the message*** *Msg* ***to*** *HM* $= \mathsf{NHX}_{Key}(Msg)$:
Let *Len* be $|Msg| \bmod 4096$, encoded as a 2-byte string.
Append to *Msg* the minimum number of 0 bits to make $|Msg|$ divisible by 8.
Let $Msg = Msg_1 \parallel Msg_2 \parallel \cdots \parallel Msg_t$ where each $Msg_i$ is 1024 words except for $Msg_t$, which has between 2 and 1024 words.
Let $HM = \mathsf{NH}_K(Msg_1) \parallel \mathsf{NH}_K(Msg_2) \parallel \cdots \parallel \mathsf{NH}_K(Msg_t) \parallel Len$

***Computing the authentication tag***:
The tag is $Tag = \text{HMAC-SHA1}_A\ (HM \parallel Nonce)$

**Fig. 1.** *An illustrative special case of* UMAC. *The algorithm above computes a 160-bit tag given Key, Msg, and Nonce. See the accompanying text for the definition of* NH.

ing which the key does not change, and so subkey-generation is usually not performance-critical.

UMAC depends on a few different *parameters*. We begin by giving a description of UMAC as specialized to one particular setting of these parameters. Then we briefly explore the role of various parameters.

### 2.1 An Illustrative Special Case

The underlying shared key *Key* (which might be, say, 128 bits) is first expanded into internal keys $K$ and $A$, where $K$ is 1024 words (a *word* being 32-bits) and $A$ is 512 bits. How *Key* determines $K$ and $A$ is a rather standard detail (it can be handled using any PRG), and so we omit its description here. See Figure 1. There we refer to the hash function NH, which is applied to each block $Msg_1, \ldots, Msg_t$ of *Msg*. We now define this function. Let $M = Msg_j$ be one of these blocks. Regard $M$ as a sequence $M = M_1 \cdots M_\ell$ of 32-bit words, where $2 \le \ell \le 1024$. The hash function is named by $K = K_1 \cdots K_{1024}$, where $K_i$ is 32 bits. We let $\mathsf{NH}_K(M)$ be the 64-bit string $\mathsf{NH}_K(M) =$

$$(M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) +_{64} \cdots +_{64} (M_{\ell-1} +_{32} K_{\ell-1}) \times_{64} (M_\ell +_{32} K_\ell)$$

where $+_{32}$ is computer addition on 32-bit strings to give their 32-bit sum, $+_{64}$ is computer addition on 64-bit strings to give their 64-bit sum, and $\times_{64}$ is computer multiplication on unsigned 32-bit strings to give their 64-bit product. This description of NH is identical to Equation (1) (for $w = 32$) but emphasizes that all the operations we are performing directly correspond to machine instructions of modern CPUs. See the left-hand side of Figure 2.

Theorem 1 says that NH is $2^{-32}$-universal on strings of equal (and appropriate) length. Combining with Proposition 1 gives that NHX (as described in Figure 1) is $2^{-32}$-universal, but now for any pair of strings. By Theorem 3, if an

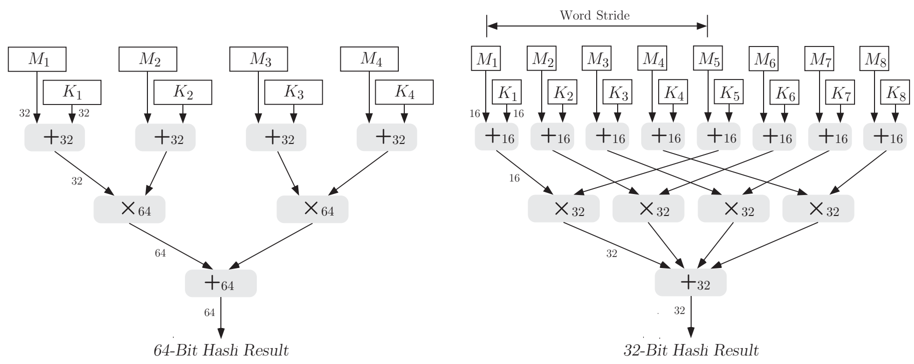



**Fig. 2.** ***Left:*** *The* NH *hash function with a wordsize of* $w = 32$ *bits, as used in* UMAC-STD-30 *and* UMAC-STD-60. ***Right:*** *The "strided" form of* NH, *with wordsize of* $w = 16$ *bits, as used in* UMAC-MMX-30 *and* UMAC-MMX-60.

adversary could forge a message with probability $2^{-32} + \delta$ then an adversary of essentially the same computational complexity could break HMAC-SHA1 (as a PRF) with advantage $\delta - 2^{-160}$. Under generally-accepted assumptions about SHA-1 only negligible values of $\delta$ can be achieved by a feasible attacker [3, 4], thus providing essentially a $2^{-32}$ upper bound on the forging probability against this version of UMAC.

### 2.2 UMAC Parameters

The full name of the version of NH just described is $\mathsf{NH}[n, w]$, where $n = 1024$ and $w = 32$: the *wordsize* is $w = 32$ bits and the *blocksize* is $n = 1024$ words. Numbers $n$ and $w$ are two of UMAC's parameters. Let us describe a few others.

Naturally enough, the *pseudorandom function* (PRF) which gets applied to *HM* $\|$ *Nonce* is a parameter. We used HMAC-SHA1 above, but any PRF is allowed. Similarly, a parameter specifies how *Key* gets mapped to $K$ and $A$.

The universal hashing used in our example had collision probability $2^{-32}$. We make provisions for lowering this. To square the collision probability one could hash the message twice, using independent hash keys, and concatenate the results. But an optimization in UMAC is that the two keys that are used are not independent; rather, one key is the "shift" of the other, with a few new words coming in. This is the well-known "Toeplitz construction." We prove in Theorem 2 that, for NH, the probability still drops according to the square.

In our example we used a long subkey $K$—it had 4096 bytes. To get good compression with a shorter subkey we can use two-level (2L) hashing. If a hash key of length $n_1$ gives compression ratio $\lambda_1$ and a hash key of length $n_2$ gives compression ratio $\lambda_2$ then using two levels of hashing gives compression ratio $\lambda_1\lambda_2$ with key size $\ell_1 + \ell_2$. Our specification allows for this. In fact, we allow 2L hashing in which the Toeplitz shift is applied at each level. It turns out that this only loses a factor of two in the collision probability. The analysis is rather complex, and is omitted.

To accommodate SIMD architectures we allow slight adjustments in indexing. For example, to use the MMX instructions of the Pentium processor, instead of multiplying $(M_1 +_{16} K_1)$ by $(M_2 +_{16} K_2)$ and $(M_3 +_{16} K_3)$ by $(M_4 +_{16} K_4)$, we compute $(M_1 +_{16} K_1) \times_{32} (M_5 +_{16} K_5) +_{32} (M_2 +_{16} K_2) \times_{32} (M_6 +_{16} K_6) +_{32} \cdots$. There are MMX instructions which treat each of two 64-bit registers as four 16-bit words, corresponding words of which can be added or multiplied to give four 16-bit sums or four 32-bit products. So reading $M_1 \| M_2 \| M_3 \| M_4$ into one MMX register and $M_5 \| M_6 \| M_7 \| M_8$ into another we are well-positioned to multiply $M_1 +_{16} K_1$ by $M_5 +_{16} K_5$, not $M_2 +_{16} K_2$. See Figure 2.

There are a few more parameters. The *sign* parameter indicates whether the arithmetic operation $\times_{64}$ is carried out thinking of the strings as unsigned (non-negative) or signed (twos-complement) integers. The MMX instructions mentioned above only operate on signed values, as does Java. If the input message is sufficiently short there is no speed savings to be had by hashing it with $\mathsf{NH}$. The *min-length-to-hash* specifies the minimum-length message which should be hashed before being passed to the PRF. Finally, an *endian* parameter indicates if the MAC should favor big-endian or little-endian computation.

Named Parameter Sets. In [7] we suggest some settings for the vector of parameters, giving rise to UMAC-STD-30, UMAC-STD-60, UMAC-MMX-30, and UMAC-MMX-60. Here we summarize their salient features.

UMAC-STD-30 and UMAC-STD-60 use a wordsize of $w = 32$ bits. They employ 2L hashing with a compression factor of 32 followed by a compression factor of 16. This corresponds to a subkey $K$ of about 400 Bytes. They employ HMAC-SHA1 as the underlying PRF. They use signed arithmetic. The difference between UMAC-STD-30 and UMAC-STD-60 is the collision bound (and therefore forgery bound): $2^{-30}$ and $2^{-60}$, respectively, which are achieved by hashing either once or twice (the latter using a Toeplitz-shifted key). These two versions of UMAC perform well on a wide range of contemporary processors.

UMAC-MMX-30 and UMAC-MMX-60 are well-suited to exploit the SIMD-parallelism available in the MMX instruction set of Intel processors. They use wordsize $w = 16$ bits. Hashing is accomplished with a single-level scheme and a hash key of about 4 KBytes, which yields the same overall compression ratio as the 2L scheme used in the UMAC-STD variants. These MACs use the CBC-MAC of a software-efficient block cipher as the basis of the underlying PRF. Our tests were performed using the block cipher RC6 [24]. Arithmetic is again signed. The difference between UMAC-MMX-30 and UMAC-MMX-60 is the maximal forgery probability: $2^{-30}$ and $2^{-60}$, respectively.

## 3 The NH Hash Family

Recall that $\mathsf{NH}$ is not, by itself, the hash-function family which UMAC uses, but the basic building block from which we construct UMAC's hash. After some brief preliminaries we define and analyze $\mathsf{NH}$.

### 3.1 Preliminaries

FUNCTION FAMILIES. A *family of functions* (with domain $A \subseteq \{0,1\}^*$ and range $B \subseteq \{0,1\}^*$) is a set of functions $\mathsf{H} = \{h : A \to B\}$ endowed with some distribution. When we write $h \leftarrow \mathsf{H}$ we mean to choose a random function $h \in \mathsf{H}$ according to this distribution. A family of functions is also called a *family of hash functions* or a *hash-function family*.

Usually we specify a family of functions $\mathsf{H}$ by specifying some finite set of strings, $\mathsf{Key}$, and explaining how each string $K \in \mathsf{Key}$ names some function $\mathsf{H}_K \in \mathsf{H}$. We may then think of $\mathsf{H}$ not as a set of functions from $A$ to $B$ but as a single function $\mathsf{H} : \mathsf{Key} \times A \to B$, whose first argument we write as a subscript. A random element $h \in \mathsf{H}$ is determined by selecting uniformly at random a string $K \in \mathsf{Key}$ and setting $h = \mathsf{H}_K$.

UNIVERSAL HASHING. We are interested in hash-function families in which "collisions" (when $h(M) = h(M')$ for distinct $M, M'$) are infrequent:

**Definition 1.** *Let $\mathsf{H} = \{h : A \to B\}$ be a family of hash functions and let $\epsilon \geq 0$ be a real number. We say that $\mathsf{H}$ is $\epsilon$-universal, denoted* $\epsilon$-AU*, if for all distinct $M, M' \in A$, we have that $\Pr_{h \leftarrow \mathsf{H}}[h(M) = h(M')] \leq \epsilon$. We say that $\mathsf{H}$ is $\epsilon$-universal on equal-length strings if for all distinct, equal-length strings $M, M' \in A$, we have that $\Pr_{h \leftarrow \mathsf{H}}[h(M) = h(M')] \leq \epsilon$.* ▮

### 3.2 Definition of NH

Fix an even $n \geq 2$ (the "blocksize") and a number $w \geq 1$ (the "wordsize"). We define the family of functions $\mathsf{NH}[n,w]$ as follows. The domain is $A = \{0,1\}^{2w} \cup \{0,1\}^{4w} \cup \cdots \cup \{0,1\}^{nw}$ and the range is $B = \{0,1\}^{2w}$. Each function in $\mathsf{NH}[n,w]$ is named by an $nw$-bit string $K$; a random function in $\mathsf{NH}[n,w]$ is given by a random $nw$-bit string $K$. We write the function indicated by $K$ as $\mathsf{NH}_K(\cdot)$.

Let $U_w$ and $U_{2w}$ represent the sets $\{0,\ldots,2^w-1\}$ and $\{0,\ldots,2^{2w}-1\}$, respectively. Arithmetic done modulo $2^w$ returns a result in $U_w$; arithmetic done modulo $2^{2w}$ returns a result in $U_{2w}$. We overload the notation introduced in Section 2.1: for integers $x, y$ let $(x +_w y)$ denote $(x+y) \bmod 2^w$. (Earlier this was an operator from strings to strings, but with analogous semantics.)

Let $M \in A$ and denote $M = M_1 \cdots M_\ell$, where $|M_1| = \cdots = |M_\ell| = w$. Similarly, let $K \in \{0,1\}^{nw}$ and denote $K = K_1 \cdots K_n$, where $|K_1| = \cdots = |K_n| = w$. Then $\mathsf{NH}_K(M)$ is defined as

$$\mathsf{NH}_K(M) \;=\; \sum_{i=1}^{\ell/2} (k_{2i-1} +_w m_{2i-1}) \cdot (k_{2i} +_w m_{2i}) \bmod 2^{2w}$$

where $m_i \in U_w$ is the number that $M_i$ represents (as an unsigned integer), where $k_i \in U_w$ is the number that $K_i$ represents (as an unsigned integer), and the right-hand side of the above equation is understood to name the (unique) $2w$-bit string which represents (as an unsigned integer) the $U_{2w}$-valued integer result. Henceforth we shall refrain from explicitly converting from strings to integers and back, leaving this to the reader's good sense. (We comment that for everything we do, one could use any bijective map from $\{0,1\}^w$ to $U_w$, and any bijective map from $U_{2w}$ to $\{0,1\}^{2w}$.) When the values of $n$ and $w$ are clear from the context, we write $\mathsf{NH}$ instead of $\mathsf{NH}[n,w]$.

### 3.3 Analysis

The following theorem bounds the collision probability of $\mathsf{NH}$. In the full version of this paper [8] we also treat the signed case, yielding a bound of $2^{t(-w+1)}$-AU.

**Theorem 1.** *For any even $n \geq 2$ and $w \geq 1$, $\mathsf{NH}[n,w]$ is $2^{-w}$-AU on equal-length strings.*

*Proof.* Let $M, M'$ be distinct members of the domain $A$ with $|M| = |M'|$. We are required to show $\Pr_{K \leftarrow \mathsf{NH}}[\,\mathsf{NH}_K(M) = \mathsf{NH}_K(M')\,] \leq 2^{-w}$. Converting the message and key strings to $n$-vectors of $w$-bit words we invoke the definition of $\mathsf{NH}$ to restate our goal as showing that

$$\Pr\left[\sum_{i=1}^{\ell/2}\ (k_{2i-1} +_w m_{2i-1})(k_{2i} +_w m_{2i}) = \sum_{i=1}^{\ell/2}\ (k_{2i-1} +_w m'_{2i-1})(k_{2i} +_w m'_{2i})\right]$$

is no more than $2^{-w}$ where the probability is taken over uniform choices of $(k_1, \ldots, k_n)$ with each $k_i$ in $U_w$. Above (and for the remainder of the proof) all arithmetic is carried out in $Z/2^{2w}$.

Since $M$ and $M'$ are distinct, $m_i \neq m'_i$ for some $1 \leq i \leq n$. Since addition and multiplication in a ring are commutative, we lose no generality in assuming $m_2 \neq m'_2$. We now prove that for any choice of $k_2, \cdots, k_n$ we have $\Pr_{k_1 \in U_w}\left[(m_1 +_w k_1)(m_2 +_w k_2) + \sum_{i=2}^{\ell/2}\ (m_{2i-1} +_w k_{2i-1})(m_{2i} +_w k_{2i}) = (m'_1 +_w k_1)(m'_2 +_w k_2) + \sum_{i=2}^{\ell/2}\ (m'_{2i-1} +_w k_{2i-1})(m'_{2i} +_w k_{2i})\right] \leq 2^{-w}$, which will imply the theorem. Collecting up the summations, let

$$y = \sum_{i=2}^{\ell/2}\ (m_{2i-1} +_w k_{2i-1})(m_{2i} +_w k_{2i}) \ - \ \sum_{i=2}^{\ell/2}\ (m'_{2i-1} +_w k_{2i-1})(m'_{2i} +_w k_{2i})$$

and let $c = (m_2 +_w k_2)$ and $c' = (m'_2 +_w k_2)$. Note that $c$ and $c'$ are in $U_w$, and since $m_2 \neq m'_2$, we know $c \neq c'$. We rewrite the above probability as

$$\Pr_{k_1 \in U_w}[\,c(m_1 +_w k_1) - c'(m'_1 +_w k_1) + y = 0\,] \leq 2^{-w}.$$

In Lemma 1 below, we prove that there can be at most one $k_1$ in $U_w$ satisfying $c(m_1 +_w k_1) - c'(m'_1 +_w k_1) + y = 0$, yielding the desired bound. ∎

**Lemma 1.** *Let $c$ and $c'$ be distinct values from $U_w$. Then for any $m, m' \in U_w$ and any $y \in U_{2w}$ there exists at most one $k \in U_w$ such that $c(k +_w m) = c'(k +_w m') + y$ in $Z/2^{2w}$.*

*Proof.* We note that it is sufficient to prove the case where $m = 0$. We proceed to therefore prove that for any $c, c', m' \in U_w$ with $c \neq c'$ and any $y \in U_{2w}$ there is at most one $k \in U_w$ such that $kc = (k +_w m')c' + y$ in $Z/2^{2w}$. Since $k, m' < 2^w$, we know that $(k +_w m')$ is either $k + m'$ or $k + m' - 2^w$, depending on whether $k + m' < 2^w$ or $k + m' \geq 2^w$ respectively. So now we have

$$k(c - c') = m'c' + y \text{ and } k < 2^w - m' \tag{2}$$

$$k(c - c') = (m' - 2^w)c' + y \text{ and } k \geq 2^w - m' \tag{3}$$

Lemma 2, presented next, shows that there is at most one solution to each of the equations above. The remainder of the proof is devoted to showing there cannot exist $k = k_1 \in U_w$ satisfying (2) and $k = k_2 \in U_w$ satisfying (3) in $Z/2^{2w}$. Suppose such a $k_1$ and $k_2$ did exist. Then we have $k_1 < 2^w - m'$ with $k_1(c - c') = m'c' + y$ and $k_2 \geq 2^w - m'$ with $k_2(c - c') = (m' - 2^w)c' + y$. Subtracting the former from the latter yields $(k_2 - k_1)(c' - c) = 2^w c'$. We show that this equation has no solutions in $Z/2^{2w}$. There are cases:

Case 1: $c' > c$. Since both $(k_2 - k_1)$ and $(c' - c)$ are positive and smaller than $2^w$, their product is also positive and smaller than $2^{2w}$. And since $2^w c'$ is also positive and smaller than $2^{2w}$, it is sufficient to show that the equation has no solutions in $Z$. But this is clear, since $(k_2 - k_1) < 2^w$ and $(c' - c) \leq c'$, and so necessarily $(k_2 - k_1)(c' - c) < 2^w c'$.

Case 2: $c' < c$. Here we show $(k_2 - k_1)(c - c') = -2^w c'$ has no solutions in $Z/2^{2w}$. As before, we convert to $Z$, to yield $(k_2 - k_1)(c - c') = 2^{2w} - 2^w c'$. But again $(k_2 - k_1) < 2^w$ and $(c - c') < (2^w - c')$, so $(k_2 - k_1)(c - c') < 2^w(2^w - c') = 2^{2w} - 2^w c'$. ∎

Let $D_w = \{-2^w + 1, \ldots, 2^w - 1\}$ be the values attainable from a difference of any two elements of $U_w$. The following lemma completes the proof.

**Lemma 2.** *Let $x \in D_w$ be nonzero. Then for any $y \in U_{2w}$, there exists at most one $a \in U_w$ such that $ax = y$ in $Z/2^{2w}$.*

*Proof.* Suppose there were two distinct elements $a, a' \in U_w$ such that $ax = y$ and $a'x = y$. Then $ax = a'x$ so $x(a - a') = 0$. Since $x$ is nonzero and $a$ and $a'$ are distinct, the foregoing product is $2^{2w}k$ for nonzero $k$. But $x$ and $(a - a')$ are in $D_w$, and therefore their product is in $\{-2^{2w} + 2^{w+1} - 1, \ldots, 2^{2w} - 2^{w+1} + 1\}$, which contains no multiples of $2^{2w}$ other than 0. ∎

Comments. The above bound is tight; let $M = 0^w 0^w$ and $M' = 1^w 0^w$ and note that any key $K = K_1 K_2$ with $K_2 = 0^w$ causes a collision.

Although we do not require any stronger properties than the above, NH is actually $2^{-w}$-A$\Delta$U under the operation of addition modulo $2^{2w}$. Only trivial modifications to the above proof are required. See [13] for a definition of $\epsilon$-A$\Delta$U.

Several variants of NH fail to preserve collision probability $\epsilon = 2^{-w}$. In particular, replacing the inner addition or the outer addition with bitwise-XOR increases $\epsilon$ substantially. However, removing the inner moduli retains $\epsilon = 2^{-w}$ (but significantly degrades performance).

There is also a version of $\mathsf{NH}[n,w]$, called $\mathsf{NHS}[n,w]$, that uses signed arithmetic. Surprisingly, the signed version of $\mathsf{NH}$ has slightly higher collision probability: the full paper proves a tight bound of $2^{-w+1}$-AU. This helps explain the $2^{-30}$ and $2^{-60}$ forging probabilities for the four UMAC versions named in Section 2.2 and performance-measured in Section 6; all four algorithms use the signed version of $\mathsf{NH}$.

# 4 Extending NH

The hash-function family $\mathsf{NH}$ is not yet suitable for use as a MAC: for one thing, it operates only on strings of "convenient" lengths ($\ell w$-bit strings for even $\ell \le n$). Also, its collision probability may be higher than desired ($2^{-w}$ when one may want $2^{-2w}$ or $2^{-4w}$), and this is guaranteed only for strings of equal length. We remedy these deficiencies in this section.

## 4.1 Reducing Collision Probability: NH-Toeplitz

Should we wish to reduce the collision probability for $\mathsf{NH}$, we have a few options. Increasing the wordsize $w$ yields an improvement, but architectural characteristics dictate the natural values for $w$. Another well-known technique is to apply several random members of our hash-function family to the message, and concatenate the results. If we concatenate the results from, say, four independent instances of the hash function, the collision probability drops from $2^{-w}$ to $2^{-4w}$. However this solution requires four times as much key material. A superior (and well-known) idea is to use the Toeplitz-extension of our hash-function families: given one key we "left shift" to get the "next" key and hash again. For example, to reduce the collision probability to $2^{-64}$ for $\mathsf{NH}[n,16]$, we choose a single key $K = (K_1, \ldots, K_{n+6})$ and hash with the four derived keys $(K_{1+2i}, \ldots, K_{n+2i})$ where $0 \le i \le 3$. This trick not only saves key material, but it can also improve performance by reducing memory accesses, increasing locality of memory references and increasing parallelism.

Since these keys are related, it is not clear that the collision probability indeed drops to the desired value of $2^{-64}$. Although there are established results which yield this bound (e.g., [18]), they only apply to linear hashing schemes over fields. Instead, $\mathsf{NH}$ is non-linear and operates over a combination of rings ($Z/2^w$ and $Z/2^{2w}$). In Theorem 2 we prove that the Toeplitz construction nonetheless achieves the desired bound in the case of $\mathsf{NH}$.

We define the hash-function family $\mathsf{NH}^\mathsf{T}[n,w,t]$ ("Toeplitz-$\mathsf{NH}$") as follows. Fix an even $n \ge 2$, $w \ge 1$, and $t \ge 1$ (the "Toeplitz iteration count"). The domain $A = \{0,1\}^{2w} \cup \{0,1\}^{4w} \cup \cdots \cup \{0,1\}^{nw}$ remains as it was for $\mathsf{NH}$, but the range is now $B = \{0,1\}^{2wt}$. A function in $\mathsf{NH}^\mathsf{T}[n,w,t]$ is named by a string $K$ of $w(n+2(t-1))$ bits. Let $K = K_1 \parallel \cdots \parallel K_{n+2(t-1)}$ (where each $K_i$ is a $w$-bit word), and let the notation $K_{i..j}$ represent $K_i \parallel \cdots \parallel K_j$. Then for any $M \in A$ we define $\mathsf{NH}^\mathsf{T}_K(M)$ as

$$\mathsf{NH}^\mathsf{T}_K(M) = \mathsf{NH}_{K_{1..n}}(M) \parallel \mathsf{NH}_{K_{3..n+2}}(M) \parallel \cdots \parallel \mathsf{NH}_{K_{(2t-1)..(n+2t-2)}}(M).$$

When clear from context we write $\mathsf{NH}^\mathsf{T}$ instead of $\mathsf{NH}^\mathsf{T}[n, w, t]$.

The following shows that $\mathsf{NH}^\mathsf{T}$ enjoys the best bound that one could hope for. The proof is in the full version [8].

**Theorem 2.** *For any $w, t \geq 1$ and any even $n \geq 2$, $\mathsf{NH}^\mathsf{T}[n, w, t]$ is $2^{-wt}$-AU on equal-length strings.*

### 4.2 Padding, Concatenation, and Length Annotation

With $\mathsf{NH}^\mathsf{T}$ we can decrease the collision probability to any desired level but we still face the problem that this function operates only on strings of "convenient" length, and that it guarantees this low collision probability only for equal-length strings. We solve these problems in a generic manner, with a combination of padding, concatenation, and length annotation.

Mechanism. Let $\mathsf{H} : \{A \to B\}$ be a family of hash functions where functions in $\mathsf{H}$ are defined only for particular input lengths, up to some maximum, and all the hash functions have a fixed output length. Formally, the domain is $A = \bigcup_{i \in I} \{0,1\}^i$ for some finite nonempty index set $I \subseteq \mathsf{N}$ and the range is $B = \{0,1\}^\beta$, where $\beta$ is some positive integer. Let $a$ (the "blocksize") be the length of the longest string in $A$ and let $\alpha \geq \lceil \lg_2 a \rceil$ be large enough to describe $|M| \bmod a$. Then we define $\mathsf{H}^* = \{h^* : \{0,1\}^* \to \{0,1\}^*\}$ as follows.

`function` $h^*(\mathit{Msg})$
`if` $M = \lambda$ `then return` $0^\alpha$
View $\mathit{Msg}$ as a sequence of "blocks", $\mathit{Msg} = \mathit{Msg}_1 \| \cdots \| \mathit{Msg}_t$,
with $|\mathit{Msg}_j| = a$ for all $1 \leq j < t$, and $1 \leq |\mathit{Msg}_t| \leq a$
Let $\mathit{Len}$ be an $\alpha$-bit string that encodes $|\mathit{Msg}| \bmod a$
Let $i \geq 0$ be the least number such that $\mathit{Msg}_t \| 0^i \in A$
$\mathit{Msg}_t = \mathit{Msg}_t \| 0^i$
`return` $h(\mathit{Msg}_1) \| \cdots \| h(\mathit{Msg}_t) \| \mathit{Len}$

Analysis. The following proposition indicates that we have correctly extended $\mathsf{H}$ to $\mathsf{H}^*$. The straightforward proof is in the full paper [8].

**Proposition 1.** *Let $I \subseteq \mathsf{N}$ be a nonempty finite set, let $\beta \geq 1$ be a number, and let $\mathsf{H} = \{h : \bigcup_{i \in I} \{0,1\}^i \to \{0,1\}^\beta\}$ be a family of hash functions. Let $\mathsf{H}^* = \{h^* : \{0,1\}^* \to \{0,1\}^*\}$ be the family of hash functions obtained from $\mathsf{H}$ as described above. Suppose $\mathsf{H}$ is $\epsilon$-AU on strings of equal length. Then $\mathsf{H}^*$ is $\epsilon$-AU (across all strings).*

## 5 From Hash to MAC

In this section we describe a way to make a secure MAC from an $\epsilon$-AU family of hash functions (with small $\epsilon$) and a secure pseudorandom function (PRF).

Definition Of The PRF(HASH, Nonce) Construction. We use a family of (hash) functions $\mathsf{H} = \{h : \{0,1\}^* \rightarrow \{0,1\}^*\}$ and a family of (random or pseudorandom) functions $\mathsf{F} = \{f : \{0,1\}^* \rightarrow \{0,1\}^\tau\}$. These are parameters of the construction. We also fix a set $\mathsf{Nonce} = \{0,1\}^\eta$ and an "encoding scheme" $\langle \cdot, \cdot \rangle$. The encoding scheme is a linear-time computable function that maps a string $HM \in \{0,1\}^*$ and $Nonce \in \mathsf{Nonce}$ into a string $\langle HM, Nonce \rangle$ of length $|HM| + |Nonce| + O(1)$ from which, again in linear time, one can recover $HM$ and $Nonce$. The MAC scheme $\mathsf{UMAC}[\mathsf{H}, \mathsf{F}] = (\text{Key}, \text{Tag})$ is defined as:

| | |
|---|---|
| `function` Key () | `function` $\text{Tag}_{(f,h)}$ $(M,\ Nonce)$ |
| $f \leftarrow \mathsf{F};\quad h \leftarrow \mathsf{H}$ | |
| `return` $(f, h)$ | `return` $f\ (\ \langle h(M),\ Nonce \rangle\ )$ |

The keyspace for this MAC is $\mathsf{Key} = \mathsf{H} \times \mathsf{F}$; that is, a random key for the MAC is a random hash function $h \in \mathsf{H}$ together with a random function $f \in \mathsf{F}$. Here we have regarded a MAC scheme as a pair consisting of a key-generation algorithm and a tag-generation algorithm. The formalization is in [8].

We point out that the use of the nonce does not, by itself, address the question of replay detection. Our definition of MAC security [8] speaks to an adversary's inability to produce a new $(M, Nonce, Tag)$ tuple, but is silent on the question of when the verifier should regard an $(M, Nonce, Tag)$ tuple as valid. Certainly $Tag$ should be the correct tag for the given message and nonce, but the verifier may demand more. In particular, the verifier may wish to reject messages for which the nonce was used before. If so, replay attacks will be thwarted. Of course the verifier will need to maintain state to do this. If the sender and receiver use a counter for detecting replays this same counter can be UMAC's nonce.

Analysis. We begin with the information-theoretic version of the scheme. The proof and relevant definitions are in the full paper [8]. Here it suffices to indicate that $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F)$ measures the chance that $F$ forges under the MAC scheme $\Sigma$.

**Lemma 3.** *Let $\epsilon \geq 0$ be a real number and let $\mathsf{H} = \{h : \{0,1\}^* \rightarrow \{0,1\}^*\}$ be an $\epsilon$-AU family of hash functions. Let $\tau \geq 1$ be a number and let $\Sigma = \mathsf{UMAC}[\mathsf{H}, \mathsf{Rand}(\tau)]$ be the MAC scheme described above. Then for every adversary $F$ we have that $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F) \leq \epsilon + 2^{-\tau}$.*

In the usual way we can extend the above information-theoretic result to the complexity-theoretic setting. Roughly, we prove that if the hash-function family is $\epsilon$-AU and no reasonable adversary can distinguish the PRF from a truly random function with advantage exceeding $\delta$ then no reasonable adversary can break the resulting MAC scheme with probability exceeding $\epsilon + \delta$.

The theorem refers to $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t, q, \mu)$, which is the maximal chance of forging by an adversary that runs in time $t$ and asks $q$ queries, these totaling $\mu$ bits. And it refers to $\mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(t', q', \mu')$, which measures the maximal possible advantage in distinguishing $\mathsf{F}$ from a random function among adversaries that run in time $t'$ and asks $q'$ queries, these totaling $\mu'$ bits. If $\mathsf{H}$ is a family of hash functions then $\mathsf{Time}_{\mathsf{H}}$ is an amount of time adequate to compute a representation for a random $h \leftarrow \mathsf{H}$, while $\mathsf{Time}_h(\mu)$ is an amount of time adequate to evaluate $h$ on strings whose lengths total $\mu$ bits. The proof of the following is standard.

**Theorem 3.** *Let $\epsilon \geq 0$ be a real number, let $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ be an $\epsilon$-AU family of hash functions, let $\tau \geq 1$ be a number and let $F : \{0,1\}^\alpha \times \{0,1\}^* \to \{0,1\}^\tau$ be a PRF. Let $\Sigma = \mathsf{UMAC[H,F]}$ be the MAC scheme described above. Then*

$$\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t,q,\mu) \;\leq\; \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(t',q',\mu') + \epsilon + 2^{-\tau}$$

*where* $t' = t + \mathsf{Time}_{\mathsf{H}} + \mathsf{Time}_h(\mu) + O(\mu)$ *and* $q' = q + 1$ *and* $\mu' = \mu + O(q)$. ▮

Discussion. The use of the nonce is important for getting a quantitatively desirable security bound. Let $\overline{\mathsf{UMAC}}[\mathsf{H},\mathsf{F}]$ be the scheme which is the same as UMAC, except that the PRF is applied directly to *HM*, rather than to $\langle HM, Nonce\rangle$. Then the analog of Lemma 3 would say: fix a number $\tau \geq 1$, let $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ be an $\epsilon$-AU family of hash functions, and let $\Sigma = \overline{\mathsf{UMAC}}[\mathsf{H}, \mathsf{Rand}(\tau)]$. Then for every adversary $F$, $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F) \leq q^2\epsilon + 2^{-\tau}$. This is a far cry from our earlier bound of $\epsilon + 2^{-\tau}$. And the problem is not with the analysis, but with the scheme itself: if one asks $\epsilon^{-1/2}$ oracle queries of $\overline{\mathsf{UMAC}}$ then, by the birthday bound, there is indeed a good chance to find distinct messages, $M_1$ and $M_2$, which yield the same authentication tag. This is crucial information about the hash function which has now been leaked.

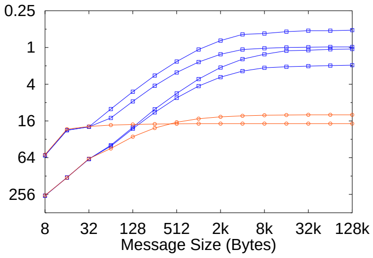
Compared to the original suggestion of [28], where one encrypts the hash of the message by XOR-ing with a one-time pad, we require a weaker assumption about the hash-function family $\mathsf{H}$: it need only be $\epsilon$-AU. (The original approach of [28] needs of $\mathsf{H}$ the stronger property of being "XOR almost-universal" [16].) Furthermore, it is no problem for us that the range of $h \in \mathsf{H}$ has strings of unbounded length, while the hash functions used for [28] should have fixed-length output. On the other hand, our cryptographic tool is potentially stronger than what the complexity-theoretic version of [28] requires: we need a PRF over the domain $\mathsf{Encoding}$ of possible $\langle HM, Nonce\rangle$ encodings.

Compared to the PRF(HASH, Nonce) method, PRF(Nonce) $\oplus$ HASH does have some advantages. One would need a final layer of almost-XOR-universal hashing, like a CRC hash [16, 26], or an $\mathsf{NMH}$ hash [13] with a conceptually infinite (PRG-generated) key. We are still investigating such possibilities.

## 6 Performance

We implemented the four flavors of UMAC named in Section 2.2 on three different platforms: a 350 MHz Intel Pentium II, a 200 MHz IBM/Motorola PowerPC 604e, and a 300 MHz DEC Alpha 21164. The code was written mostly in C, with a few functions done in assembly. For the Pentium II we wrote assembly for RC6, SHA-1, and the first-level NH hashes. For the PowerPC we did assembly for just the first-level NH hashes. In both cases, the number of lines of assembly written was small: about 80 lines.

For each combination of options we determined the scheme's throughput on variously sized messages, eight bytes through 512 KBytes. The experimental setup ensured that messages resided in level-1 cache regardless of their length.




| | Pentium II | | PowerPC | | Alpha | |
|---|---|---|---|---|---|---|
| UMAC-STD-30 | 2.79 | (1.03) | 2.28 | (1.26) | 1.79 | (1.60) |
| UMAC-STD-60 | 1.49 | (1.93) | 1.81 | (1.58) | 1.03 | (2.78) |
| UMAC-MMX-30 | 5.66 | (0.51) | 1.19 | (2.40) | 0.571 | (5.02) |
| UMAC-MMX-60 | 2.94 | (0.98) | 0.684 | (4.19) | 0.287 | (10.0) |
| CBC-MAC-RC6 | 0.162 | (17.7) | 0.210 | (13.7) | 0.068 | (42.5) |
| HMAC-SHA1 | 0.227 | (12.6) | 0.228 | (12.6) | 0.117 | (24.5) |

**Fig. 3.** UMAC Performance. ***Left:*** *Performance over various message lengths on a Pentium II, measured in machine cycles/byte. The lines in the graph correspond to the following MACs (beginning at the top-right and moving downward):* UMAC-MMX-30, UMAC-MMX-60, UMAC-STD-30, UMAC-STD-60, HMAC-SHA1 *and* CBC-MAC-RC6. ***Right:*** *Peak performance for three platforms, measured in Gbits/sec (cycles/byte). The Gbits/sec numbers are normalized to 350 MHz.*

For comparison the same tests were run for HMAC-SHA1 [11] and the CBC-MAC of a fast block cipher, RC6 [24].

The graph in Figure 3 shows the throughput of the four versions of UMAC, as well as HMAC-SHA1 and CBC-MAC-RC6, all run on our Pentium II. The table gives peak throughput for the same MACs, but on all three platforms. The performance curves for the Alpha and PowerPC look similar to the Pentium II—they perform better than the reference MACs at around the same message length, and level out at around the same message length.

As the data shows, the MMX versions are much faster than the STD versions on the Pentium. Going from words of $w = 32$ bits to $w = 16$ bits might appear to increase the amount of work needed to get to a given collision bound, but a single MMX instruction can do four 16-bit multiplications and two 32-bit additions. This is more work per instruction than the corresponding 32-bit instructions.

UMAC-STD uses only one-tenth as much hash key as UMAC-MMX to achieve the same compression ratio. The penalty for such 2L hashing ranges from 8% on small messages to 15% on long ones. To lower the amount of key material we could have used a one-level hash with a smaller compression ratio, but experiments show this is much less effective: relative to UMAC-MMX-60, which uses about 4 KBytes of hash key, a 2 KBytes scheme goes 85% as fast, a 1 KByte scheme goes 66% as fast, and a 512 bytes scheme goes 47% as fast.

Another experiment replaced the NH hash function used in UMAC-STD-30 by MMH [13]. Peak performance dropped by 24%. We replaced the NH hash function of UMAC-MMX-30 by a 16-bit MMH and performance dropped by 5%.

We made a $2^{-15}$-forgery probability UMAC-MMX-15 in the natural way, which ran in 0.32 cycles/bytes on our Pentium II.

We tried UMAC-STD-30 and UMAC-STD-60 on a Pentium processor which lacked MMX. Peak speeds were 2.2 cycles/byte and 4.3 cycles/byte—still well ahead of methods like HMAC-SHA1.

## 7 Directions

An interesting possibility (suggested to us by researchers at NAI Labs—see acknowledgments) is to restructure UMAC so that a receiver can verify a tag to various forgery probabilities—e.g., changing UMAC-MMX-60 to allow tags to be verified, at increasing cost, to forging probabilities of $2^{-15}$, $2^{-30}$, $2^{-45}$, or $2^{-60}$. Such a feature is particularly attractive for authenticating broadcasts to receivers of different security policies or computational capabilities.

## Acknowledgments

Thanks to Mihir Bellare and the CRYPTO '99 Committee for their suggestions. Thanks to Dave Balenson and Dave Carman (of NAI Labs, the security research division of Network Associates; DARPA F30602-98-C-0215) for the idea mentioned in Section 7. Thanks to them, and to Bill Aiello, for pointing out the usefulness of high-forgery-probability MACs, such as UMAC-MMX-15, to applications like telephony and multimedia.

Rogaway, Black, and Krovetz were supported by Rogaway's NSF CAREER Award CCR-962540, and by MICRO grants 97-150 and 98-129, funded by RSA Data Security, Inc., and ORINCON Corporation. Much of Rogaway's work was carried out while on sabbatical at Chiang Mai University, under Prof. Krisorn Jittorntrum and Prof. Darunee Smawatakul.

## References

1. Afanassiev, V., Gehrmann, C., and Smeets, B. Fast message authentication using efficient polynomial evaluation. In *Proceedings of the 4th Workshop on Fast Software Encryption* (1997), vol. 1267, Springer-Verlag, pp. 190–204.
2. Ansi X9.9. American national standard — Financial institution message authentication (wholesale). ASC X9 Secretariat – American Bankers Association, 1986.
3. Bellare, M., Canetti, R., and Krawczyk, H. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.
4. Bellare, M., Canetti, R., and Krawczyk, H. Pseudorandom functions revisited: The cascade construction. In *37th Annual Symposium on Foundations of Computer Science* (1996), IEEE Computer Society, pp. 514–523.
5. Bellare, M., Kilian, J., and Rogaway, P. The security of cipher block chaining. In *Advances in Cryptology – CRYPTO '94* (1994), vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 341–358.
6. Bernstein, D. Guaranteed message authentication faster than MD5. Unpublished manuscript, 1999.
7. Black, J., Halevi, S., Hevia, A., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC — Message authentication code using universal hashing. Unpublished specification, `www.cs.ucdavis.edu/~rogaway/umac`, 1999.
8. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), Lecture Notes in Computer Science, Springer-Verlag. Full version of this paper, available at `www.cs.ucdavis.edu/~rogaway/umac`.

9. Brassard, G. On computationally secure authentication tags requiring short secret shared keys. In *Advances in Cryptology – CRYPTO '82* (1983), Springer-Verlag, pp. 79–86.
10. Carter, L., and Wegman, M. Universal hash functions. *J. of Computer and System Sciences*, 18 (1979), 143–154.
11. FIPS 180-1. Secure hash standard. NIST, US Dept. of Commerce, 1995.
12. H. Krawczyk, M. B., and Canetti, R. HMAC: Keyed hashing for message authentication. IETF RFC-2104, 1997.
13. Halevi, S., and Krawczyk, H. MMH: Software message authentication in the Gbit/second rates. In *Proceedings of the 4th Workshop on Fast Software Encryption* (1997), vol. 1267, Springer-Verlag, pp. 172–189.
14. Johansson, T. Bucket hashing with small key size. In *Advances in Cryptology – EUROCRYPT '97* (1997), Lecture Notes in Computer Science, Springer-Verlag.
15. Kaliski, B., and Robshaw, M. Message authentication with MD5, 1995. Technical newsletter of RSA Laboratories.
16. Krawczyk, H. LFSR-based hashing and authentication. In *Advances in Cryptology – CRYPTO '94* (1994), vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 129–139.
17. Krovetz, T. UMAC reference code (in ANSI C with Pentium assembly). Available from `www.cs.ucdavis.edu/~rogaway/umac`, 1999.
18. Mansour, Y., Nissan, N., and Tiwari, P. The computational complexity of universal hashing. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing* (1990), ACM Press, pp. 235–243.
19. Nevelsteen, W., and Preneel, B. Software performance of universal hash functions. In *Advances in Cryptology — EUROCRYPT '99* (1999), vol. 1592 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 24–41.
20. Patel, S., and Ramzan, Z. Square hash: Fast message authentication via optimized universal hash functions. In *Advances in Cryptology — CRYPTO '99* (1999), Lecture Notes in Computer Science, Springer-Verlag.
21. Petrank, E., and Rackoff, C. CBC MAC for real-time data sources. Manuscript 97-10 in `http://philby.ucsd.edu/cryptolib.html`, 1997.
22. Preneel, B., and van Oorschot, P. MDx-MAC and building fast MACs from hash functions. In *Advances in Cryptology — CRYPTO '95* (1995), vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–14.
23. Preneel, B., and van Oorschot, P. On the security of two MAC algorithms. In *Advances in Cryptology — EUROCRYPT '96* (1996), vol. 1070 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 19–32.
24. Rivest, R., Robshaw, M., Sidney, R., and Yin, Y. The RC6 block cipher. Available from `http://theory.lcs.mit.edu/~rivest/publications.html`, 1998.
25. Rogaway, P. Bucket hashing and its application to fast message authentication. In *Advances in Cryptology – CRYPTO '95* (1995), vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 313–328.
26. Shoup, V. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology – CRYPTO '96* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 74–85.
27. Tsudik, G. Message authentication with one-way hash functions. In *Proceedings of Infocom '92* (1992), IEEE Press.
28. Wegman, M., and Carter, L. New hash functions and their use in authentication and set equality. In *J. of Comp. and System Sciences* (1981), vol. 22, pp. 265–279.