Software-Optimized Universal Hashing and Message Authentication

by

THEODORE D. KROVETZ

B.S. (Stanford University) 1988
M.Sc. (University of Oxford) 1990

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Phillip Rogaway, Chair

Daniel Gusfield

Daniel Boneh

Committee in Charge

September 2000

To Emma, Hannah and Avery,

the milk in my coconut.

Abstract

We describe a message authentication algorithm, UMAC, which can authenticate messages in software, on contemporary machines, at a rate faster than one Pentium cycle per byte of authenticated message. This is roughly an order of magnitude faster than current practice (e.g., HMAC-SHA1 or CBC-MAC-RC6). We explore three designs, all variations on the style of Wegman and Carter [39], which achieve such speeds. As a first step, all three designs utilize NH, a universal hash-function family which allows effective exploitation of SIMD parallelism. Unlike conventional, inherently serial authentication algorithms, UMAC will have ever-faster implementation speeds as machines offer up increasing amounts of parallelism. UMAC was designed and analyzed in a "provable-security" framework which has allowed its design to be minimal and therefore conducive to high-speeds.

Along with the description of UMAC, we develop some new concepts. We introduce "verifier-selectable assurance" for message authentication, where the receiver of a message and authentication tag can verify the message up to a level of their own choosing, trading computation time for assurance level. Also introduced is the notion of "variationally-universal" hash-function families, a relaxation of strongly-universal which measures universal hash-function families in terms of variational distance from the uniform distribution. Also described is an almost-universal hash-function family based on polynomial evaluation that hashes arbitrary-length messages into short strings using short keys.

Contents

Acknowledgements

I have lived a blessed life, and for that I owe many thanks to many people. Ever supportive and encouraging, my wife and children, parents and siblings, have been a constant source of love and affection. From an early age my parents have taught me the value of education and have provided me with a supportive environment in which to thrive. I now have the great fortune to offer those same lessons, with my marvelous wife Emma, to our own children, Hannah and Avery. I believe most thoroughly that I could not have accomplished my goals without such a wonderful family, and for their help I am thankful.

Having chosen to attend the University of California at Davis for mostly logistical reasons, I stumbled into a situation that could not have been better. As a new graduate student, I had the notion that I would research computer architecture. But luckily I was assigned to be Phillip Rogaway's teaching assistant during my first quarter at the University, and he quickly changed my mind. He opened my eyes to the beautiful world of computational theory and cryptography, and I was soon over my head in that world. In the ensuing months and years, I have learned a great deal from Phil, and not only academically. Phil's generosity of time and spirit has been inspirational. I thank him for always being available and for working so hard on helping me make sense of it all. I was also lucky to have been paired in my research with John Black, one of the most clever and pleasant people I have yet to meet. Although I did not choose to come to U.C. Davis because of its academic environment, I might as well have. My teachers have been enthusiastic, knowledgeable and available. Davis has been a wonderful place to be.

None of my research was done alone, and I have had the great fortune to work with some of the best cryptographers around. I appreciate very much their patience and brilliance: Mihir Bellare, John Black, Shai Halevi, Hugo Krawczyk and, of course, Phillip Rogaway.

Finally, thanks are owed to Dan Boneh, Dan Gusfield and Phillip Rogaway for taking the time to review this tome and offer valuable feedback.

This research was partially supported by Phillip Rogaway's CAREER award CCR-962540, and by MICRO grants 97-150, 98-129, and 99-103 funded by RSA Data Security Inc., ORINCON Corporation, Certicom Corporation, and the State of California. Also supporting this research was the GAANN fellowship program underwritten by the U.S. Department of Education and participating universities, including U.C. Davis.

Chapter 1

Introduction and Background

One of the most common uses of cryptography is message authentication. When Alice receives a message purportedly to be from Bob, how can she know that the message received is actually from Bob and has not been tampered with during transport? Reliable systems exist for ensuring the authenticity of messages written and delivered on paper — these messages can be sealed with a signature in a tamper-resistant envelope. But what about electronic messages delivered by digital computer networks?

Each day billions of messages, from IP packets at the network level to bulk file-transfers at the application level, are delivered by the Internet and other networks. The open nature of many of these networks, however, leave them vulnerable to mischief. Clever adversaries have the ability to read any message being transported across the network and to delete, interject or alter any desired message. To combat these adversaries, cryptography can be used to ensure message privacy and authenticity. Ensuring that a message remains private is done with the use of *encryption*. An encrypted message intercepted by a malicious adversary should yield him no information other than, perhaps, the length of the message. Ensuring message authenticity is usually done by an algorithm called a *message authentication code*

(MAC). A message that has been verified by a MAC will (almost certainly) have originated from the reported sender and will not have been altered during transport. This dissertation addresses only the authentication of messages and not their encryption.

Many tasks carried out on networks require both privacy and authentication: banking transactions, electronic mail, and purchases, to name three. But many transactions, which would hurt or embarrass nobody if made public, require only verification of authenticity: a store's price-list, an acknowledgement of message receipt, or a newsfeed, for example. Some argue that *all* messages should be authenticated as a means of ensuring the integrity of all communications.

With so much demand for authentication, it is critical that MAC schemes be fast and that the guarantees afforded by them be credible. If they are extremely fast then there will be little reason not to use MACs ubiquitously, because they would take only a small percentage of communication resources. And, if they are provably good then we can feel confident that the goal of integrity in network communication has actually been achieved.

A (deterministic) MAC scheme works as follows. The sender and receiver both share some short secret string (the "key"). The sender processes the key and a message with a tag-generation algorithm which produces another short string (the "tag"). The sender then sends the message and tag to the receiver. When the receiver receives a message and tag, she then processes their shared key and the received message with the tag-generation algorithm. If her generated tag matches the tag received along with the received message, then it is very likely that the received message is authentic. An adversary is said to create a message *forgery* if, without knowledge of the secret key, he is able to create a tag for a new message which together pass the receiver's test of authenticity. Note that this way of modelling message authentication disallows so-called "replay attacks", where the adversary

simply replays an intercepted message and tag. Such attacks need to be dealt with but are usually beyond the scope of the MAC itself. We will formalize this model more after the next section.

Related to MACs are *digital signatures*. In a MAC, both the sender and receiver share the same secret key. This is a symmetric, or private-key, model. In a digital signature scheme, the sender and receiver each have a different key. This is an asymmetric, or public-key, model. Usually, one of the keys will be privately known by its holder while the other key is publicly known by possibly many people. Any message "signed" using the private-key can be verified by the public-key. Such a scheme allows an entity to broadcast a message, along with its signature, which can be verified by anyone who has access to the sender's public-key. Although more flexible than a MAC scheme, insofar as the verifier cannot generate signed messages, the fastest digital signature schemes are much slower than the fastest MAC schemes, and so MAC schemes are used whenever possible.

1.1 Summary of Results

This dissertation examines the progression of UMAC, the fastest, provably-secure MAC in the open literature. The evolution of UMAC has broadly occurred in three phases: initial research on UMAC published in 1999 [10], and then two follow-up projects designed to enhance UMAC performance and flexibility.

UMAC as initially published used a fast universal hash function, NH, as an accelerant to a cryptographic pseudorandom function. This general approach has its roots in the ideas of Carter and Wegman [14, 39], although their ideas were not originally intended for high speeds. Because NH requires a key as long as the message it is hashing, UMAC would break the message to be authenticated into 4KB chunks and hash each chunk using the same 4KB

key. Concatenating the results of each NH hash would result in a much shorter message which could then be processed, along with a short "nonce" string[1], by the pseudorandom function to generate the MAC tag.

The NH hash function being so much faster than the pseudorandom function, this resulted in a very fast MAC: as fast as 0.5 Pentium II cycles per byte of message being authenticated for a forgery probability of around 2^{-30}. But UMAC was exceptionally fast only on messages of moderate size and larger. Also, the time spent executing cryptographic functions in UMAC (the pseudorandom function) was proportional to the length of the message. Chapter 2 describes this initial UMAC and proves its forgery probabilities. This chapter is extracted from the full version of [10]; joint work of Black, Halevi, Krawczyk, Krovetz and Rogaway.

After its initial publication, we identified three primary goals to improve UMAC. We wished to make UMAC faster on short messages, more conservative in its use of cryptography, and allow the receiver of a message and MAC tag to verify the authenticity of the message to a receiver-selected level of security. We did this, in part, by moving to a MAC in the more classical style of Wegman and Carter [39]. Their method is described more fully later in this section. Our first attempt at achieving these goals shared the same initial step as the original UMAC: the message being authenticated was "compressed" by breaking it into chunks and then NH hashing each chunk. After concatenating the outputs of hashing these chunks, we were left with a string which was proportional in length to the original message. This string was too long for a Wegman-Carter MAC, so we hashed it again with NH, this time using an infinite-length key. The infinite-length key, generated on-the-fly by a stream- or block-cipher, allowed NH to return a fixed-length output appropriate for a

[1] A nonce is an extra input to a computation which must be distinct for each invocation of the computation. Usually, analysis of the algorithm will depend on distinct nonces for each invocation.

Wegman-Carter MAC.

Chapter 3 explores the motivation and result of this first UMAC revision. Although mostly successful in achieving our goals for revision, this revision still used an unbounded amount of cryptography because of its need for an unbounded-in-length key in the second layer of NH hashing. An alternative to using NH in the second-layer would be to use another hash function which used a fixed-length key and returned a fixed-length output. Chapter 4 investigates the use of optimized polynomial hashing. In our polynomial hash, the message is broken into words which are then viewed as the coefficients of a polynomial over some field. The evaluation of this polynomial on some random point yields a hash result that can be computed as fast as 3.9 cycles per byte. While polynomial hashing is not novel, and many of the methods we use to make it faster and more flexible may be attributed to "folklore", our construction is new and the overall result is a hash function with a short fixed key suitable for the second layer of UMAC hashing. Some of the optimizations we use in our polynomial hashing are: (1) to pick a prime field with prime close to a power of 2; (2) to restrict the keys used to ones which minimize 32-bit register carries during evaluation; and (3) to dynamically enlarge the size of the field over which the polynomial is being evaluated in order to avoid high collision probabilities. Chapter 4 is extracted from a draft paper by Krovetz and Rogaway which is in preparation for publication [26].

A secondary goal of the UMAC revisions was to produce a universal hash function, with freely-available source code, to be useful in domains beyond message authentication. In pursuing this goal we identified a weakness in current definitions used to describe universal hash functions. If a hash function-family is "strongly-universal" (SU), then the images of any two distinct values under a randomly chosen member of the family are uniformly distributed. A common relaxation of this notion, "almost-strongly-universal" (ASU), says

that the image of a second point, given knowledge of the image of a first point, is "hard to guess". In Chapter 5 we suggest a new definition, "variationally-universal", stronger than ASU but still weaker than SU. The new notion measures the quality of the second image in terms of its variational distance from the uniform distribution. A hash function which is variationally-universal (VU) produces outputs which are more "generally useful" than the outputs from an ASU hash function because they are known to be "close to random" rather than "hard to guess". Also in this chapter are presented methods and proofs of how to construct a VU hash family by composing other universal hash functions. This chapter is extracted from a draft paper by Krovetz and Rogaway which is in preparation for publication [26].

Chapter 6 defines a second revision of UMAC which uses NH as a first-layer accelerant to a second-layer polynomial hash. The output from the second layer is then hashed with a final inner-product hash which results in a three-layer hash function that is VU and appropriate for a Wegman-Carter MAC. The resulting UMAC is twice as fast as the original on short messages, the same speed as the original UMAC on long messages, uses only a small fixed amount of cryptography when generating tags, and is defined in terms of a hash function that is useful in other domains. The goal of receiver-selectable levels of security is met, too. Namely, UMAC iterates multiple times the hashing procedure that we have just sketched, concatenating the output of each, to improve security. For the receiver to verify a message to a lesser degree, she need only iterate the hash a fewer number of times. Chapters 3 and 6 will be integrated into a future paper describing UMAC (ie., the journal version of [10]).

The UMAC project is significant in that its design principals were intended to produce practical, "optimized" results. Extensive experimentation was coupled with provable-

security goals. Provable-security imbued in our designs a minimalism which allowed us to optimize our experiments with conviction, just as our experimental work imbued in our proofs a confidence that the constructions we were proving were practical and fast. No experiment was done without the knowledge that it met the minimal requirements provided by our proofs, and no proof was undertaken without the knowledge that the construct being proven was fast under experimentation. These are, of course, circular requirements, but they are intended to portray the fine granularity of intertwining of theoretical and experimental work in the design of UMAC. The extensive experimental work also paid dividends by providing a reference implementation which provided a basis for specification documents and public-domain code.

Full specifications for the original UMAC (1999) and the second revision of UMAC (2000) are provided in Appendices C and A, while the implementation used for testing and timing UMAC (2000) can be found in Appendix B.

1.2 MAC Model

We now look more closely at how we judge the security provided by a MAC scheme.

A (deterministic) MAC is a function $\mathsf{MAC} : \mathsf{Keys} \times \mathsf{Messages} \rightarrow \mathsf{Tags}$ which when given a key $k \in \mathsf{Keys}$ and a message $M \in \mathsf{Messages}$ produces a tag $v \in \mathsf{Tags}$. An adversary has *forged* the MAC if, without knowledge of random key k, she is able to produce a new message M and tag v such that $\mathsf{MAC}_k(M) = v$. To accomplish this, the adversary is allowed to adaptively obtain the tags of messages of her choosing. We say that the MAC scheme is (t, q, ϵ)-*secure* if, under a randomly chosen key k, the adversary cannot forge a new message in time t with probability better than ϵ even if she is provided with the tags of q other messages of her choice.

Some MACs, including UMAC, require a nonce to calculate the tag. If so, then a MAC is a function $\mathsf{MAC} : \mathsf{Keys} \times \mathsf{Nonces} \times \mathsf{Messages} \to \mathsf{Tags}$, and it is forged if, without knowledge of random key k, an adversary is able to produce a new message M and nonce n pair and tag v such that $\mathsf{MAC}_k(n, M) = v$. We say that this MAC scheme is (t, q, ϵ)-*secure* if no adversary successfully forges in time t with probability greater than ϵ. In this setting a successful forgery occurs if a random key $k \in \mathsf{Keys}$ is chosen; the adversary is allowed to get the tags $v_1, \ldots, v_q$ for q message-nonce pairs $(M_1, n_1), \ldots, (M_q, n_q)$, provided that the nonces $n_1, \ldots, n_q$ are all distinct; and the adversary then produces a valid (M^*, n^*, v^*) triple in which (M^*, n^*) was not one of the previously requested message-nonce pairs.

This sense of security is very strong. It gives an adversary q *chosen-message* tags, and then asks what is the chance that she can then forge a message. Contrast this with what is likely to be a real-life scenario. An adversary may be able to eavesdrop on messages and their tags being transmitted over a network, in which case she is privy to *known-message* tags. After seeing some number of these known-message tags, she may attempt to either interject or alter a message. In either case, she is attempting a forgery. The receiver *may* indicate whether the forgery was successful or not, but this is still less information than the adversary could glean from the chosen-message scenario that we model. In a chosen-message model, the adversary could make a forgery attempt and then ask for the tag corresponding with the attempted forgery. Comparison of the forged tag and real one will give the adversary at least as much information as a real-life receiver who indicates whether a forgery attempt was successful. That our model is stronger than the real-life scenario is true so long as the real-life receiver does not allow duplicate nonces. Nonces are intended to be non-repeating.

A MAC need not be deterministic. If, however, a MAC is probabilistic, then the above model is not sufficient. In particular, a MAC verifier is needed to check that a particular

tag v is a *possible* tag for message M. If $\Pr[\mathsf{MAC}_k(M) = v] > 0$, then the tag must be declared valid. All of the MACs we discuss in this dissertation are deterministic MACs, so we stick with the simpler deterministic model.

To use a MAC algorithm, the sender and receiver must share a randomly chosen MAC key. How the key is distributed to the sender and receiver without any adversary knowing anything about the key is outside the scope of our MAC model and outside the scope of this dissertation. Several efficient methods do exist for secure key distribution; see, for example [6, 7, 8].

1.3 Some MAC Examples

A perfect MAC is one in which an adversary has no better chance of forging a message, even with infinite computing resources, than by simply guessing a random tag for the message. This can be achieved in an information-theoretic sense if the sender and receiver share a random function. Let $\mathsf{Rand}[\mathsf{Messages}, \mathsf{Tags}]$ be the set of *all* (total) functions which map the elements of $\mathsf{Messages}$ to elements of Tags. If we choose a function f uniformly from $\mathsf{Rand}[\mathsf{Messages}, \mathsf{Tags}]$, denoted $f \xleftarrow{\mathrm{R}} \mathsf{Rand}[\mathsf{Messages}, \mathsf{Tags}]$, then f is a random function in which the image of each element of $\mathsf{Messages}$ is uniform under f.

If we construct a MAC using a random element of $\mathsf{Rand}[\mathsf{Messages}, \mathsf{Tags}]$ as its key, meaning $\mathsf{MAC}_f(M) \stackrel{\mathrm{def}}{=} f(M)$ is our MAC, then no adversary can do better under our MAC model than one who guesses random tags as their forgery. Imagine a random function is selected $f \xleftarrow{\mathrm{R}} \mathsf{Rand}[\mathsf{Messages}, \mathsf{Tags}]$, and an adversary if given $\mathsf{MAC}_f(\cdot) \stackrel{\mathrm{def}}{=} f(\cdot)$ as her oracle. No matter what sequence of queries the adversary makes $f(M_1), \ldots, f(M_q)$, all she gets in return for her oracle responses are q independent random elements from Tags. When the adversary finally guesses (M^*, v^*) as her forgery, $f(M^*)$ will be uniform because

$M^* \notin \{M_1, \ldots, M_q\}$ and so $f(M^*)$ will match v^* with probability $1/|\mathsf{Tags}|$. This MAC is $(\infty, \infty, \frac{1}{|\mathsf{Tags}|})$-secure, but cannot be realized for reasonable message lengths because truly random functions allowing messages longer than just a few bytes require too much storage to be practical. We often design MACs (and other cryptographic objects) with information-theoretic primitives like a random function and then relate via reductions the security of the MAC to a complexity-theoretic object like a pseudorandom function. We will discuss this "reductionist" view of cryptographic analysis further in a moment.

Another way to build a MAC is to begin with a publicly-known random function. Let $\mathsf{Rand}[\{0,1\}^*, \{0,1\}^\ell]$ be the set of all functions mapping arbitrary strings to ℓ-bit strings. Let $H \xleftarrow{R} \mathsf{Rand}[\{0,1\}^*, \{0,1\}^\ell]$ be a publicly-known (by receiver, sender and adversary) random function. One can imagine choosing H from this infinite set by constructing an infinite table in which each possible string is paired with a uniform ℓ-bit string. We can build a MAC using H which takes κ-bit keys, arbitrary message strings and returns ℓ-bit tags by defining $\mathsf{MACH} : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^l$ as $\mathsf{MACH}_k(M) \stackrel{\text{def}}{=} H(k \,\|\, M)$. Analysis of this MAC is straightforward. Imagine a random string $k \in \{0,1\}^\kappa$ is chosen and an adversary is given $f(\cdot) = \mathsf{MACH}_k(\cdot)$ as an oracle. The adversary asks up to q queries of her oracle and then produces (M^*, v^*) as her message-tag forgery. If $H(k \,\|\, M^*) = v^*$, then she has successfully forged. But, H is a random function, and unless the adversary has asked $f(M^*)$ as one of her queries or she has calculated $H(k \,\|\, M^*)$ on her own, then $H(k \,\|\, M^*)$ will be uniformly distributed and the chance that v^* is correct will be $2^{-\ell}$. She could not have asked $f(M^*)$ because our MAC model does not allow forgeries based on previous oracle queries. So, what is the chance she calculated $H(k \,\|\, M^*)$ on her own? She can certainly use her time t by guessing a sequence of keys $k_1, \ldots, k_t$ and seeing if any of $H(k_1 \,\|\, M), \ldots, H(k_t \,\|\, M)$ matches $f(M)$. If any do, then she can forge, assuming the

suspected key k is correct, by calculating $H(k \,\|\, M^*)$ for any new message M^*. But, the chance of finding k this way is only $t2^{-\kappa}$, and because each oracle query yields a uniform ℓ-bit string, they do not assist in the search for k except to verify guesses of the key k. So, this MAC is (t, ∞, ϵ)-secure for $\epsilon = \frac{t}{2^\kappa} + \frac{1}{2^\ell}$, which is perfect for a keyed MAC because it indicates that the best an adversary can do is to spend time guessing keys and, if that fails, guess the tag.

There are no publicly-known random functions, but a complexity-theoretic approximation can be found in a cryptographic hash function like MD5 [33] or SHA-1 [17]. Many MACs have been designed based on cryptographic hash functions [24, 37], the most popular being HMAC [20, 3]. The speeds of MACs designed in this paradigm are lower-bounded by the speeds of the cryptographic hash function in use. The maximum reported throughput, measured in Pentium processor cycles per byte (cpb), are 5.3 cpb for MD5 and 13.1 cpb for SHA-1 [13]. Recently some weakness has been detected in MD5 [15], making its use in such constructs suspect in the eyes of some.

Another commonly used MAC is the CBC-MAC [2]. In the CBC-MAC, the sender and receiver share a function $f : \{0,1\}^\ell \to \{0,1\}^\ell$. Given f and a message $M_1 \,\|\, \cdots \,\|\, M_n = M$ where each M_i is ℓ bits long, CBCMAC can be defined recursively as

$$\mathsf{CBCMAC}_f(M_1 \,\|\, \cdots \,\|\, M_m) = \begin{cases} f(\mathsf{CBCMAC}_f(M_1 \,\|\, \cdots \,\|\, M_{m-1}) \oplus M_m) & \text{if } m > 1 \\ f(M_1) & \text{if } m = 1. \end{cases}$$

The MAC result being $\mathsf{CBCMAC}_f(i, M)$. The function f is typically realized by a block cipher like DES [18] or RC6 [34] on a random key. Once again, the speed of the MAC is lower-bounded by the speed of the cryptographic primitive used to instantiate it. So, CBCMAC using DES as its family of functions goes no faster than 42 cpb, and if using RC6, no faster than 15 cpb [10, 13]. Analysis of CBCMAC is complex and not given in this

dissertation [5].

1.4 Wegman-Carter MACs

Each of the above MACs use a cryptographic primitive directly on each byte of message being authenticated. If the primitive is RC6 or SHA-1, both widely regarded as conservative and secure, then this limits MAC throughput to no faster than around 13 Pentium cycles-per-byte.

In 1979 Wegman and Carter introduced the idea of a "universal hash-function family", and in 1981 they suggested the use of universal hashing to authenticate messages [14, 39]. A universal hash-function family is a set of functions endowed with some combinatoric property. For example, a hash family $H = \{h : A \to B\}$ is "ϵ-almost-universal" if for any distinct $x, x' \in A$, the probability that $h(x)$ and $h(x')$ are equal is no more than ϵ, when $h \in H$ is chosen at random. There are other types of universal hash-function properties, and some will be discussed later. In a Wegman-Carter MAC a message is hashed with a function drawn from a universal hash-function family, the output of the hash function is encrypted, and then the encrypted hash-output is produced as the MAC tag. Because universal hash functions are usually non-cryptographic, this process has the benefit of requiring only a small, fixed amount of cryptography for each message being authenticated: as little as 75 cycles to encrypt each 32-bit tag if one carefully uses RC6. Because the amount of (relatively slow) cryptography is fixed in a Wegman-Carter MAC, the key to making a fast MAC in this paradigm is to choose a fast universal hash-function. Recent research has been remarkably successful at designing fast hash functions for the purpose of message authentication, achieving throughput rates as fast as 0.5 cpb [1, 9, 10, 21, 25, 35, 36].

The reason for such disparity in throughput speeds when comparing cryptographic

primitives with universal hash functions is that universal hash functions need not use cryptography in achieving their goal. As indicated above, a universal hash function need only satisfy a simple combinatoric property involving collisions in its output. The combinatoric property is often not difficult to prove and, once proven, is absolute. No "safety margin" in the hash function's design is necessary. Contrast this with cryptographic primitives like SHA-1 and RC6. They have a much more difficult goal than universal hashing: they must approximate as closely as possible random objects. This is done by iteration of a series of non-linear operations which scramble bits. With our current knowledge, their goal cannot be provably achieved, and so their designers are usually conservative and do more iterations than may be necessary for making a secure MAC. It is no wonder that universal hashing with its simple and provably achieved goal can be designed to be extremely fast while cryptographic primitives have not been.

A family of hash functions is a finite set of functions with common domain A and (finite) range B. We denote a family H using either set notation, as in $H = \{h : A \to B\}$ where it is not made explicit the number of functions from A to B that are in the set; or we denote the family using functional notation, as in $H : K \times A \to B$ where $H_k : A \to B$ is a function in the family for each $k \in K$. In the later case, choosing a random function f from the family is done by choosing $k \in K$ uniformly and letting $f = H_k$.

Hash-function family $H : K \times A \to B$ is ϵ-strongly-universal, written ϵ-SU, if for any $M \neq M' \in A$, and any $c, c' \in B$, the following two conditions hold: (1) $\Pr[H_k(M) = c] = \frac{1}{|B|}$, and (2) $\Pr[H_k(M) = c \mid H_k(M') = c'] \leq \epsilon$, with all probabilities taken over $k \in K$. The first condition states that $H_k(M)$ must be uniformly distributed, and the second condition says that $H_k(M)$ must not produce any output with probability higher than ϵ regardless of the value of H_k for one other point. Constructing a function which is random on a single

point and which also appears close-to-random on a second point is considerably easier than constructing a function that needs to look random on a much larger number of points, as cryptographic primitives must.

Given an ϵ-SU hash function family $H : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\ell$, a Wegman-Carter MAC, $\mathsf{WCMAC} : \left(\mathsf{Rand}[\mathsf{Nonces}, \{0,1\}^\ell] \times \{0,1\}^\kappa\right) \times \mathsf{Nonces} \times \{0,1\}^* \to \{0,1\}^\ell$, can be defined as $\mathsf{WCMAC}_{f,k}(n, M) = H_k(M) \oplus f(n)$, where f, k jointly serve as the key to WCMAC, n is the nonce and M is the message for which we want a tag. The hash function H serves to quickly hash the message down to an ℓ-bit string, and then f serves to encrypt the hash output with a random pad. Wegman-Carter MACs can be designed with weaker universal-hashing definitions, "ϵ-xor-universal" or "ϵ-delta-universal", but we will later be defining a MAC using only ϵ-SU, so we focus on authenticating with that definition.

We now take a look at the security of WCMAC. We imagine random values for $k \xleftarrow{\text{R}} \{0,1\}^\kappa$ and $f \xleftarrow{\text{R}} \mathsf{Rand}[\mathsf{Nonces}, \{0,1\}^\ell]$ are chosen and an oracle $g(n, M) \stackrel{\text{def}}{=} H_k(M) \oplus f(n)$ is given to an adversary. He may ask $g(n_1, M_1), \ldots, g(n_q, M_q)$, with each nonce distinct, receiving from the oracle tags $v_1, \ldots, v_q$ and then output an attempted forgery (n^*, M^*, v^*). But because f is a random function unknown to the adversary, all oracle responses $v_1, \ldots, v_q$ will appear uniformly distributed. If, in his forgery attempt, the adversary uses a nonce that he previously presented to his oracle, say $n^* = n_i$ for some $1 \le i \le q$, then $g(n^*, M^*) = v^*$, indicating a successful forgery, only if $H_k(M^*) = H_k(M_i) \oplus v_i \oplus v^*$. But, because H is assumed ϵ-SU, this event is no more than ϵ-likely. If, on the other hand, the adversary uses a never-before-seen nonce in his forgery attempt, then $f(n^*)$ is uniform, meaning $g(n^*, M^*) = v^*$ has probability exactly $2^{-\ell}$. The adversary must either reuse a nonce or not, so his probability of forgery success is $\max\left\{\epsilon, \frac{1}{2^\ell}\right\} = \epsilon$. This means the Wegman-Carter MAC described here is $(\infty, \infty, \epsilon)$-secure.

1.5 Complexity-Theoretic Reductions

In our discussion about MACH, CBCMAC and Wegman-Carter MACs we couched the definitions in information-theoretic terms, assuming perfectly random primitive functions. Such objects do not exist. In practice, these MACs use complexity-theoretic substitutes like SHA-1 and RC6. These objects, appropriately used, are hard to distinguish from their information-theoretic counterparts, and serve well in their stead. For example, RC6 is a family of functions $\mathsf{RC6} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ such that it is hard for an adversary with limited time and computing resources to distinguish an oracle instantiated as a random function $f : \{0,1\}^{128} \to \{0,1\}^{128}$ from one instantiated as $\mathsf{RC6}_k : \{0,1\}^{128} \to \{0,1\}^{128}$ when $k \in \{0,1\}^{128}$ is chosen randomly.

If, for example, CBCMAC is shown to be (t_1, q_1, ϵ_1)-secure in an information-theoretic setting and it was known that RC6 was (t_2, q_2, ϵ_2)-secure in the sense that no adversary could distinguish with q_2 oracle queries in time t_2 with probability better that ϵ_2 an oracle instantiated as a randomly chosen RC6 function from an oracle instantiated as a random function, then it would also be known that CBCMAC instantiated with RC6 as its random function would have to be $(t_3, q_3, \epsilon_1 + \epsilon_2)$-secure. If there did exist an adversary who could forge the MAC with better than $\epsilon_1 + \epsilon_2$ probability, then that same adversary could be used to differentiate RC6 from a random function with better than ϵ_2 probability. The exact values for t_3 and q_3 depend on the reduction used, but are often comparable to t_1 and q_1.

Taking this view, we analyze our constructions in information-theoretic terms, since any weakness found in the MAC construction after instantiation with a complexity-theoretic object will give rise to an attack on the underlying primitive with success proportional to the difference. This view of reductionist, provable cryptography has a long history [19, 5, 4].

Chapter 2

UMAC (1999)

In 1997 Halevi and Krawczyk introduced a family of hash functions called MMH [21]. The family was designed for use on processors which support well multiplication of 32-bit integers into 64-bit products, and was the first universal hash family to claim gigabit-per-second throughput. Halevi and Krawczyk gave a collision analysis of MMH and reported throughput timings but did not make explicit how to embed it in any practical MAC. The MMH family requires a key whose length is proportional to that of the message being hashed, making it unclear how to use it for message authentication of arbitrary messages. In 1998, Black, Krovetz and Rogaway simplified MMH and began embedding it into a fully specified MAC. The hash family NH was developed and embedded in a MAC inspired by the Wegman-Carter paradigm. Halevi and Krawczyk joined the effort and the resulting message authentication code, UMAC, was published in 1999 [10]. Refinements were later made in 2000. The name "UMAC" applies to both the 1999 and 2000 versions. Where it is not apparent from the context, the versions may be referred to as UMAC (1999) and UMAC (2000).

This chapter discusses UMAC (1999) exclusively and is largely extracted from [10]. It

is the only part of this dissertation which is not majority-written by the author.

A detailed, parameterized specification for UMAC (1999) is given in Appendix C.

2.1 Introduction

We now explore UMAC in detail and the theory that lies behind it. UMAC was been designed with two main goals in mind:

- **Extreme speed.** We have aimed to create the fastest MAC ever described, and by a wide margin. Speed is with respect to software implementations on contemporary general-purpose computers.
- **Provable security.** We insist that the MAC be demonstrably secure, by virtue of proofs carried out in the sense of provable-security cryptography. Namely, if the underlying cryptographic primitives are secure (we use pseudorandom functions) then the MAC is secure, too.

There were of course other goals, avoiding excessive conceptual and implementation complexity being principal among them.

UMAC is certainly fast. On our 350 MHz Pentium II PC, one version of UMAC (where the adversary has 2^{-60} chance of forgery) achieves 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. Note that 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 [21], 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 [39], making innovations in its realization. We now review this approach and its advantages, and then describe what we have done to make it fly.

2.1.1 Universal-Hashing Approach

Universal Hashing And Authentication. Our starting point is a universal hash-function family [14]. (Indeed the "U" in UMAC is meant to suggest the central role that universal hash-function families play in this MAC.) In this chapter we define a MAC which is inspired by the Wegman-Carter method described in the Introduction, but requires its hash function only to be ϵ-universal rather than ϵ-SU. Remember that a set of hash functions is said to be "ϵ-universal" if for any pair of distinct messages, the probability that they collide (hash to the same value) is at most ϵ. The probability is over the random choice of hash function. This notion is weaker than ϵ-SU, but is sufficient for our revised use of the Carter-Wegamn MAC paradigm.

The Quest For Fast Universal Hashing. At least in principle, the universal-hashing paradigm has reduced the problem of fast message authentication to the problem of fast universal hashing. Thus there has been much recent work on the design of fast-to-compute universal hash-function families. Here is a brief overview of some of this work; a more complete overview is given in Section 2.1.3. Krawczyk [25] describes a "cryptographic CRC" construction, which has very fast hardware implementations and reasonably fast software implementations; it needs about 6 cycles/byte, as shown by Shoup [36]. Rogaway's "bucket

hashing" construction [35] was the first universal hash-function family explicitly targeted for fast software implementation; it runs in about 1.5–2.5 cycles/byte. Halevi and Krawczyk devised the MMH hash-function family [21], which takes advantage of current CPU trends and hashes 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. However, authenticating information at the rate that it is generated or transmitted is not the real goal. The goal is to use the smallest possible fraction of the CPU's cycles by the simplest possible hash mechanism, and having the best proven bounds. In this way most of the machine's cycles are available for other work.

2.1.2 Our Contributions

UMAC represents the next step in the quest for a fast and secure MAC. Here we describe the main contributions of this work.

New Hash Function Families And Their Tight Analyses. A hash-function family named NH underlies hashing in UMAC. It is a simplification of the MMH and NMH families described in [21]. The family NH works like this. The message M to hash is regarded as a sequence of an even number ℓ of integers, $M = (m_1, \ldots, m_\ell)$, where each $m_i \in \{0, \ldots, 2^w - 1\}$ corresponds to a w-bit word (eg., $w = 16$ or $w = 32$). A particular hash function is named by a sequence of $n \geq \ell$ w-bit integers $K = (k_1, \ldots, k_n)$. The NH function is then computed as

$$\mathsf{NH}_K(M) = \left(\sum_{i=1}^{\ell/2} ((m_{2i-1} + k_{2i-1}) \bmod 2^w) \cdot ((m_{2i} + k_{2i}) \bmod 2^w) \right) \bmod 2^{2w} . \quad (2.1.1)$$

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 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 and desirable bound on the collision probability: 2^{-w}. Earlier analyses of related hash-function families had to give up a small constant in the analysis [21]. We give up nothing.

After proving our bounds on NH we go on and extend the method using the "Toeplitz construction." This is a well-known approach to reduce the error probability without much lengthening of the key [28, 25]. 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. Here again our proof achieves a tight bound for the collision probability. We then make further extensions to handle length-issues, and allow other optimizations, 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 the next step of the process—how to embed a fast-to-compute hash function into a *concrete, practical, and fully analyzed* MAC. For some hash-function constructions (eg., cryptographic CRCs) this step would be straightforward. But for the fastest hash families it is not, since these hash functions have some unpleasant characteristics, including specialized domains, long key-lengths, long output-lengths, or good performance only on very long messages. It had never been demonstrated that these difficulties could be overcome in a practical way which would deliver on the promised speed. This chapter shows that they can. We provide a complete specification of UMAC, a ready-to-use MAC, in Appendix C. The technical difficulties encountered in embedding our hash-function family into a MAC are not minimized; they are treated with the same care as the

hash-function family itself. The construction is fully analyzed, beginning-to-end. What is analyzed is exactly what is specified; there is no "gap" which separates them. 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 to some fixed length string (eg., 64 bits) and then the cryptographic primitive is applied. But using universal hashing to reduce a very long message into a fixed-length one can be complex, require long keys, or reduce the quantitative security. Furthermore, hashing the message down by a factor of 128, say, already provides much of the speed and security benefits. So we reduce the length of the message by some pre-set constant, concatenate a sender-generated nonce, and then apply a pseudorandom function (PRF) to the shorter (but still unbounded-length) string. The nonce is communicated along with the image of the PRF. We call this the PRF(HASH, Nonce)-construction. We analyze it and compare it with alternatives.

Extensive Experimentation. In defining UMAC we have been guided by extensive experimentation. Through this we have identified several parameters that influence the speed which UMAC delivers. Our studies illuminate how these algorithmic details shape performance. Our experiments have made clear that while any reasonable version of UMAC (ie., any reasonable setting of the parameters) should out-perform any conventional MAC, the fastest version of UMAC for one platform will often be different from the fastest version for another platform. We have therefore kept UMAC a parameterized construction, allowing some specific choices to be fine-tuned to the application or platform at hand. In this chapter and Appendix C we consider a few reasonable settings for these parameters.

SIMD Exploitation. Unlike conventional, inherently serial MACs, UMAC is paralleliz-

able, and will have ever-faster implementations as machines offer up increasing amounts of parallelism. Our algorithm and our specification were specifically designed to allow implementations to exploit the form of parallelism offered up in current and emerging SIMD architectures (Single Instruction Multiple Data). These architectures provide some long registers that can, in certain instructions, be treated as vectors of smaller-sized words. For NH to run well we must be able to quickly multiply w-bit numbers ($w = 16$ or $w = 32$) into their w-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 computations. One of our fastest implementations 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. UMAC is the first MAC specifically constructed to exploit SIMD designs.

2.1.3 Related Work

MMH Paper. Halevi and Krawczyk investigated fast universal hashing in [21]. 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. Besides MMH, [21] describes a (formerly-unpublished) hash-function family of Carter and Wegman, NMH*, and a variation of it called NMH. Our NH function, as described in formula Equation 2.1.1, is a (bound-improving) simplification of NMH. The difference between NH and NMH is that NMH uses an additional modular reduction by a prime close to 2^w, followed by a reduction modulo 2^w. Similarly, NMH* is the same as NH, as given in formula Equation 2.1.1 except the mods are omitted and the arithmetic is in Z/p, where p is prime.

Other Work On Universal Hashing For MACs. Krawczyk describes a "crypto-

graphic CRC" which has very fast hardware implementations [25]. Shoup later studied the software performance of this construction, and gave several related ones [36]. In [25] one also 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 [28].

A hash-function family specifically targeted for software was Rogaway's "bucket hashing" [35]. 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 [29]. Patel and Ramzan give an MMH-variant that can be more efficient than MMH in certain settings [16].

Bernstein reports he has designed and implemented a polynomial-evaluation style hash-function family that has a collision probability of around 2^{-96} and runs in 4.3 Pentium cycles/byte [9]. Other recent work about universal hashing for authentication includes [1, 23].

Other Types Of MACs. Most concrete MACs have been constructed from other cryptographic primitives. The most popular choice of cryptographic primitive has been a block cipher, with the most popular construction being the CBC-MAC [2]. This MAC was analyzed by [5]. An extension of this analysis was carried out by [30], this extension being relevant for how we suggest using a block cipher to make a PRF.

More recently, MACs have been constructed from cryptographic hash-functions. It has usually been assumed that this would lead to faster MACs than the CBC-MAC. A few such methods are described in [38, 24], and analysis appears in [31, 32]. An increasingly popular MAC of this cryptographic-hash-function variety is HMAC [3, 20]. 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.

OTHER VERSIONS. The proceedings version of this chapter appears in [11].

2.2 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*[1]. The sender must not reuse the nonce within the communications session. 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 tag actually received. The receiver might also wish to verify that the nonce has not been used already; doing this is a way to avoid replay attacks.

Like many modern ciphers, UMAC employs a *subkey generation process* in which the underlying (convenient-length) key is mapped into UMAC's internal keys. In typical applications subkey generation is done just once, at the beginning of a long-lived session, 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.

[1]Because the nonce must be distinct for each UMAC invocation, the sender must maintain some information to disallow repeated nonces. This maintained information is the sender's "state".

Subkey generation:
Using a PRG, map *Key* to $K = K_1 K_2 \cdots K_{1024}$, 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 $|Msg|$ mod 4096, encoded as a 2-byte string.
Append to *Msg* the minimum number of 0 bits to make $|Msg|$ divisible by 64.
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 $Tag = \text{HMAC-SHA1}_A\ (Nonce \parallel HM)$

Figure 2.1: *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.

2.2.1 An Illustrative Special Case

Refer to Figure 2.1. The underlying 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 unimportant and standard detail (it can be handled using any PRG), and so we omit its description.

Figure 2.1 refers to the hash function NH, which is applied to each block $Msg_1, \ldots, Msg_t$ of *Msg*. 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 Equation 2.1.1 (for $w = 32$) but emphasizes that all the operations we are performing directly correspond to machine instructions of modern CPUs. See the left-hand

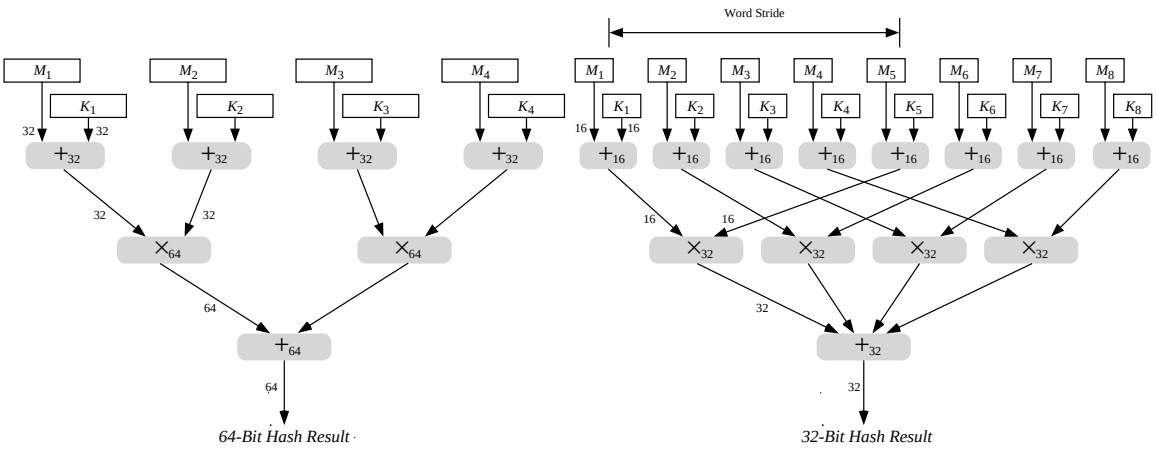


Figure 2.2: ***Left:*** *The* NH *hash function in its "basic" form, using a wordsize of* $w = 32$ *bits. This type of hashing underlies* UMAC-STD-30 *and* UMAC-STD-60. ***Right:*** *The "strided" form of* NH*, using a wordsize of* $w = 16$ *bits. This type of hashing underlies* UMAC-MMX-30 *and* UMAC-MMX-60.

side of Figure 2.2 for a picture.

Theorem 2.4.2 says that NH is 2^{-32}-universal with respect to strings of equal and appropriate length. Combining with Proposition 2.6.1 gives that NHX (as described in Figure 2.1) is 2^{-32}-universal, but now for any pair of strings. Now by Theorem 2.8.2, if an 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 $\delta - 2^{-160}$. But analysis in [3] indicates that one can't break HMAC-SHA1 (as a PRF) given generally-accepted assumptions about SHA-1.

2.2.2 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. The values of n and w are examples of two of UMAC's parameters. Let us describe a few others.

Naturally enough, the *pseudorandom function* (PRF) which is applied to $\mathit{Nonce} \,\|\, \mathit{HM}$

is a parameter. We used HMAC-SHA1, but any PRF is allowed. Similarly, it is a parameter of UMAC how *Key* gets mapped to K and A.

The universal hashing we used in our example had collision probability 2^{-32}. We make provisions for lowering this (eg., to 2^{-64}). To square the collision probability one could of course hash the message twice, using independent hash keys, and concatenate the results. But an important 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 2.5.1 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 λ_1 and a hash key of length n_2 gives compression ratio λ_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 in the computation of NH we allow slight adjustments in the indexing of NH. 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

$$(M_1 +_{16} K_1) \times_{32} (M_5 +_{16} K_5) \;+_{32}\; (M_2 +_{16} K_2) \times_{32} (M_6 +_{16} K_6) \;+_{32}\; \ldots$$

There are MMX instructions which treat each of two 64-bit words 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. 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 2.2.

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 (two's-complement) integers. The signed version of NH requires a slightly different analysis and loses a factor of two in the collision probability. This loss is inherent in the method; our analysis is tight.

If the input message is sufficiently short there is no speed savings to be had by hashing it. The *min-length-to-hash* parameter specifies the minimum-length message which should be hashed.

An *endian* parameter indicates if the MAC should favor big-endian or little-endian computation, where "endian" refers to the memory-orientation of multibyte quantities (ie. the most-significant byte of a multibyte quantity has the lowest address for big-endian computers while it has the highest address for little-endian computers).

Named Parameter Sets. In Appendix C we suggest five different settings for the vector of parameters, giving rise to UMAC-STD-30, UMAC-STD-60, UMAC-MMX-15, UMAC-MMX-30, and UMAC-MMX-60. 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 are the collision bounds (and therefore forgery bounds): 2^{-30} and 2^{-60}, respectively, which are achieved by hashing either once or twice. These two versions of UMAC perform well on a wide range of contemporary processors.

UMAC-MMX-15, UMAC-MMX-30 and UMAC-MMX-60 are well-suited for exploiting 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 [34]. Arithmetic is again signed. The difference between UMAC-MMX-15, UMAC-MMX-30 and UMAC-MMX-60 is the maximal forgery probability: 2^{-15}, 2^{-30} and 2^{-60}, respectively.

2.3 UMAC Performance

Fair performance comparisons are always difficult to make, but they are particularly difficult for UMAC. First, UMAC depends on a number of parameters, and the most desirable setting for these parameters differs from platform to platform. In some usage scenarios it is realistic to assume the most desirable set of parameters, though often it is not. Thus it is unclear which parameters should be selected and tested on which platforms. Second, UMAC performance varies with message length. The advantage UMAC delivers over conventional MACs increases as messages get longer. What message lengths should be chosen for performance tests?

To address the first issue we report performance characteristics for a few different parameter choices. To address the second issue we report timing data for a variety of message lengths.

Test Environment. We implemented the four UMAC variants named in Section 2.2.2 on three representative hardware platforms: a 350 MHz Intel Pentium II, a 200 MHz

IBM/Motorola PowerPC 604e, and a 300 MHz DEC Alpha 21164. We also implemented the UMAC-MMX variants on a pre-release PowerPC 7400 equipped with AltiVec SIMD instructions and running at 266 MHz. Performing well on these platforms is important for acceptance of UMAC but also is an indication that the schemes will work well on future architectures. The SIMD operations in the MMX and AltiVec equipped processors and the 64-bit register size of the Alpha are both features we expect to become more prevalent in future processors.

All tests were written in C with a few functions written in assembly. For the Pentium II we wrote assembly for RC6, SHA-1, and the first-level NH hashes. For the PowerPC 604e we wrote in assembly just the first-level NH hashes. In both cases, the number of lines of assembly written was small—about 70–90 lines. No assembly code was written for the Alpha or AltiVec implementations.

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. For comparison the same tests were run for HMAC-SHA1 [17] and the CBC-MAC of a fast block cipher, RC6 [34].

Results. The graph in Figure 2.3 shows the throughput of five versions of UMAC, as well as HMAC-SHA1 and CBC-MAC-RC6, all run on our Pentium II. The table in Figure 2.4 gives peak throughput for the same MACs, but does so for all three architectures. When reporting throughput in Gbits/second the meaning of "Gbit" is 10^9 bits (as opposed to 2^{30} bits). 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.

UMAC performs best on long messages because the hash function is then most effective

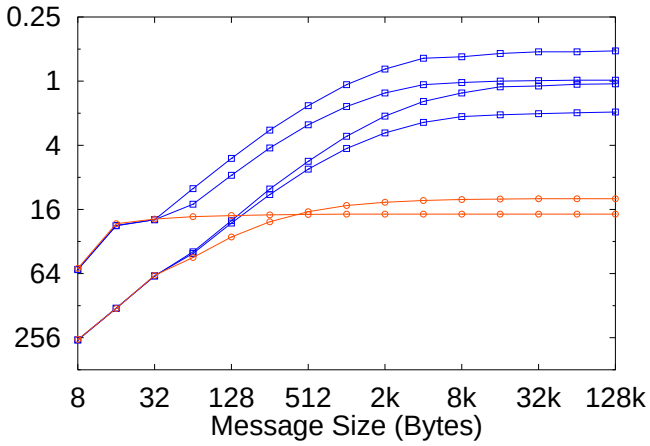


Figure 2.3: *UMAC Performance. Performance over various message lengths on a Pentium II, measured in machine cycles/byte. The lines in the graph correspond to the following MACs (beginning at the top-right and moving downward):* UMAC-MMX-30, UMAC-MMX-60, UMAC-STD-30, UMAC-STD-60, HMAC-SHA1 *and* CBC-MAC-RC6.

at reducing the amount of data acted upon by the PRF. For our choice of tested parameters the maximum compression ratio is achieved on messages of 4 KBytes and up, which is why the curves in Figure 2.3 level out at around 4 KBytes. Still, reasonably short messages (a couple hundred bytes) already see substantial benefit from the hashing. For short messages about half the time spent by UMAC-MMX-60 authenticating is spent in the PRF, but that ratio drops to around 7% for longer messages.

When messages are short the portion of time spent computing the PRF is higher, making the choice of PRF more significant. Our tests show that, all other parameters being the same, an RC6-based UMAC is about 50% faster than one based on SHA-1 for messages of length 32 bytes, but this advantage drops to 10% - 20% for 4 KByte messages and nearly vanishes for 64 KByte messages.

In general, the amount of work spent hashing increases as word size decreases because the number of arithmetic operations needed to hash a fixed-length message is inversely related to word size. The performance of UMAC on the Alpha demonstrates this clearly.

	Pentium II	PowerPC	Alpha
UMAC-STD-60	1.49 (1.93)	1.81 (1.58)	1.03 (2.78)
UMAC-STD-30	2.79 (1.03)	2.28 (1.26)	1.79 (1.60)
UMAC-MMX-60	2.94 (0.98)	4.21 (0.66)	0.287 (10.0)
UMAC-MMX-30	5.66 (0.51)	7.20 (0.39)	0.571 (5.02)
UMAC-MMX-15	8.47 (0.33)	10.5 (0.27)	0.981 (2.85)
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)

Figure 2.4: UMAC Performance. *Peak performance for three architectures measured in Gbits/sec (cycles/byte). The Gbits/sec numbers are normalized to 350 MHz.*

On UMAC-MMX-60 ($w = 16$) it requires 10 cycles per byte to authenticate a long message, while it requires 2.8 for UMAC-STD-60 ($w = 32$) and only 1.7 for a test version which uses $w = 64$ bit words[2]. Perhaps our most surprising experimental finding was how, on some processors, we could dramatically improve performance by going from words of $w = 32$ bits to words of $w = 16$ bits. Such a reduction in word size might appear to vastly increase the amount of work needed to get to a given collision bound. But a single MMX instruction which UMAC uses heavily can do four 16-bit multiplications and two 32-bit additions, and likewise a single AltiVec instruction can do eight 16-bit multiplications and eight 32-bit additions. This is much more work per instruction than the corresponding 32-bit instructions.

UMAC-STD requires an internal hash key only one-tenth as long as the one used by 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

[2]Another reason UMAC-MMX-60 does poorly on non-SIMD processors is because of additional overhead required to load small words into registers and then sign-extend or zero the upper bits of those registers.

bytes scheme goes 47% as fast.

The endian-orientation of the UMAC version being executed has little effect on performance for the PowerPC and Pentium II systems. Both support swapping the endian-orientation of words efficiently. On the Alpha, and most other systems, such reorientation is not part of the architecture and must be done with an expensive series of primitive operations. Key setup on our 350 MHz Pentium II took 450 μs for UMAC-MMX-60 and 130 μs for UMAC-STD-60, or about 158,000 and 46,000 cycles, respectively.

To answer questions we had about UMAC performance we implemented it with several variations. In one experiment we replaced the NH hash function used in UMAC-STD-30 by MMH [21]. Peak performance dropped by 24%. We replaced the NH hash function of UMAC-MMX-30 by a 16-bit MMH (implemented using MMX instructions) and peak performance dropped by 5%. Thus, for the case of 16-bit words, our performance gains (compared to prior art) are due more to SIMD exploitation than the difference between NH and MMH.

The hash family MMH can be simplified by eliminating its last two modular reductions. We refer to the resulting hash function family as MH, and it is defined by $\mathsf{MH}_K(M) = \sum_{i=1}^{\ell} m_i k_i \bmod 2^{2w}$. An MH-version of UMAC-MMX-30 ($w = 16$) was about 5% slower than (our NH-version of) UMAC-MMX-30, while an MH-version of UMAC-STD-30 ($w = 32$) was about 26% slower than (our NH-version of) UMAC-STD-30. Finally, we coded UMAC-STD-30 and UMAC-STD-60 for a Pentium processor which lacked MMX. Peak speeds were 2.2 cycles/byte and 4.3 cycles/byte—still 3 to 6 times faster than methods like HMAC-SHA1. Even though these versions of UMAC do not use MMX instructions, Intel processors without MMX are also slower than the Pentium II at integer multiplication, an important operation in UMAC.

2.4 The NH Hash Family

We now define and analyze NH, the hash-function family which underlies UMAC. Recall that NH is not, by itself, the hash-function family which UMAC uses, but the basic building block from which we construct UMAC's hash families.

2.4.1 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 H by specifying some finite set of strings, Key, and by explaining how each string $K \in \mathsf{Key}$ names some function $\mathsf{H}_K \in \mathsf{H}$. We may then think of 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. The formal definition follows.

Definition 2.4.1 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 H is ϵ-universal, denoted ϵ-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 H is ϵ-universal on equal-length strings if for all distinct, equal-length strings $M, M' \in A$, we have that $\Pr_{h \leftarrow \mathsf{H}}[h(M) = h(M')] \leq \epsilon$.

∎

2.4.2 Definition of NH

Fix an even $n \geq 2$ (the "blocksize") and a number $w \geq 1$ (the "wordsize"). We define the family of hash 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 a string K of nw bits; a random function in $\mathsf{NH}[n, w]$ is given by a random nw-bit string K. We write the function indicated by string 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 2.2.1: 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 NH instead of $\mathsf{NH}[n, w]$.

2.4.3 Analysis

The following theorem bounds the collision probability of NH.

Theorem 2.4.2 *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

$$\Pr_{K \leftarrow \mathsf{NH}} \left[\mathsf{NH}_K(M) = \mathsf{NH}_K(M') \right] \leq 2^{-w}.$$

Converting the message and key strings to n-vectors of w-bit words we invoke the definition of NH to restate our goal as requiring

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

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

$$\Pr_{k_1 \in U_w} \Bigg[(m_1 +_w k_1)(m_2 +_w k_2) + \sum_{i=2}^{\ell/2} (m_{2i-1} +_w k_{2i-1})(m_{2i} +_w k_{2i}) = (m'_1 +_w k_1)(m'_2 +_w k_2) + \sum_{i=2}^{\ell/2} (m'_{2i-1} +_w k_{2i-1})(m'_{2i} +_w k_{2i}) \Bigg] \leq 2^{-w}$$

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

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

In Lemma 2.4.3 below, we prove 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. ∎

The above proof reduced establishing our bound to the following useful lemma, which is used again for Theorem 2.5.1.

Lemma 2.4.3 *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 $m = 0$: to see this, notice that if $c(k +_w m) = c'(k +_w m') + y$, then also $c(k^* +_w 0) = c'(k^* +_w m^*) + y$, where we define $k^* = (k +_w m)$ and $m^* = (m' -_w m)$. If follows that if there exist $k_1 \neq k_2$ satisfying the former equality, then there must also exist $k_1^* \neq k_2^*$ satisfying the latter.

Assuming $m = 0$, 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

$$k(c - c') = m'c' + y \quad \text{and} \quad k < 2^w - m' \tag{2.4.1}$$

$$k(c - c') = (m' - 2^w)c' + y \quad \text{and} \quad k \geq 2^w - m' \tag{2.4.2}$$

A simple lemma (Lemma 2.4.4, 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 Equation 2.4.1 and $k = k_2 \in U_w$ satisfying Equation 2.4.2 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

$$(k_2 - k_1)(c' - c) = 2^w c' \tag{2.4.3}$$

We show Equation 2.4.3 has no solutions in $Z/2^{2w}$. To accomplish this, we examine two 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 Equation 2.4.3 has no solutions in $\mathbb{Z}$. But this clearly holds, 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 $\mathbb{Z}$, 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 $D_w = \{-2^w + 1, \ldots, 2^w - 1\}$ contain the values attainable from a difference of any two elements of U_w. We now prove the following lemma, used in the body of the preceding proof.

Lemma 2.4.4 *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. ∎

REMARKS. The bound given by Theorem 2.4.2 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}-AΔU under the operation of addition modulo 2^{2w}; only trivial modifications to the proof of Theorem 2.4.2 are required. In particular, in that proof y can be offset by an arbitrary difference. See [25] for a definition of ϵ-AΔU.

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 ϵ substantially. However, removing the inner moduli retains $\epsilon = 2^{-w}$ (but significantly degrades performance).

2.4.4 The Signed Construction: NHS

To this point we have assumed our strings are interpreted as sequences of *unsigned* integers. But often we will prefer they be *signed* integers. Surprisingly, the signed version of NH has slightly higher collision probability: we shall now prove that the collision bound increases by a factor of two to 2^{-w+1}-AU on equal-length strings. This helps explain the 2^{-30} and 2^{-60} message forging probabilities for the four UMAC versions named in Section 2.2.2 and the performance measured in Section 2.3; in actuality, all four algorithms use the signed version of NH.

Let S_w and S_{2w} be the sets $\{-2^{w-1}, \ldots, 2^{w-1} - 1\}$ and $\{-2^{2w-1}, \ldots, 2^{2w-1} - 1\}$, respectively. For this section, assume all arithmetic done modulo 2^w returns a result in S_w and all arithmetic done modulo 2^{2w} returns a result in S_{2w}. The family of hash functions

NHS is defined exactly like NH except that each $M_i \in \{0,1\}^w$ and $K \in \{0,1\}^w$ is bijectively mapped to an $m_i \in S_w$ and a $k_i \in S_w$.

Theorem 2.4.5 *For any even $n \geq 2$ and $w \geq 1$, $\mathsf{NHS}[n,w]$ is 2^{-w+1}-AU on equal-length strings.*

The proof of Theorem 2.4.5 is identical to the proof for Theorem 2.4.2 with two exceptions: instead of U_w we take elements from S_w, and in the end we use Lemma 2.4.7, which guarantees a bound of *two* on the number of k_1 values satisfying $c(m_1 +_w k_1) - c'(m_1' +_w k_1) + y = 0$. To prove Lemma 2.4.7 we begin by restating Lemma 2.4.4 for the signed case. Recall that $D_w = \{-2^w + 1, \ldots, 2^w - 1\}$.

Lemma 2.4.6 *Let $x \in D_w$ be nonzero. Then for any $y \in S_{2w}$, there exists at most one $a \in S_w$ such that $ax = y$ in $Z/2^{2w}$.*

Proof: Notice that the set of possible differences of any two elements of S_w is again D_w. (This follows from the fact that the obtainable differences for two elements of *any* set of j consecutive integers is always the same, namely $\{-j+1, \ldots, j-1\}$.) Since the proof of Lemma 2.4.4 depends only on D_w, we may recycle the proof without modification. ∎

Lemma 2.4.7 *Let c and c' be distinct values from S_w. Then for any $m, m' \in S_w$ and any $y \in S_{2w}$ there exist at most two $k \in S_w$ such that $c(k +_w m) = c'(k +_w m') + y$ in $Z/2^{2w}$.*

Proof: As in the proof to Lemma 2.4.3, we again notice it is sufficient to prove the case where $m = 0$. We now consider two cases depending on whether $m' < 0$ or $m' \geq 0$. We show in either case there can be at most two values of k satisfying $c(k +_w m) = c'(k +_w m') + y$ in $Z/2^{2w}$.

CASE 1: $m' < 0$. Since $k \in S_w$, we know that $(k +_w m')$ is either $k + m' + 2^w$ (if $k + m' < -2^{w-1}$), or $k + m'$ (if $-2^{w-1} \leq k + m'$). Clearly we cannot have $k + m' \geq 2^{w-1}$. Substituting these values and moving k to the left yields

$$k(c - c') = (m' + 2^w)c' + y \quad \text{and} \quad k + m' < -2^{w-1}$$

$$k(c - c') = m'c' + y \quad \text{and} \quad k + m' < 2^{w-1}$$

But Lemma 2.4.6 tells us there can be at most one solution to each of the above equations.

CASE 2: $m' \geq 0$. Since $k \in S_w$, we now have that $(k +_w m')$ is either $k + m'$ (if $k + m' < 2^{w-1}$) or $k + m' - 2^w$ (if $k + m' \geq 2^{w-1}$). Clearly we cannot have $k + m' < -2^{w-1}$. Substituting these values and moving k to the left yields

$$k(c - c') = m'c' + y \quad \text{and} \quad k + m' < 2^{w-1}$$

$$k(c - c') = (m' - 2^w)c' + y \quad \text{and} \quad k + m' \geq 2^{w-1}$$

But again Lemma 2.4.6 tells us there can be at most one solution to each of the above equations. ∎

A LOWER BOUND. We now show that the bound for NHS is nearly tight by exhibiting two equal-length messages where the probability of collision is very close to 2^{-w+1}.

Theorem 2.4.8 *Fix an even $n \geq 2$ and let $m_i = 0$ for $1 \leq i \leq n$. Let $m'_1 = m'_2 = -2^{w-1}$ and $m'_i = 0$ for $3 \leq i \leq n$. Then*

$$\Pr_{K \leftarrow \mathsf{NHS}}\left[\, \mathsf{NHS}_K(M) = \mathsf{NHS}_K(M') \,\right] \;\geq\; 2^{-w+1} - 2^{1-2w}$$

where the M and M' are mapped in the usual way from m_i and m'_i, respectively.

Proof: As usual, assume henceforth that all arithmetic is in $Z/2^{2w}$. Invoking the definition

of NHS, we will show

$$(k_1 +_w m_1)(k_2 +_w m_2) = (k_1 +_w m_1')(k_2 +_w m_2') + y$$

has at least $2^{w+1} - 2$ solutions in $k_1, k_2 \in S_w$. This will imply the theorem. As in the proof to Theorem 2.4.2, y is the collection of terms for the m_i, m_i', and k_i where $i > 2$. Since we have $m_i = m_i' = 0$ for $i > 2$, y is clearly 0. Therefore we wish to prove

$$k_1 k_2 = (k_1 -_w 2^{w-1})(k_2 -_w 2^{w-1})$$

has at least $2^{w+1} - 2$ solutions. To remove the inner moduli, we let $i_1 = 1$ if $k_1 < 0$ and $i_1 = 0$ otherwise. Define $i_2 = 1$ if $k_2 < 0$ and $i_2 = 0$ otherwise. Now we may re-write the above as

$$k_1 k_2 = (k_1 - 2^{w-1} + i_1 2^w)(k_2 - 2^{w-1} + i_2 2^w).$$

Multiplying the right side out and rearranging terms we have

$$2^{w-1}(k_1(2i_2 - 1) + k_2(2i_1 - 1)) + 2^{2w-2} = 0.$$

Multiplying through by 4, we arrive at

$$2^{w+1}(k_1(2i_2 - 1) + k_2(2i_1 - 1)) = 0.$$

We notice that all $k_1, k_2 \in S_w$ such that $|k_1| + |k_2| = 2^{w-1}$ will satisfy the above. Now we count the number of k_1, k_2 which work: we see that for each of the $2^w - 2$ choices of $k_1 \in S_w - \{0, -2^{w-1}\}$ there are two values of k_2 causing $|k_1| + |k_2| = 2^{w-1}$, yielding $2^{w+1} - 4$ solutions. And of course two further solutions are $k_1 = -2^{w-1},\ k_2 = 0$ and $k_1 = 0,\ k_2 = -2^{w-1}$. And so there are at least $2^{w+1} - 2$ total solutions. ∎

2.5 Reducing the Collision Probability: Toeplitz Extensions

The hash-function families NH and NHS are not yet suitable for use in a MAC: they operate only on strings of "convenient" lengths (ℓw-bit strings for even $\ell \leq n$); their collision probability may be higher than desired (2^{-w} when one may want 2^{-2w} or 2^{-4w}); and they guarantee low collision probability only for strings of equal length. We begin the process of removing these deficiencies, here describing a method to square the collision probability (and thus lower them) at a cost far less than doubling the key length.

2.5.1 The Toeplitz Approach

To reduce the collision probability for 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. For example, if we concatenate the results from, say, four independent instances of the hash function, the collision probability drops from 2^{-w} to 2^{-4w}. But 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 another key, and we hash again.

For example, to reduce the collision probability to 2^{-64} for $\mathsf{NH}[n, 16]$, we can choose an underlying hash key $K = K_1 \| \cdots \| K_{n+6}$ and then hash with the four derived keys $K_1 \| \cdots \| K_n$, $K_3 \| \cdots \| K_{n+2}$, $K_5 \| \cdots \| K_{n+4}$, and $K_7 \| \cdots \| K_{n+6}$. 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 the derived keys are related it is not clear that the collision probability drops to the desired value of 2^{-64}. Although there are established results which yield this bound

(eg., [28]), they only apply to linear hashing schemes over fields. Instead, NH is non-linear and operates over a combination of rings ($Z/2^w$ and $Z/2^{2w}$). In Theorem 2.5.1 we prove that the Toeplitz construction nonetheless achieves the desired bound in the case of NH.

2.5.2 The Unsigned Case

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

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

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

The following shows that NH^{T} enjoys the best bound that one could hope for.

Theorem 2.5.1 *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.*

Proof: We refer to $\mathsf{NH}^{\mathsf{T}}[n, w, t]$ as NH^{T} and to $\mathsf{NH}[n, w]$ as NH, where the parameters n, w, and t are as in the theorem statement.

Let M and M' be distinct members of the domain A with $|M| = |M'|$. We are required to show

$$\Pr_{K \leftarrow \mathsf{NH}^{\mathsf{T}}} \left[\mathsf{NH}^{\mathsf{T}}_K(M) = \mathsf{NH}^{\mathsf{T}}_K(M') \right] \leq 2^{-wt}.$$

As usual, we view M, M', K as sequences of w-bit words, $M = M_1 \| \cdots \| M_\ell$, $M' = M'_1 \| \cdots \| M'_\ell$ and $K = K_1 \| \cdots \| K_{n+2(t-1)}$, where the M_i, M'_i and K_i are each w-

bits long. We denote by m_i, m'_i, and k_i the w-bit integers corresponding to M_i, M'_i, and K_i, respectively. Again, for this proof assume all arithmetic is carried out in $Z/2^{2w}$. For $j \in \{1, \ldots, t\}$, define E_j as

$$E_j : \sum_{i=1}^{\ell/2} (k_{2i+2j-3} +_w m_{2i-1})(k_{2i+2j-2} +_w m_{2i}) = \sum_{i=1}^{\ell/2} (k_{2i+2j-3} +_w m'_{2i-1})(k_{2i+2j-2} +_w m'_{2i})$$

and invoke the definition of NH^{T} to rewrite the above probability as

$$\Pr_{K \leftarrow \mathsf{NH}^{\mathsf{T}}}[E_1 \wedge E_2 \wedge \cdots \wedge E_t]. \tag{2.5.1}$$

We call each term in the summations of the E_j a "clause" (for example, one of the clauses is $(k_1 +_w m_1)(k_2 +_w m_2)$). We refer to the jth equality in Equation 2.5.1 as Equality E_j.

Without loss of generality, we can assume that M and M' disagree in the last clause (ie., that $m_{\ell-1} \neq m'_{\ell-1}$ or $m_\ell \neq m'_\ell$). To see this, observe that if M and M' agree in the last few clauses, then each E_j is satisfied by a key K if and only if it is also satisfied when omitting these last few clauses. Hence, we could truncate M and M' after the last clause in which they disagree, and still have exactly the same set of keys causing collisions.

Assume now that $m_{\ell-1} \neq m'_{\ell-1}$. (The proof may easily be restated if we instead have $m_\ell \neq m'_\ell$. This case is symmetric due to the fact we shift the key by two words.) We proceed by proving that for all $j \in \{1, \ldots, t\}$, $\Pr[E_j \text{ is true} \mid E_1, \ldots, E_{j-1} \text{ are true}] \leq 2^{-w}$, implying the theorem.

For E_1, the claim is satisfied due to Theorem 2.4.2. For $j > 1$, notice that Equalities E_1 through E_{j-1} depend only on key words $k_1, \ldots, k_{\ell+2j-4}$, whereas Equality E_j depends also on key words $k_{\ell+2j-3}$ and $k_{\ell+2j-2}$. Fix k_1 through $k_{\ell+2j-4}$ such that Equalities E_1 through E_{j-1} are satisfied, and fix any value for $k_{\ell+2j-3}$. We prove that there is at most one value of $k_{\ell+2j-2}$ satisfying E_j. To achieve this we follow the same technique used in

Theorem 2.4.2. Let

$$y = \sum_{i=1}^{\ell/2-1} (k_{2i+2j-3} +_w m_{2i-1})(k_{2i+2j-2} +_w m_{2i}) - \sum_{i=1}^{\ell/2-1} (k_{2i+2j-3} +_w m'_{2i-1})(k_{2i+2j-2} +_w m'_{2i})$$

and let $c = (k_{\ell+2j-3} +_w m_{\ell-1})$ and $c' = (k_{\ell+2j-3} +_w m'_{\ell-1})$, and then rewrite E_j as

$$c(k_{\ell+2j-2} +_w m_\ell) + y = c'(k_{\ell+2j-2} +_w m'_\ell).$$

Since we assumed $m_{\ell-1} \neq m'_{\ell-1}$ we know $c \neq c'$; clearly $m_\ell, m'_\ell, c, c' \in U_w$ so Lemma 2.4.3 tells us there is at most one value of $k_{\ell+2j-2}$ satisfying this equation, which completes the proof. ∎

COMMENT. With UMAC, we sometimes use shift amounts of more than two words; this clearly does not interfere with the validity of our proof, provided the number of shifted words is even.

2.5.3 The Signed Case

Now we consider the Toeplitz construction on NHS, the signed version of NH. The only significant change in the analysis for NH^T will be the effect of the higher collision probability of NHS.

We define the hash family $\mathsf{NHS}^\mathsf{T}[n, w, t]$ ("Toeplitz-NHS") exactly as we did for NH, but we instead use NHS as the underlying hash function. Now we restate Theorem 2.5.1 for the signed case.

Theorem 2.5.2 *For any $w, t \geq 1$ and any even $n \geq 2$, $\mathsf{NHS}^\mathsf{T}[n, w, t]$ is $2^{t(-w+1)}$-AU on equal-length strings.*

Proof: The proof is precisely the same as the proof to Theorem 2.5.1 except we use Theorem 2.4.5 in place of Theorem 2.4.2 and Lemma 2.4.7 in place of Lemma 2.4.3. Then,

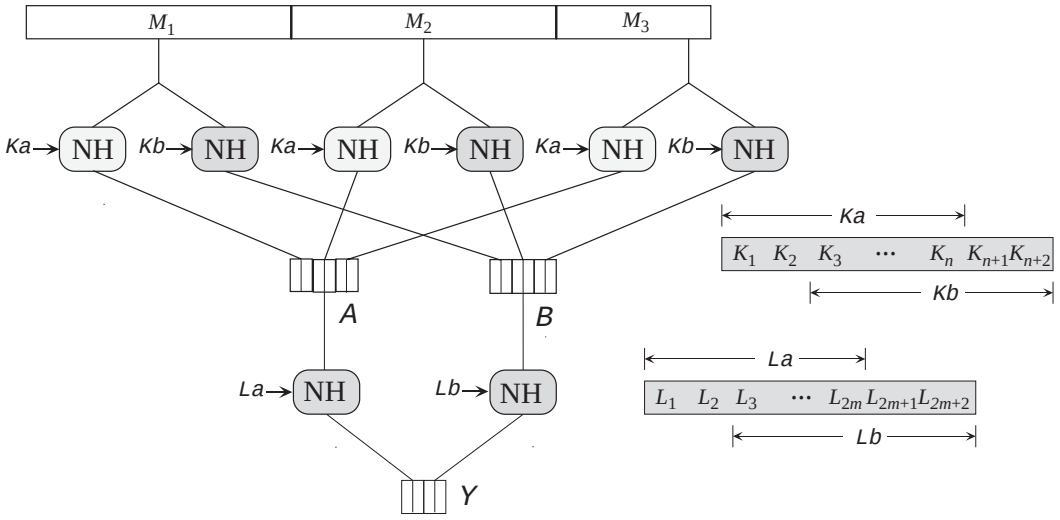


Figure 2.5: *The T2 scheme. Each block of the message is hashed twice, first using a key Ka, and then using a key Kb. Key Kb is the shift of key Ka. The outputs of the Ka-hashes are concatenated and La hashed, and the outputs of the Kb-hashes are concatenated and then Lb hashed. Key Lb is once again the shift of key La.*

using the same notation as Theorem 2.5.1, $\Pr_{K \leftarrow \mathsf{NHS^{T}}}[E_j \mid E_1, \ldots, E_{j-1}] \leq 2^{-w+1}$, and so $\Pr_{K \leftarrow \mathsf{NHS^{T}}}[E_1 \wedge E_2 \wedge \cdots \wedge E_t] \leq 2^{t(-w+1)}$, yielding our result. ∎

2.5.4 Shorter Keys: T2

Fix an even $n \geq 2$ and $m, w \geq 1$. We describe the family of hash functions $\mathsf{NH^{T2}}[n, m, w]$ and the family of hash functions $\mathsf{NHS^{T2}}[n, m, w]$. Both families have domain $A = \{0,1\}^{2w} \cup \{0,1\}^{4w} \cup \cdots \cup \{0,1\}^{nmw}$ and range $B = \{0,1\}^{4w}$. (That is, the input consists of an even number of w-bit words—at least two words and at most mw words. The output is four words.) A function $\mathsf{NH^{T2}}(K \parallel L, \cdot)$ from the family $\mathsf{NH^{T2}}[n, m, w]$ is named by a key $K \parallel L$ having $(n+2)w + (2m+2)w$ bits. (The part of the key denoted K has $n+2$ words, and the part of the key denoted L has $2m+2$ words.) We define $\mathsf{NH^{T2}}(K \parallel L,\ M)$ below. See Figure 2.5 as well.

function $\mathsf{NH^{T2}}(K \parallel L,\ M)$
1. Let $Ka = K[1..nw]$
2. Let $Kb = K[2w+1..(n+2)w]$

3. Let $La = L[1..2mw]$
4. Let $Lb = L[2w + ..(2m + 2)w]$
5. Let $M_1 \parallel \cdots \parallel M_t = M$, where $t = \lceil |M|/(nw) \rceil$,
   $|M_1| = \cdots = |M_{t-1}| = nw$, and $2w \leq |M_t| \leq nw$
6. $A = \mathsf{NH}_{Ka}(M) \parallel \cdots \parallel \mathsf{NH}_{Ka}(M_t)$
7. $B = \mathsf{NH}_{Kb}(M) \parallel \cdots \parallel \mathsf{NH}_{Kb}(M_t)$
8. Return $\mathsf{NH}_{La}(A) \parallel \mathsf{NH}_{Lb}(B)$

The family of hash functions $\mathsf{NHS}^{\mathsf{T2}}[n, m, w]$ is defined exactly as above except for using NHS instead of NH throughout the construction. Proofs of the following two theorems are omitted.

Theorem 2.5.3 *For any $m, w \geq 1$ and any even $n \geq 2$, $\mathsf{NH}^{\mathsf{T2}}[n, m, w]$ is 2^{-2w+2}-AU on equal-length strings.*

Theorem 2.5.4 *For any $m, w \geq 1$ and any even $n \geq 2$, $\mathsf{NHS}^{\mathsf{T2}}[n, m, w]$ is 2^{-2w+4}-AU on equal-length strings.*

2.6 Padding, Concatenation, and Length Annotation

With NH^{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 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 β 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. Each function $h \in \mathsf{H}$ gives rise to a corresponding function $h^* \in \mathsf{H}^*$, A random function of H^* is determined by taking a random function $h \in \mathsf{H}$ and following the procedure below.

First partition the message *Msg* into some number of "full blocks" (each containing exactly a bits) and then a "last block" (which might contain fewer bits). Hash each of the full blocks by applying h, and then hash the last block by first zero-padding it to the next domain point and then applying h. Finally, concatenate all the hash values (each having β bits), and append (an encoding of) the length of the last block prior to padding.[3] Pseudocode follows.

function $h^*(Msg)$
1. If $M = \lambda$ then return 0^α
2. View *Msg* as a sequence of "blocks", $Msg = Msg_1 \| \cdots \| Msg_t$, with $|Msg_j| = a$ for all $1 \le j < t$, and $1 \le |Msg_t| \le a$
3. Let *Len* be an α-bit string that encodes $|Msg| \bmod a$
4. Let $i \ge 0$ be the least number such that $Msg_t \| 0^i \in A$
5. $Msg_t = Msg_t \| 0^i$
6. Return $h(Msg_1) \| \cdots \| h(Msg_t) \| Len$

Analysis. The following proposition indicates that we have correctly extended H to H^*.

Proposition 2.6.1 *Let $I \subseteq \mathsf{N}$ be a nonempty finite set, let $\beta \ge 1$ 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 H as described above. Suppose H is ϵ-AU on strings of equal length. Then H^* is ϵ-AU (across all strings).*

Proof: The idea is to note that when distinct messages *Msg* and *Msg′* have different lengths then, by our definition of H^*, their hashes can never collide; and if, on the other

[3] Since the length of the last block (prior to padding) is between 1 and a, it suffices to take this number modulo a; it still specifies the length of the last block. The length annotation can equivalently be regarded as the length of the original message *Msg* modulo a.

hand, distinct messages Msg and Msg' have the same length, then, because H is ϵ-AU on equal-length strings, the hash under $h^* \leftarrow \mathsf{H}^*$ of Msg and Msg' agree with probability at most ϵ even if one fixes attention on some particular spot where Msg and Msg' differ.

We must show for all distinct $Msg, Msg' \in \{0,1\}^*$ that $\Pr_{h^* \in \mathsf{H}^*}[h^*(Msg) = h^*(Msg')]$ is no more than ϵ. So fix distinct $Msg, Msg' \in \{0,1\}^*$ and consider the following cases, based on the relative lengths of these strings:

Case 1. Let $t = \lceil |Msg|/a \rceil$ and let $t' = \lceil |Msg'|/a \rceil$. If the length of Msg and Msg' are sufficiently different that $t \neq t'$ then the probability that $h^*(Msg) = h^*(Msg')$ is zero, since $h^*(Msg)$ and $h^*(Msg')$ have different lengths.

Case 2. With t and t' as above, assume $|Msg|$ differs from $|Msg'|$ but in a manner such that $t = t'$. This means that the lengths of the last blocks of Msg and Msg' differ, and therefore the last α bits of $h^*(Msg)$ (representing the length of the last block) differ from the last α bits of $h^*(Msg')$, so the probability of collision is zero.

Case 3. As the final case, assume $|Msg| = |Msg'|$. Suppose Msg is partitioned into $Msg_1 \| \cdots \| Msg_t$ and Msg' is partitioned into $Msg'_1 \| \cdots \| Msg'_t$. Since $Msg \neq Msg'$ there exists some index i such that $Msg_i \neq Msg'_i$. Fix such an i. Of course $|Msg_i| = |Msg'_i|$. If $i = t$ then we replace Msg_i and Msg'_i by what we get after zero-padding them, but the lengths of the (now padded) blocks Msg_i and Msg'_i will still be identical and these blocks will still differ. Hence we have that $\Pr_{h \in \mathsf{H}}[h(Msg_i) = h(Msg'_i)] \leq \epsilon$. Moreover, $h(Msg_i) \neq h(Msg'_i)$ implies that $h^*(Msg) \neq h^*(Msg')$, since one can infer $h(Msg_i)$ from $h^*(Msg)$ and one can infer $h(Msg'_i)$ from $h^*(Msg')$. (This uses the fact the range of h is fixed-length strings.) It follows that $h^*(Msg)$ and $h^*(Msg')$ coincide with probability at most ϵ. ∎

COMMENT. For implementation convenience the specification for UMACin Appendix C is byte-oriented, and so the domain of the hash function defined in the specification is actually $(\{0,1\}^8)^*$, not $\{0,1\}^*$.

2.7 Final Extensions: Stride, Endianness, Key Shifts

There are a few more useful adjustments to enhance the performance, generality, or and ease-of-implementation of the hash families we have constructed. Although we have defined NHX with parameters n, w, t, what is specified in Appendix C actually has additional parameters: "message stride," "endianness," "key shift," "signed/unsigned," and parameters to handle two-level (2L) hashing. We have already analyzed the effect of using signed integers and two-level hashing in Section 2.4.4 and Section 2.5.4. Let us now describe the parameters we have not touched on and argue that they do not affect the analysis.

First we make the simple observation that any bijection π applied to the inputs of an ϵ-AU hash-function family leaves the collision bound unchanged. This can be observed by simply looking at the definition of ϵ-AU: the statement requires a certain probability hold for any two distinct inputs M, M' from the domain. But if $M \neq M'$ then certainly $\pi(M) \neq \pi(M')$ and so the bound still holds.

MESSAGE STRIDE. In NH we multiplied $(k_{2i-1} +_w m_{2i-1})$ by $(k_{2i} +_w m_{2i})$ for $1 \leq i \leq \ell/2$. As described in Section 2.2.2, we may prefer to pair these multiplications differently, like

$$(k_1 +_w m_1)(k_5 +_w m_5) + (k_2 +_w m_2)(k_6 +_w m_6) + \cdots$$

to take advantage of machine architectural characteristics. The number of words separating words paired in multiplication is the "message stride". In this example it is 4: m_1 and m_5 are separated by 4 words. Such rearrangements do not affect our collision bound since they

amount to applying a fixed permutation to the input and key; since the keys are random, permuting them is irrelevant.

ENDIANNESS. Whether we read a message using big-endian or little-endian conventions again can be viewed as a bijection on the messages.

KEY SHIFT. In our proof of Theorem 2.5.1 we shifted our key by two words to acquire the "next" key. In some cases we may wish to shift by more than two words for performance reasons (specifically in the MMX implementations). As noted in the comment following that proof, a larger shift does not invalidate the argument.

2.8 From Hash to MAC

In this section we describe a way to make a secure MAC from an ϵ-AU family of hash functions (with small ϵ) and a secure pseudorandom function (PRF). We also describe some constructions for suitable PRFs. We begin with the formal definitions for MACs and PRFs.

2.8.1 Security Definitions

MACS AND THEIR SECURITY. For this chapter a MAC scheme $\Sigma = (\text{KEY}, \text{TAG})$ consists of two things: a *key generator* KEY and a *tag generator* TAG. There are also four associated nonempty sets: Key, which denotes the set of possible keys, and Message, Nonce, and Tag, which are sets of strings. The sets Nonce and Tag are finite, with $\mathsf{Nonce} = \{0,1\}^{\eta}$ and $\mathsf{Tag} = \{0,1\}^{\tau}$. These different sets are used to describe the domain and range of KEY and TAG, as we now explain.

The key generator KEY takes no arguments and probabilistically produces an element $Key \in \mathsf{Key}$. This process is denoted $Key \leftarrow \text{KEY}()$. The tag generator TAG takes $Key \in$

Key, $M \in \mathsf{Message}$, and $\mathit{Nonce} \in \mathsf{Nonce}$, and deterministically yields $\mathit{Tag} \in \mathsf{Tag}$. The first argument to Tag will be written as a subscript, as in $\mathit{Tag} = \textsc{Tag}_{\mathit{Key}}(M, \mathit{Nonce})$.

An adversary F for attacking MAC scheme $\Sigma = (\textsc{Key}, \textsc{Tag})$ is an algorithm with access to two oracles, denoted Tag and Vf. Specifically, F's oracles behave as follows. First the key generator is run to compute $\mathit{Key} \leftarrow \mathsf{Key}()$. From then on, when presented with a query $(M, \mathit{Nonce}) \in \mathsf{Message} \times \mathsf{Nonce}$, the Tag oracle returns $\textsc{Tag}_{\mathit{Key}}(M, \mathit{Nonce})$. (If the query (M, Nonce) is outside of the indicated domain, the oracle returns the empty string.) When presented with a query $(M, \mathit{Nonce}, \mathit{Tag}) \in \mathsf{Message} \times \mathsf{Nonce} \times \mathsf{Tag}$, the Vf oracle returns 1 if $\textsc{Tag}_{\mathit{Key}}(M, \mathit{Nonce}) = \mathit{Tag}$, and it returns 0 otherwise. (If the query $(M, \mathit{Nonce}, \mathit{Tag})$ is outside of the indicated domain, the oracle returns 0.) To disallow replay attacks and meet other goals one can, alternatively, make the Vf oracle stateful and more restrictive with when it returns 1. We will not pursue these possibilities here.

In an execution of an adversary with her oracles, she is said to *forge*, or *succeed*, if she asks the Vf oracle some query $(M, \mathit{Nonce}, \mathit{Tag})$ which returns 1 even though (M, Nonce) was not an earlier query to the Tag oracle. The success of adversary F in attacking Σ, denoted $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F)$, is the probability that F succeeds. Informally, a MAC scheme Σ is good if for every reasonable adversary F the value $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F)$ is suitably small. To talk of this conveniently, we overload the notation and let $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t, q_s, q_v, \mu)$ be the maximal value of $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F)$ among adversaries that run in time at most t, ask at most q_s Tag queries, ask at most q_v Vf queries, and all of these queries total at most μ bits. One assumes some fixed RAM model of computation and running time is understood to mean the actual running time plus the size of the program description.

Remarks. Our notion for MAC security is very strong, insofar as we have let the adversary manipulate the nonce (just so long as they *are* nonces), and then we regarded the adversary

as successful if she could forge any new message (or even an old message, but with a nonce different from any already used for it). In actuality we expect the nonce to be a counter which is not under the adversary's control. Our notion of security says that even if the adversary *could* control the nonce, *still* she would be unable to forge, even in a very weak sense.

The definition above explicitly allows verification queries. This choice is pretty inconsequential, in the following sense: $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t, q_s, q_v, \mu) \leq q_v \cdot \mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t + O(\mu), q_s, 1, \mu)$ for any MAC scheme Π. That is, suppose we allowed only one verification query. This is equivalent to having the adversary output her one and only attempt at a forgery, $(M, \mathit{Nonce}, \mathit{Tag})$, at the end of her execution. Then generalizing (as our definition does) to $q_v \geq 1$ verification queries will increase the adversary's chance of success by at most q_v. The proof is simple and the observation is well known, so a proof is omitted. But we will use this fact in the proof of Lemma 2.8.1.

For substantial q_v the value $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t, q_s, q_v, \mu)$ may become larger than desired. For this reason it may be necessary to architecturally limit q_v to 1—for example, by tearing down a connection when a forgery attempt is detected. There are further approaches to keeping $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(t, q_s, q_v, \mu)$ small when one is imagining q_v large.

PRFs And Their Security. A pseudorandom function (PRF) (with key-length α, arbitrary argument length, and output-length β) is a family of functions $\mathbb{F} : \{0,1\}^{\alpha} \times \{0,1\}^{*} \to \{0,1\}^{\beta}$. (See Section 2.4.1). Let $\mathsf{Rand}(\beta)$ be the family of functions from $\{0,1\}^{*}$ to $\{0,1\}^{\beta}$ in which choosing a random $\rho \leftarrow \mathsf{Rand}(\beta)$ means associating to each string $x \in \{0,1\}^{*}$ a random string $\rho(x) \in \{0,1\}^{\beta}$.

An adversary D for attacking the PRF $\mathbb{F} : \{0,1\}^{\alpha} \times \{0,1\}^{*} \to \{0,1\}^{\beta}$ is given an oracle g which is either a random element of $\mathbb{F}$ or a random element of $\mathsf{Rand}(\beta)$.

The adversary tries to distinguish these two possibilities. Her advantage is defined as $\mathsf{Adv}^{\mathrm{prf}}_{\mathbb{F}}(D) = \Pr_{a\in\{0,1\}^{\alpha}}[D^{\mathbb{F}_a(\cdot)} = 1] - \Pr_{\rho\in\mathsf{Rand}(\beta)}[D^{\rho(\cdot)} = 1]$. Informally, a PRF $\mathbb{F}$ is good if for every reasonable adversary D the value $\mathsf{Adv}_{\mathbb{F}}(D)$ is small. To talk of this conveniently we overload the notation and let $\mathsf{Adv}^{\mathrm{prf}}_{\mathbb{F}}(t, q, \mu)$ be the maximal value of $\mathsf{Adv}^{\mathrm{prf}}_{\mathbb{F}}(D)$ among adversaries that run in time at most t, ask at most q oracle queries, and these queries total at most μ bits.

2.8.2 Definition of the PRF(HASH, Nonce) Construction

We use a family of (hash) functions $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ and a family of (random or pseudorandom) functions $\mathbb{F} = \{f : \{0,1\}^* \to \{0,1\}^{\tau}\}$. 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 $\mathsf{UMAC}[\mathsf{H}, \mathbb{F}] = (\textsc{Key}, \textsc{Tag})$ is then defined as follows:

function KEY ()	**function** $\textsc{Tag}_{(f,h)}$ (M, $Nonce$)
$f \leftarrow \mathbb{F}$	
$h \leftarrow \mathsf{H}$	**return** f ($\langle h(M),\ Nonce\rangle$)
return (f, h)	

The keyspace for this MAC is $\mathsf{Key} = \mathsf{H} \times \mathbb{F}$; that is, a random key for the MAC is a random hash function $h \in \mathsf{H}$ together with a random function $f \in \mathbb{F}$.

ANALYSIS. We begin with the information-theoretic version of the scheme.

Lemma 2.8.1 *Let $\epsilon \geq 0$ be a real number and let $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ be an ϵ-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 that makes at most q_v verification queries we have that $\mathsf{Succ}^{\mathrm{mac}}_{\Sigma}(F) \leq q_v(\epsilon + 2^{-\tau})$.*

Proof: First assume that F prepares a single verification query which, without loss of generality, is its last oracle query. Run adversary F in the experiment which defines success in attacking $\Sigma = (\textsc{Key}, \textsc{Tag})$. In this experiment the adversary is provided an oracle which behaves as follows. First it chooses a random $\rho \leftarrow \mathsf{Rand}(\tau)$ and $h \leftarrow \mathsf{H}$. Now F asks the sequence of tag-generation queries $(M_1, \mathit{Nonce}_1), \ldots, (M_q, \mathit{Nonce}_q)$, getting responses $\mathit{Tag}_1, \ldots, \mathit{Tag}_q$, where $\mathit{Tag}_i = \textsc{Tag}_{(\rho,h)}(\langle h(M_i), \mathit{Nonce}_i\rangle)$. Based on these answers (and possibly internal coin flips) the adversary produces its verification query $(M, \mathit{Nonce}, \mathit{Tag})$. Since we are bounding from above the probability that F is successful we may assume of F's behavior that $\mathit{Nonce}_1, \ldots, \mathit{Nonce}_q$ are distinct and that $(M, \mathit{Nonce}) \notin \{(M_1, \mathit{Nonce}_1), \ldots, (M_q, \mathit{Nonce}_q)\}$ for otherwise, by definition, F does not succeed. Let Forge be the event that the adversary is successful: $\mathit{Tag} = \rho(\langle h(M), \mathit{Nonce}\rangle)$. Let $y_j = \langle M_j, \mathit{Nonce}_j\rangle$ for $1 \le j \le q$ and let $y = \langle M, \mathit{Nonce}\rangle$. Note that $y_1, \ldots, y_q$ are distinct, since $\mathit{Nonce}_1, \ldots \mathit{Nonce}_q$ are distinct and $\langle \cdot, \cdot \rangle$ is a (reversible) encoding.

Let Repeat be the event that Nonce is one of $\{\mathit{Nonce}_1, \ldots, \mathit{Nonce}_q\}$. Observe that $\Pr[\mathsf{Forge}|\overline{\mathsf{Repeat}}] \le 2^{-\tau}$ since when $\overline{\mathsf{Repeat}}$ holds the adversary must predict $\sigma(y)$ given $\sigma(y_1), \ldots, \sigma(y_q)$, where $y_1, \ldots, y_q$ are all distinct from y.

We wish to bound $\Pr[\mathsf{Forge}|\mathsf{Repeat}]$. Let i be the index such that $\mathit{Nonce} = \mathit{Nonce}_i$. There is a unique such i because $\mathit{Nonce}_1, \ldots, \mathit{Nonce}_q$ are distinct and Nonce is among them. Let $\mathsf{Collision}$ be the event that Repeat holds and $h(M)$ and $h(M_i)$ are equal. We note that $\Pr[\mathsf{Forge}|\overline{\mathsf{Collision}}] \le 2^{-\tau}$ since, as before, the adversary must predict $\sigma(y)$ having seen only $\sigma(y_1), \ldots, \sigma(y_q)$, where $y \notin \{y_1, \ldots, y_q\}$.

We claim $\Pr[\mathsf{Collision}] \le \epsilon$. By definition of H being ϵ-AU we know if one chooses a random $h \leftarrow \mathsf{H}$ and gives the adversary no information correlated to h then the adversary's

chance of producing distinct M and M_i which collide under h is at most ϵ. The point to note is that the adversary has, in fact, obtained no information correlated to h. In response to her queries she obtains the images under ρ of the distinct points $y_1, \ldots, y_q$. These values, $Tag_1, \ldots, Tag_q$, are random τ-bit strings; the adversary F would be provided an identical distribution of views if the oracle were replaced by one which, in response to a query, returned a random τ-bit string.

The result follows. We have that $\Pr[\mathsf{Forge}] \leq \max\{\Pr[\mathsf{Forge}|\mathsf{Repeat}],\ \Pr[\mathsf{Forge}|\overline{\mathsf{Repeat}}]\}$, with the second expression being at most $2^{-\tau}$. On the other hand,

$$\begin{aligned}
\Pr[\mathsf{Forge}|\mathsf{Repeat}] &= \Pr[\mathsf{Forge}|\mathsf{Collision}]\Pr[\mathsf{Collision}] + \Pr[\mathsf{Forge}|\overline{\mathsf{Collision}}]\Pr[\overline{\mathsf{Collision}}] \\
&\leq \Pr[\mathsf{Collision}] + \Pr[\mathsf{Forge}|\overline{\mathsf{Collision}}] \\
&\leq \epsilon + 2^{-\tau},
\end{aligned}$$

as we have argued. We conclude that $\Pr[\mathsf{Forge}] \leq \epsilon + 2^{-\tau}$. To finish the proof, use the observation, mentioned in the Remarks of Section 2.8.1, that allowing q_v verification queries at most increases the adversary's chance of success by a multiplicative factor of q_v. ∎

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

To make this formal, we use the following notations. If 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 μ bits.

The proof of the following, being standard, is omitted.

Theorem 2.8.2 *Let $\epsilon \geq 0$ be a real number, let $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ be an ϵ-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 $\Sigma = \mathsf{UMAC}[\mathsf{H}, \mathbb{F}]$ be the MAC scheme described above. Then*

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

where $t' = t + \mathsf{Time}_{\mathsf{H}} + \mathsf{Time}_h(\mu) + O(\mu)$ and $q' = q_s + q_v$ and $\mu' = O(\mu)$. ∎

2.8.3 Discussion

We have now defined the PRF(HASH, Nonce) construction, but we wish to make a couple of comments about it.

Comparison With Carter-Wegman Method. Compared to the original suggestion of [39], 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 H: it need only be ϵ-AU. The original approach of [39] needs of H the stronger property of being "XOR almost-universal" [25]: for all distinct M, M' and for all C, one must have that $\Pr_h[h(M) \oplus h(M') = C] \leq \epsilon$. 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 [39] should have fixed-length output. On the other hand, our cryptographic tool is effectively stronger than what the complexity-theoretic version of [39] requires: we need a PRF over $\{0,1\}^*$ (or at least over the domain $\mathsf{Encoding}$ of possible $\langle HM, Nonce \rangle$ encodings), while the complexity-theoretic analog of [39] could conveniently use a (fixed-output-length) PRF on the fixed-length strings Nonce. The security bound one gets is the same; both constructions are as good as one could hope for, from the point of view of concrete security.

Comparison With PRF(HASH) Construction. Let us now clarify the role of the nonce in the PRF(HASH, Nonce) scheme. The nonce is essential, in the sense that in its absence the quantitative security bounds are much worse and unacceptable for many applications. Let $\overline{\mathsf{UMAC}}[\mathsf{H}, \mathbb{F}]$ 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 2.8.1 would say the following: Fix a number $\tau \geq 1$, let $\mathsf{H} = \{h : \{0,1\}^* \to \{0,1\}^*\}$ be an ϵ-AU family of hash functions, and let $\Sigma = \mathsf{UMAC}[\mathsf{H}, \mathsf{Rand}(\tau)]$ be the MAC scheme described above. 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 $\epsilon + 2^{-\tau}$. Instead of security which is independent of the number of tags acquired, q, security degrades with the square of that number. When $q = \epsilon^{-1/2}$, there is no security left.

It is important to note that this is not a problem 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 a good chance to find distinct messages, M_1 and M_2, which yield the same authentication tag. If τ is large then this implies that, with high probability, the hash of M_1 equals the hash of M_2. This is crucial information about the hash function which has now been leaked. In particular, for some ϵ-AU hash-function families knowing that M_1 and M_2 collide immediately tells us some third message M_3 which collides with both of them. The adversary can now forge an authentication tag for this message.

The conclusion is that the nonce in the PRF(HASH, Nonce)-scheme cannot be removed. For example, without it, using a 2^{-32}-AU hash function would let you authenticate fewer than 2^{16} messages, which is usually insufficient.

2.8.4 Realizing the PRF

A complete specification of UMAC must describe how to make the requisite PRF. Under our construction the domain of the PRF needs to be strings of arbitrary lengths. The specification associated to this chapter (Appendix C) suggests two ways of realizing such PRFs: one using a cryptographic hash function and using a secure block cipher. Let us briefly describe the two suggested constructions. The second is somewhat novel.

From A Cryptographic Hash Function. Given a cryptographic hash function H (eg., SHA-1) we can use HMAC based on H as our PRF [3, 20]. This method defines the PRF F to be $F_A(M) = H(A \oplus \text{opad} \parallel H(A \oplus \text{ipad} \parallel M))$, where $|A|$ is, typically, the blocksize of H (ie., 512 bits for all well-known constructions) and opad and ipad are distinct $|A|$-bit constants. This construction is shown in [3] to be a PRF under weak (though nonstandard) assumptions about H.

From A Block Cipher. We use a new variant of the CBC-MAC in order to turn a block cipher E into a PRF F which operates over arbitrary-length inputs. Our CBC-MAC variant is more efficient than earlier suggestions but can still be proven secure when E is a good block cipher.

Let $E : \{0,1\}^\alpha \times \{0,1\}^\beta \to \{0,1\}^\beta$ be the block cipher. We define $F_{K1 \parallel K2 \parallel K3}(M)$, where $|K_1| = |K_2| = |K_3| = \alpha$ and $M \in \{0,1\}^*$, as follows.

If M contains an integral number of blocks (ie., $|M|$ is a nonzero multiple of β) then apply the "basic" CBC-MAC construction to M except use key $K2$ for encrypting the last block and use key $K1$ for encrypting the earlier ones. More precisely, writing $M = M_1 \parallel \cdots \parallel M_n$, where $|M_1| = \cdots = |M_n| = \beta$, and letting $y_0 = 0^\beta$, set $y_i = E_{K1}(M_i \oplus y_{i-1})$ for $1 \le i < n$, and then define $F_{K1 \parallel K2 \parallel K3}(M)$ as $E_{K2}(M_n \oplus y_{n-1})$.

If $|M| = 0$ or $|M|$ is not a multiple of β then append to M a 1-bit and then the minimum number of 0-bits so that the resulting string M' will have length which is a multiple of β. Now proceed as above, but using key K_3 in lieu of K_2: that is, writing $M' = M'_1 \| \cdots \| M'_n$, where $|M'_1| = \cdots = |M'_n| = \beta$, and letting $y_0 = 0^\beta$, set $y_i = E_{K1}(M'_i \oplus y_{i-1})$ for $1 \leq i < n$, and then define $F_{K1 \| K2 \| K3}(M)$ as $E_{K3}(M'_n \oplus y_{n-1})$.

Building on [30], which in turn builds on [5], it is possible to show that if E is a good PRF (on its domain) then F is a good PRF (on its domain). The quantitative bounds are as in [30]. Further details are omitted.

We comment that since F_A is being applied to adversarially-unknown points, even if it were to have some cryptographic weakness as a PRF, still this might not, by itself, lead to the resulting MAC being insecure. This is true regardless of how the PRF F is realized.

Chapter 3

Improving UMAC

UMAC (1999) is certainly fast, but aspects of its performance and design could benefit from further attention. Immediately upon the publication of UMAC (1999), its creators began to address the following goals.

- **Short message speeds.** David McGrew and Scott Fluhrer (Cisco Systems) pointed out that much needed message authentication occurs on very short messages, from 43 bytes to 1500 bytes. Whereas UMAC (1999) did optimize for messages shorter than 33 bytes by switching to a conventional MAC faster than UMAC on those lengths, there remained scope for improvement on messages larger than this.

- **Less cryptography.** UMAC (1999) was designed to use NH as a compression function on the message being authenticated before the compressed message (along with a nonce) was acted on by a cryptographic pseudorandom function. The compression function acted to reduce by a fixed ratio the amount of cryptography used in the MAC, but the amount of cryptography was still unbounded and proportional in length to the message being authenticated. This potentially increases UMAC's susceptibility to

cryptographic attack because of the substantial use of cryptography. Reduction to a small, fixed amount of cryptography per tag generated would be desirable for better speed and security.

- **Verifier-selectable assurance.** A new goal in message authentication was suggested to us by David Balenson and David Carman (NAI Labs). If a server generates an authentication tag for some message and then broadcasts the message and tag on a network, it would be nice if each receiver of the message were able to verify the tag to an assurance level well-matched to its available time and computation resources, with lower assurance guarantees requiring less time to achieve.

- **Generally useful hash function.** The NH hash family was very fast, but was not "generally" useful because it needed key material equal in length to the data being hashed. Embedding NH into a compression function was not "generally" useful either because the output of a compression function is not fixed in length. To make NH useful to a wider community it would be nice to embed it efficiently in a function which used a short fixed-length key and produced a short fixed-length output.

Analysis of how and where UMAC (1999) was failing in fulfilling these goals focussed on the pseudorandom function used to process the nonce and compressed message. Even the very shortest messages required either two SHA-1 invocations or two RC6 invocations after NH compression, requiring about 1500 or 500 cycles respectively. Any attempt to make authentication of short messages faster would have to greatly reduce these PRF-related numbers. The way the PRF was used in UMAC (1999) also made the goal of verifier-selectable assurance improbable: every bit of the PRF output depends on every bit of the PRF input, meaning that verifying any bit of the MAC tag would likely require calculation

of the entire PRF input.

3.1 A First Attempt

An obvious solution to many of the PRF problems is to change the way the PRF was used in UMAC (1999) and return to the original Wegman-Carter MAC scheme. Recall that in their scheme if $H = \{h : \{0,1\}^* \to \{0,1\}^\ell\}$ is a universal hash function and $\mathsf{Rand}[\mathsf{Nonces}, \{0,1\}^\ell]$ is the set of all functions from Nonces to ℓ-bit strings, then given members $h \xleftarrow{R} H$ and $f \xleftarrow{R} \mathsf{Rand}[\mathsf{Nonces}, \{0,1\}^\ell]$ the tag for message M and nonce n is $v = f(n) \oplus h(M)$. Such a change makes the requirements of the hash family and the PRF very different from those of UMAC (1999).

In UMAC (1999) the hash function was viewed as a compression function producing a variable length string, whereas in a true Wegman-Carter MAC the hash function must produce a fixed-length string. Likewise, in UMAC (1999) the PRF was required to accept arbitrary-length inputs while in a true Wegman-Carter MAC the PRF need only accept nonces as input and produce strings as long as the hash function's outputs.

The requirement of the PRF in a Wegman-Carter MAC is easily and efficiently accomplished using a block cipher. To approximate a random function $f : \mathsf{Nonces} \to \{0,1\}^\ell$ using, say, $\mathsf{RC6} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$, one first chooses a random $k \xleftarrow{R} \{0,1\}^{128}$ and defines the string $S = \mathsf{RC6}_k(\langle 0 \rangle) \,\|\, \cdots \,\|\, \mathsf{RC6}_k(\langle 2^{128} - 1 \rangle)$ where $\langle i \rangle$ is a 128-bit encoding of the integer i. Then, assuming $\mathsf{Nonces} \subseteq \{0, \ldots, 2^{128} - 1\}$ and $\ell \leq 128$, $f(i)$ returns the i-th ℓ-bit substring of S. When ℓ is shorter than 128, say 32 or 64, and the nonce is maintained as a sequential counter, this PRF can be very efficiently implemented because multiple sequential nonce values can be serviced with a single invocation to $\mathsf{RC6}$. The PRF may now account for as little 75 cycles per 32-bit tag generated. This potentially means

very little cryptography use per tag generated and also removes the PRF as a bottleneck for fast performance on short messages.

How to efficiently hash messages to a fixed length using NH has less obvious solutions. Doing so would certainly advance our aim of providing a "generally" useful hash function for use outside of message authentication. As we will soon see, it also facilitates our goal of allowing verifier-selected assurances.

At a stroke, it appears, moving from the PRF(HASH $\|$ Nonce) method of UMAC (1999) to the more traditional HASH-xor-PRF(Nonce) method shows promise in achieving all of our goals. But first an appropriate hash function must be designed.

3.1.1 Hashing to a Fixed Length

One promising idea is to hash messages using NH and an infinite-length key. In fact, this would also solve the verifier-selectable security goal as well. Imagine that distinct messages M_1 and M_2 are each hashed twice using NH, first with infinite key K and then again with K', a Toeplitz shifted version of K. If NH based on 32-bit words is used, then as we saw in Chapter 2, the probability that $\mathsf{NH}_K(M_1) = \mathsf{NH}_K(M_2)$ is no more than 2^{-32} and the probability that $\mathsf{NH}_K(M_1) \,\|\, \mathsf{NH}_{K'}(M_1) = \mathsf{NH}_K(M_2) \,\|\, \mathsf{NH}_{K'}(M_2)$ is no more than 2^{-64}. This means that if we define our MAC tags for message M and nonce n as the string $v = f(n) \oplus (\mathsf{NH}_K(M) \,\|\, \mathsf{NH}_{K'}(M))$, then a verifier can verify to a 2^{-32} forgery guarantee by computing just $f(n)$, $\mathsf{NH}_K(M)$ and examining the first half of v; or to a 2^{-64} forgery guarantee by computing $f(n)$, $\mathsf{NH}_K(M)$, $\mathsf{NH}_{K'}(M)$ and looking at all of v.[1] The only problem with this scenario is finding the source for an infinite key. The key for this

[1]The sharp reader may notice that we are constructing a Wegman-Carter MAC from an ϵ-AU hash family, which normally does not work. The family NH, however, is actually ϵ-delta-universal, as mentioned in Chapter 2, and is therefore appropriate for message authentication. One might replace $\oplus$ with addition modulo 2^{64}.

suggested MAC would likely need to be generated on-the-fly, as needed, by a cryptographic primitive such as a block or stream cipher. Unfortunately, since the length of the key used in this MAC needs to be the same length as the message being authenticated, this means that "cryptography" is being applied to the message at a one-to-one ratio, and we are back to a MAC speed upper bounded by the speed of the cryptographic primitive.

In UMAC (1999), NH was used as a compression function which reduced the use of cryptography by asking the PRF to act on a shorter string. This same approach can also be used by the MAC of the previous paragraph. Instead of acting cryptographically on every byte of the message, we can compress the message using NH and a fixed-length string, and then act on this compressed string with NH and an infinite key. The use of cryptography is then reduced by the ratio of compression achieved by the first-layer of NH hashing. To make things more clear, let us be more careful in our description.

We would break a message into 2KB chunks and hash each block using NH with a 2KB key. Concatenating the NH outputs would yield a compressed (but unbounded in length) string. This string we would then hash again with NH, but this time using an infinite key. Because this second key is unbounded, NH produces a fixed length output which we can easily use in a Carter-Wegman MAC. The infinite key could be produced on-the-fly using a block- or stream-cipher, but could also have a significant prefix cached, allowing moderate length messages to be hashed without any on-line cryptography during hashing. Multiple iterations of this approach using Toeplitzed keys could be used to achieve lower collision probabilities and maintain selective-assurance verifiability. But, how well does this MAC fulfill our four reasons for refining UMAC?

This new MAC should certainly be faster than UMAC (1999) on short messages. UMAC (1999) used a fast NH layer followed by an expensive PRF layer, whereas this

MAC uses a fast NH layer followed by another fast NH layer followed by a short, fixed cryptographic pad-generation. Test implementations showed this new MAC to be about twice as fast as UMAC (1999) on messages in the 33–512 byte range, and the two MACs performing equally on messages longer than 4KB. This seems to indicate that once full compression is being achieved by the first NH layer (around 2–4KB), it makes no difference whether the second layer is handled by a fast NH or a slower PRF. But, when compression is poor, the disparity between fast NH and slow PRF becomes much more pronounced.

The hash function is both more-generally useful and appropriate for selective-assurance verifiability because it produces a short fixed output that is produced by iteration over independent (except for Toeplitz-based recycling) hashes. To verify a prefix of a tag, simply calculate the hash, iterating a fewer number of times.

The reduction in cryptography is only partial. Because the first-layer NH is still just a compression layer, and the second layer envisions an infinite key (produced cryptographically), an unbounded amount of cryptography is still potentially called for during each hash. But, things are better than in UMAC (1999). The cryptography is used for keying purposes only, meaning that an adversary never gets to inspect the output of a cryptographic primitive, presumably making her job harder.

3.1.2 t-Wise Universality

In order to achieve the verifier-selectable assurance goal for the tags produced by a MAC, we need to refine the granularity of claim made by its hash. We now define a framework to prove a hash function, when used in a Wegmac-Carter MAC, fulfills the selective-assurance verifiability goal. First, we need some notation. Given a function $h : A \to B^t$ and integer $1 \leq i \leq t$, we define $h_i(x)$ to be the i-th element of the t-vector $h(x)$.

We also define $h_{<i}(x)$ as the vector $(h_1(x), h_2(x), \ldots, h_{i-1}(x))$. If $i \leq 1$, then $h_{<i}(x)$ is defined to be the empty string.

Let us say that the functions of a hash family $\mathsf{H} = \{A \to B^t\}$ all return tuples of t elements from B. If we were to claim that H were ϵ-AU, for example, that would be a claim that for distinct m and m', and randomly chosen $h \in \mathsf{H}$, the probability that the *entire* outputs of $h(m)$ and $h(m')$ collide is no more than ϵ. But, what of the collision probability of the i-th components, $h_i(m)$ and $h_i(m')$, for some particular value i? It is conceivable that $h_1(m)$ and $h_1(m')$, for example, collide with a much different probability than say $h_2(m)$ and $h_2(m')$. It is also conceivable that components are far from independent of one another, that, say, $h_1(m)$ and $h_1(m')$ collide exactly when $h_2(m)$ and $h_2(m')$ do so.

One way to refine the granularity of our claim is to say that for distinct m and m', number i, and randomly chosen $h \in \mathsf{H}$, the probability that $h_i(m) = h_i(m')$ is low no matter what the values of $h_{<i}(m)$ and $h_{<i}(m')$. That is, the collision probability for the i-th components of $h(m)$ and $h(m')$ is low regardless of the values of the first $i-1$ components.

More formally, let $\mathsf{H} = \{A \to B^t\}$ be any family of functions. From H we define a family $\mathsf{H}' = \{A \to B\}$ which, given $1 \leq i \leq t$, $m \neq m' \in A$, and $c, c' \in B^{i-1}$ is populated as follows. For all functions $h \in \mathsf{H}$, if $h_{<i}(m) = c$ and $h_{<i}(m') = c'$, then $h_i(\cdot)$ is in H'.

Hash family H is **t-wise ϵ-almost-universal** If H' is ϵ-AU over m and m' for any choice of i, m, m', c, c' which defines a non-empty H' as described above. Likewise, if H' is ϵ-SU over m and m' for any choice of i, m, m', c, c' which defines a non-empty H', then H is **t-wise ϵ-strongly-universal**.

If the domain of a hash function is bit-strings $\{0,1\}^n$ then the hash family may be called t-wise universal so long as t divides n, in which case the output strings may be viewed as elements from $(\{0,1\}^{n/t})^t$.

3.2 Moving On

The MAC sketched in this chapter nearly fulfills completely the goals stated at the beginning. Still not fully resolved is the use of a cryptographic primitive during hashing. To hash arbitrary messages to a fixed-length, the NH hash function requires a cryptographically-produced key of unbounded length, and so seems to be not suitable as the final layer in our hash function. In Chapter 4 we develop a fast universal hash function based on polynomial evaluation. This polynomial hash uses a fixed-length key, produces a fixed-length output, is fast (although not as fast as NH), and will prove to be a suitable replacement for NH in producing fixed-length outputs.

Also not fully resolved is the meaning of "generally useful hash function". We have declared this to be one of our goals and given a small amount of intuition as to what is meant by this phrase, but the idea needs more development. Chapter 5 defines what is meant by "generally useful" and defines a new type of universal hashing applicable to this goal.

The ideas in these chapters, along with the general MAC design developed in this chapter are all brought together in Chapter 6 where we define and investigate UMAC (2000).

Chapter 4

Fast Polynomial Hashing

In Chapter 3 we saw that after compressing a message using NH with a fixed key-length, we are left with a shorter, but still unbounded string. This string needs to be hashed to a fixed-length to be useful in a Wegman-Carter MAC. In this chapter we develop a universal hash function suitable for this purpose. This chapter is extracted from a preliminary version of [26].

4.1 Introduction

Ever since its introduction by Carter and Wegman in 1979, universal hashing has been an important tool in computer science [14]. Recent attention has been paid to universal hashing as a method to authenticate messages, an idea also proposed by Carter and Wegman [39]. Its use in authentication has resulted in several very fast universal hash functions with very low collision probabilities. The implementations of these fastest universal hash functions tend to require either significant precomputed data, long keys or special-purpose hardware to achieve their impressive results.

Our contribution is a polynomial-based hash function we call RPHash. This hash

function is not as fast as hash functions which were designed for message authentication—speed is about 3.9–6.9 cpb—but that is still very fast, and RPHash has different and desirable characteristics. First, it hashes messages of essentially any length (and varying lengths are fine). The key is short (say 28 bytes), independent of the message length. The key requires no preprocessing: the natural representation of the key is the desirable one for achieving good efficiency. Quite pleasantly, the hash function is fastest, per byte, on *short* messages—it actually gets slower, per byte, as the message gets longer (the rates are constant until particular threshold lengths are crossed, like 2^{11} and 2^{33} bytes). This is the exact reverse of most optimized hash functions having short output lengths, which do better and better as the message gets longer. If used for authentication, working best for short messages is desirable insofar as *most* network traffic is short. Finally, implementation of our hash function family is simple and requires no special hardware like floating-point or multimedia execution units.

The hash function family RPHash was originally designed for use in a multi-layer hashing construction, UMAC [10]. In UMAC, a very fast first layer is applied to an incoming message to compress it to a small fraction of its original length. This compressed message is then passed to RPHash. When used as a second hash-layer in this manner, it can be expected that the *vast* majority of messages fed to RPHash will be short, since messages must be quite huge indeed before the second-layer compressed message gets long.

The hash function RPHash is a refinement to the classical suggestion of Carter and Wegman where one treats the message as specifying the coefficients of a polynomial, and one evaluates that polynomial at a point which is the key. Our refinements involve choosing the base field to be a prime just smaller than a power of 2^{32}, 2^{64}, or 2^{128}; limiting the key space to being in a "convenient" subset of all the field points; using a simple "translation"

Hash Function	Collision Bound	Code + Data Size	Speed (cpb)	Output (bits)
Poly32	$n2^{-28}$	$(124 + 8)$ bytes	3.9	32
Poly64	$n2^{-49}$	$(409 + 16)$ bytes	6.9	64
Division Hash	$n2^{-59}$	$\sim (? + 8)$ KB	7.5	64
UHASH-16	2^{-60}	$\sim (7 + 2)$ KB	1.0	64
UHASH-32	2^{-60}	$\sim (8 + 2)$ KB	2.0	64
hash127	$n2^{-127}$	$\sim (4 + 1.5)$ KB	4.3	127
MD5	?	1.7 KB	5.3	128
SHA-1	?	4.3 KB	13.1	160

Table 4.1: Poly32 and Poly64 with other hash families. *Various reported attributes are given for a variety of fast hash functions. Sizes marked with "∼" are conservative estimates. All timings are for the fastest Pentium/Pentium II timings reported.*

trick to take care of the problem that some messages will now give rise to coefficients not in the field (because our field is just smaller than a power of two); and using a "ramping-up" trick so that we don't have to pay in efficiency for short messages in order that the method can handle long ones. The result is a simple, flexible, fast-to-compute hash function.

4.1.1 Related Work

Carter and Wegman introduced the ideas of universal hashing and using polynomials in universal hashing in 1979 [14]. Since that time polynomials have been used for fast hashing in many other works. In his "Cryptographic CRC", Krawczyk views messages to be hashed as polynomials over GF(2) which are divided modulo a random irreducible polynomial [25]. The division can be done quickly in hardware using linear feedback shift registers. Shoup describes several variants of polynomial hash and provides implementation results [36]. The "generalized division hash", which bounds collisions between $64n$-bit messages as no more than $n2^{-59}$, views messages as polynomials over GF(2^{64}), uses an 8KB precomputed table, and has a throughput of 7.5 cpb on a Pentium [36]. Afanassiev, Gehrmann and Smeets discuss fast polynomial hashing modulo random 3-, 4- and 5-nomials [1]. Their methods

use small keys, but no implementation results are provided. Bernstein defines *hash127*, a polynomial hash defined over a large prime field $\mathbb{Z}_{p(127)}$ [9]. Over $32n$-bit messages it has a collision probability of no more than $n2^{-127}$. Bernstein's implementation uses floating-point operations and a 1.5KB precomputed table to achieve a throughput of 4.3 Pentium II cpb.

Other software efficient universal hash functions include Rogaway's bucket hash [35]; MMH of Halevi and Krawczyk [21]; and NH of Black, Halevi, Krawczyk, Krovetz and Rogaway [10]. NH is the current speed champion, providing collision probabilities of 2^{-60} with 4KB of precomputed data and achieving throughput of 1.0 Pentium II cpb (using Pentium MMX instructions) and 1.9 cpb (without MMX).

If one does not require the combinatoric certainties of universal hashing, one could employ cryptographic hashing to construct hash functions with short output lengths, short keys and little preprocessing. Bosselaers, Govaerts and Vandewalle report on optimized Pentium timing for several cryptographic hash functions: MD4 (3.8 cpb), MD5 (5.3) and SHA-1 (13.1) [13]. Simple methods can be used to convert these function into universal hash functions by, for example, keying their initial values [38]. For such a transformation to result in a universal hash function, certain unproved assumptions must be made about the cryptographic hash function.

4.1.2 Notation

The algorithms described in this chapter manipulate both bit-strings and integers. The i-th bit of string M is denoted $M[i]$ (bit-indices begin with 1). The substring consisting of the i-th through j-th bits of M is denoted $M[i \ldots j]$. The concatenation of string M_1 followed by string M_2 is denoted $M_1 \parallel M_2$. The length in bits of string M is $|M|$. The string of n zero-bits is denoted 0^n.

```
algorithm PolyCW[𝔽](k, m)
// Parameter: 𝔽 is a finite field.
// Input: k ∈ 𝔽 and m = (m_0, ..., m_n) where m_i ∈ 𝔽 for 0 ≤ i ≤ n.
// Output: y ∈ 𝔽.
Let n be the number of elements in m
y = 0
for i ← 0 to n do
    y ← ky + m_i                    // Arithmetic in 𝔽
return y
```

Figure 4.1: *The basic polynomial-hashing method of Carter and Wegman on which we build. The message* $\mathbf{m} = (m_0, \ldots, m_n)$ *is hashed to* $\sum_{i=0}^{n} m_i k^{n-i}$.

Given $b > 1$, the constant $p(b)$ is the largest prime smaller than 2^b. Given string M and $b > 0$, $\mathsf{padonezero}(M, b)$ returns the string $M \parallel 1 \parallel 0^n$, where n is the smallest number that makes the length of $M \parallel 1 \parallel 0^n$ divisible by b. Given a string M, the function $\mathsf{str2num}(M)$ returns the integer that results when M is interpreted as an unsigned binary number. Similarly, $\mathsf{num2str}(n, b)$ produces the unique b-bit string which is the binary representation for the non-negative number n.

The number of elements in a set S is denoted $|S|$.

4.1.3 Organization

In the next few sections we develop a fast polynomial hash function. In the appendix we generalize the function using arbitrary parameters. Theorems are given in both cases, but proven only for the concrete case. Proofs for the parameterized cases are straightforward adaptations of the ones for the concrete version, so they are omitted. Understanding the algorithms, theorems and proofs is easier in the concrete examples.

4.2 Carter-Wegman Polynomial Hashing: PolyCW

We begin by reviewing the "standard" approach for polynomial hashing. Let $\mathbb{F}$ be a finite field, let $k \in \mathbb{F}$ be a point in that field (the "key") and let $\mathbf{m} = (m_0, \ldots, m_n)$ be a vector of points in $\mathbb{F}$ that we want to hash (the "message"). We can hash message $\mathbf{m}$ to a point y in $\mathbb{F}$ (the "hash value") by computing $y = m_0 k^n + \cdots + m_{n-1} k^1 + m_n k^0$, where all arithmetic is done in $\mathbb{F}$. We denote this family of hash functions as $\mathsf{PolyCW}[\mathbb{F}]$. The computation of this hash function (with $n+1$ multiplications in the field and $n+1$ additions in the field) is described in Figure 4.1.[1]

$\mathsf{PolyCW}[\mathbb{F}]$ is one of the most well-known universal hash-function families. It was described by Carter and Wegman in the paper that introduced that notion [14]. The main property it has is as follows. If $\mathbf{m} = (m_n, \ldots, m_0)$ and $\mathbf{m}' = (m'_n, \ldots, m'_0)$ are distinct vectors with the same number of components then $\Pr[H \leftarrow \mathsf{PolyCW}[\mathbb{F}]\,;\, k \xleftarrow{R} \mathbb{F} : H_k(\mathbf{m}) = H_k(\mathbf{m}')] \leq \frac{n}{|\mathbb{F}|}$. This result is due to the Fundamental Theorem of Algebra which states that a nonzero polynomial of degree at most n can have at most n roots. Rewriting the above probability as $\Pr[k \xleftarrow{R} \mathbb{F} : \sum_{i=0}^{n} m_i k^{n-i} = \sum_{i=0}^{n} m_i k^{n-i}] = \Pr[k \xleftarrow{R} \mathbb{F} : \sum_{i=0}^{n} (m_i - m'_i) k^{n-i} = 0]$, and applying the Fundamental Theorem, we see that there can be at most n values for k which cause $\sum_{i=0}^{n} (m_i - m'_i) k^{n-i}$ to evaluate to zero.

4.3 Making $\mathsf{PolyCW}[\mathbb{F}]$ Fast

Care must be taken in the implementation of $\mathsf{PolyCW}[\mathbb{F}]$. A naive implementation is unlikely to perform well. Many choices of $\mathbb{F}$ and the set from which the hash-key is chosen can result in sub-optimal performance. We investigate the effect that shrewd choices for $\mathbb{F}$

[1] All algorithms depicted in this paper which evaluate polynomials do so by using Horner's Rule which says that polynomial $m_0 k^n + \cdots + m_{n-1} k^1 + m_n k^0$ can be rewritten as $m_n + k(m_{n-1} + k(m_{n-2} + k(m_{n-3} + \cdots)))$. This allows for simple iteration with one multiplication and one addition for each element of the message.

```
algorithm PolyP32(k, m)
// Input: k ∈ K_32 and m = (m_1, ..., m_n) where m_i ∈ Z_{p(32)} for 1 ≤ i ≤ n.
// Output: y ∈ Z_{p(32)}.
Let n be the number of elements in m
p ← 2^32 − 5                                  // The largest prime smaller than 2^32
y = 0
for i ← 1 to n do
    y ← ky + m_i mod p
return y
```

Figure 4.2: *The* PolyP32 *algorithm. A variant of the* PolyCW *hash, accelerated by choosing a field* $\mathbb{Z}_{p(32)}$ *in which calculations can be performed quickly and choosing a key-set* K_{32} *which reduces arithmetic overflow on 32-bit processors. The* **for** *loop could be rewritten as the polynomial:* $y = \sum_{i=1}^{n}(m_i k^{n-i}) \bmod p$.

and the key-set have on performance.

Field Selection. To make an efficient and practical hash function out of PolyCW $[\mathbb{F}]$ we should carefully choose the finite field $\mathbb{F}$. Fields like GF$[2^{64}]$ make natural candidates, because we are ultimately interested in hashing bit strings which are easily partitioned into 64-bit substrings. But arithmetic in GF$[2^w]$ turns out to be less convenient for contemporary CPUs than a well-chosen alternative. In this chapter we will do better by using prime fields in which the prime is just smaller than a power of two.

Consider first the use of the prime $p(32) = 2^{32} - 5$, which is the largest prime less than 2^{32}. To implement PolyCW $[\mathbb{Z}_{p(32)}]$ efficiently, we need a good way to calculate $y \leftarrow ky + m \bmod p(32)$, where $y, k, m \in \mathbb{Z}_{p(32)}$. There are several options. One's first instinct is to use the native "mod" operand of a high-level programming language (like "%" in C), or to use a corresponding operator in the hardware architecture. But these choices are usually slow. For example, PolyCW $[\mathbb{Z}_{p(32)}]$, implemented in assembly using the native mod operator runs in **12.4** cpb (cycles/byte) on a Pentium II.

A faster method exploits the fact that since $p(32) = 2^{32} - 5$, the numbers 2^{32} and 5 are

equivalent in the field $\mathbb{Z}_{p(32)}$, so $2^{33} = 10$, $2^{34} = 20$ and, more generally, $a2^{32} = 5a$ in $\mathbb{Z}_{p(32)}$. So, to calculate $ky \bmod p(32)$, first compute the 64-bit product $z = ky$ and separate z into a 32-bit high-word a and a 32-bit low-word b so that $z = a2^{32} + b$. We can then use the observation just made and rewrite $z \bmod p(32)$ as $5a + b$. This means that the calculation $y = ky + m \bmod p(32)$ can be done by computing $y = 5a + b + m \bmod p(32)$, which can be implemented more cheaply than the original approach because it does not require division to perform the modular reduction.

Key-Set Selection. When implemented on a 32-bit architecture, the values a, b and m just discussed fit conveniently into 32-bit registers, making these quantities easy to manipulate. On most such architectures, the calculation of y is going to be fastest if it is done with minimal register overflow. To calculate $y = 5a + b + m \bmod p(32)$ using only 32-bit registers, we need one multiplication, two additions and then some additional instructions to handle register overflow. Each operation that can result in register overflow requires several instructions, including a conditional move or branch, to check and deal with the potential overflow event. To accelerate the calculation of y we reduce the number of potential overflows. Little can be done about overflow from the additions because both b and m can be nearly 2^{32}, but overflow from the multiplication can be eliminated. If a is larger than $\lfloor 2^{32}/5 \rfloor \approx 2^{29.7}$, then the term $5a$ will overflow a 32-bit register. We can restrict a to safe values by restricting k to values less than 2^{29}. This eliminates the possibility of $5a$ overflowing and allows for a faster implementation. The expense for this optimization is a higher collision probability because the key is chosen from a set of 2^{29} elements instead of a set of 2^{32} elements.

Divisionless modular reduction. Another optimization over a naive implementation is the elimination of division to calculate modular reductions. This technique is not new.

```
; Calculate y = y * k + m mod p(32)
; Assume y is in register eax before and after code segment.
mul    k                        ; edx:eax = k * y
lea    edx, [edx*4+edx]         ; edx = 5 * edx
add    eax, edx                 ; eax = edx + eax
lea    edx, [eax+5]             ; edx = eax + 5
cmovc  eax, edx                 ; if (carried) then eax = edx
add    eax, m                   ; eax = eax + m
lea    edx, [eax+5]             ; edx = eax + 5
cmovc  eax, edx                 ; if (carried) then eax = edx
```

Figure 4.3: *The $y = ky + m$ calculation of the* PolyP32 *algorithm written in Pentium II assembly. The flag "*`carried`*" is true only if the previous* `add` *instruction causes a register overflow. The conditional-move instruction (*`cmovc`*) is used to avoid any branches during execution of the routine, and the load-effective-address instruction (*`lea`*) is used for addition and multiplication of small constants. The result of the routine could possibly be in the range $p \leq y < 2^{32}$, which is outside of the field $\mathbb{Z}_{p(32)}$, but this is easily fixed with a single subtraction after hashing the final word of the entire message.*

In calculating $y = 5a + b + m \bmod p(32)$, each of the $5a$, b and m terms are less than 2^{32}. As we sum them using computer arithmetic with 32-bit registers, we can easily detect 32-bit overflows. Each such overflow indicates a 2^{32} term which is not accounted for in the resulting register. But, because $2^{32} \equiv 5$, these overflows are easily accounted for by adding 5 for each overflow to the resulting register. Done carefully, this observation results in a number y, derived without any division, which is representable in 32-bits (ie. $0 \leq y < 2^{32}$). See Figure 4.3 for implementation details. Do we then need to reduce y to a number in $\mathbb{Z}_{p(32)}$? No. All of the discussion so far requires only that y be representable in a 32-bit register. Instead of reducing y to $\mathbb{Z}_{p(32)}$ after every intermediate calculation, we defer all such reductions until a final single reduction is performed as a final step.

Speed. Taken together, the selection of a convenient prime field and the restriction of the key-set to keys which eliminate some register overflows allow great speed-up over a naive implementation of PolyCW. Figure 4.2 shows a version of polynomial hash based upon

PolyCW which hashes over the field $\mathbb{Z}_{p(32)}$ and restricts key selection to the set $K_{32} = \{a : 0 \leq a < 2^{29}\}$. Our implementation of the core $y = ky + m \bmod p(32)$ calculation uses 8 lines of Pentium II assembly (Figure 4.3) and achieves a peak throughput of **3.69** cpb.

We state here a proposition establishing the collision bound of the PolyP32 hash function.

Proposition 4.3.1 *For any positive integer n and any distinct messages $\mathbf{m} = (m_0, \ldots, m_n)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$, consisting of elements from $\mathbb{Z}_{p(32)}$,* $\Pr[k \leftarrow K_{32} : \mathsf{PolyP32}(k, \mathbf{m}) = \mathsf{PolyP32}(k, \mathbf{m}')] \leq n/|K_{32}| = n2^{-29}$.

64-Bit Hashing and Key Restriction. We also implemented an analogous PolyP64 hash function whose core calculation is $y = ky + m \bmod p(64)$ where $p(64) = 2^{64} - 59$ and k, y and m are all elements of $\mathbb{Z}_{p(64)}$. As in the 32-bit case, it is cheapest to calculate the result without using division. If we let $2^{32}k_h + k_l$ represent k and $2^{32}y_h + y_l$ represent y, then ky can be calculated as $ky = 2^{64}k_h y_h + 2^{32}(k_h y_l + k_l y_h) + k_l y_l$. Again, restricting the set of values that k can take on allows for faster implementations by eliminating some 32-bit register overflows. We define key-set $K_{64} = \{a2^{32} + b : 0 \leq a, b < 2^{25}\}$. This restriction allows an implementation of PolyP64 which has a collision probability of $(n/2^{50})$, uses 40 lines of assembly and has a peak throughput of **6.86** cpb.

4.4 Expanding the Domain to Arbitrary Strings

The hash function PolyP32 is not generally useful. It only works on same-length messages, and those messages must be made of elements from the field $\mathbb{Z}_{p(32)}$. We now remove these limitations and develop Poly32. The result, depicted in Figure 4.4, hashes most messages at a rate of **3.86** cpb.

```
algorithm Poly32(k, M)
// Input: k ∈ K_32 and M ∈ ({0,1}^32)^+.
// Output: y ∈ Z_{p(32)}.
p ← 2^32 − 5                          // The largest prime smaller than 2^32
offset ← 5                            // Constant for translating out-of-range words
marker ← 2^32 − 6                     // Constant for indicating out-of-range words
n ← |M|/32
M_1 ‖ ... ‖ M_n ← M,                  // Break M into 32-bit chunks
     where |M_1| = ··· = |M_n| = 32
y ← 1                                 // Set highest coefficient to 1
for i ← 1 to n do
     m ← str2num(M_i)
     if (m ≥ p − 1) then              // If word is not in range, then
          y ← ky + marker mod p       // Marker indicates out-of-range
          y ← ky + (m − offset) mod p // Offset m back into range
     else
          y ← ky + m mod p            // Otherwise hash in-range word
return y
```

Figure 4.4: *The* Poly32 *algorithm. The* PolyP32 *hash extended to hash strings instead of vectors of field elements and to allow good collision probabilities over two strings which differ in length.*

Allowing Variations in Length. It is a trivial exercise to produce two different length messages which collide when hashed with PolyP32 using *any* key: Under PolyP32, the hash of a message $\mathbf{m} = (m_0, \ldots, m_n)$ using key k is simply $h(k, \mathbf{m}) = m_0 k^n + \cdots + m_n k^0 \bmod p(32)$, so prepending 0 to the vector $\mathbf{m}$ results in a message $\mathbf{m}' = (0, m_0, \ldots, m_n)$ which is hashed as $h(k, \mathbf{m}') = 0k^{n+1} + m_0 k^n + \cdots + m_n k^0 \bmod p(32)$ and is equal to $h(k, \mathbf{m})$ because the additional zero-term has no effect on the hash value. For the Fundamental Theorem of Algebra to guarantee a low number of roots (and hence a low collision probability), it is essential that the difference between $\mathbf{m}$ and $\mathbf{m}'$ be non-zero. This means that if the two vectors differ only in length, then at least one of the initial elements of the longer vector must be non-zero. To guarantee this we employ a standard trick and implicitly prepend a "1" to the vectors being hashed. Thus, the hash of $\mathbf{m} = (m_0, \ldots, m_n)$ implicitly becomes the hash of $\mathbf{m} = (1, m_0, \ldots, m_n)$, and the hash of $\mathbf{m}' = (0, m_0, \ldots, m_n)$ implicitly becomes the

hash of $\mathbf{m}' = (1, 0, m_0, \ldots, m_n)$. The difference between these two vectors is non-zero. The following theorem assures that augmenting PolyP32 in this way results in a hash with nearly the same collision probability as PolyP32, but works over messages of different lengths.

Proposition 4.4.1 *Let $\ell < n$ be positive integers. Let $\mathbf{m} = (m_0, \ldots, m_l)$ and $\mathbf{m}' = (m'_0, \ldots, m'_n)$ be any two vectors of elements from the field $\mathbb{F}$. Then there are at most $n+1$ values for $k \in \mathbb{F}$ such that $k^{\ell+1} + \sum_{i=0}^{\ell} m_i k^{\ell-i} = k^{n+1} + \sum_{i=0}^{n} m'_i k^{n-i}$.*

Proof. Beginning with $k^{\ell+1} + \sum_{i=0}^{\ell} m_i k^{\ell-i} = k^{n+1} + \sum_{i=0}^{n} m'_i k^{n-i}$, and moving all of its terms to the right side of the equation we get $0 = k^{n+1} - k^{\ell+1} + \sum_{i=0}^{n} m'_i k^{n-i} - \sum_{i=0}^{\ell} m_i k^{\ell-i}$. But, the right side of this equations is now a non-zero polynomial, is of degree $n+1$, and therefore has at most $n+1$ roots. $\diamondsuit$

Alternative Method. Another method of augmenting PolyP32 to allow variable length messages is to use a second key $k' \in \mathbb{Z}_{p(32)}$ and add it to each element of the message being hashed. Thus, $h(k, k', \mathbf{m})$ would be computed as $\sum_{i=0}^{\ell}(m_i + k')k^{\ell-i}$. This method requires an extra addition per message word being hashed and so the first method seems favorable.

Allowing Bit-Strings. To make the function PolyP32 of Figure 4.2 more useful, it must be adapted to allow bit-strings rather than only vectors from $\mathbb{Z}_{p(32)}$. The field $\mathbb{Z}_{p(32)}$ was chosen because it contains nearly all the numbers representable as 32-bit strings. Thus, when we desire to hash a bit-string, we may partition the string into 32-bit words and treat the partition as a vector of 32-bit numbers. PolyP32 can then hash the vast majority of the vector's elements without any modification. But, some of the 32-bit numbers may be in the range $p(32) \ldots 2^{32} - 1$, outside $\mathbb{Z}_{p(32)}$. What should be done with them?

One approach is to transform a vector of 32-bit numbers, which may have some elements outside of $\mathbb{Z}_{p(32)}$, into a vector which does not. The transformation must map distinct

vectors into distinct vectors.

We solve this problem by examining a vector of 32-bit numbers and replacing each vector element m_i that is greater than $p(32) - 2$ with *two* numbers, $p(32) - 1$ and $m_i - 5$. Note that both of these numbers are in $\mathbb{Z}_{p(32)}$. Each such replacement lengthens the resulting vector by one element. Thus, the vector $\mathbf{m} = (4, 2^{32}-3, 10)$, whose second element is greater than $p(32)-2$, would be transformed into the vector $\mathbf{m'} = (4, 2^{32}-6, 2^{32}-8, 10)$. We call this transformation $\mathsf{DoubleTransform} : (\mathbb{Z}_{2^{32}})^+ \rightarrow (\mathbb{Z}_{p(32)})^+$. The following proposition assures that $\mathsf{DoubleTransform}$ is correct.

Proposition 4.4.2 *For positive integers ℓ and n, and distinct messages $\mathbf{m} = (m_0, \ldots, m_l)$ and $\mathbf{m'} = (m'_0, \ldots, m'_n)$ consisting of elements from $\mathbb{Z}_{2^{32}}$, the vectors $\mathsf{DoubleTransform}(\mathbf{m})$ and $\mathsf{DoubleTransform}(\mathbf{m'})$ consist of elements from $\mathbb{Z}_{p(32)}$ and are distinct.*

Proof. Let ℓ and n be positive integers and let $\mathbf{m} = (m_0, \ldots, m_l)$ and $\mathbf{m'} = (m'_0, \ldots, m'_n)$ be distinct vectors consisting of elements from $\mathbb{Z}_{2^{32}}$. Let $\mathbf{t} = \mathsf{DomainTransform}(\mathbf{m})$ and $\mathbf{t'} = \mathsf{DomainTransform}(\mathbf{m'})$. Let i be the smallest number such that $m_i \neq m'_i$. If such an i does not exist then one of $\mathbf{m}$ or $\mathbf{m'}$ must be a proper prefix of the other. In this case, any lengthening of the shorter vector by $\mathsf{DoubleTransform}$ must be mirrored by the transformation of the longer vector ensuring that the two remain different lengths after transformation.

If m_i and m'_i are both less than $p(32) - 1$, then after transformation $t_i = m_i$ and $t'_i = m'_i$, ensuring that $\mathbf{t} \neq \mathbf{t'}$. If only one of m_i and m'_i is less than $p(32) - 1$, say m_i, then after transformation $t_i = m_i$ and $t'_i = p(32) - 1$, again ensuring that $\mathbf{t} \neq \mathbf{t'}$. Finally, if both m_i and m'_i are greater than $p(32) - 2$, then after transformation $t_{i+1} = m_i - 5$ and $t'_{i+1} = m_i - 5'$ again ensuring that $\mathbf{t} \neq \mathbf{t'}$. $\diamondsuit$

ALTERNATIVE METHOD. There are many ways to patch PolyP32 to allow out-of-range elements. One probabilistic alternative is to offset every out-of-range number by a randomly chosen $k' \in \{5, \ldots, 2^{32}-5\}$. All out-of-range numbers are in $\{2^{32}-5, \ldots, 2^{32}-1\}$, so k', when subtracted from an out-of-range number, will always yield a number in $\mathbb{Z}_{p(32)}$. This method has the advantage of not increasing message length upon transformation, but requires an extra key element, and in practice does not speed hashing with respect to the method of Proposition 4.4.2.

Together, Propositions 4.4.1 and 4.4.2 prove the following corollary.

Corollary 4.4.3 *For any positive integers $\ell \leq n$ and distinct messages $M \in \{0,1\}^{32l}$ and $M' \in \{0,1\}^{32n}$, $\Pr_{k \in K_{32}}[\mathsf{Poly32}(k, M) = \mathsf{Poly32}(k, M')] \leq 2n/|K_{32}| = n2^{-28}$.*

The discussion so far has focussed on Poly32, a hash function defined on 32-bit words. An analogous 64-bit variant, Poly64, yields the following bound.

Corollary 4.4.4 *For any positive integers $\ell \leq n$ and distinct messages $M \in \{0,1\}^{64l}$ and $M' \in \{0,1\}^{64n}$, $\Pr_{k \in K_{64}}[\mathsf{Poly64}(k, M) = \mathsf{Poly64}(k, M')] \leq 2n/|K_{64}| = n2^{-49}$.*

Two things are worth noting. First, the factor of two introduced in the $2n/|K_{32}|$ term is due to the potential doubling of message length by the DoubleTransform function. And, second, standard message padding techniques are not addressed in this chapter. It is assumed that messages being hashed have been properly padded to a 32-bit boundary.

It should also be noted that the probability that Poly32 or Poly64 hash any message to a particular result is also low. Consider a message made of n 32-bit words $\mathbf{x} = (x_1, \ldots, x_n)$ and a constant c. If $c \geq p(32)$ then $\mathsf{Poly32}(k, \mathbf{x})$ cannot hash to c, and if $c < p(32)$, then $\mathsf{Poly32}(k, \mathbf{x})$ will hash to c only if $\mathsf{Poly32}(k, \mathbf{x}')$ hashes to zero where $\mathbf{x}' = (x_1, \ldots, x_n - c)$.

```
algorithm RPHash32_64(k, M)
// Input: k = (k1, k2) with k1 ∈ K32 and k2 ∈ K64, and M ∈ {0,1}*.
// Output: Y ∈ {0,1}^64.
if (|M| ≤ 2^14) then                          // 2^9 32-bit words
    M ← padonezero(M, 32)
    y ← Poly32(k1, M)                         // Hash in Z_p(32)
else if (|M| ≤ 2^36) then                     // 2^30 64-bit words
    M1 ← M[1 ... 2^14]
    M2 ← M[2^14 + 1 ... |M|]
    M2 ← padonezero(M2, 64)
    y ← Poly32(k1, M1)                        // Hash in Z_p(32)
    y ← Poly64(k2, num2str(y, 64) || M2)      // Hash in Z_p(64), prepending y
else
    return Error                              // Message too long
Y ← num2str(y, 64)                            // Convert to string
return Y
```

Figure 4.5: *The* RPHash32_64 *algorithm. Combining the* PolyP32 *and* PolyP64 *hashes into a hash function which is fast on short messages but also performs well on long ones.* RPHash32_64 *also extends the domain to messages which are not a multiple of the constituent hashes word-lengths.*

After the DoubleTransform transformation of $\mathbf{x}'$, the Fundamental Theorem tells us that there are no more than $2n$ keys which allow this to happen.

Claim 4.4.5 *Let n and c be numbers, and message $M \in \{0,1\}^{32\ell}$,* $\Pr_{k \in K_{32}}[\mathsf{Poly32}(k, M) = c] \leq 2n/|K_{32}|$.

4.5 RPHash: Overcoming Polynomial Hash Length Limitations

Taking a closer look at the bounds established for each of the polynomial hash functions, one can see that the collision bounds degrade linearly along with the length of the messages being hashed. This is a byproduct of the use of polynomials in hashing: As messages get longer, so do the degrees of the polynomials get higher, resulting in more potential collision-

causing roots. This introduces a trade-off in application design. If one wants to guarantee some maximum collision probability ϵ and the hash-key is chosen from a set of k elements, then the length of messages to be hashed must be limited to around $k\epsilon$ words. The larger the key-set size k used in the hashing polynomial, the more words can be hashed before reaching the allowable collision probability q. But, to make the key-set size significantly larger requires the polynomial to be computed over a larger prime-field, and in general, as the prime p is increased, so is the time needed to evaluate the polynomials in $\mathbb{Z}_p$. As one can see by examining the timing results for Poly32 and Poly64, the move from a prime close to 2^{32} to one close to 2^{64} increases the number of cycles-per-byte required to hash a message by nearly 50%.[2]

Can we have the best of both worlds: A hash function which is as fast as Poly32 but can hash messages as long as Poly64, without having intollerably high collision probability? This is the goal which motivates this section. We approach the problem with the belief that most strings being hashed are short, but that a generalized hash function should be able to handle well long messages too.

Our idea is to hash short messages (up to some fixed number of bits ℓ) directly with Poly32, but hash messages longer than ℓ bits with a hybrid scheme. Let us say that message M is longer than ℓ bits. To hash M we first partition it into its ℓ bit prefix M_1 and the remainder M_2, so that $M_1 \parallel M_2 = M$. The hash of M under our hybrid scheme is then $\mathsf{Poly64}(k_2, \mathsf{Poly32}(k_1, M_1) \parallel M_2)$. In this manner, the first ℓ bits of M is hashed with a fast hash function (which cannot safely hash long messages), and if there is any of the string left after hashing its prefix, the remainder is hashed with a slower hash function (which can safely hash longer messages). The parameter ℓ depends on the maximum desirable collision

[2]Some of this difference is an artifact of the fact that the Pentium II natively supports multiplication of 32-bit operands to a 64-bit result, but not the multiplication of 64-bit operands to a 128-bit result. Most processors will display this type of threshold behavior when operands exceed well-supported lengths.

bound and how long a message can be before the fast hash function approaches this bound.

As an example, let us say that we want to hash messages and have a collision bound of no more than 2^{-16}. If we were to hash solely with Poly32, then we could hash no messages longer than around 2^{17} bits. Alternatively, we could hash with only Poly64 and would then be able to hash strings as long as 2^{39} bits before allowing 2^{-16} collision probability, but at a much slower rate than Poly32. Under our scheme, if a message M is shorter than 2^{17} bits, then the hash result is simply $\mathsf{Poly32}(M)$; whereas if M is longer than 2^{17} bits, then the hash is calculated as $\mathsf{Poly64}(\mathsf{Poly32}(M_1) \parallel M_2)$ where M_1 is the 2^{17}-bit prefix of M. Such a construction is fast on short messages, but handles well long messages too. If messages were anticipated to be longer than 2^{39}, then a function Poly128, employing a 128-bit prime modulus, could be defined analogously and be employed as a third-stage polynomial. This ramping-up of the prime modulus used in the polynomial evaluations gives the construction its name: *Ramped polynomial hashing.*

One might expect the collision bound of such a hybrid approach to be approximately the *sum* of the collision bounds of each of its constituent functions, but as the following theorem shows, the overall collision bound is instead only the *maximum* of the functions.

The following theorem and proof address RPHash32_64, the ramped polynomial hash of Figure 4.5. This concrete hash function hashes up to 2^{14} bits (equivalent to 2^9 32-bit words) using the fast Poly32 function, and allows a total message length of up to 2^{36} bits (or 2^{30} 64-bit words). In the following theorem and proof, for increased generality, we use parameters ℓ and m instead of the numbers of words 2^9 and 2^{30}.

Theorem 4.5.1 *Let $\ell = 2^9$ and $m = 2^{30}$. Let $M \neq M'$ be messages (possibly of unequal length) no longer than $64m$ bits. Then* $\Pr[k_1 \xleftarrow{R} K_{32}; k_2 \xleftarrow{R} K_{64} : \mathsf{RPHash32\_64}(k_1, k_2, M) = \mathsf{RPHash32\_64}(k_1, k_2, M')] \leq \max(\ell 2^{-28}, m2^{-49}) + 2^{-50}$.

Proof. Let M and M' be messages, and imagine partitioning them into $M = M_1 \| M_2$ and $M' = M'_1 \| M'_2$ so that M_1 and M'_1 are the first 32ℓ bits of M and M'. If M is shorter than 32ℓ bits, then $M_1 = M$ and M_2 is empty. Likewise, if M' is shorter than 32ℓ bits, then $M'_1 = M'$ and M'_2 is empty. We ignore all padding issues in this discussion, assuming that standard padding techniques are used to bring M and M' to appropriate lengths. Let $k_1 \xleftarrow{R} K_{32}$ and $k_2 \xleftarrow{R} K_{64}$ be randomly chosen keys. We define the following values here for convenience, but all probabilities in this proof are assumed to be taken over these random choices of k_1 and k_2.

$$h_1 = \mathsf{num2str}(\mathsf{Poly32}(k_1, M_1), 64) \qquad \text{and} \qquad h'_1 = \mathsf{num2str}(\mathsf{Poly32}(k_1, M'_1), 64)$$

$$h_2 = \mathsf{num2str}(\mathsf{Poly64}(k_2, h_1 \| M_2), 64) \qquad \text{and} \qquad h'_2 = \mathsf{num2str}(\mathsf{Poly64}(k_1, M'_2), 64)$$

Depending on the lengths of M and M', the result of hashing M will be h_1 or h_2 and the result of hashing M' will be h'_1 or h'_2. We examine several cases for the relative lengths of M and M'.

Case 1: Different lengths, same ramp. Here we examine the case where the messages M and M' are different lengths, but are both either longer than 32ℓ bits or both no longer. If both are no longer than 32ℓ bits then a collision occurs if $h_1 = h'_1$. But, $M_1 = M$ and $M'_1 = M'$ differ in length which means (by Proposition 4.4.3) that $\Pr[h_1 = h'_1] \leq \ell 2^{-28}$. If both M and M' are longer than 32ℓ bits then a collision occurs if $h_2 = h'_2$. But, $h_1 \| M_2$ and $h'_1 \| M'_2$ also differ in length which means (by Proposition 4.4.4) that $\Pr[h_2 = h'_2] \leq m2^{-49}$

Case 2: Different lengths, different ramp. If M is longer than 32ℓ bits and M' is not, then a collision occurs only if $h'_1 = h_2$. Expanding the h_2 term, we see that a collision only occurs if $h'_1 = \mathsf{num2str}(\mathsf{Poly64}(k_2, h_1 \| M_2), 64)$. If we fix k_1 to an arbitrary value, then h_1 and h'_1 become fixed as well, and the probability of collision then depends only on

the selection of k_2. The string $h_1 \parallel M_2$ is partitioned by the Poly64 algorithm into 64-bit strings and then transformed by DoubleTransform into some sequence $x_0, x_1, \ldots, x_{n \leq 2m}$ of elements from $\mathbb{Z}_{p(64)}$. This sequence is then used in the summation $\sum_{i=0}^{n} x_i k_2^{n-i} \bmod p(64)$ to calculate the final hash result. A collision occurs if the result of this summation is h'_1, or alternatively when $\sum_{i=0}^{n} x_i k_2^{n-i} - h'_1 \bmod p(64) = 0$. The Fundamental Theorem of Algebra applies to this last polynomial, meaning there are no more than $n \leq 2m$ values for k_2 which satisfy it. Thus, $\Pr[h'_1 = h_2] \leq 2m2^{-29} = m2^{-28}$.

Case 3: Equal length messages, last ramp different. If M and M' are equal length, longer than 32ℓ bits and $M_2 \neq M'_2$, then (by Proposition 4.4.4) $\Pr[h_2 = h'_2] \leq m2^{-49}$ because $h_1 \parallel M_2$ and $h'_1 \parallel M'_2$ are distinct. Similarly, if M and M' are the same length, no longer than 32ℓ bits and $M_1 \neq M'_1$, then (by Proposition 4.4.3) $\Pr[h_1 = h'_1] \leq m2^{-28}$ because M_1 and M'_1 are distinct.

Case 4: Equal length messages, last ramp same. If M and M' are equal length and longer than 32ℓ bits, and $M_1 \neq M'_1$ but $M_2 = M'_2$, then there are two opportunities for a collision to take place. First, if $\mathsf{Poly32}(k_1, M_1) = \mathsf{Poly32}(k_1, M'_1)$, then the strings $h_1 \parallel M_2$ and $h'_1 \parallel M'_2$ are equal, guaranteeing that h_2 and h'_2 collide. The probability of this event is no more than $\ell 2^{-28}$. Second, if $h_1 \neq h'_1$, then a collision can still occur if $\mathsf{Poly32}(k_2, h_1 \parallel M_1) = \mathsf{Poly32}(k_2, h'_2 \parallel M'_2)$. One might think that this is an event with up to $m2^{-49}$ probability, but it is not. Because $M_2 = M'_2$, the strings $h_1 \parallel M_1$ and $h'_1 \parallel M'_1$ only differ in their first 64-bit word. The collision event when hashing such strings takes the form $(h_1 - h'_1)k_2^n = 0$, which can only be satisfied if $k_2 = 0$, a 2^{-50} probability event. Thus, the total probability of collision in this case is bounded by $\ell 2^{-28} + 2^{-50}$. $\diamondsuit$

4.5.1 Security Notes

If a lower collision probability is desired, one can hash messages multiple times, using a different key for each message hash. A hash function which has an ϵ collision bound when hashing once with a random key, has an ϵ^2 collision bound when hashing twice with two random keys, and an ϵ^3 collision bound when hashed with three keys, etc.

Also, all of the theorems in this work have been stated in terms of collisions (ie. the difference between the result of evaluating the hash of two distinct messages is zero). It is a simple matter to tweek the algorithms and proofs to show that the probability that the difference between the hash of two distinct messages being a particular constant is bounded by the same ϵ. This version of universal hashing ("delta"-universal) is required in some message authentication schemes.

4.6 Fully Parameterized: Poly, RPHash

This chapter developed a two-stage ramped polynomial hash function RPHash32_64 using polynomials over 32- and 64-bit prime fields. The concrete choices made for RPHash32_64 were designed especially for the UMAC message authentication code. In UMAC, we needed a universal hash function which would guarantee a collision bound of at most 2^{-16} and would typically be applied to messages no longer than a few dozen bytes. But, the hash function must also be able to process huge inputs, too, and still guarantee a bound of at most 2^{-16}. These requirements led to the development of ramped polynomial hashing in general, and in the choice of the 32- and 64-bit prime fields, and associated crossover points, used in the body of this chapter.

Other collision bounds and expected message lengths not addressed by RPHash32_64

$r \geq 1$	:	Length of $\mathbf{v}$, $\mathbf{l}$, $\mathbf{K}$ vectors used in RPHash.
$\mathbf{v} = (v_1, \ldots, v_r)$	:	Word-lengths used in RPHash, with $1 < v_1 < \cdots < v_r$.
$\mathbf{l} = (\ell_1, \ldots, \ell_r)$	:	Message lengths used in RPHash, with $\ell_i \geq 1$ for $1 \leq i \leq r$.
$\mathbf{d} = (d_1, \ldots, d_r)$	:	Domain bounds used in RPHash, with $2^{v_i - 1} \leq d_i \leq p(v_i)$.
$\mathbf{K} = (K_1, \ldots, K_r)$	:	Key-sets used in RPHash, with $K_i \subseteq \mathbb{Z}_{p(v_i)}$ for $1 \leq i \leq r$.

Figure 4.6: *Parameters used in the fully parameterized* RPHash *algorithm. Fixing these parameters fixes the algorithm definition specified in Figure 4.8.*

are likely, and so we present in this appendix fully parameterized versions of the hashes called Poly and RPHash. For each of the algorithms we state their collision bounds as a theorem, but give no proofs. The proofs are straightforward extensions of those given in the body of the chapter.

Proposition 4.6.1 *Let v be any positive integer, let $K \subseteq \mathbb{Z}_{p(v)}$ be any subset of points in the field $\mathbb{Z}_{p(v)}$, and let $2^{v-1} \leq d \leq p(v)$. For any positive integers $\ell \leq n$ and distinct messages $M \in \{0,1\}^{lv}$ and $M' \in \{0,1\}^{nv}$,* $\Pr[k \xleftarrow{\mathrm{R}} K : \mathsf{Poly}[K, v, d](k, M) = \mathsf{Poly}[K, v, d](k, M')] \leq 2n/|K|$.

Proposition 4.6.2 *Let all of the parameters from Figure 4.6 be fixed. For any distinct messages M and M', each shorter than $\sum_{1 \leq i \leq r} \ell_i v_i$ bits,* $\Pr[\mathbf{k} \xleftarrow{\mathrm{R}} \mathbf{K} : \mathsf{RPHash}(\mathbf{k}, M) = \mathsf{RPHash}(\mathbf{k}, M')] \leq$

$$\max_{1 \leq i \leq r} \left\{ \frac{2\ell_i}{|K_i|} \right\} + \sum_{i=2}^{r} \frac{1}{|K_i|} \; .$$

algorithm $\mathsf{Poly}[K, v, d](k, M)$
// *Parameters: "Key set"* $K \subseteq \mathbb{Z}_{p(v)}$*, "word length"* $v \geq 1$ *and "domain-bound"* d*.*
// *Input:* $k \in K$ *and* $M \in (\{0,1\}^v)^+$*.*
// *Output:* $y \in \mathbb{Z}_{p(v)}$*.*
$offset \leftarrow 2^v - p(v)$ // *Constant for translating out-of-range words*
$marker \leftarrow p(v) - 1$ // *Constant for indicating out-of-range words*
$n \leftarrow |M|/v$
$M_1 \parallel \ldots \parallel M_n \leftarrow M$, // *Break* M *into word size chunks*
where $|M_1| = \cdots = |M_n| = v$
$y \leftarrow 1$ // *Set highest coefficient to 1*
for $i \leftarrow 1$ **to** n **do**
$m \leftarrow \mathsf{str2num}(M_i)$
if $(m \geq d)$ **then** // *If word is not in range, then*
$y \leftarrow ky + marker \bmod p(v)$ // *Marker indicates out-of-range*
$y \leftarrow ky + (m - offset) \bmod p(v)$ // *Offset* m *back into range*
else
$y \leftarrow ky + m \bmod p(v)$ // *Otherwise hash in-range word*
return y

Figure 4.7: *The* Poly *algorithm, parameterized on key-set* K*, word-length* v *and domain-bound* d*.*

algorithm $\mathsf{RPHash}(\mathbf{k}, M)$
// *Parameters: Uses "vector length"* r*, "word-length vector"* $\mathbf{v}$*,*
// *"message-length vector"* $\mathbf{l}$*, "domain-bounds vector"* $\mathbf{d}$*, and "key-set vector"* $\mathbf{K}$*.*
// *Input:* $\mathbf{k} = (k_1, \ldots, k_r)$ *with* $k_i \in K_i$ *for* $1 \leq i \leq r$ *and* $M \in \{0,1\}^*$*.*
// *Output:* $Y \in \{0,1\}^{v_r}$*.*
$prepend \leftarrow \varepsilon$ // *Initially no string to prepend*
$i \leftarrow 1$ // *Index for* $\mathbf{v}$, $\mathbf{l}$, $\mathbf{K}$ *vectors*
while $(|M| > \ell_i v_i)$ **do** // *While more than one ramp-level remains*
if $(i = r)$ **then return** Error // *Message too long*
$T \leftarrow M[1 \ldots \ell_i v_i]$ // *Extract the string to be hashed under* $p(v_i)$
$M \leftarrow M[\ell_i v_i + 1 \ldots |M|]$
$y \leftarrow \mathsf{Poly}[K_i, v_i, d_i](k_i, prepend \parallel T)$ // *Hash in* $\mathbb{Z}_{p(v_i)}$*, prepending previous hash*
$prepend \leftarrow \mathsf{num2str}(y, v_{i+1})$ // *Update* prepend *for next ramp-level*
$i \leftarrow i + 1$
$M \leftarrow \mathsf{padonezero}(M, v_i)$ // *Final ramp-level needs bijective padding*
$y \leftarrow \mathsf{Poly}[K_i, v_i, d_i](k_i, prepend \parallel M)$ // *Hash in* $\mathbb{Z}_{p(v_i)}$*, prepending previous hash*
$Y \leftarrow \mathsf{num2str}(y, v_r)$ // *Convert to string*
return Y

Figure 4.8: *The* RPHash *algorithm. Parameters are described in Figure 4.6.*

Chapter 5

Variationally Universal Hashing

In Chapter 3 we stated as one of our goals for UMAC that it should use a universal hash function useful outside of message authentication. We called such a function a "generally useful" hash function. In this chapter we develop this idea more fully and define a type of universal hash function which fulfils the goal. This chapter is extracted from a preliminary version of [27].

5.1 Introduction

The UMAC message authentication code of Black, Halevi, Krovetz, Krawczyk and Rogaway uses a software-optimized universal hash family called UHASH to authenticate messages in the Wegman-Carter paradigm [10, 12, 39]. One of the goals in creating UHASH was to provide a "generally-useful" hash function, including public-domain code, which could be used off-the-shelf by the computer science community. In saying "generally-useful" we mean that no assumptions should be made about how the user of the hash function will use its output bits. If a "generally-useful" hash function outputs 32 bits, say, and the user only needs 16, then the user should be able to take an arbitrary collection of 16 out of the

32 output bits and have some assurance that they are good bits. The very strongest notion of universality, that a hash family is "strongly universal" (SU) does give this assurance, but common relaxations of SU do not. We offer a new relaxation for SU, one that looks at statistical variance, which allows the claim that a hash family is "generally-useful" without being SU.

Several flavors of universal hash functions have been suggested. The weakest one which seems to be useful is that of being "almost universal" (ϵ-AU). Given an ϵ-AU family H of hash functions, the probability that two distinct messages m and m' collide when hashed by a randomly selected member of H is at most ϵ. The notion of being strongly universal is at the other extreme. This notion says that for all distinct pairs of messages m and m', the images of these points, $h(m)$ and $h(m')$, are uniformly distributed under a randomly chosen hash function $h \in H$. This notion is often too *strong*: known constructions that achieve this are relatively inefficient, and applications usually require a less stringent property anyway.

Many relaxations of SU have been suggested—hash-function families which are almost-universal, almost-xor-universal, almost-delta-universal, and almost-strongly-universal. The last of these is the strongest of these relaxations. Let us recall its definition [14]. A hash-function family $H = \{h : A \to B\}$ is **ϵ-strongly universal**, written ϵ-SU, if for all distinct $m, m' \in A$ and all $c, c' \in B$,

(1) $\Pr_{h \in H}[h(m) = c] \;=\; \frac{1}{|B|}$, and

(2) $\Pr_{h \in H}[h(m) = c \mid h(m') = c'] \;\leq\; \epsilon$.

The first condition says that $h(m)$ is uniformly distributed over B given a randomly chosen h, and the second condition says that you can't guess $h(m)$ very well, even if you know $h(m')$. The number ϵ bounds how well $h(m)$ might be guessed. If ϵ is "low", we may say that H is "almost-SU".

While the notion of an almost-SU hash-function family might seem quite strong, we suggest that it is weaker and less generally useful than one might imagine. In particular, a hash-function family can be ϵ-SU, for a very small value of ϵ, and yet $h(m)$ does not look at all random once you know the value of $h(m')$, for some $m \neq m'$.

Let us give a concrete example. Consider the following "degenerate" hash-function family, $\tilde{H}_{16}$ The domain and range of $\tilde{H}_{16}$ are $\{0,1\}^{64}$. A random element $h \in \tilde{H}_{16}$ can be described as follows. First, let y be a random 16 bits. Then for any 64-bit m, let the first 16-bits of $h(m)$ be y, and let the remaining 48 bits be completely random. Then $\tilde{H}_{16}$ is 2^{-48}-SU. Yet $\tilde{H}_{16}$ has the undesirable property that once you know $h(m')$, for any value of m', $h(m)$ isn't random-looking at all—indeed you immediately know one quarter of the bits of $h(m)$. In applications, this can be a problem. Suppose, for example, that we are exporting a hash function to be used for some unknown purpose. We don't know which bits of the hash will be used how. Suppose that the user happens to need only 16 bits of hash, and so grabs the first 16. Though we exported a 2^{-48}-SU hash function, what the user sees in his application is a hash function in which all points collide. This is almost certainly not what the user had in mind!

For an example which more directly demonstrates the cryptographic implications, suppose that one is using the hash-function family within the Wegman-Carter construction. Authentication tags of 64-bits are being computed by the sender. Suppose the hash function has a structure (as UHASH does [12]) that allows one to compute the first 16 bits faster than one can compute the whole thing. Then the verifier, to save time, might wish to check only the first 16 bits of the authentication tag. But if the underlying hash function is (only) 2^{-48}-SU, it is entirely possible that the verifier's check will have no value. Intuitively, this is wrong. Checking the first 16 bits should be allowing the verifier to detect forgeries with

chance around $1 - 2^{-16}$. But this guarantee will definitely not follow from the 64-bit hash having been 2^{-48}-SU.

The conclusion we draw is that a hash family being almost-SU does not give adequate assurance that the hash output string appears random. The definition of ϵ-SU focusses on ones inability to know the *entire* value of $h(m)$ once one knows $h(m')$. We want instead to formalize that *everything* about $h(m)$ looks (almost) random, even if one knows $h(m')$.

NOTATION. If A is a finite set then $x \xleftarrow{R} A$ denotes the experiment of choosing a random element of A. A distribution on strings, D, is a probability measure over the universe $\Omega = \{0,1\}^*$. That is, each string $y \in \{0,1\}^*$ has an associated probability $D(y) \in [0,1]$, and $\sum_{y \in \{0,1\}^*} D(y) = 1$. By $[\![\, \mathit{experiment} \,:\, \mathit{outcome} \,]\!]$ we denote the distribution which is determined by performing the indicated experiment and then returning the specified outcome. The experiment may involve a semicolon-separated list of steps, which are performed in order. If A is a finite set then Uniform(A) is the uniform distribution on A. So in the notation just introduced, Uniform(A) $= [\![\, x \xleftarrow{R} A \,:\, x \,]\!]$. Let D and D' be distributions on strings. Then the *variational distance* between D and D', written $\mathit{dist}(D, D')$, is defined as

$$\mathit{dist}(D, D') \;=\; \sum_{y \in \{0,1\}^*,\; D(y) > D'(y)} (D(y) - D'(y)) = \frac{1}{2} \sum_{y \in \{0,1\}^*} \Big| D(y) - D'(y) \Big| .$$

5.2 ϵ-VU Hash Functions

The definition of ϵ-SU establishes ϵ as the maximum probability that $h(m)$ produces any *particular* value given the knowledge of $h(m')$. The definition takes no account of the remaining less likely output points, and in particular, only provides a very weak bound on how far from uniform is $h(m)$, given knowledge of $h(m')$. We suggest strengthening the definition of ϵ-SU by measuring the variational distance between $h(m)$ and the uniform

distribution, given knowledge of $h(m')$.

We say that hash-function family $H = \{h : A \to B\}$ is **ϵ-variational-universal**, written ϵ-VU, if for all distinct $m, m' \in A$, and all $c' \in B$,

(1) $[\![h \xleftarrow{R} H : h(m)]\!] = \text{Uniform}(B)$, and

(2) $dist([\![\textbf{repeat } h \xleftarrow{R} H \textbf{ until } h(m') = c' : h(m)]\!], \text{Uniform}(B)) \leq \epsilon$.

Roughly speaking, we demand that for any distinct m, m', $h(m)$ should look random, even if you know the value of $h(m')$. The value ϵ measures how far from random $h(m)$ might look (under the L_1 norm). If ϵ is "low", we may call H "almost-VU". If ϵ is zero, then we may call H "perfect-VU".

Equivalently, an ϵ-VU hash function can be characterized by bounding an adversary's ability to differentiate hash outputs from random values. Hash-function family $H = \{h : A \to B\}$ is ϵ-VU if for all distinct $m, m' \in A$, for all $c, c' \in B$, and for all functions $f : B \to \{0, 1\}$,

(1) $\Pr_{h \in H}[h(m) = c] = \frac{1}{|B|}$, and

(2) $\Pr_{h \in H}[f(h(m)) = 1 \mid h(m') = c'] - \Pr_{y \in B}[f(y) = 1] \leq \epsilon$.

The second condition can be interpreted as follows. Adversary f sees one of two things: y, where y is a random string in the range of the hash function; or $h(m)$, where h is a random hash function such that $h(m') = c'$. Then the adversary can't distinguish which of these two scenarios is in effect with advantage exceeding ϵ. Note that the adversary can be assumed to "know" the values m' and c', in that we are universally quantifying over all adversaries.

Here is a second interpretation. The algorithm f is a user application which uses the result of hashing. We don't know any details about this application. One imagines two scenarios: the algorithm f is fed the result of hashing m, or else it is fed a random range

value y. The second condition says that, whatever the algorithm f may be doing, the difference in behavior in these two scenarios will be at most ϵ, even if the algorithm has already made use of one other $h(m')$ value.

For comparison, think back to hash-function family $\tilde{H}_{16}$ described in the introduction. It was 2^{-48}-SU. But it is not ϵ-VU for a small value of ϵ. Letting $f(y)$ be the function which returns 1 if the first 16 bits of y are the same as the first 16 bits of c', then the difference in probabilities associated to condition (2) is $1 - 2^{-16}$; informally said, $\tilde{H}_{16}$ is almost-SU, but it is not almost-VU.

The first condition of both definitions are simple restatements of each other. The first (variationally-based) definition can be easier to work with than the second since we do not have to quantify over all functions f.

5.2.1 Some Common Hash Functions

One might wonder if any common hash functions are almost-VU. One common class, those which are SU, are all perfect-VU because the SU definition requires that the first two outputs from a randomly chosen function be perfectly uniform. But, beyond SU functions, almost-VU function families are hard to find.

Consider hashing using polynomial evaluation [1, 9, 14, 36]. In one form of this paradigm [14], messages are broken up into words and the words are interpreted as coefficients in a polynomial over some field. Given a prime p, the following hash family $H = \{h_{a,b} : \mathbb{Z}_p^n \to \mathbb{Z}_p\}$ hashes n-vectors and is (n/p)-SU. Given vector $\mathbf{m} = (m_n, \ldots, m_1)$ and keys $a, b \in \mathbb{Z}_p$, the hash of $\mathbf{m}$ is

$$h_{a,b}(\mathbf{m}) = \left(b + \sum_{i=1}^{n} m_i a^i \right) \bmod p \; .$$

Choosing a random element of H is done by choosing a random $a, b \in \mathbb{Z}_p$.

Although this family is (n/p)-SU, which is good if n is small and p is large, the hash family is not even (1/3)-VU. Let $n = 2$ and $p > 3$ be a prime. Let $\mathbf{m} = (1, 0)$, $\mathbf{m}' = (0, 0)$ and $c' = 0$. Because $n = 2$ the hash function we are interested in is $h_{a,b}(\mathbf{m}) = (b + m_2 a^2 + m_1 a) \bmod p$. We look at this hash function in light of the variational definition of ϵ-VU. Condition (1) of the definition requires $[\![\, a, b \xleftarrow{R} \mathbb{Z}_p \,:\, h_{a,b}(\mathbf{m}) \,]\!] = \text{Uniform}(\mathbb{Z}_p)$, which is satisfied because of the random translation b. Condition (2) requires computation of $dist([\![\, \textbf{repeat } a, b \xleftarrow{R} \mathbb{Z}_p \textbf{ until } h_{a,b}(\mathbf{m}') = c' \,:\, h(m) \,]\!], \text{Uniform}(\mathbb{Z}_p))$. However, because of the values we have selected for $\mathbf{m}$, $\mathbf{m}'$ and c', this computation simplifies to $dist([\![\, a \xleftarrow{R} \mathbb{Z}_p \,:\, a^2 \,]\!], \text{Uniform}(\mathbb{Z}_p))$, which is exactly $(p-1)/2p$, a number greater than 1/3 for any $p > 3$.

5.3 ϵ-VU Constructions

We now examine some constructions which result in almost-VU hash functions likely more efficient than SU functions. SU hash families tend to be compute intensive and often require long keys to specify members of the family, while recent research into software-efficient AU hash functions has produced families which process messages at rates as low as 0.5 CPU cycles per byte [10, 21, 35]. These families can be used to first hash a message into a much smaller string, which can then be hashed by an SU hash family to produce an output. This approach results in a hash family nearly as fast as the software-optimized hash, but with an almost-VU guarantee.

Let $H = \{h : A \to B\}$ and $G = \{g : B \to C\}$ be two families of functions. We define the composed family of functions $G \circ H = \{f : A \to C\}$ as follows. For all $g \in G$ and all $h \in H$, the composed function $f = g \circ h$ is in $G \circ H$.

Theorem 5.3.1 *If* $H^{\mathrm{au}} = \{h : A \to B\}$ *is* ϵ*-AU, and* $H^{\mathrm{su}} = \{g : B \to C\}$ *is SU, then* $H^{\mathrm{su}} \circ H^{\mathrm{au}}$ *is* $(\epsilon - \frac{\epsilon}{|C|})$*-VU.*

Proof. Let $c' \in C$ and $m \neq m' \in A$ be arbitrary values such that $g \circ h(m') = c'$ for at least one pair of $g \in H^{\mathrm{su}}$ and $h \in H^{\mathrm{au}}$. In choosing a random element from $H^{\mathrm{su}} \circ H^{\mathrm{au}}$, we do so by choosing random elements $h \in H^{\mathrm{au}}$ and $g \in H^{\mathrm{su}}$, and considering $g \circ h$ to be the random element from $H^{\mathrm{su}} \circ H^{\mathrm{au}}$. Because H^{su} is SU, $g \circ h(m)$ is uniformly distributed over C for randomly chosen $g \in H^{\mathrm{su}}$ and any $h \in H^{\mathrm{au}}$. For convenience, let f be shorthand for $g \circ h$, let all probability measures be over the choice of g and h, and let $p = \Pr[h(m) = h(m')]$. We need to show that $dist(\llbracket \textbf{repeat } g \xleftarrow{R} H^{\mathrm{su}}; h \xleftarrow{R} H^{\mathrm{au}} \textbf{ until } f(m') = c' \ : \ f(m) \rrbracket, \mathrm{Uniform}(C))$ is no more than $(\epsilon - \frac{\epsilon}{|C|})$. We begin by using the definition of *dist* to get an equivalent equation

$$\frac{1}{2} \sum_{c \in C} \left| \Pr[f(m) = c | f(m') = c'] - \frac{1}{|C|} \right| .$$

We rewrite this equation with the $c = c'$ term extracted from the summation,

$$\frac{1}{2} \left(\left| \Pr[f(m) = c' | f(m') = c'] - \frac{1}{|C|} \right| + \sum_{c \in C, c \neq c'} \left| \Pr[f(m) = c | f(m') = c'] - \frac{1}{|C|} \right| \right) , \tag{5.3.1}$$

and then solve each of the two terms in the resulting equation, conditioning on the event $h(m) = h(m')$. First we look at $\Pr[f(m) = c' | f(m') = c'] - 1/|C|$ which can be rewritten as

$$\begin{aligned} & \Pr[f(m) = c' | f(m') = c' \wedge h(m) = h(m')] \cdot \Pr[h(m) = h(m')] \qquad (5.3.2) \\ + \quad & \Pr[f(m) = c' | f(m') = c' \wedge h(m) \neq h(m')] \cdot \Pr[h(m) \neq h(m')] - \frac{1}{|C|} . \end{aligned}$$

Notice that if $h(m) = h(m')$, then $f(m)$ must be equal to $f(m')$, and that if $h(m)$ and $h(m')$ are distinct, then $\Pr[f(m) = f(m')] = 1/|C|$ because H^{su} is strongly universal. Letting $p = \Pr[h(m) = h(m')]$, we can simplify the Equation 5.3.2 to

$$p + \frac{1}{|C|}(1 - p) - \frac{1}{|C|} = p - \frac{p}{|C|} . \tag{5.3.3}$$

Next, we look at the second term from Equation 5.3.1, $\Pr[f(m) = c | f(m') = c'] - \frac{1}{|C|}$, with $c \neq c'$, which can be rewritten as

$$\begin{aligned} & \Pr[f(m) = c | f(m') = c' \wedge h(m) = h(m')] \cdot \Pr[h(m) = h(m')] \\ + \quad & \Pr[f(m) = c | f(m') = c' \wedge h(m) \neq h(m')] \cdot \Pr[h(m) \neq h(m')] - \frac{1}{|C|} \,. \end{aligned} \tag{5.3.4}$$

This time, because $c \neq c'$ and H^{su} is SU, $\Pr[f(m) = c | f(m') = c' \wedge h(m) = h(m')] = 0$ and $\Pr[f(m) = c | f(m') = c' \wedge h(m) \neq h(m')] = 1/|C|$. Again letting $p = \Pr[h(m) = h(m')]$, we can simplify Equation 5.3.4 to

$$0 + \frac{1}{|C|}(1 - p) - \frac{1}{|C|} = -\frac{p}{|C|} \,. \tag{5.3.5}$$

Substituting Equations (5.3.3) and (5.3.5) into Equation 5.3.1 results in

$$\frac{1}{2}\left(\left|p - \frac{p}{|C|}\right| + \sum_{c \in C, c \neq c'} \left|-\frac{p}{|C|}\right|\right) = \frac{1}{2}\left(p - \frac{p}{|C|} + (|C| - 1)\frac{p}{|C|}\right) = p - \frac{p}{|C|} \,.$$

And, finally, because $p = \Pr[h(m) = h(m')]$, and H^{au} is ϵ-AU, we know that $p \leq \epsilon$ and so $\mathit{dist}([\![\textbf{repeat } g \xleftarrow{\text{R}} H^{\text{su}}; h \xleftarrow{\text{R}} H^{\text{au}} \textbf{ until } f(m') = c' \,:\, f(m)]\!], \text{Uniform}(C)) \leq (\epsilon - \frac{\epsilon}{|C|})$. $\diamondsuit$

We now know two ways to achieve almost-VU hashing: Use an SU hash function directly, or combine an AU hash function with an SU one. Unfortunately, this does not completely solve the problem we wish to address: The arbitrary use of a hash function's output bits. Many convenient SU hash functions produce values in a range set which is not $\{0,1\}^w$ for any number w. For instance, the inner-product over a prime field $\mathbb{Z}_p$ produces values from the set $\mathbb{Z}_p$. Although the hash is SU (and therefore 0-VU), extracting arbitrary bits for use in an application cannot be done without violating the nature of the VU guarantee. If w bits are required to represent all the values in $\mathbb{Z}_p$, then the hash results will likely be represented as w-bit strings. None of the w bits in these strings will be uniformly distributed due to the fact that not all 2^w possible w-bit strings can be produced by the inner-product hash. This

introduces small biases into each of the w bits. If arbitrary bit extraction from the w-bit string is desired, then the 0-VU hash over $\mathbb{Z}_p$ must be converted into an almost-VU hash over $\{0,1\}^w$. This is straightforward. The next two theorems show, first, how to enlarge the domain of an almost-VU hash function, and second, that it is okay to extract arbitrarily the bits from the enlarged domain.

Let $H = \{h : A \to \mathbb{Z}_a\}$ be a family of functions. For $b > a$ we define $H_b = \{g : A \to \mathbb{Z}_b\}$ as follows: For every $h \in H$ and $r \in \mathbb{Z}_b$, the function $g(x) = (h(x) + r) \bmod b$ is in H_b. In the following theorem think of b as 2^w for some $w > 1$ and think of a as a prime just smaller than b.

Theorem 5.3.2 *For any positive $a < b$ and ϵ-VU hash family $H = \{h : A \to \mathbb{Z}_a\}$, the hash family $H_b = \{g : A \to \mathbb{Z}_b\}$ is $(\epsilon + 1 - \frac{a}{b})$-VU.*

Proof. Let $a < b$ be arbitrary positive integers and let $H = \{h : A \to \mathbb{Z}_a\}$ be ϵ-VU. Choosing $g \in H_b$ randomly yields a function $g(x) = (h(x) + r) \bmod b$ where $h \in H$ and $r \in \mathbb{Z}_b$ are both uniformly distributed. Let $m, m' \in A$ be distinct and let $c' \in \mathbb{Z}_b$ be arbitrary. Clearly, for randomly chosen g, $g(m)$ is uniformly distributed in Z_b because of the random translation r. We need only show $dist([\![\textbf{repeat } g \xleftarrow{R} H_b \textbf{ until } g(m') = c' \,:\, g(m)]\!], \mathrm{Uniform}(\mathbb{Z}_b)) \leq \epsilon + 1 - \frac{a}{b}$. This distance can be rewritten as

$$\frac{1}{2} \sum_{c \in \mathbb{Z}_b} \left| \Pr_{g \in H_b}[g(m) = c \mid g(m') = c'] - \frac{1}{b} \right| .$$

For any fixed r, the value of the function $g(x) = (h(x) + r) \bmod b$ can take on only one of a numbers determined by the value of $h(x)$. Call the set of these a values (for a particular r) $S_r \subseteq \mathbb{Z}_b$. The values $\mathbb{Z}_b - S_r$ are unreachable by $g(x) = (h(x) + r) \bmod b$. Now let's condition on the event that r is chosen to be some $k \in \mathbb{Z}_b$. We can rewrite the above distance as

$$
\begin{aligned}
& \frac{1}{2}\left(\sum_{c \in S_k} \left| \Pr_{g \in H_b}[g(m) = c \mid g(m') = c' \wedge r = k] - \frac{1}{b} \right| + \right. \\
& \left. \sum_{c \in \mathbb{Z}_b - S_k} \left| \Pr_{g \in H_b}[g(m) = c \mid g(m') = c' \wedge r = k] - \frac{1}{b} \right| \right) \\
= \; & \frac{1}{2}\left(\sum_{c \in S_k} \left| \Pr_{g \in H_b}[g(m) = c \mid g(m') = c' \wedge r = k] - \frac{1}{a} + \frac{1}{a} - \frac{1}{b} \right| + \sum_{c \in \mathbb{Z}_b - S_k} \left| 0 - \frac{1}{b} \right| \right) \\
\leq \; & \frac{1}{2}\left(\sum_{c \in S_k} \left| \Pr_{g \in H_b}[g(m) = c \mid g(m') = c' \wedge r = k] - \frac{1}{a} \right| + \sum_{c \in S_k} \left| \frac{1}{a} - \frac{1}{b} \right| + (b-a)\left(\frac{1}{b}\right) \right) \\
= \; & \epsilon + \frac{1}{2}\left(a\left(\frac{1}{a} - \frac{1}{b}\right) + (b-a)\left(\frac{1}{b}\right) \right) = \epsilon + 1 - \frac{a}{b} \; .
\end{aligned}
$$

The final ϵ term needs some explaining. The set S_k is simply a translated version of the set $\mathbb{Z}_a$ (modulo $\mathbb{Z}_b$). The distribution of $g(m)$ over S_k is therefore isomorphic to that of $h(m)$ over $\mathbb{Z}_a$, allowing us to use the ϵ bound due to H being ϵ-VU. The above derivation holds no matter what the value of $k \in \mathbb{Z}_b$, so we conclude that the derivation is true regardless of the value of r. $\diamondsuit$

If we have an almost-VU hash family $H = \{h : A \to B\}$ with a range set B, but only a smaller range set C is desired, the next theorem states that any function $f : B \to C$ can be used to transform hash output points into C so long as the preimage set for every element of C is equal in size under f. For example, if the range of H were $\{0,1\}^{64}$ and we wished to use only the first 16 bits of each hash output, then we would be implicitly defining a function $f : \{0,1\}^{64} \to \{0,1\}^{16}$ that simply returned the first 16 bits.

Some notation. If $H = \{h : A \to B\}$ is a hash family and $f : B \to C$ is a function, then we define $f \circ H = \{g : A \to C\}$ as a hash family which is defined as follows: If h is in H, then $g = f \circ h : A \to C$ is in $f \circ H$.

Theorem 5.3.3 *Let $H = \{h : A \to B\}$ be ϵ-VU and let $f : B \to C$ be a function such that the preimage set for every element of C has equal size under f. Then, $f \circ H$ is also ϵ-VU.*

Proof. Let $H = \{h : A \to B\}$ be ϵ-VU and let $f : B \to C$ be a function such that the preimage set for every element of C has equal size under f. There are two parts to showing $f \circ H$ is VU. First, the function f maps an equal number of elements in B to each element in C. This ensures that for randomly chosen $h \in H$ the value $f \circ h(m)$ is uniform in C because $h(m)$ is assumed uniform in B. For the second part of the VU definition, we need to show that $dist([\![\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' \ : \ f \circ h(m)]\!], \text{Uniform}(C)) \leq \epsilon$ for any f and $c' \in C$. Let f and c' maximize this distance. Let $S \subseteq C$ be the set of elements such that for every $c \in S$, $\Pr[f \circ h(m) = c \mid f \circ h(m') = c'] > 1/|C|$. Following the definition of *dist*, we can rewrite the distance as

$$\sum_{c \in S} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' : f \circ h(m) = c] - \frac{1}{|C|} \right) .$$

Our goal is to show this summation (and therefore the distance in question) is no greater than ϵ. We will do so by writing a series of equivalences, the last of which is no more than ϵ. First we look at the event $f \circ h(m) = c$. Because f and h are composed, for each $c \in S$ there may be multiple values for $h(m)$ which map to c. So, we rewrite the equation with an additional summation over the preimage set $f^{-1}(c)$,

$$\sum_{c \in S} \left(\left(\sum_{c^* \in f^{-1}(c)} \Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' : h(m) = c^*] \right) - \frac{1}{|C|} \right) .$$

But, the cardinality of $f^{-1}(c)$ is $|B|/|C|$, so may adjust the $1/|C|$ term and bring it inside the second summation and instead write

$$\sum_{c \in S} \left(\sum_{c^* \in f^{-1}(c)} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' : h(m) = c^*] - \frac{1}{|B|} \right) \right) .$$

The summations can be consolidated. We define $S' = \{b \in B | f(b) \in S\}$, which is the union of all preimage sets for all elements in S, and rewrite as

$$\sum_{c \in S'} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' : h(m) = c] - \frac{1}{|B|} \right) , \tag{5.3.6}$$

which is still equal to $dist([\![\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' \ : \ f \circ h(m)]\!], \text{Uniform}(C))$. We know because H is ϵ-VU

$$\sum_{c \in S'} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } h(m') = c'' : h(m) = c] - \frac{1}{|B|} \right) \leq \epsilon \tag{5.3.7}$$

for any choice of c''. All that is left is to show Equation 5.3.6 is less than Equation 5.3.7. Notice that in Equation 5.3.6, the condition enforced by the repetition is that $f \circ h(m') = c'$, which is satisfied when $h(m') \in f^{-1}(c')$. Let us choose c'' to be the one from $f^{-1}(c')$ that maximizes Equation 5.3.6. This choice ensures

$$\begin{aligned} & \sum_{c \in S'} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } f \circ h(m') = c' : h(m) = c] - \frac{1}{|B|} \right) \\ \leq & \sum_{c \in S'} \left(\Pr[\textbf{repeat } h \xleftarrow{R} H \textbf{ until } h(m') = c'' : h(m) = c] - \frac{1}{|B|} \right) \\ \leq & \ \epsilon \ . \qquad \diamondsuit \end{aligned}$$

As an example look at UHASH [12]. Let $p(x)$ be the largest prime smaller than 2^x. Roughly speaking, UHASH combines a fast (2^{-31})-AU hash function, let's call it $H_1 = \{h_1 : \{0,1\}^* \rightarrow \mathbb{Z}_{p(128)}\}$, with a slower perfect-SU inner-product hash function $H_2 = \{h_2 : \mathbb{Z}_{p(128)} \rightarrow \mathbb{Z}_{p(36)}\}$, followed by a random translation into the set $\{0,1\}^{36}$. Theorem 5.3.1 says that $H_2 \circ H_1$ is $2^{-31}(1 - \frac{1}{2^{36}-5})$-VU, and Theorem 5.3.2 says that the random translation makes the overall composed hash function $\left(2^{-31}(1 - \frac{1}{2^{36}-5}) + 1 - \frac{2^{36}-5}{2^{36}}\right)$-VU, very nearly $(2^{-31} + 5 \cdot 2^{-36})$-VU, over $\{0,1\}^{36}$. Because the AU hash function does most of the work on long data inputs, the overall speed of the hash is approximately the speed of the AU hash function: Approximately 1.0 Pentium II cycles per input byte.

Chapter 6

UMAC (2000)

6.1 Introduction

The UMAC (1999) message authentication code (MAC) proposed by Black, Halevi, Krawczyk, Krovetz and Rogaway combined a fast software-optimized hash family, NH, with a pseudorandom function, either CBC-RC6 or HMAC-SHA1, to produce a MAC with high throughput [10]. For a MAC with forging probability of 2^{-60} their MAC had peak speeds of 1.0 Pentium II cycles per byte (cpb) with the use of Pentium MMX SIMD parallelism and 1.9 cpb without. These headline rates tell an impressive performance story for authenticating messages longer than 4KB, but a lot of traffic requiring MACs, particularly IP flows, is heavily geared towards short messages. According to contacts at Cisco Systems, a fair rule-of-thumb for the distribution on message-sizes on an Internet backbone is that roughly one-third of messages are 43 bytes (TCP ACKs), one-third are about 256 bytes (common PPP dialup MTU), and one-third are 1,500 bytes (common Ethernet MTU).

Using the optimized code found at the UMAC website, a different story unfolds when looking at UMAC (1999) performance on these message sizes. As many as 52.9 cpb are

required to authenticate 43-byte messages. Clearly there is scope to improve UMAC performance on shorter messages. We took as our job to revise UMAC so that it is substantially faster on small messages, while maintaining comparably fast peak performance. We name our versions UMAC32 (the non-SIMD variant) and UMAC16 (the SIMD variant), and achieve throughput as many as three times faster than the original versions (UMAC-STD and UMAC-MMX).

	43 bytes	256 bytes	1500 bytes	256 kbytes
UMAC32	**16.3**	**3.8**	**2.1**	**1.9**
UMAC-STD	52.9	12.3	3.8	1.9
UMAC16	**14.0**	**2.7**	**1.2**	**1.0**
UMAC-MMX	35.9	4.5	1.7	1.0

We accomplish this speedup by using the same software-optimized hash function NH used by the original UMAC, but in a different manner. The original UMAC used NH as an accelerant to CBC-RC6 or HMAC-SHA1 [2, 3, 34]. A function from the NH family was used to compress into a shorter string the message being authenticated. This shorter string was then processed by CBC-RC6 or HMAC-SHA1. Because NH hashes messages much faster than CBC-RC6 or HMAC-SHA1, the combination of first using NH on a long message and then using CBC-RC6 or HMAC-SHA1 on a short string results in a much faster overall processing time than just MACing the entire message with CBC-RC6 or HMAC-SHA1. This is true for long messages, but the compression is much less effective on short messages and so achieves a much less impressive speedup because the expensive PRF becomes dominant.

Instead of using NH as an accelerant for an expensive PRF, we instead use NH as an accelerant for another universal hash function appropriate for use in a Carter-Wegman MAC. In a Carter-Wegman MAC one first applies to a message a hash function drawn from some "universal" hash-function family, and then one encrypts the resulting hash [14, 39]. Compared to more "traditional" approaches, like CBC-MAC and HMAC, Carter-Wegman

MACs offer substantial benefits in both speed and demonstrated security. See [10, 21, 35] for discussions on the advantages of this approach, particularly when one is focussed on software efficiency.

In this chapter, and an accompanying specification document (Appendix A), we revise UMAC with the following three main goals:

• ***Minimizing "cryptography."*** The original UMAC would compress with the NH hash function the message being authenticated and then process the compressed (but still unbounded in length) message with a PRF. The use of a cryptographic primitive, such as RC6 or SHA1, is unbounded in this PRF and proportional to the message being authenticated. We suggest following the Carter-Wegman approach and hashing the message all the way to a fixed length before using any cryptography. This approach has the heuristic benefit that it minimizes the use of cryptography, and the provable-security benefit that the security analysis is now completely independent of the length of the messages being authenticated.

• ***Faster MACing of short messages.*** As previously mentioned, many short messages require authentication, and our UMAC revision pays close attention to this need without sacrificing peak performance on longer messages. Being fast on short messages is partially a side-effect of minimizing the cryptography used during authentication: Hash functions can be made much faster than common cryptographic primitives.

• ***Selective-assurance verifiability.*** The revised UMAC is constructed so that if one computes a 64-bit MAC, say, then one can verify the first 32 bits at nearly twice the speed of verifying the whole thing, and (with UMAC16) one can verify the first 16 bits at nearly four times the speed. Most MACs, including the original UMAC, do not have this property. This feature was suggested to us by David Balenson and David Carman (NAI Labs).

A secondary goal of ours is to describe a fast universal hash function useful in domains outside of message authentication. Our move to Carter-Wegman MACing has allowed us to achieve all of these goals.

6.1.1 UMAC Basic Description

UMAC16 and UMAC32 are Carter-Wegman MACs [39]. As such, a MAC tag is generated for a particular message by hashing the message with a universal hash function and then encrypting the hash function's output. In this chapter and the accompanying specification, we suggest doing the encryption by enciphering a nonce with a block-cipher and using the block-cipher output as a pad to be XOR'ed with the hash output. The tag, nonce and message are then sent to the receiver who can verify the message authenticity by comparing the received tag with one he generates using the received message and nonce. The sender and receiver share a secret key for both the universal hash function and block-cipher.

In the scenario just described, the block-cipher does a fixed amount of work for each message authenticated. The key to maximizing the speed for a Carter-Wegman MAC is therefore to maximize the speed of the universal hash function used. This chapter is primarily dedicated to describing UHash, a software-optimized universal hash function, and showing it has good security properties when used in a Carter-Wegman MAC. We describe two variants of UHash, UHash16 and UHash32, but for simplicity we concentrate on UHash16.

UHash16 uses three families of hash function in its work. See Figure 6.1. At a very high level, when presented with message M, UHash16 first breaks the message into 2KB chunks so that $M_1 \parallel \cdots \parallel M_n = M$ (with M_n possibly shorter than 2KB). A key k_1 is used to select a member of the NH hash family, and it is used to hash each message chunk. The concatenation of the NH hash outputs becomes the string $A = \mathsf{NH}_{k_1}(M_1) \parallel \cdots \parallel \mathsf{NH}_{k_1}(M_n)$.

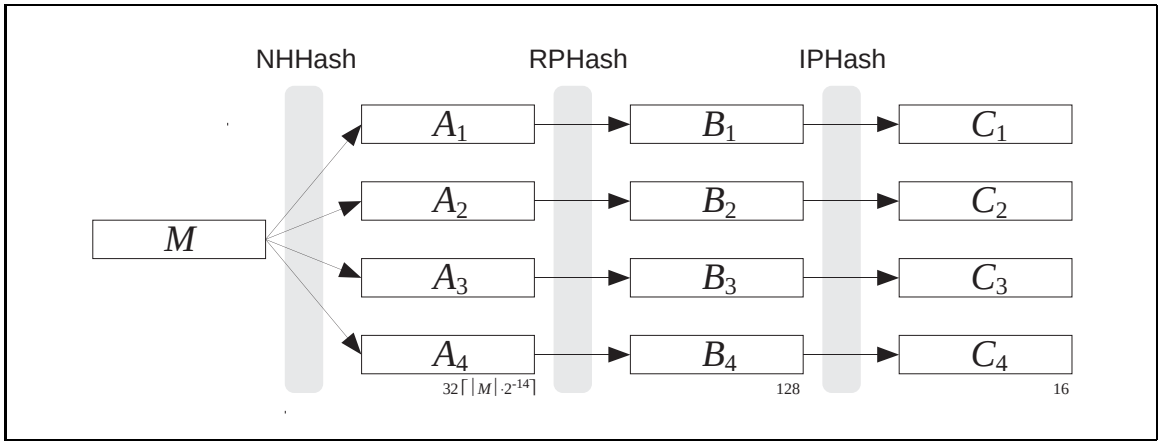


Figure 6.1: *General organization of* UHash16. *Message M is hashed by* NHHash16 *into strings A_1, A_2, A_3 and A_4, each of length $32\lceil |M| \cdot 2^{-14} \rceil$ bits. Each of these messages is then hashed independently by* RPHash32_64_128 *into strings B_1, B_2, B_3 and B_4, each of length* 128 *bits. These strings are finally hashed independently by* IPHash16 *into strings C_1, C_2, C_3 and C_4. The result of the hash is (C_1, C_2, C_3, C_4). If $|M| \leq 2^{14}$, then* RPHash32_64_128 *is bypassed and $B_i \leftarrow A_i$ for all $1 \leq i \leq 4$.*

Because this first NH hash layer may produce long output, a second hash layer is employed to hash A into a fixed length. Key k_2 is used to select a member of the RPHash family, and A is hashed to a fixed length 128-bit string $B = \mathsf{RPHash}_{k_2}(A)$. The hash family RPHash is polynomial-based and analyzed in Chapter 4. At this point, string B is longer than needed and lacks a requisite security guarantee, so a key k_3 is used to select a member of one last hash family, IPHash, and B is hashed to the string $C = \mathsf{IPHash}_{k_3}(B)$. The hash family IPHash is based on an inner-product hash and is "strongly universal". If M is no longer than 2KB, then RPHash can be omitted as an optimization and B can instead be defined as A. Regardless, the string C is appropriate for use in a Carter-Wegman MAC. If a MAC is desired with a lower forging probability, then the steps taken to generate C can be repeated with new keys, the output string from each iteration being concatenated together. The key k_1 used in each iteration can be related by being "Toeplitz" shifted between iterations, thus reducing the need for new key material. Later sections give more detailed descriptions of

these algorithms and of exactly what security guarantee is provided.

6.1.2 Related Work

This work began as a continuation of the originally proposed UMAC [10]. Before settling on the ideas contained in this chapter, we vetted one other promising approach to fast universal hashing. In this other approach, described more fully in Chapter 3, we would break a message into 2KB chunks and hash each block using NH with a 2KB key. Concatenating the NH outputs would yield a compressed (but unbounded in length) string. This string we would then hash again with NH, but this time using a conceptually-infinite key. Because this second key was unbounded, NH produced a fixed-length output which we could easily use in a Carter-Wegman MAC. The infinite key could be produced "on-the-fly" using a block- or stream-cipher. As in UHash, multiple iterations of this approach using Toeplitzed keys could be used to achieve lower collision probabilities and selective-assurance verifiability. This approach yields a universal hash of approximately the same speed as UHash, including good performance on short messages, but requires unbounded use of cryptography and, for maximum speed, requires significant caching of the second-layer NH key.

Over the years several universal hash-function families have been proposed, each intended for constructing an efficient MAC. Some software-efficient constructions are the CRC-like method of Krawczyk [25], some variants of this by Shoup[36], the bucket-hash family of Rogaway [35], the MMH family of Halevi and Krawczyk [21], and a polynomial-based function of Bernstein [9].

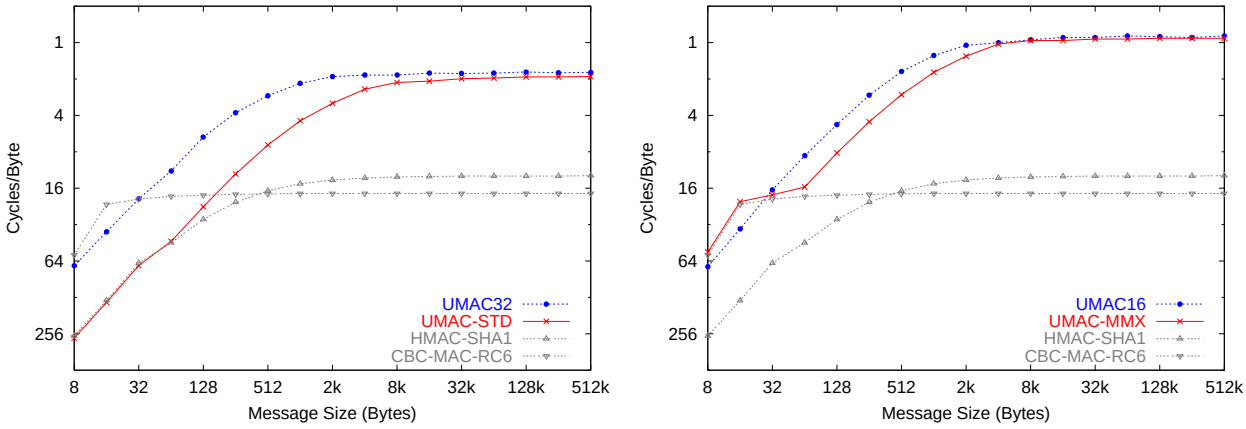


Figure 6.2: *Throughput rates on various message sizes for the revised UMAC compared with those of the original UMAC. Data was collected on a 700 MHz Pentium III and measured in cycles per byte of message authenticated.* ***Left:*** *UMAC32 versus UMAC-STD.* ***Right:*** *UMAC16 versus UMAC-MMX.*

6.2 Performance

We implemented UMAC16 and UMAC32 as specified in Appendix A and gathered timings on a 700 MHz Pentium III. The code was written mostly in C with about 140 lines of assembly implementing critical parts of RC6, NH, RPHash, and IPHash. Compilation was under `gcc` version 2.95. We also gathered timings for UMAC-MMX and UMAC-STD using optimized public-domain code. Results for generating 64-bit MAC tags with about 2^{-60} forgery probability are depicted in Figure 6.2. For each tested MAC, we determined it's throughput on variously sized messages, eight bytes through 512 KBytes. The experimental setup ensured that messages resided in level-one cache regardless of their length. Comparison results are given from [10] for HMAC-SHA1 [17] and the CBC-MAC of a fast block cipher, RC6 [34].

UMAC16 and UMAC32 are faster than their respective counterparts, UMAC-MMX and UMAC-STD, almost everywhere. Importantly, the new versions are up to three times faster on messages in the 32 – 2KB range. Our current implementation of UMAC16 is slower

than UMAC-MMX on messages shorter than 32 bytes. This is because UMAC-MMX uses a highly optimized CBC-MAC as a special case for very short messages and UMAC16 does not. We believe that a more careful UMAC16 implementation will overcome most, or all, of this differential.

The main reason for improvement in performance over the original UMAC is the reduction in use of cryptography. In the case of UMAC-STD, even very short messages incurred the cost of one HMAC-SHA1 invocation (about 1500 cycles). This cost is mitigated as message lengths increase, but remains very expensive on a cycle-per-byte basis for short messages.

NH Key-Length. The results depicted for UMAC16 and UMAC32 in Figure 6.2 are from UMAC variants which use 2KB of key in the first-layer NH hash. This key-length has a big impact on performance because it determines the amount of compression achieved in the first layer of hashing. Larger key-lengths mean greater compression, and so higher performance, until the key becomes so large that the data being hashed and the key used by NH no longer simultaneously fit the processor cache. Larger keys also aid performance because second-layer hashing is not used if a message is not longer than the first-layer NH key. The following table shows performance (in cpb) on 1500 byte and 256KB messages when authenticated with UMAC under various different first-layer key-lengths.

		512 Bytes	1KB	2KB	4KB	8KB
UMAC16	1500	1.47	1.38	1.17	1.17	1.17
	256KB	1.11	0.96	0.88	0.84	0.85
UMAC32	1500	2.50	2.34	2.08	2.10	2.10
	256KB	2.10	1.85	1.75	1.69	1.71

We chose a 2KB first-layer key because it encompassed our three "rule-of-thumb" common message lengths: 43, 256 and 1500 bytes.

```
algorithm NHHash16(K, M)
// Input: K ∈ {0,1}^(2^14) and M ∈ {0,1}*.
// Output: Y ∈ {0,1}^(32m) where m = ⌈|M|/2^14⌉.
Y ← ε                                          // emptystring
m ← ⌈|M|/2^14⌉                                 // Number of hash-blocks
M_1 ‖ ... ‖ M_m ← M,                           // Break M in 2^14-bit chunks
     where |M_1| = ··· = |M_{m−1}| = 2^14
for i ← 1 to m do                              // For each hash-block
     Len ← num2str(|M_i|, 32)                  // Length of |M_i| before any padding
     M_i ← padzero(M_i, 32)                    // A no-op if |M_i| divisible by 32
     Y ← Y ‖ (NH[2^10, 16](K, M_i) +_32 Len)   // Hash, add length and append to Y
return Y
```

Figure 6.3: *The* NHHash16 *algorithm. The message M is broken into chunks no longer than 2^{14} bits. Each chunk is hashed by* NH*, and to each of these hashed values the length of the chunk being hashed is added. If M is the emptystring, then* padzero*(M,w) transforms M into 0^w. Any* NH *variant (signed or unsigned, strided or not) may be used in* NHHash16*, and these choices are left as parameters in the formal specification of UMAC in Appendix A. The theorems in this chapter address* NHS*, the signed variant, because of all the* NH *variants,* NHS *has the most adverse impact on security.*

6.3 UMAC Description

In this chapter we do not give a complete specification of UMAC16. That would require the description of several fairly standard details. Instead we focus on UHash16, the universal hash function at the heart of UMAC16. We refer the reader to Appendix A for complete details of how UHash16 is combined with a block-cipher to create UMAC16. Very simply, a message is hashed with UHash16 and a nonce is enciphered with a block-cipher. The bitwise exclusive-or of these two fixed-length strings is then produced as the MAC tag.

Figures 6.3 – 6.7 give complete algorithmic descriptions of UHash16. In this section we discuss UHash16 in a functional manner. A message being hashed by UHash16 is processed by three layers: A compression layer, a hash-to-fixed-length layer and a strengthen-and-fold layer. We look more closely at each of these layers. But first we define some notation used in the figures. Let ε be the empty string. Let $|S|$ return the bit-length of string S.

```
algorithm Poly[w, b](k, M)
// Input: k ∈ K_w and M ∈ ({0,1}^w)^+.
// Output: y ∈ Z_{p(w)}.
p ← p(w)                                  // The largest prime smaller than 2^w
offset ← 2^w − p                          // Constant for translating out-of-range words
marker ← offset − 1                       // Constant for indicating out-of-range words
n ← |M|/w
M_1 ‖ ... ‖ M_n ← M,                      // Break M into w-bit chunks
     where |M_1| = ··· = |M_n| = w
y ← 1                                     // Set highest coefficient to 1
for i ← 1 to n do
     m ← str2num(M_i)
     if (m ≥ b) then                      // If word is not in range, then
          y ← ky + marker mod p           // Marker indicates out-of-range
          y ← ky + (m − offset) mod p     // Offset m back into Z_p
     else
          y ← ky + m mod p                // Otherwise hash in-range word
return y
```

Figure 6.4: *The* Poly *algorithm parameterized on "word-length" w and "word-bound" b. Given a string made of n words, each w-bits long,* Poly *hashes the string into an element of $\mathbb{Z}_{p(w)}$ using a polynomial hash. The value of w is expected to be 32, 64 or 128, corresponding to the key-sets K_{32}, K_{64} and K_{128}.*

Let $S \,\|\, T$ be the concatenation of S and T. Let $p(x)$ be the largest prime smaller than 2^x. Let K_{32}, K_{64} and K_{128} be the sets $\{x : 0 \leq x < 2^{29}\}$, $\{a2^{32} + b : 0 \leq a, b < 2^{25}\}$ and $\{a2^{96} + b2^{64} + c2^{32} + d : 0 \leq a, b, c, d < 2^{25}\}$, respectively. Let 0^n be the string made of n zero-bits. Let $\mathsf{padonezero}(S,w)$ return the string $S \,\|\, 1 \,\|\, 0^n$ where n is the smallest number so that the length of $S \,\|\, 1 \,\|\, 0^n$ is divisible by w. Let $\mathsf{padzero}(S,w)$ return the string $S \,\|\, 0^n$ where n is the smallest number so that the length of $S \,\|\, 0^n$ is a positive multiple of w. This definition causes $\mathsf{padzero}(\varepsilon,w)$ to return 0^w. Let $\mathsf{num2str}(y,w)$ return the w-bit binary representation of y. And, let $\mathsf{str2num}(S)$ return the integer for which S is the binary representation.

First layer: Compression. The central defining element of UMAC, present in all revisions, is its use of the NH hash function for initial message compression [10]. The NH

algorithm RPHash32_64_128($\mathbf{k}, M$)
// *Input:* $\mathbf{k} = (k_1, k_2, k_3)$ *with* $k_1 \in K_{32}$, $k_2 \in K_{64}$ *and* $k_3 \in K_{128}$*; and* $M \in (\{0,1\}^{32})^+$.
// *Output:* $Y \in \{0,1\}^{128}$.
if ($|M| \leq 2^{14}$) **then**
    $y \leftarrow \mathsf{Poly}[32, p(32)-1](k_1, M)$ // *Hash in* $\mathbb{Z}_{p(32)}$
else if ($|M| \leq 2^{36}$) **then**
    $M_1 \leftarrow M[1 \ldots 2^{14}]$; $M_2 \leftarrow M[2^{14}+1 \ldots |M|]$
    $M_2 \leftarrow \mathsf{padonezero}(M_2, 64)$
    $y \leftarrow \mathsf{Poly}[32, p(32)-1](k_1, M_1)$ // *Hash in* $\mathbb{Z}_{p(32)}$
    $y \leftarrow \mathsf{Poly}[64, 2^{64}-2^{32}](k_2, \mathsf{num2str}(y, 64) \parallel M_2)$ // *Hash in* $\mathbb{Z}_{p(64)}$
else if ($|M| \leq 2^{61}$) **then**
    $M_1 \leftarrow M[1 \ldots 2^{14}]$; $M_2 \leftarrow M[2^{14}+1 \ldots 2^{36}]$; $M_3 \leftarrow M[2^{36}+1 \ldots |M|]$
    $M_3 \leftarrow \mathsf{padonezero}(M_3, 128)$
    $y \leftarrow \mathsf{Poly}[32, p(32)-1](k_1, M_1)$ // *Hash in* $\mathbb{Z}_{p(32)}$
    $y \leftarrow \mathsf{Poly}[64, 2^{64}-2^{32}](k_2, \mathsf{num2str}(y, 64) \parallel M_2)$ // *Hash in* $\mathbb{Z}_{p(64)}$
    $y \leftarrow \mathsf{Poly}[128, 2^{128}-2^{96}](k_3, \mathsf{num2str}(y, 128) \parallel M_3)$ // *Hash in* $\mathbb{Z}_{p(128)}$
else
    return Error // *Message too long*
$Y \leftarrow \mathsf{num2str}(y, 128)$
return Y

Figure 6.5: *The* RPHash32_64_128 *algorithm, combines* Poly32, Poly64 *and* Poly128 *into a hash function which is fast on short messages but also performs well on longer ones.*

hash function is extremely fast and easily accelerated using the type of SIMD parallelism available through Intel's MMX and Motorola's AltiVec instruction sets. The NH hash is unsuitable for message authentication on it own, however, because it requires its key to be at least as long as the message being hashed. Instead, in NHHash16 (Figure 6.3), we break the message to be hashed into 2KB chunks and hash each of them using the same 2KB key. The outputs from each of these NH hashes are augmented with some length information and concatenated into a string which is output by NHHash16. Because each 2KB of input message yields 32 bits of output, NHHash16 compresses long messages by about 512-to-1. We know from [10] that given two distinct, same-length messages M_1 and M_2 and a randomly chosen 2KB key K, that the chance that $\mathsf{NHHash16}(K, M_1)$ and $\mathsf{NHHash16}(K, M_2)$ are equal is no more than 2^{-15} (using signed NH, referred to as NHS in Chapter 2). In Section 6.5 we show that the length annotation in NHHash16 allows M_1 and M_2 to differ

algorithm IPHash16($\mathbf{k}, T, M$)
// *Input:* $\mathbf{k} = (k_1, \ldots, k_8)$ *with each element from* $\mathbb{Z}_{p(19)}$, $T \in \{0,1\}^{19}$ *and* $M \in \{0,1\}^{128}$.
// *Output:* $Y \in \{0,1\}^{19}$.
$M_1 \parallel \cdots \parallel M_8 \leftarrow M$, // *Break* M *into* 16*-bit chunks*
where $|M_1| = \cdots = |M_8| = 16$
for $i \leftarrow 1$ **to** 8 **do**
$m_i \leftarrow$ str2num(M_i)
$y \leftarrow m_1 k_1 + \cdots + m_8 k_8$ **mod** $p(19)$ // *Inner-product hash*
$Y \leftarrow$ num2str$(y, 19)$ // *Return* 19*-bit string*
$Y = Y +_{19} T$ // *Offset result by translation-key* T
$Y = Y[4 \ldots 19]$ // *Truncate to 16 bits*
return Y

Figure 6.6: *The* IPHash16 *algorithm. This hash combines a "strongly-universal" inner-product hash function along with a random translation to the domain* $\{0,1\}^{19}$ *followed by a truncation to the domain* $\{0,1\}^{16}$. *The result, as described in Chapter 5, is an "almost-VU" hash function.*

in length while adding no more than 2^{-30} to the collision bound (Theorem 6.4.1).

Second layer: Hash-to-fixed-length. After compression by NHHash16 a string is still left which is proportional in length to the original message. The purpose of RPHash32_64_128 (Figure 6.5), the second layer in UHash16, is to efficiently hash the output of NHHash16 into a fixed-length string using a fixed-length key. This is done using the "ramped-polynomial hash" described in Chapter 4. In our polynomial-based hash, a string made of n words, of length w bits each, is viewed as an n-degree polynomial over a prime field with each word of the input string serving as a coefficient. To compute the hash, one evaluates the polynomial for a randomly chosen point (the key). Figure 6.4 shows an optimized and parameterized polynomial hash, Poly, which evaluates the polynomial using Horner's rule [22]. The RPHash32_64_128 hash was designed to add little (around 2^{-19}) to the approximately 2^{-15} collision probability created by NHHash16 (Theorem 6.4.3).

Third layer: Strengthen-and-fold. The third layer of hashing in UHash16, performed

```
algorithm UHash16(NHKey, PolyKey, IPKey, IPKeyTrans, M)
// Input: NHKey ∈ {0,1}^(2^14+96),
//        PolyKey = (a_1, ..., a_12), where a_1, a_4, a_7, a_10 ∈ K_32, a_2, a_5, a_8, a_11 ∈ K_64,
//                a_3, a_6, a_9, a_12 ∈ K_128,
//        IPKey = (b_1, ..., b_32), where b_i ∈ Z_p(19) for 1 ≤ i ≤ 32,
//        IPKeyTrans ∈ {0,1}^76, M ∈ {0,1}*.
// Output: Y ∈ {0,1}^64.
for i ← 1 to 4 do
    K ← NHKey[32(i−1)+1 ... 32(i−1)+2^14]   // Extract i-th NHHash16 key
    k1 ← (a_3i−2, ..., a_3i)                // Extract i-th RPHash key
    k2 ← (b_8i−7, ..., b_8i)                // Extract i-th IPHash16 key
    T ← IPKeyTrans[19(i−1)+1 ... 19i]       // Extract i-th IPHash16 translation key
    A_i ← NHHash16(K, M)
    if |M| > 2^14 then                      // If M is longer than K,
        B_i ← RPHash32_64_128(k1, A_i)      // Then A_i must be hashed again
    else
        B_i ← padzero(A_i, 128)             // Otherwise, B_i is an extension of A_i
    C_i ← IPHash16(k2, T, B_i)              // Strengthen-and-fold B_i into C_i
Y ← C_1 ‖ ··· ‖ C_4                         // Concatenate output from 4 iterations
return Y
```

Figure 6.7: *The* UHash16 *algorithm combines* NHHash16, RPHash32_64_128, *and* IPHash16 *in an iterated, three-layer scheme. The* NHHash16 *key is shifted between iterations.*

by IPHash16 (Figure 6.6), serves two main purposes. First, the input to IPHash16 is 128 bits while the output from NHHash16 has only about 2^{-15} collision guarantee. The IPHash16 hash folds its 128-bit input into a 16-bit output while maintaining the nearly 2^{-15} collision guarantee. The second purpose served by IPHash16 is to strengthen the guarantee made of the output bits. Whereas the bits input to IPHash16 have an AU guarantee (addressing collision probabilities), the IPHash16 function employs a strongly-universal inner-product hash, meaning that the bits output from IPHash16 have an ϵ-VU guarantee. This VU guarantee ensures that the output from UHash16 is useful for more than message authentication. See Theorem 6.4.3. See Chapter 5 for the definition and discussion of VU hashing and "generally useful" hash functions. Although the string input to IPHash16 is 128 bits, on all but the longest messages being authenticated, many of the leading bits of the 128

will be zeros. This reduces the amount of work needed to calculate IPHash16.

TYING IT TOGETHER. As we will see in Section 6.4, the 16 bits produced by IPHash16 have a nearly (2^{-15})-VU guarantee. This is good for a 16-bit output, but is not sufficient for most authentication needs. Tag lengths of 32, 64 or 96 bits, with forgery probabilities close to 2^{-30}, 2^{-60} or 2^{-90} are more useful in practice. The function UHash16 (Figure 6.7) beats down the forgery probability by iterating over the three hash-layers four times, each time using a different key. The outputs of the four iterations are concatenated to form a 64-bit output. This output is 4-wise ϵ-VU, where ϵ is about 2^{-15}, which corresponds to about 2^{-58}-SU. See Section 3.1.2 for the definition of t-wise VU. The purpose of the RPHash32_64_128 hash is to handle long outputs from NHHash16. If the message needing authentication is short to begin with, then the RPHash32_64_128 hash is not needed, and it is skipped as an optimization. The next section specifies what security guarantees are provided by this multi-level, iterated hash.

6.4 Security

The UHash16 algorithm does its work by iterating four times over a three-layer hash function. During the i-th iteration, intermediate values A_i, B_i and C_i are produced by the hash function. We are ultimately interested in the properties that can be proven about the concatenation of the third-layer outputs $C_1 \parallel C_2 \parallel C_3 \parallel C_4$. We do this by proving properties about the concatenation of the first- and second-level outputs that are generated along the way. Let UHash16A be defined identically to UHash16 (Figure 6.7) except that its output $Y \in \{0,1\}^*$ is defined to be $A_1 \parallel A_2 \parallel A_3 \parallel A_4$. Let UHash16B be defined identically to UHash16 except that its output $Y \in \{0,1\}^{512}$ is defined to be $B_1 \parallel B_2 \parallel B_3 \parallel B_4$. Finally,

for $1 \leq i \leq 4$, let $\mathsf{UHash16C}_i$ be defined identically to UHash16 except that its output $Y \in \{0,1\}^{16i}$ is defined to be $C_1 \,\|\, \cdots \,\|\, C_i$. The following series of theorem statements regarding these contrived hash functions, justified in Section 6.5, prove that UHash16 is almost-strongly universal and therefore appropriate for a Wegman-Carter MAC.

NOTE REGARDING NH VARIANT. The NH hash family has many variations determined by things like word-length, key-length, sign-interpretation, stride and endian-interpretation. See Chapter 2 for a thorough discussion of these parameters. In the following proofs we fix on the use of NHS, a variant discussed in Section 2.4.4 which interprets its inputs as signed integers. We choose NHS for several reasons: it has the greatest negative impact on our security results and it is the variant chosen for UMAC16 in Appendix A.

Theorem 6.4.1 *Hash family* NHHash16 *is* (2^{-15})*-AU.*

Theorem 6.4.2 *Hash family* UHash16A *is* 4*-wise* (2^{-15})*-AU.*

Theorem 6.4.3 *Hash family* UHash16B *is* 4*-wise* $(2^{-15} + 2^{-19} + 2^{-28})$*-AU.*

Theorem 6.4.4 *Hash family* UHash16 *is* 4*-wise* $(2^{-15} + 2^{-18} + 2^{-28})$*-VU.*

Theorem 6.4.5 *Let* $i \in \{1, \ldots, 4\}$*. Hash family* $\mathsf{UHash16C}_i$ *is* $(3 \cdot 2^{-16} + 2^{-18} + 2^{-28})^i$*-SU.*

This last claim is the one we are after for selective-assurance verifiability. It implies that the verifier of a tag can calculate any number of quarters of $C_1 \,\|\, C_2 \,\|\, C_3 \,\|\, C_4$ and have an ϵ-SU assurance on that part that he calculates where ϵ scales with the number of quarters.

These theorems address only the case when signed NHS is used. If NHHash16 had instead been defined using unsigned NH, then NHHash16 would be $(2^{-16} + 2^{-30})$-AU and UHash16 would be 4-wise $(2^{-16} + 2^{-18} + 2^{-28})$-VU.

6.4.1 UHash32 Differences

The primary difference between UHash32 and UHash16 is that UHash32 uses NHS based on 32-bit words and iterates over the three-layer scheme only twice. These changes mandate the use of bigger prime fields in the polynomial hash and inner-product hash, but the analysis is essentially the same as that of UHash16. Figures 6.8–6.11 depict UHash32.

We give below theorems for UHash32 mirroring Theorems 6.4.4 and 6.4.5. Because the proofs would be nearly identical to those given for UHash16, we omit them. For $1 \leq i \leq 2$, let $\mathsf{UHash32C}_i$ be defined identically to UHash32 except that its output $Y \in \{0,1\}^{32i}$ is defined to be $C_1 \parallel \cdots \parallel C_i$.

Theorem 6.4.6 *Hash family* UHash32 *is* 2*-wise* $(2^{-31} + 2^{-33})$*-VU.*

Theorem 6.4.7 *Let* $i \in \{1, 2\}$. *Hash family* $\mathsf{UHash32C}_i$ *is* $(3 \cdot 2^{-32} + 2^{-33})^i$*-SU.*

6.5 Proofs

In each of the following proofs, we consider the hash-evaluation of messages M and M' by the algorithm depicted in Figure 6.7. When discussing the hash of M', we will attach a "prime" notation when referring to variables used in its evaluation. For example, the output when hashing M' with UHash16B will be referred to as $B'_1 \parallel \cdots \parallel B'_4$ while the output when hashing M will be $B_1 \parallel \cdots \parallel B_4$. All probabilities are over the choice of keys used in the hash unless explicitly noted otherwise. Although UHash16 is defined over $\{0,1\}^*$, security degrades too much when messages are extremely long. We assume throughout that messages are never longer than 2^{64} bits long.

6.5.1 Proof of Theorem 6.4.1

Theorem 6.4.1 *Hash family* NHHash16 *is* (2^{-15})*-AU.*

Proof: Recall from Chapter 2 that $\mathsf{NHS}[n,w]$ is 2^{-w+1}-AΔU over same-length messages from the domain $\{0,1\}^{2w} \cup \{0,1\}^{4w} \cup \cdots \cup \{0,1\}^{nw}$, and is defined as

$$\mathsf{NHS}[n,w](K,M) = (M_1 +_w K_1) \times_{2w} (M_2 +_w K_2) +_{2w} \cdots +_{2w} (M_{\ell-1} +_w K_{\ell-1}) \times_{2w} (M_\ell +_w K_\ell)\,,$$

where $M_1 \| \cdots \| M_\ell = M$, $K_1 \| \cdots \| K_\ell$ is a prefix of K, the length $|K|$ is nw bits and $|K_i| = |M_i| = w$ for all $1 \le i \le \ell$. We use NHS as a primitive to define $\mathsf{NHL}[n,w]$, an almost-universal hash function over all strings no longer than nw bits and requiring a key K of length nw. We define

$$\mathsf{NHL}[n,w](K,M) = \mathsf{NHS}[n,w](K, \mathsf{padzero}(M,2w)) +_{2w} \mathsf{num2str}(|M|, 2w)\,,$$

where padzero(M,$2w$) returns the string $M \| 0^m$ and m is the smallest positive number so that the length of $M \| 0^m$ is divisible by $2w$. To show that NHHash16 is (2^{-15})-AU over arbitrary strings, it is sufficient to show that $\mathsf{NHL}[n,w]$ is (2^{-w+1})-AU over arbitrary non-empty strings no longer than nw bits.

Let n and w be numbers such that $nw < 2^{2w}$. Let M and M' be distinct non-empty strings no longer than nw bits. We wish to show that the probability of $\mathsf{NHL}[n,w](K,M)$ and $\mathsf{NHL}[n,w](K,M')$ being equal, when $K \in \{0,1\}^{nw}$ is chosen uniformly, is no more than 2^{-w+1}. We do so by looking at three cases: when M and M' are the same length, nearly the same length, or very different lengths.

Equal length messages. If $|M| = |M'|$, then $\mathsf{NHL}(K,M)$ and $\mathsf{NHL}(K,M')$ are equal only when $\mathsf{NHS}(K, \mathsf{padzero}(M,2w))$ and $\mathsf{NHS}(K, \mathsf{padzero}(M',2w))$ are equal. We know from Chapter 2 that the collision of distinct, same-length messages under NHS happens with no

more than 2^{-w+1} probability. If using unsigned NH, this collision probability would be decreased to 2^{-w}.

Almost equal length messages. If M and M' differ in length, but $\mathsf{padzero}(M, 2w)$ and $\mathsf{padzero}(M', 2w)$ are the same length, then $\mathsf{NHL}(K, M)$ and $\mathsf{NHL}(K, M')$ are equal only when $\mathsf{NHS}(K, \mathsf{padzero}(M, 2w)) -_{2w} \mathsf{NHS}(K, \mathsf{padzero}(M', 2w)) = \mathsf{num2str}(|M'| - |M|, 2w)$. If $\mathsf{padzero}(M, 2w)$ and $\mathsf{padzero}(M', 2w)$ are identical (because one message is a prefix of the other), then $\mathsf{NHS}(K, \mathsf{padzero}(M, 2w)) -_{2w} \mathsf{NHS}(K, \mathsf{padzero}(M', 2w)) = 0$ while $|M'| - |M| \neq 0$. Otherwise, $\mathsf{padzero}(M, 2w)$ and $\mathsf{padzero}(M', 2w)$ are distinct, same-length messages. That $\mathsf{NHS}[n, w]$ is 2^{-w+1}-AΔU means that for any constant $c \in \{0,1\}^{2w}$ and distinct, same-length messages M and M', the probability that $\mathsf{NHS}[n, w](K, M) -_{2w} \mathsf{NHS}[n, w](K, M') = c$ is no more than 2^{-w+1} when $K \in \{0,1\}^{nw}$ is chosen uniformly. Thus, the probability that $\mathsf{NHS}(K, \mathsf{padzero}(M, 2w)) -_{2w} \mathsf{NHS}(K, \mathsf{padzero}(M', 2w))$ is equal to $\mathsf{num2str}(|M'| - |M|, 2w)$ is no more than 2^{-w+1}. Again, if using unsigned NH, this collision probability would be decreased to 2^{-w}.

Different length messages. In this case, $\mathsf{padzero}(M, 2w)$ and $\mathsf{padzero}(M', 2w)$ differ in length. First, we state two lemmas which we will need.

Lemma 6.5.1 *Let X be an arbitrary $2w$-bit string, and let S_1 and S_2 be arbitrary w-bit strings. For randomly chosen w-bit strings K_1 and K_2, $\Pr_{K_1,K_2}[(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) = X]$ is no more than 2^{-w} when $X \neq 0$ and no more than 2^{-w+1} when $X = 0$.*

Proof: If $X = 0$ then $(S_1 +_w K) \times_{2w} (S_2 +_w K) = X$ only if $(S_1 +_w K_1) = 0$ or $(S_2 +_w K_1) = 0$. The chance that at least one of the two terms is zero is no more than 2^{-w+1}. In the case that $X \neq 0$, if we allow K_1 to be arbitrarily fixed and $(S_1 +_w K_1) = 0$ then there is no

K_2 which achieves $(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) = X$. On the other hand, if K_1 is fixed so that $(S_1 +_w K_1) \neq 0$, then by Lemma 2.4.6 there is at most one value for K_2 which causes $(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) = X$. ∎

Lemma 6.5.2 *Let X be an arbitrary $2w$-bit string, and let $S_1, \ldots, S_4$ be arbitrary w-bit strings. For randomly chosen w-bit strings $K_1, \ldots, K_4$, $\Pr_{K_1,\ldots,K_4}[(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) +_{2w} (S_3 +_w K_3) \times_{2w} (S_4 +_w K_4) = X]$ is no more than $2^{-w} + 2^{-2w+2}$.*

Proof: Let Q be the event that $(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) +_{2w} (S_3 +_w K_3) \times_{2w} (S_4 +_w K_4) = X$, and let R be the event that $(S_1 +_w K_1) \times_{2w} (S_2 +_w K_2) = X$. Then, we can compute the probability as

$$\Pr[Q] = \Pr[Q|R] \cdot \Pr[R] + \Pr[Q|\overline{R}] \cdot \Pr[\overline{R}] \; .$$

But, according to Lemma 6.5.1 $\Pr[Q|R]$ and $\Pr[R]$ are each no more than 2^{-w+1}, and so is $\Pr[Q|\overline{R}]$ no more than 2^{-w}. The final term, $\Pr[\overline{R}]$ is trivially bounded by 1, so we conclude that $\Pr[Q] \leq 2^{-w} + 2^{-2w+2}$. ∎

We continue with our proof. If $\mathsf{padzero}(M, 2w)$ and $\mathsf{padzero}(M', 2w)$ differ in length, then they differ by at least $2w$ bits. To simplify exposition without loss of generality, we consider M to be longer than M' and each to be a multiple of $2w$ bits long. We break M and K into pieces. Let $K_\mathrm{f} \parallel K_\mathrm{b} = K$, let $M_\mathrm{f} \parallel M_\mathrm{b} = M$, and let $|K_\mathrm{f}| = |M_\mathrm{f}| = |M'|$. This new notation allows us to rewrite the collision event as $\mathsf{NHS}(K_\mathrm{f}, M_\mathrm{f}) +_{2w} \mathsf{NHS}(K_\mathrm{b}, M_\mathrm{b}) +_{2w} |M| = \mathsf{NHS}(K_\mathrm{f}, M') +_{2w} |M'|$, or $\mathsf{NHS}(K_\mathrm{b}, M_\mathrm{b}) = \mathsf{NHS}(K_\mathrm{f}, M') -_{2w} \mathsf{NHS}(K_\mathrm{f}, M_\mathrm{f}) +_{2w} |M'| -_{2w} |M|$. If M_b is at least $4w$ bits long, then we can consider K_f to be fixed and can apply Lemma 6.5.2. In this case, the collision event happens with no more than $2^{-w} + 2^{-2w+2}$ probability. If M_b is exactly $2w$ bits long, then we will apply Lemma 6.5.1. Let CollB be the event

$\mathsf{NHS}(K_\mathrm{b}, M_\mathrm{b}) = \mathsf{NHS}(K_\mathrm{f}, M') -_{2w} \mathsf{NHS}(K_\mathrm{f}, M_\mathrm{f}) +_{2w} |M'| -_{2w} |M|$, and let CollF be the event $\mathsf{NHS}(K_\mathrm{f}, M') -_{2w} \mathsf{NHS}(K_\mathrm{f}, M_\mathrm{f}) +_{2w} |M'| -_{2w} |M| = 0$. The event CollF happens only if $\mathsf{NHS}(K_\mathrm{f}, M') -_{2w} \mathsf{NHS}(K_\mathrm{f}, M_\mathrm{f})$ equals a fixed constant, an event that is bounded by the fact that NHS is 2^{-w+1}-AΔU. We can bound the probability of CollB, using this 2^{-w+1}-AΔU bound and Lemma 6.5.1, as

$$\begin{aligned}\Pr[\mathsf{CollB}] &= \Pr[\mathsf{CollB}|\mathsf{CollF}] \cdot \Pr[\mathsf{CollF}] + \Pr[\mathsf{CollB}|\overline{\mathsf{CollF}}] \cdot \Pr[\overline{\mathsf{CollF}}] \\ &\leq (2^{-w+1}) \cdot (2^{-w+1}) + (2^{-w}) \cdot (1) \leq 2^{-w} + 2^{-2w+2} .\end{aligned}$$

If using unsigned NH, then the event CollF would have only 2^{-w} probability, and so overall the collision probability in this case would be decreased to $2^{-w} + 2^{-2w+1}$.

Putting it together. The collision probability for $\mathsf{NHL}[n, w]$ is maximized in the first two cases examined above and is thus 2^{-w+1}. This implies that $\mathsf{NHHash16}$ is (2^{-15})-AU. If we were instead using unsigned NH, then the collision probability is maximized in the third case examined above, and would thus decrease to $2^{-w} + 2^{-2w+1}$. ∎

6.5.2 Proof of Theorem 6.4.2

Theorem 6.4.2 *Hash family* $\mathsf{UHash16A}$ *is* 4*-wise* (2^{-15})*-AU.*

Proof: Let M and M' be arbitrary-length, distinct messages, and let $1 \leq i \leq 4$ be any number. To show $\mathsf{UHash16A}$ is 4-wise (2^{-15})-AU, we need only show $\Pr[A_i = A'_i] \leq 2^{-15}$ regardless of the values of $A_1 \parallel \cdots \parallel A_{i-1}$ and $A'_1 \parallel \cdots \parallel A'_{i-1}$. It suffices to show that NHL, and therefore $\mathsf{NHHash16}$, is (2^{-15})-AU in each iteration within $\mathsf{UHash16A}$.

The proof of Theorem 6.4.1 applies for the i-th iteration just as well as the first. The only difference between successive invocations of $\mathsf{NHHash16}$ is the use of a Toeplitz-shifted key, and as proven in Chapter 2, all claims regarding NHS remain true across Toeplitz

shifts. Additionally, all non-NHS claims made in the proof of Theorem 6.4.1 involve only independent key material, unused in the calculation of $A_1 \parallel \cdots \parallel A_{i-1}$. ∎

6.5.3 Proof of Theorem 6.4.3

Theorem 6.4.3 *Hash family* UHash16B *is* 4*-wise* $(2^{-15} + 2^{-19} + 2^{-28})$*-AU.*

Proof: The hash function UHash16B is identical to UHash16 except that instead of outputting the value $C_1 \parallel \cdots \parallel C_4$, it outputs $B_1 \parallel \cdots \parallel B_4$. But $B_1 \parallel \cdots \parallel B_4$ is defined in two different ways, depending on the length of message being hashed. If message M is longer than 2^{14} bits (the length of the first-layer NHS key), then M is first hashed by NHHash16, and then the NHHash16 output is hashed by a second-layer polynomial hash to produce the B_i values. If M is not longer than 2^{14} bits, then the polynomial hash step is skipped as an optimization and the B_i values are defined as the zero-padded output of the first-layer NHHash16 hash. We must therefore consider three length-based cases in this proof.

Let M and M' be arbitrary-length, distinct messages, and let $1 \leq i \leq 4$ be any number. To show UHash16B is 4-wise $(2^{-15} + 2^{-19} + 2^{-28})$-AU, we need only show $\Pr[B_i = B'_i] \leq 2^{-15} + 2^{-19} + 2^{-28}$ regardless of the values of $B_1 \parallel \cdots \parallel B_{i-1}$ and $B'_1 \parallel \cdots \parallel B'_{i-1}$.

If M and M' are both no longer than 2^{14} bits, then B_i and B'_i collide only if A_i and A'_i do so. But, from Theorem 6.4.2 we know that the probability of this is no more than 2^{-15}.

If M and M' are both longer than 2^{14} bits, then B_i and B'_i collide if A_i and A'_i collide or if there is a collision in the polynomial hash given no collision of A_i and A'_i. Let CollA_i be the event that $A_i = A'_i$, and let CollB_i be the event that $B_i = B'_i$. Then $\Pr[\mathsf{CollB}_i]$ can be rewritten as

$$\Pr[\mathsf{CollB}_i] \;=\; \Pr[\mathsf{CollB}_i \mid \mathsf{CollA}_i] \cdot \Pr[\mathsf{CollA}_i] + \Pr[\mathsf{CollB}_i \mid \overline{\mathsf{CollA}_i}] \cdot \Pr[\overline{\mathsf{CollA}_i}]$$

$$\leq \quad 1 \cdot (2^{-15}) + \Pr[\mathsf{CollB}_i \mid \overline{\mathsf{CollA}_i}] \cdot 1$$

$$\leq \quad (2^{-15}) + (2^{-19} + 2^{-29} + 2^{-50} + 2^{-75}) \leq 2^{-15} + 2^{-19} + 2^{-28} \, .$$

The last line of the derivation is a direct application of Proposition 4.6.2, which bounds the collision probability for RPHash32_64_128.

The final length-based case is when one message M is longer than 2^{14} bits while the other message M' is not. In this case a collision occurs if B_i collides with $\mathsf{padzero}(A_i, 128)$. If we consider *NHKey* to be fixed, then $\mathsf{padzero}(A_i, 128)$ becomes a fixed quantity, and B_i will only collide with it if $\mathsf{RPHash32\_64\_128}(\mathbf{k1}, A_i)$ hashes to a constant. Claim 4.4.5, which bounds the probability that RPHash32_64_128 hashes to a fixed value, tells us that this is no more likely than 2^{-19}.

Over the three length-based cases, the one in which both messages are longer than 2^{-14} yields the highest collision probability. Thus $\Pr[B_i = B'_i] \leq 2^{-15} + 2^{-19} + 2^{-28}$. ∎

6.5.4 Proof of Theorem 6.4.4

Theorem 6.4.4 *Hash family* UHash16 *is* 4*-wise* $(2^{-15} + 2^{-18} + 2^{-28})$*-VU.*

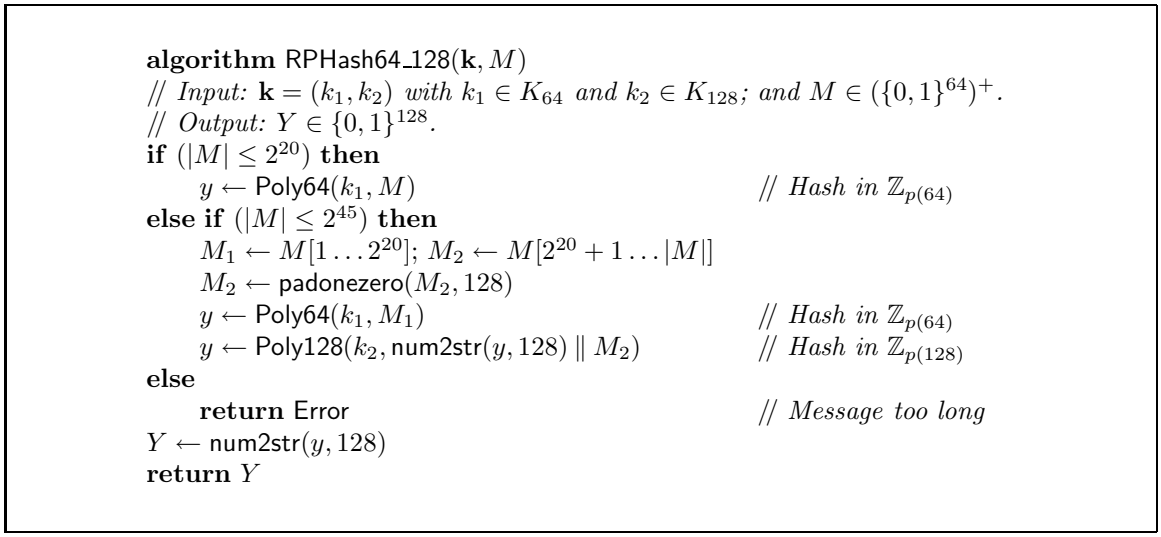
Proof: Each value C_i is produced by an $(2^{-15} + 2^{-19} + 2^{-28})$-AU hash function, followed by an SU inner-product hash function with range $\mathbb{Z}_{2^{19}-1}$, followed by a translation into $\{0,1\}^{19}$, followed by a truncation to 16 bits. Chapter 5 investigates the construction of almost-VU hash functions using these steps. Theorems 5.3.1–5.3.3 indicate that the result is ϵ-VU where $\epsilon = 2^{-15} + 2^{-18} + 2^{-28}$. Furthermore, because UHash16B is 4-wise AU and all of the key-material used in IPHash16 is independent between iterations, UHash16 is guaranteed to be 4-wise ϵ-VU. ∎

6.5.5 Proof of Theorem 6.4.5

Theorem 6.4.5 *Let $i \in \{1, \ldots, 4\}$. Hash family* $\mathsf{UHash16C}_i$ *is* $(3 \cdot 2^{-16} + 2^{-18} + 2^{-28})^i$*-SU.*

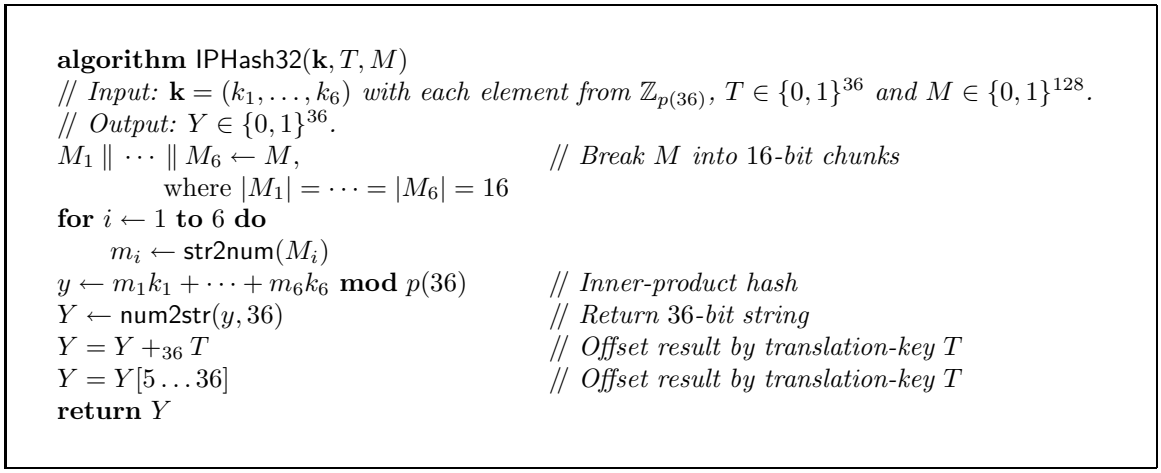
Proof: Let M and M' be arbitrary-length distinct messages, let $1 \leq i \leq 4$ be any number, and let $c, c' \in \{0,1\}^{16i}$ be arbitrarily chosen. We know that $\mathsf{UHash16C}_i$ is i-wise almost-VU because $\mathsf{UHash16}$ is 4-wise almost-VU. Due to the definition of almost-VU, this means that $\mathsf{UHash16C}_i(k, M)$ is uniformly distributed when k is chosen randomly. So, to show that $\mathsf{UHash16C}_i$ is $(3 \cdot 2^{-16} + 2^{-18} + 2^{-28})^i$-SU, we need only show that $\Pr_k[\mathsf{UHash16C}_i(k, M) = c \mid \mathsf{UHash16C}_i(k, M') = c']$ is no more than $(3 \cdot 2^{-16} + 2^{-18} + 2^{-28})^i$.

This is a natural result of $\mathsf{UHash16C}_i$ being i-wise $(2^{-15} + 2^{-18} + 2^{-28})$-VU. Let c and c' be split into 16-bit chunks, $c_1 \parallel \cdots \parallel c_i = c$ and $c'_1 \parallel \cdots \parallel c'_i = c'$. The definition of almost-VU tells us that $\Pr[C_1 = c_1 \mid C'_1 = c'_1] - 2^{-16}$ is no more than $2^{-15} + 2^{-18} + 2^{-28}$. Which means $\Pr[C_1 = c_1 \mid C'_1 = c'_1] \leq 3 \cdot 2^{-16} + 2^{-18} + 2^{-28}$. Likewise, by the definition of i-wise almost-VU, $\Pr[C_2 = c_2 \mid C'_2 = c'_2 \wedge C_1 = c_1 \wedge C'_1 = c'_1] - 2^{-16}$ is no more than $2^{-15} + 2^{-18} + 2^{-28}$. This continues i times until we have $\Pr_k[\mathsf{UHash16C}_i(k, M) = c \mid \mathsf{UHash16C}_i(k, M') = c']$ is no more than $(3 \cdot 2^{-16} + 2^{-18} + 2^{-28})^i$. ∎



algorithm RPHash64_128$(\mathbf{k}, M)$
// *Input:* $\mathbf{k} = (k_1, k_2)$ *with* $k_1 \in K_{64}$ *and* $k_2 \in K_{128}$*; and* $M \in (\{0,1\}^{64})^+$.
// *Output:* $Y \in \{0,1\}^{128}$.
if $(|M| \leq 2^{20})$ **then**
    $y \leftarrow$ Poly64(k_1, M) // *Hash in* $\mathbb{Z}_{p(64)}$
else if $(|M| \leq 2^{45})$ **then**
    $M_1 \leftarrow M[1 \ldots 2^{20}]$; $M_2 \leftarrow M[2^{20}+1 \ldots |M|]$
    $M_2 \leftarrow$ padonezero$(M_2, 128)$
    $y \leftarrow$ Poly64(k_1, M_1) // *Hash in* $\mathbb{Z}_{p(64)}$
    $y \leftarrow$ Poly128$(k_2,$ num2str$(y, 128) \,\|\, M_2)$ // *Hash in* $\mathbb{Z}_{p(128)}$
else
    return Error // *Message too long*
$Y \leftarrow$ num2str$(y, 128)$
return Y

Figure 6.8: *The* RPHash64_128 *algorithm. Combining the* Poly64 *and* Poly128 *hashes into a hash function which is fast on short messages but also performs well on long ones.* RPHash64_128 *also extends the domain to messages which are not a multiple of the constituent hashes word-lengths.*



algorithm IPHash32$(\mathbf{k}, T, M)$
// *Input:* $\mathbf{k} = (k_1, \ldots, k_6)$ *with each element from* $\mathbb{Z}_{p(36)}$, $T \in \{0,1\}^{36}$ *and* $M \in \{0,1\}^{128}$.
// *Output:* $Y \in \{0,1\}^{36}$.
$M_1 \,\|\, \cdots \,\|\, M_6 \leftarrow M$, // *Break* M *into* 16*-bit chunks*
    where $|M_1| = \cdots = |M_6| = 16$
for $i \leftarrow 1$ **to** 6 **do**
    $m_i \leftarrow$ str2num(M_i)
$y \leftarrow m_1 k_1 + \cdots + m_6 k_6$ **mod** $p(36)$ // *Inner-product hash*
$Y \leftarrow$ num2str$(y, 36)$ // *Return* 36*-bit string*
$Y = Y +_{36} T$ // *Offset result by translation-key* T
$Y = Y[5 \ldots 36]$ // *Offset result by translation-key* T
return Y

Figure 6.9: *The* IPHash32 *algorithm.*

algorithm NHHash32(K, M)
// *Input:* $K \in \{0,1\}^{2^{12}}$ *and* $M \in \{0,1\}^*$.
// *Output:* $Y \in (\{0,1\}^{64})^+$.
$Y \leftarrow \varepsilon$ // *emptystring*
$m \leftarrow \lceil |M|/2^{12} \rceil$ // *Number of hash-blocks*
$M_1 \parallel \ldots \parallel M_m \leftarrow M$, // *Break* M *in* 2^{12}*-bit chunks*
where $|M_1| = \cdots = |M_{m-1}| = 2^{12}$
for $i \leftarrow 1$ **to** m **do** // *For each hash-block*
$Len \leftarrow$ num2str$(|M_i|, 64)$ // *Length of* $|M_i|$ *before any padding*
$M_i \leftarrow$ padzero$(M_i, 64)$ // *A no-op if* $|M_i|$ *divisible by* 64
$Y \leftarrow Y \parallel (\mathsf{NH}[2^7, 32](K, M_i) +_{64} Len)$ // *Hash, add length and append to* Y
return Y

Figure 6.10: *The* NHHash32 *algorithm.*

algorithm UHash32$(NHKey, PolyKey, IPKey, IPKeyTrans, M)$
// *Input:* $NHKey \in \{0,1\}^{2^{12}+64}$,
// $PolyKey = (a_1, \ldots, a_4)$, where $a_1, a_3 \in K_{64}$, $a_2, a_4 \in K_{128}$,
// $IPKey = (b_1, \ldots, b_7)$, where $b_i \in \mathbb{Z}_{p(36)}$ for $1 \le i \le 7$,
// $IPKeyTrans \in \{0,1\}^{72}$, $M \in \{0,1\}^*$.
// *Output:* $Y \in \{0,1\}^{64}$.
for $i \leftarrow 1$ **to** 2 **do**
$K \leftarrow NHKey[64(i-1)+1 \ldots 64(i-1)+2^{12}]$ // *Extract* i*-th* NHHash16 *key*
k1 $\leftarrow (a_{2i-1}, \ldots, a_{2i})$ // *Extract* i*-th* RPHash *key*
k2 $\leftarrow (b_i, \ldots, b_{i+5})$ // *Extract* i*-th* IPHash16 *key*
$T \leftarrow IPKeyTrans[36(i-1)+1 \ldots 36i]$ // *Extract* i*-th* IPHash16 *translation key*
$A \leftarrow$ NHHash32(K, M)
if $|M| \le 2^{12}$ **then**
$B \leftarrow$ RPHash64_128(**k1**, A)
else
$B \leftarrow$ padzero$(A, 128)$
$C_i \leftarrow$ IPHash32(**k2**, T, B)
$Y \leftarrow C_1 \parallel C_2$
return Y

Figure 6.11: *The* UHash32 *algorithm.*

Bibliography

[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., Guérin, R., and Rogaway, P. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology – CRYPTO '95* (1995), vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–28.

[5] Bellare, M., Kilian, J., and Rogaway, P. The security of the cipher block chaining message authentication code. In *Advances in Cryptology – CRYPTO '94* (1994), vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 341–358.

[6] Bellare, M., Pointcheval, D., and Rogaway, P. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology – Eurocrypt '00* (2000), vol. 1807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 139–155.

[7] Bellare, M., and Rogaway, P. Entity authentication and key distribution. In *Advances in Cryptology – CRYPTO '93* (1994), vol. 773 of *Lecture Notes in Computer Science*, Springer-Verlag.

[8] Bellovin, S., and Merritt, M. Encrypted key exchange: Password-based protocols secure against dictionary dttacks. In *Proc. of the Symposium on Security and Privacy* (1992), IEEE, pp. 72–84.

[9] Bernstein, D. Floating-point arithmetic and message authentication. Unpublished manuscript, `http://cr.yp.to/papers.html`, 2000.

[10] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–233.

[11] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–233.

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

[13] Bosselaers, A., Govaerts, R., and Vandewalle, J. Fast hashing on the Pentium. In *Advances in Cryptology – CRYPTO '96* (1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–312. Updated timing at `http://www.esat.kuleuven.ac.be/ bosselae/fast.html`.

[14] Carter, L., and Wegman, M. Universal classes of hash functions. *J. of Computer and System Sciences 18* (1979), 143–154.

[15] Dobbertin, H. The status of MD5 after a recent attack, 1996. Technical newsletter of RSA Laboratories.

[16] Etzel, M., Patel, S., and Ramzan, Z. Square hash: Fast message authentication via optimized universal hash functions. In *Advances in Cryptology – CRYPTO '99* (1999), vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 234–251.

[17] FIPS 180-1. Secure hash standard. NIST, US Dept. of Commerce, 1995.

[18] FIPS 46. Data encryption standard. NIST, US Dept. of Commerce, 1977.

[19] Goldreich, O., Goldwasser, S., and Micali, S. How to construct random functions. *Journal of the ACM 33*, 4 (1986), 210–217.

[20] H. Krawczyk, M. B., and Canetti, R. HMAC: Keyed hashing for message authentication. IETF RFC-2104, 1997.

[21] 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.

[22] Harrison, P., and While, R. Transformation of polynomial evaluation to a pipeline via horner's rule. *Science of Computer Programming 24*, 1 (1995), 83–95.

[23] Johansson, T. Bucket hashing with small key size. In *Advances in Cryptology – EUROCRYPT '97* (1997), Lecture Notes in Computer Science, Springer-Verlag.

[24] Kaliski, B., and Robshaw, M. Message authentication with MD5, 1995. Technical newsletter of RSA Laboratories.

[25] 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.

[26] Krovetz, T., and Rogaway, P. Fast universal hashing with small keys and no preprocessing. In preparation, 2000.

[27] Krovetz, T., and Rogaway, P. Variationally universal hashing. In preparation, 2000.

[28] 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.

[29] 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.

[30] Petrank, E., and Rackoff, C. CBC MAC for real-time data sources. Manuscript 97-10 in `http://philby.ucsd.edu/cryptolib.html`, 1997.

[31] 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.

[32] 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.

[33] RFC 1321. The MD5 message digest algorithm. IETF RFC-1321, R. Rivest, 1992.

[34] Rivest, R., Robshaw, M., Sidney, R., and Yin, Y. The RC6 block cipher. Available from `http://theory.lcs.mit.edu/~rivest/publications.html`, 1998.

[35] 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.

[36] 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. 313–328.

[37] Stinson, D. Universal hashing and authentication codes. *Designs, Codes and Cryptography 4* (1994), 369–380.

[38] Tsudik, G. Message authentication with one-way hash functions. *Computer Communications Review 22* (1992), 29–38.

[39] Wegman, M., and Carter, L. New hash functions and their use in authentication and set equality. *J. of Computer and System Sciences 22* (1981), 265–279.

Appendix A

UMAC (2000) Specification

One goal from the very beginning of the UMAC project has been to fully specify an authentication code that was provably secure and provably fast. This appendix provides that specification as a working-draft to be submitted to the IETF. Once submitted, we will entertain comments from the security community. After that, a revised version of this document will be submitted as an RFC with the goal of standardization. The page numbers of the specification have been altered to match those of the dissertation.

Network Working Group
INTERNET-DRAFT
Expires December 2000

T. Krovetz, U.C. Davis
J. Black, U.C. Davis
S. Halevi, IBM
A. Hevia, U.C. San Diego
H. Krawczyk, Technion
P. Rogaway, U.C. Davis
July 2000

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

Status of this Memo

Abstract

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

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

Table of Contents

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1 Organization

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

2 Named parameter sets: UMAC16 and UMAC32

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

2.1 Named parameters

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

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

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

WORD-LEN: Specifies the size in bytes of a "word". UMAC will be significantly faster in execution if the executing machine supports well certain operations on datatypes of this size. Note that WORD-LEN is not necessarily the native wordsize of the target machine (and on some machines a smaller value turns out to be preferable).

UMAC-OUTPUT-LEN: Specifies the length of the authentication tag generated by UMAC, in bytes. This is also the length of the output of UHASH.

L1-KEY-LEN: Specifies the "block length," in bytes, on which the hash-function initially operates. This much storage (and then some) will be needed in the run-time environment for UMAC's internal keys.

UMAC-KEY-LEN: Specifies the length in bytes of the user-supplied UMAC key.

ENDIAN-FAVORITE: Specifies which endian-orientation will be followed in the reading of data to be hashed. This need not be equal to the native endianess of any specific machine running UMAC.

L1-OPERATIONS-SIGN: Specifies whether the strings manipulated in the hash-function are to be initially considered as signed or unsigned integers.

2.2 Alternative instantiations

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

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

Roughly speaking, doubling UMAC-OUTPUT-LEN approximately doubles

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

2.3 Naming convention

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

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

Some examples

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

3 Notation and basic operations

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

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

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

3.1 Operations on strings

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

length(S):	The length of string S in bytes.
zeroes(n):	The string made of n zero-bytes.
S xor T:	The string which is the bitwise exclusive-or of S and T. Strings S and T must have the same length.
S and T:	The string which is the bitwise conjunction of S and T. Strings S and T must have the same length.
S[i]:	The i-th byte of the string S (indices begin at 1).
S[i..j]:	The substring of S consisting of bytes i through j.
S || T:	The string S concatenated with string T.
zeropad(S,n):	The string S, padded with zero-bytes to the nearest non-zero multiple of n bytes. Formally, zeropad(S,n) = S || zeroes(i), where i is the smallest nonnegative integer such that S || zeroes(i) is non-empty and n divides length(S)+i.

3.1.1 ENDIAN-SWAP: Adjusting endian orientation

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

3.1.1.1 Discussion

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

3.1.1.2 Interface

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

3.1.1.3 Algorithm

Compute T using the following algorithm.

```
  //
  // Break S into word-size chunks
  //
  n = length(S) / WORD-LEN
  Let S_1, S_2, ..., S_n be strings of length WORD-LEN bytes
     so that S_1 || S_2 || .. || S_n = S.

  //
  // Byte-reverse each chunk, and build-up T
```

```
//
T = <empty string>
for i = 1 to n do
  Let W_1, W_2, ..., W_{WORD-LEN}  be bytes
     so that W_1 || W_2 || ... || W_{WORD-LEN} = S_i
  SReversed_i = W_{WORD-LEN} || W_{WORD-LEN - 1} || ... || W_1
  T = T || SReversed_i

Return T
```

3.2 Operations on integers

In this specification, we generally use standard notation for mathematical operations, such as "*" for multiplication, "+" for addition and "mod" for modular reduction. Some less standard notations are defined here.

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

lg a: The base-2 logarithm of integer a.

floor(x): The largest integer less than or equal to x.

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

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

The prime numbers used in UMAC are:

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

3.3 String-Integer conversion operations

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

bit(S,n): Returns the integer 1 if the n-th bit of the string S is 1, otherwise returns the integer 0 (indices begin at 1). Here n must be between 1 and the bit-length os S.

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

uint2str(n,i): The i-byte string S such that str2uint(S) = n. If no such string exists then uint2str(n,i) is undefined.

str2sint(S): The integer whose binary representation in two's-complement is the string S. More formally, if S is t bits long then str2sint(S) = -2^{t-1} * bit(S,1) + 2^{t-2} * bit(S,2) + ... + 2^{1} * bit(S,t-1) + bit(S,t).

sint2str(n,i): The i-byte string S such that str2sint(S) = n. If no such string exists then sint2str(n,i) is undefined.

3.4 Mathematical operations on strings

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

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

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

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

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

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

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

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

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

4 Key and pad derivation functions

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

4.1 KDF: Key derivation function

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

4.1.1 Interface

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

4.1.2 Algorithm

Compute Y using the following algorithm.

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

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

Y = Y[1..numbytes]

Return Y
```

4.2 PDF: Pad-derivation function

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

4.2.1 Discussion

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

effective only when nonces are sequential. We do so by using the low bits of the nonce as an index into the AES output, which is generated using the higher bits of the nonce which are not used for indexing. This speeds message authentication by reducing the average time spent by AES for each authentication. If UMAC-OUTPUT-LEN is larger than 16, then two AES invocations are required to produce a sufficient number of bits.

4.2.2 Interface

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

4.2.3 Algorithm

Compute Y using the following algorithm.

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

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

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

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

      //
```

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

      else

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

      Return Y
```

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

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

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

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

UHASH does its work in three layers. First, a hash function called NH [3] is used to compress input messages into strings which are typically many times smaller than the input message. Second, the compressed message is hashed with an optimized "polynomial hash function" into a fixed-length 16-byte string. Finally, the 16-byte

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

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

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

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

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

5.1.1 Interface

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

5.1.2 Algorithm

Compute Y using the following algorithm.

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

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

Return Y
```

5.2 L1-HASH-32: First-layer hash

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

5.2.1 Discussion

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

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

5.2.2 Interface

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

5.2.3 Algorithm

Compute Y using the following algorithm.

```
  //
  // Break M into L1-KEY-LEN byte chunks (final chunk may be shorter)
  //
  t = ceil(length(M) / L1-KEY-LEN)
  Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || .. ||
     M_t, and length(M_i) = L1-KEY-LEN for all 0 < i < t.

  //
  // For each chunk, except the last: endian-adjust, NH hash
  // and add bit-length.  Use results to build Y.
  //
  Len = uint2str(L1-KEY-LEN * 8, 8)
  Y = <empty string>
  for i = 1 to t-1 do
    if (ENDIAN-FAVORITE == LITTLE) then  // See endian discussion
       ENDIAN-SWAP(M_i)                  // in section 3.1.1
    Y = Y || (NH-32(K, M_i) +_64 Len)

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

  return Y
```

5.3 POLY: Polynomial hash

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

5.3.1 Discussion

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

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

5.3.2 Interface

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

5.3.3 Algorithm

Compute y using the following algorithm.

```
  //
  // Define constants used for fixing out-of-range words
  //
  wordbytes = wordbits / 8
```

```
p = prime(wordbits)
offset = 2^wordbits - p
marker = p - 1

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

//
// For each input word, compare it with maxwordrange.  If larger
// then hash the words 'marker' and (m - offset), both in range.
//
y = 1
for i = 1 to n do
   m = str2uint(M_i)
   if (m >= maxwordrange) then
      y = (k * y + marker) mod p
      y = (k * y + (m - offset)) mod p
   else
      y = (k * y + m) mod p

Return y
```

5.4 L2-HASH-32: Second-layer hash

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

5.4.1 Discussion

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

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

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

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

For more information see [10].

5.4.2 Interface

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

5.4.3 Algorithm

Compute y using the following algorithm.

```
  //
  //  Extract keys and restrict to special key-sets
  //
  Mask64  = uint2str(0x01ffffff01ffffff, 8)
  Mask128 = uint2str(0x01ffffff01ffffff01ffffff01ffffff, 16)
  k_64    = str2uint(K[1..8]  and Mask64)
  k_128   = str2uint(K[9..24] and Mask128)

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

     //
     // View M as an array of 64-bit words, and use POLY modulo
```

```
        // prime(64) (and with bound 2^64 - 2^32) to hash it.
        //
        y = POLY(64, 2^64 - 2^32,  k_64, M)

     else

        M_1 = M[1 .. 2^17]
        M_2 = M[2^17 + 1 .. length(M)]
        M_2 = zeropad(M || uint2str(0x80,1), 16)
        y = POLY(64, 2^64 - 2^32, k_64, M_1)
        y = POLY(128, 2^128 - 2^96, k_128, uint2str(y, 16) || M_2)

     Y = uint2str(y, 16)

     Return Y
```

5.5 L3-HASH-32: Third-layer hash

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

5.5.1 Interface

```
Function Name:
  L3-HASH-32
Input:
  K1, string of length 64 bytes.
  K2, string of length 4 bytes.
  M, string of length 16 bytes.
Output:
  Y, string of length 32 bytes.
```

5.5.2 Algorithm

Compute Y using the following algorithm.

```
  y = 0

  //
  // Break M and K1 into 8 chunks and convert to integers
  //
  for i = 1 to 8 do
    M_i = M [(i - 1) * 2 + 1 .. i * 2]
    K_i = K1[(i - 1) * 8 + 1 .. i * 8]
```

```
      m_i = str2uint(M_i)
      k_i = str2uint(K_i) mod prime(36)

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

    Return Y
```

5.6 UHASH-32: Three-layer universal hash

The hash functions L1-HASH, L2-HASH and L3-HASH are used together in a straightforward manner. A message is first hashed by L1-HASH, its output is then hashed by L2-HASH, whose output is then hashed by L3-HASH. If the message being hashed is no longer than L1-KEY-LEN bytes, then L2-HASH is skipped as an optimization. Because L3-HASH outputs a string whose length is only WORD-LEN bytes long, multiple iterations of this three-layer hash are used, with different keys each time, until UMAC-OUTPUT-LEN have been generated. To reduce memory requirements, L1-HASH and L3-HASH both reuse most of their key-material between iterations.

5.6.1 Interface

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

5.6.2 Algorithm

Compute Y using the following algorithm.

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

  //
```

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

//
// For each iteration, extract key and three-layer hash.
// If length(M) <= L1-KEY-LEN, then skip L2-HASH.
//
Y = <empty string>
for i = 1 to streams do
  L1Key_i  = L1Key [(i-1) * 16 + 1 .. (i-1) * 16 + L1-KEY-LEN]
  L2Key_i  = L2Key [(i-1) * 24 + 1 .. i * 24]
  L3Key1_i = L3Key1[(i-1) * 64 + 1 .. i * 64]
  L3Key2_i = L3Key2[(i-1) * 4  + 1 .. i * 4]

  A = L1-HASH-32(L1Key_i, M)
  if (length(M) <= L1-KEY-LEN) then
    B = zeroes(8) || A
  else
    B = L2-HASH-32(L2Key_i, A)
  C = L3-HASH-32(L3Key1_i, L3Key2_i, B)
  Y = Y || C
Y = Y[1 .. UMAC-OUTPUT-LEN]

Return Y
```

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

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

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

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

6.1.1 Interface

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

6.1.2 Algorithm

Compute Y using the following algorithm.

```
//
// Break M and K into 2-byte chunks
//
t = length(M) / 2
Let M_1, M_2, ..., M_t be 2-byte strings
  so that M = M_1 || M_2 || .. || M_t.
Let K_1, K_2, ..., K_t be 2-byte strings
  so that K_1 || K_2 || .. || K_t  is a prefix of K.

//
// Perform NH hash on the chunks, pairing words for multiplication
// which are 8 apart to accommodate vector-parallelism.
//
Y = zeroes(4)
i = 1
while (i < t) do
  Y = Y +_32 ((M_{i+0} +_16 K_{i+0}) *_32 (M_{i+ 8} +_16 K_{i+ 8}))
  Y = Y +_32 ((M_{i+1} +_16 K_{i+1}) *_32 (M_{i+ 9} +_16 K_{i+ 9}))
  Y = Y +_32 ((M_{i+2} +_16 K_{i+2}) *_32 (M_{i+10} +_16 K_{i+10}))
  Y = Y +_32 ((M_{i+3} +_16 K_{i+3}) *_32 (M_{i+11} +_16 K_{i+11}))
  Y = Y +_32 ((M_{i+4} +_16 K_{i+4}) *_32 (M_{i+12} +_16 K_{i+12}))
  Y = Y +_32 ((M_{i+5} +_16 K_{i+5}) *_32 (M_{i+13} +_16 K_{i+13}))
  Y = Y +_32 ((M_{i+6} +_16 K_{i+6}) *_32 (M_{i+14} +_16 K_{i+14}))
  Y = Y +_32 ((M_{i+7} +_16 K_{i+7}) *_32 (M_{i+15} +_16 K_{i+15}))
  i = i + 16

Return Y
```

6.2 L1-HASH-16: First-layer hash

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

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

6.2.1 Interface

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

6.2.2 Algorithm

Compute Y using the following algorithm.

```
//
// Break M into L1-KEY-LEN byte chunks (final chunk may be shorter)
//
t = ceil(length(M) / L1-KEY-LEN)
Let M_1, M_2, ..., M_t be strings so that M = M_1 || M_2 || .. ||
  M_t, and length(M_i) = L1-KEY-LEN for all 0 < i < t.

//
// For each chunk, except the last: endian-adjust, NH hash
// and add bit-length.  Use results to build Y.
//
Len = uint2str(L1-KEY-LEN * 8, 4)
Y = <empty string>
for i = 1 to t-1 do
  if (ENDIAN-FAVORITE == LITTLE) then  // See endian discussion
     ENDIAN-SWAP(M_i)                  // in section 3.1.1
  Y = Y || (NH-16(K, M_i) +_32 Len)

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

return Y
```

6.3 L2-HASH-16: Second-layer hash

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

6.3.1 Interface

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

6.3.2 Algorithm

Compute Y using the following algorithm.

```
  //
  //  Extract keys and restrict to special key-sets
  //
  Mask32  = uint2str(0x1fffffff, 4)
  Mask64  = uint2str(0x01ffffff01ffffff, 8)
  Mask128 = uint2str(0x01ffffff01ffffff01ffffff01ffffff, 16)
  k_32    = str2uint(K[1..4]   and Mask32)
  k_64    = str2uint(K[5..12]  and Mask64)
  k_128   = str2uint(K[13..28] and Mask128)

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

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

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

     M_1 = M[1 .. 2^11]
     M_2 = M[2^11 + 1 .. length(M)]
     M_2 = zeropad(M || uint2str(0x80,1), 8)
     y = POLY(32, 2^32 - 6,    k_32, M_1)
     y = POLY(64, 2^64 - 2^32, k_64, uint2str(y, 8) || M_2)

  else
```

```
        M_1 = M[1 .. 2^11]
        M_2 = M[2^11 + 1 .. 2^33]
        M_3 = M[2^33 + 1 .. length(M)]
        M_3 = zeropad(M || uint2str(0x80,1), 16)
        y = POLY(32,  2^32 - 6,     k_32,  M_1)
        y = POLY(64,  2^64 - 2^32,  k_64,  uint2str(y,  8) || M_2)
        y = POLY(128, 2^128 - 2^96, k_128, uint2str(y, 16) || M_3)

     Y = uint2str(y, 16)

     Return Y
```

6.4 L3-HASH-16: Third-layer hash

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

6.4.1 Interface

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

6.4.2 Algorithm

Compute Y using the following algorithm.

```
  y = 0

  //
  // Break M and K1 into 8 chunks and convert to integers
  //
  for i = 1 to 8 do
    M_i = M[(i - 1) * 2 + 1 .. i * 2]
    K_i = K1[(i - 1) * 4 + 1 .. i * 4]
    m_i = str2uint(M_i)
    k_i = str2uint(K_i) mod prime(19)

  //
```

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

    Return Y
```

6.5 UHASH-16: Three-layer universal hash

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

6.5.1 Interface

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

6.5.2 Algorithm

Compute Y using the following algorithm.

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

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

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

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

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

    Return Y
```

7 UMAC tag generation

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

7.1 Interface

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

7.2 Algorithm

Compute AuthTag using the following algorithm.

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

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

Return AuthTag
```

8 Security considerations

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

8.1 Resistance to cryptanalysis

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

The core of the UMAC design, the UHASH function, does not depend on any "cryptographic assumptions": its strength is specified by a purely mathematical property stated in terms of collision probability, and this property is proven in an absolute sense. In this way, the strength of UHASH is guaranteed regardless of future advances in cryptanalysis.

The analysis of UMAC [3, 4] shows this scheme to have "provable security", in the sense of modern cryptography, by way of tight reductions. What this means is that an adversarial attack on UMAC which forges with probability significantly exceeding the established collision probability will give rise to an attack of comparable complexity which breaks the AES, in the sense of distinguishing AES from a family of random permutations. This design approach essentially obviates the need for cryptanalysis on UMAC: cryptanalytic efforts might as well focus on AES, the results imply.

8.2 Tag lengths and forging probability

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

the MAC means that the attacker is able to generate, on its own, a new message M (i.e. one not previously transmitted between the legitimate parties) and to compute on M a correct authentication tag under the key K. This is called a forgery. Note that if the authentication tag is specified to be of length t then the attacker can trivially break the MAC with probability 1/2^t. For this the attacker can just generate any message of its choice and try a random tag; obviously, the tag is correct with probability 1/2^t. By repeated guesses the attacker can increase linearly its probability of success.

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

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

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

With UMAC, off-line computation aimed at exceeding the forging probability is hopeless, regardless of tag length, as long as the underlying cipher is not broken. The only way to forge is to interact with the entity that verifies the MAC and to try a huge amount of forgeries before one is likely to succeed. The system

architecture will determine the extent to which this is possible. In a well-architected system there should not be any high-bandwidth capability for presenting forged MACs and determining if they are valid. In particular, the number of authentication failures at the verifying party should be limited. If a large number of such attempts are detected the session key in use should be dropped and the event be recorded in an audit log.

Let us reemphasize: a forging probability of 1 / 2^60 does not mean that there is an attack that runs in 2^60 time - as long as AES maintains its believed security there is no such attack for UMAC. Instead, a 1 / 2^60 forging probability means that if an attacker could try out 2^60 MACs, then the attacker would probably get one right. But the architecture of any real system should render this infeasible. One can certainly imagine an attacker having a high bandwidth channel (e.g., 1 Gbit/second or more) over which it can continually present attempted forgeries, the attacker being signaled when a correct tag is found, but realizing such a scenario in a real system would represent a major lapse in the security architecture.

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

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

8.3 Selective-assurance authentication

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

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

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

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

8.4 Nonce considerations

The function UMAC (Section 7) requires a nonce with length in the range 1 to 16 bytes. All nonces in an authentication session must be equal in length. For secure operation, no nonce value should be repeated within the life of a single UMAC session-key.

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

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

1. The nonce is an eight-byte [resp., four-byte] unsigned number, Counter, which is initialized to zero, which is incremented by one following the generation of each authentication tag, and which is always communicated along with the message and the authentication tag. An error occurs at the sender if there is an attempt to authenticate more than 2^64 [resp., 2^32] messages within a session.

2. The nonce is a 16-byte unsigned number, Counter, which is initialized to zero and which is incremented by one following the generation of each authentication tag. The Counter is not explicitly communicated between the sender and receiver. Instead, the two are assumed to communicate over a reliable transport, and each maintains its own counter so as to keep track of what the current nonce value is.

3. The nonce is a 16-byte random value. (Because repetitions in a random n-bit value are expected at around 2^(n/2) trials, the number of messages to be communicated in a session using n-bit nonces should not be allowed to approach 2^(n/2).)

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

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

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

8.5 Guarding against replay attacks

A replay attack entails the attacker repeating a message, nonce, and authentication tag. In systems, replay attacks may be quite damaging, and many applications will want to guard against them. In UMAC, this would normally be done at the receiver by having the receiver check that no nonce value is used twice. On a reliable connection, when the nonce is a counter, this is trivial. On an unreliable connection, when the nonce is a counter, one would

normally cache some "window" of recent nonces. Out-of-order message delivery in excess of what the window allows will result in rejecting otherwise valid authentication tags.

We emphasize that it is up to the receiver when a given message, nonce and tag will be deemed authentic. Certainly the tag should be valid for the message and nonce, as determined by UMAC, but the message may still be deemed inauthentic because the nonce is detected to be a replay.

9 Acknowledgments

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

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

10 References

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

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

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and provably secure message authentication", Advances in Cryptology - CRYPTO '99, LNCS vol. 1666, pp. 216-233. Full version available from http://www.cs.ucdavis.edu/~rogaway/umac

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "The UMAC message authentication code", work in progress, 2000. To be available from http://www.cs.ucdavis.edu/~rogaway/umac

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

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

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

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

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

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

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

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

11 Author contact information

Authors' Addresses

John Black
Department of Computer Science
University of California
Davis CA 95616
USA

EMail: blackj@cs.ucdavis.edu

Shai Halevi
IBM T.J. Watson Research Center
P.O. Box 704

Yorktown Heights NY 10598
USA

EMail: shaih@watson.ibm.com

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

EMail: ahevia@cs.ucsd.edu

Hugo Krawczyk
Deprtment of Electrical Engineering
Technion
Haifa 32000
ISRAEL

EMail: hugo@ee.technion.ac.il

Ted Krovetz
Department of Computer Science
University of California
Davis CA 95616
USA

EMail: tdk@acm.org

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

EMail: rogaway@cs.ucdavis.edu

A Suggested application programming interface (API)

```
/* umac.h */

typedef struct UMAC_CTX *umac_ctx_t;

umac_ctx_t umac_alloc(char key[]);
  /* Dynamically allocate UMAC_CTX struct */
  /* initialize variables and generate   */
  /* subkeys for default parameters.     */
```

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

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

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

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

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

Each routine returns zero if unsuccessful.

B Reference code and test vectors

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

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

Appendix B

UMAC (2000) Implementation

This appendix is a verbatim copy of the source code written to implement the algorithm specified in Appendix A. The code is available at the UMAC website for public-domain consumption.

There are three sections in this chapter. First is the programming interface that we offer. Second is the implementation of the specification written in standard ANSI C. Third is a file which can be included in the ANSI C implementation which causes certain functions written in the ANSI C version to be overridden by Intel IA-32 assembly-language versions. This third file accelerates execution by a factor of two or more.

B.1 UMAC ANSI C Header

```
/* -----------------------------------------------------------------------
 *
 * umac.c -- C Implementation UMAC Message Authentication
 *
 * Version 0.04 of draft-krovetz-umac-00.txt -- 2000 August
 *
 * For a full description of UMAC message authentication see the UMAC
 * world-wide-web page at http://www.cs.ucdavis.edu/~rogaway/umac
 * Please report bugs and suggestions to the UMAC webpage.
 *
 * Copyright (c) 1999-2000 Ted Krovetz (tdk@acm.org)
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee, is hereby granted,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation, and that the names of the University of
 * California and Ted Krovetz not be used in advertising or publicity
 * pertaining to distribution of the software without specific,
 * written prior permission.
 *
 * The Regents of the University of California and Ted Krovetz disclaim
 * all warranties with regard to this software, including all implied
 * warranties of merchantability and fitness.  In no event shall the
 * University of California or Ted Krovetz be liable for any special,
 * indirect or consequential damages or any damages whatsoever resulting
 * from loss of use, data or profits, whether in an action of contract,
 * negligence or other tortious action, arising out of or in connection
 * with the use or performance of this software.
 *
 * ----------------------------------------------------------------------- */

/* umac.h */

#ifdef __cplusplus
    extern "C" {
#endif

typedef struct umac_ctx *umac_ctx_t;

umac_ctx_t umac_new(char key[]);
/* Dynamically allocate a umac_ctx struct, initialize variables,
 * generate subkeys from key.
 */

int umac_reset(umac_ctx_t ctx);
/* Reset a umac_ctx to begin authenicating a new message */

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

int umac_final(umac_ctx_t ctx, char tag[], char nonce[8]);
```

```
/* Incorporate any pending data and the ctr value, and return tag.
 * This function returns error code if ctr < 0.
 */

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

int umac(umac_ctx_t ctx, char *input,
         long len, char tag[],
         char nonce[8]);
/* All-in-one implementation of the functions Reset, Update and Final */


/* uhash.h */


typedef struct uhash_ctx *uhash_ctx_t;
  /* The uhash_ctx structure is defined by the implementation of the    */
  /* UHASH functions.                                                   */

uhash_ctx_t uhash_alloc(char key[16]);
  /* Dynamically allocate a uhash_ctx struct and generate subkeys using */
  /* the kdf and kdf_key passed in. If kdf_key_len is 0 then RC6 is     */
  /* used to generate key with a fixed key. If kdf_key_len > 0 but kdf  */
  /* is NULL then the first 16 bytes pointed at by kdf_key is used as a */
  /* key for an RC6 based KDF.                                          */

int uhash_free(uhash_ctx_t ctx);

int uhash_set_params(uhash_ctx_t ctx,
                     void       *params);

int uhash_reset(uhash_ctx_t ctx);

int uhash_update(uhash_ctx_t ctx,
                 char       *input,
                 long        len);

int uhash_final(uhash_ctx_t ctx,
                char        ouput[]);

int uhash(uhash_ctx_t ctx,
          char       *input,
          long        len,
          char        output[]);

#ifdef __cplusplus
    }
#endif
```

B.2 UMAC ANSI C Source

```
/* ---------------------------------------------------------------------
 *
 * umac.c -- C Implementation UMAC Message Authentication
 *
 * Version 0.04 of draft-krovetz-umac-00.txt -- 2000 August
 *
 * For a full description of UMAC message authentication see the UMAC
 * world-wide-web page at http://www.cs.ucdavis.edu/~rogaway/umac
 * Please report bugs and suggestions to the UMAC webpage.
 *
 * Copyright (c) 1999-2000 Ted Krovetz (tdk@acm.org)
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee, is hereby granted,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation, and that the names of the University of
 * California and Ted Krovetz not be used in advertising or publicity
 * pertaining to distribution of the software without specific,
 * written prior permission.
 *
 * The Regents of the University of California and Ted Krovetz disclaim
 * all warranties with regard to this software, including all implied
 * warranties of merchantability and fitness.  In no event shall the
 * University of California or Ted Krovetz be liable for any special,
 * indirect or consequential damages or any damages whatsoever resulting
 * from loss of use, data or profits, whether in an action of contract,
 * negligence or other tortious action, arising out of or in connection
 * with the use or performance of this software.
 *
 * ------------------------------------------------------------------- */

/* ------------------------------------------------------------------- */
/* -- Global Includes ---------------------------------------------- */
/* ------------------------------------------------------------------- */

#include "umac.h"
#include <string.h>
#include <stdlib.h>

/* ------------------------------------------------------------------- */
/* --- User Switches ------------------------------------------------ */
/* ------------------------------------------------------------------- */

/* Following is the list of UMAC parameters supported by this code.
 * The following parameters are fixed in this implementation.
 *
 *      ENDIAN_FAVORITE_LITTLE  = 1
 *      L1-OPERATIONS-SIGN      = SIGNED   (when WORD_LEN == 2)
 *      L1-OPERATIONS-SIGN      = UNSIGNED (when WORD_LEN == 4)
 */
#define WORD_LEN                4   /* 2  | 4                            */
```

```
#define UMAC_OUTPUT_LEN         8   /* 4  | 8  | 12  | 16                */
#define L1_KEY_LEN           1024   /* 32 | 64 | 128 | ... | 2^28         */
#define UMAC_KEY_LEN           16   /* 16 | 32                           */

/* To produce a prefix of a tag rather than the entire tag defined
 * by the above parameters, set the following constant to a number
 * less than UMAC_OUTPUT_LEN.
 */
#define UMAC_PREFIX_LEN  UMAC_OUTPUT_LEN

/* This file implements UMAC in ANSI C as long as the compiler supports 64-
 * bit integers. To accellerate the execution of the code, architecture-
 * specific replacements have been supplied for some compiler/instruction-
 * set combinations. To enable the features of these replacements, the
 * following compiler directives must be set appropriately. Some compilers
 * include "intrinsic" support of basic operations like register rotation,
 * byte reversal, or vector SIMD manipulation. To enable these intrinsics
 * set USE_C_AND_INTRINSICS to 1. Most compilers also allow for inline
 * assembly in the C code. To allow intrinsics and/or assembly routines
 * (whichever is faster) set only USE_C_AND_ASSEMBLY to 1.
 */
#define USE_C_ONLY            1  /* ANSI C and 64-bit integers req'd */
#define USE_C_AND_INTRINSICS  0  /* Intrinsics for rotation, MMX, etc.    */
#define USE_C_AND_ASSEMBLY    0  /* Intrinsics and assembly */

#if (USE_C_ONLY + USE_C_AND_INTRINSICS + USE_C_AND_ASSEMBLY != 1)
#error -- Only one setting may be nonzero
#endif

#define RUN_TESTS             0  /* Run basic correctness/speed tests    */
#define HASH_ONLY             0  /* Only universal hash data, don't MAC   */

/* ---------------------------------------------------------------------- */
/* --- Primitive Data Types ---                                           */
/* ---------------------------------------------------------------------- */

#ifdef _MSC_VER
typedef unsigned char      UINT8;   /* 1 byte   */
typedef __int16            INT16;  /* 2 byte   */
typedef unsigned __int16   UINT16; /* 2 byte   */
typedef __int32            INT32;  /* 4 byte   */
typedef unsigned __int32   UINT32; /* 4 byte   */
typedef unsigned __int64   UINT64; /* 8 bytes  */
#else
typedef unsigned char      UINT8;  /* 1 byte   */
typedef short              INT16;  /* 2 byte   */
typedef unsigned short     UINT16; /* 2 byte   */
typedef int                INT32;  /* 4 byte   */
typedef unsigned int       UINT32; /* 4 byte   */
typedef unsigned long long UINT64; /* 8 bytes  */
#endif
typedef unsigned long      UWORD;  /* Register */

/* ---------------------------------------------------------------------- */
```

```
/* --- Derived Constants ------------------------------------------------- */
/* ----------------------------------------------------------------------- */

#if (WORD_LEN == 4)

typedef UINT32  SMALL_UWORD;
typedef UINT64  LARGE_UWORD;

#elif (WORD_LEN == 2)

typedef UINT16  SMALL_UWORD;
typedef UINT32  LARGE_UWORD;

#endif

/* How many iterations, or streams, are needed to produce UMAC_OUTPUT_LEN
 * and UMAC_PREFIX_LEN bytes of output
 */
#define PREFIX_STREAMS    (UMAC_PREFIX_LEN / WORD_LEN)
#define OUTPUT_STREAMS    (UMAC_OUTPUT_LEN / WORD_LEN)

/* Three compiler environments are supported for accellerated
 * implementations: GNU gcc and Microsoft Visual C++ (and copycats) on x86,
 * and Metrowerks on PowerPC.
 */
#define GCC_X86         (__GNUC__ && __i386__)      /* GCC on IA-32       */
#define MSC_X86         (_MSC_VER && _M_IX86)       /* Microsoft on IA-32 */
#define MW_PPC          ((__MWERKS__ || __MRC__) && __POWERPC__)
                                                    /* Metrowerks on PPC  */
/* ----------------------------------------------------------------------- */
/* --- Host Computer Endian Definition ----------------------------------- */
/* ----------------------------------------------------------------------- */

/* Message "words" are read from memory in an endian-specific manner.     */
/* For this implementation to behave correctly, __LITTLE_ENDIAN__ must    */
/* be set true if the host computer is little-endian.                     */

#if __i386__ || __alpha__ || _M_IX86 || __LITTLE_ENDIAN
#define __LITTLE_ENDIAN__ 1
#else
#define __LITTLE_ENDIAN__ 0
#endif

/* ----------------------------------------------------------------------- */
/* ----- RC6 Function Family Constants ------------------------------------ */
/* ----------------------------------------------------------------------- */

/* These constants belong to RC6. They will change when AES is announced  */
#define RC6_KEY_BYTES    UMAC_KEY_LEN
#define RC6_ROUNDS       20
#define RC6_KEY_WORDS    (UMAC_KEY_LEN/4)
#define RC6_TABLE_WORDS  (2*RC6_ROUNDS+4)
#define RC6_P            0xb7e15163u
#define RC6_Q            0x9e3779b9u
```

```
/* These constants are for the KDF and PDF only */
#define AES_BLOCK_LEN            16
#define AES_INTERNAL_KEY_LEN     (RC6_TABLE_WORDS*4)

/* ---------------------------------------------------------------------- */
/* ----- Poly hash and Inner-Product hash Constants ---------------------- */
/* ---------------------------------------------------------------------- */

/* Primes and masks */
#define p19    ((UINT32)0x0007FFFFu)             /* 2^19 -  1 */
#define p32    ((UINT32)0xFFFFFFFBu)             /* 2^32 -  5 */
#define m19    ((UINT32)0x0007FFFFu)  /* The low 19 of 32 bits */

#if _MSC_VER  /* no support for ull suffix in Visual C++ */
const UINT64 p36 = (((UINT64)1 << 36) - (UINT64)5);         /* 2^36 -  5 */
const UINT64 p64 = ((UINT64)0 - (UINT64)59);                /* 2^64 - 59 */
const UINT64 m36 = (((UINT64)1 << 36) - 1);  /* The low 36 of 64 bits */
#else
#define p36    ((UINT64)0x0000000FFFFFFFFBull)             /* 2^36 -  5 */
#define p64    ((UINT64)0xFFFFFFFFFFFFFFC5ull)             /* 2^64 - 59 */
#define m36    ((UINT64)0x0000000FFFFFFFFFull)  /* The low 36 of 64 bits */
#endif

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Architecture Specific Routines ---------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* These are the optional, architecture-specific accelleration files      */

#if (GCC_X86 && ! USE_C_ONLY)
#include "umac_gcc_x86_incl.c"
#elif (MSC_X86 && ! USE_C_ONLY)
#include "umac_msc_x86_incl.c"
#elif (MW_PPC && ! USE_C_ONLY)
#include "umac_mw_ppc_incl.c"
#endif

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Primitive Routines ---------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
/* --- 32-Bit Rotation operators                                          */
/* ---------------------------------------------------------------------- */

/* Good compilers can detect when a rotate
 * is being constructed from bitshifting and bitwise OR and output the
 * assembly rotates. Other compilers require assembly or C intrinsics.
 * There are two versions because some intrinsics differentiate between
```

```
 * constant rotation and variable rotation. "n" must be on 0..31.
 */

#if (USE_C_ONLY || ! ARCH_ROTL)

#define ROTL32_VAR(r,n)   (((r) <<  (n))       | \
                           ((UINT32)(r) >> (32 -  (n))))
#define ROTL32_CONST(r,n) (((r) <<  (n))       | \
                           ((UINT32)(r) >> (32 -  (n))))

#endif

/* ---------------------------------------------------------------------- */
/* --- 32-bit by 32-bit to 64-bit Multiplication ------------------------ */
/* ---------------------------------------------------------------------- */

#if (USE_C_ONLY || ! ARCH_MUL64)

#define MUL64(a,b) ((UINT64)((UINT64)(UINT32)(a) * (UINT64)(UINT32)(b)))

#endif

/* ---------------------------------------------------------------------- */
/* --- Endian Conversion --- Forcing assembly on some platforms           */
/* ---------------------------------------------------------------------- */

/* Lots of endian reversals happen in UMAC. PowerPC and Intel Architechture
 * both support efficient endian conversion, but compilers seem unable to
 * automatically utilize the efficient assembly opcodes. The architechture-
 * specific versions utilize them.
 */

#if (USE_C_ONLY || ! ARCH_ENDIAN_LS)

static UINT32 LOAD_UINT32_REVERSED(void *ptr)
{
    UINT32 temp = *(UINT32 *)ptr;
    temp = (temp >> 24) | ((temp & 0x00FF0000) >> 8 )
         | ((temp & 0x0000FF00) << 8 ) | (temp << 24);
    return (UINT32)temp;
}

static void STORE_UINT32_REVERSED(void *ptr, UINT32 x)
{
    UINT32 i = (UINT32)x;
    *(UINT32 *)ptr = (i >> 24) | ((i & 0x00FF0000) >> 8 )
                   | ((i & 0x0000FF00) << 8 ) | (i << 24);
}

static UINT16 LOAD_UINT16_REVERSED(void *ptr)
{
    UINT16 temp = *(UINT16 *)ptr;
    temp = (temp >> 8) | (temp << 8);
    return (UINT16)temp;
```

```
}

static void STORE_UINT16_REVERSED(void *ptr, UINT16 x)
{
    UINT16 temp = (UINT16)x;
    *(UINT16 *)ptr = (temp >> 8) | (temp << 8);
}

#endif

/* The following definitions use the above reversal-primitives to do the right
 * thing on endian specific load and stores.
 */

#if (__LITTLE_ENDIAN__)
#define LOAD_UINT16_LITTLE(ptr)     (*(UINT16 *)(ptr))
#define LOAD_UINT32_LITTLE(ptr)     (*(UINT32 *)(ptr))
#define STORE_UINT16_LITTLE(ptr,x)  (*(UINT16 *)(ptr) = (UINT16)(x))
#define STORE_UINT32_LITTLE(ptr,x)  (*(UINT32 *)(ptr) = (UINT32)(x))
#define LOAD_UINT16_BIG(ptr)        LOAD_UINT16_REVERSED(ptr)
#define LOAD_UINT32_BIG(ptr)        LOAD_UINT32_REVERSED(ptr)
#define STORE_UINT16_BIG(ptr,x)     STORE_UINT16_REVERSED(ptr,x)
#define STORE_UINT32_BIG(ptr,x)     STORE_UINT32_REVERSED(ptr,x)
#else
#define LOAD_UINT16_LITTLE(ptr)     LOAD_UINT16_REVERSED(ptr)
#define LOAD_UINT32_LITTLE(ptr)     LOAD_UINT32_REVERSED(ptr)
#define STORE_UINT16_LITTLE(ptr,x)  STORE_UINT16_REVERSED(ptr,x)
#define STORE_UINT32_LITTLE(ptr,x)  STORE_UINT32_REVERSED(ptr,x)
#define LOAD_UINT16_BIG(ptr)        (*(UINT16 *)(ptr))
#define LOAD_UINT32_BIG(ptr)        (*(UINT32 *)(ptr))
#define STORE_UINT16_BIG(ptr,x)     (*(UINT16 *)(ptr) = (UINT16)(x))
#define STORE_UINT32_BIG(ptr,x)     (*(UINT32 *)(ptr) = (UINT32)(x))
#endif


/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Begin Cryptographic Primitive Section -------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* The UMAC specification requires the use of AES for it's cryptographic
 * component. Until AES is finalized, we use RC6, an AES finalist, in its
 * place.
 */

/* ---------------------------------------------------------------------- */
#if (USE_C_ONLY || ! ARCH_RC6)
/* ---------------------------------------------------------------------- */

#define RC6_BLOCK(a,b,c,d,n)    \
        t = b*(2*b+1);          \
        t = ROTL32_CONST(t,5);    \
        u = d*(2*d+1);          \
```

```
        u = ROTL32_CONST(u,5);    \
        a ^= t;                  \
        a = ROTL32_VAR(a,u&31);  \
        a += s[n];               \
        c ^= u;                  \
        c = ROTL32_VAR(c,t&31);  \
        c += s[n+1];

static void RC6(UINT32 S[], void *pt, void *ct)
{
    const UINT32 *s = (UINT32 *)S;
    UINT32 A = LOAD_UINT32_LITTLE((UINT32 *)pt  );
    UINT32 B = LOAD_UINT32_LITTLE((UINT32 *)pt+1) + s[0];
    UINT32 C = LOAD_UINT32_LITTLE((UINT32 *)pt+2);
    UINT32 D = LOAD_UINT32_LITTLE((UINT32 *)pt+3) + s[1];
    UINT32 t,u;

    RC6_BLOCK(A,B,C,D, 2)
    RC6_BLOCK(B,C,D,A, 4)
    RC6_BLOCK(C,D,A,B, 6)
    RC6_BLOCK(D,A,B,C, 8)

    RC6_BLOCK(A,B,C,D,10)
    RC6_BLOCK(B,C,D,A,12)
    RC6_BLOCK(C,D,A,B,14)
    RC6_BLOCK(D,A,B,C,16)

    RC6_BLOCK(A,B,C,D,18)
    RC6_BLOCK(B,C,D,A,20)
    RC6_BLOCK(C,D,A,B,22)
    RC6_BLOCK(D,A,B,C,24)

    RC6_BLOCK(A,B,C,D,26)
    RC6_BLOCK(B,C,D,A,28)
    RC6_BLOCK(C,D,A,B,30)
    RC6_BLOCK(D,A,B,C,32)

    RC6_BLOCK(A,B,C,D,34)
    RC6_BLOCK(B,C,D,A,36)
    RC6_BLOCK(C,D,A,B,38)
    RC6_BLOCK(D,A,B,C,40)

    A += s[42];
    C += s[43];
    STORE_UINT32_LITTLE((UINT32 *)ct  , A);
    STORE_UINT32_LITTLE((UINT32 *)ct+1, B);
    STORE_UINT32_LITTLE((UINT32 *)ct+2, C);
    STORE_UINT32_LITTLE((UINT32 *)ct+3, D);
}

/* ---------------------------------------------------------------------- */
#endif
/* ---------------------------------------------------------------------- */
```

```
static void RC6_SETUP(UINT8 *K, UINT32 S[])
{
    UWORD i, j, k;
    UINT32 A, B, L[RC6_KEY_WORDS];

    /* Load little endian key into L */
    #if (__LITTLE_ENDIAN__)
    memcpy(L, K, RC6_KEY_BYTES);
    #else
    for (i = 0; i < RC6_KEY_WORDS; i++)
        STORE_UINT32_LITTLE(L+i, LOAD_UINT32_BIG((UINT32 *)K+i));
    #endif

    /* Preload S with P and Q derived constants */
    S[0]=RC6_P;
    for (i = 1; i < RC6_TABLE_WORDS; i++)
        S[i] = S[i-1] + RC6_Q;

    /* Mix L into S */
    A=B=i=j=k=0;
    for (k = 0; k < 3*RC6_TABLE_WORDS; k++) {
        A = S[i] = ROTL32_CONST(S[i]+(A+B),3);
        B = L[j] = ROTL32_VAR(L[j]+(A+B),(A+B)&31);
        i=(i+1)%RC6_TABLE_WORDS;
        j=(j+1)%RC6_KEY_WORDS;
    }
}

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Begin KDF & PDF Section ---------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* The user-supplied UMAC key is stretched using AES in an output feedback
 * mode to supply all random bits needed by UMAC. The kdf function takes
 * and AES internal key representation 'key' and writes a stream of
 * 'nbytes' bytes to the memory pointed at by 'buffer_ptr'. Each distinct
 * 'index' causes a distinct byte stream.
 */
static void kdf(void *buffer_ptr, UINT8 *key, UINT8 index, int nbytes)
{
    UINT8 chain[AES_BLOCK_LEN] = {0};
    UINT8 *dst_buf = (UINT8 *)buffer_ptr;

    chain[AES_BLOCK_LEN-1] = index;

    while (nbytes >= AES_BLOCK_LEN) {
        RC6((UINT32 *)key,chain,chain);
        memcpy(dst_buf,chain,AES_BLOCK_LEN);
        nbytes -= AES_BLOCK_LEN;
        dst_buf += AES_BLOCK_LEN;
    }
    if (nbytes) {
```

```
        RC6((UINT32 *)key,chain,chain);
        memcpy(dst_buf,chain,nbytes);
    }
}

/* The final UHASH result is XOR'd with the output of a pseudorandom
 * function. Here, we use AES to generate random output and
 * xor the appropriate bytes depending on the last bits of nonce.
 * This scheme is optimized for sequential, increasing big-endian nonces.
 */

typedef struct {
    UINT8 cache[AES_BLOCK_LEN];  /* Previous AES output is saved      */
    UINT8 nonce[AES_BLOCK_LEN];  /* The AES input for the above cache */
    UINT8 prf_key[AES_INTERNAL_KEY_LEN]; /* Expanded AES key for PDF  */
} pdf_ctx;

static void pdf_init(pdf_ctx *pc, UINT8 *prf_key)
{
    UINT8 buf[UMAC_KEY_LEN];

    kdf(buf, prf_key, 128, UMAC_KEY_LEN);
    RC6_SETUP(buf, (UINT32 *)pc->prf_key);

    /* Initialize pdf and cache */
    memset(pc->nonce, 0, sizeof(pc->nonce));
    RC6((UINT32 *)pc->prf_key, pc->nonce, pc->cache);
}

static void pdf_gen_xor(pdf_ctx *pc, UINT8 nonce[8], UINT8 buf[8])
{
    /* This implementation requires UMAC_OUTPUT_LEN to divide AES_BLOCK_LEN
     * or be at least 1/2 its length. 'index' indicates that we'll be using
     * the index-th UMAC_OUTPUT_LEN-length element of the AES output. If
     * last time around we returned the index-1 element, then we may have
     * the result in the cache already.
     */
    UINT8 tmp_nonce_lo[4];
    int index = nonce[7] % (AES_BLOCK_LEN / UMAC_OUTPUT_LEN);

    *(UINT32 *)tmp_nonce_lo = ((UINT32 *)nonce)[1];
    tmp_nonce_lo[3] ^= index; /* zero some bits */

    if ( (((UINT32 *)tmp_nonce_lo)[0] != ((UINT32 *)pc->nonce)[1]) ||
         (((UINT32 *)nonce)[0] != ((UINT32 *)pc->nonce)[0]) )
    {
        ((UINT32 *)pc->nonce)[0] = ((UINT32 *)nonce)[0];
        ((UINT32 *)pc->nonce)[1] = ((UINT32 *)tmp_nonce_lo)[0];
        RC6((UINT32 *)pc->prf_key, pc->nonce, pc->cache);
    }

    #if (UMAC_OUTPUT_LEN == 2)
        *((UINT16 *)buf) ^= ((UINT16 *)pc->cache)[index];
    #elif (UMAC_OUTPUT_LEN == 4)
```

```
        *((UINT32 *)buf) ^= ((UINT32 *)pc->cache)[index];
    #elif (UMAC_OUTPUT_LEN == 8)
        *((UINT64 *)buf) ^= ((UINT64 *)pc->cache)[index];
    #elif (UMAC_OUTPUT_LEN == 12)
        ((UINT64 *)buf)[0] ^= ((UINT64 *)pc->cache)[0];
        ((UINT32 *)buf)[2] ^= ((UINT32 *)pc->cache)[2];
    #elif (UMAC_OUTPUT_LEN == 16)
        ((UINT64 *)buf)[0] ^= ((UINT64 *)pc->cache)[0];
        ((UINT64 *)buf)[1] ^= ((UINT64 *)pc->cache)[1];
    #else
        #error only 2,4,8,12,16 byte output supported.
    #endif
}

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Begin NH Hash Section ------------------------------------------ */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* The NH-based hash functions used in UMAC are described in the UMAC paper
 * and specification, both of which can be found at the UMAC website.
 * The interface to this implementation has two
 * versions, one expects the entire message being hashed to be passed
 * in a single buffer and returns the hash result immediately. The second
 * allows the message to be passed in a sequence of buffers. In the
 * muliple-buffer interface, the client calls the routine nh_update() as
 * many times as necessary. When there is no more data to be fed to the
 * hash, the client calls nh_final() which calculates the hash output.
 * Before beginning another hash calculation the nh_reset() routine
 * must be called. The single-buffer routine, nh(), is equivalent to
 * the sequence of calls nh_update() and nh_final(); however it is
 * optimized and should be prefered whenever the multiple-buffer interface
 * is not necessary. When using either interface, it is the client's
 * responsability to pass no more than L1_KEY_LEN bytes per hash result.
 *
 * The routine nh_init() initializes the nh_ctx data structure and
 * must be called once, before any other PDF routine.
 */

 /* The "nh_aux_*" routines do the actual NH hashing work. They
  * expect buffers to be multiples of L1_PAD_BOUNDARY. These routines
  * produce output for all PREFIX_STREAMS NH iterations in one call,
  * allowing the parallel implementation of the streams.
  */
#if   (UMAC_PREFIX_LEN == 2)
#define nh_aux   nh_aux_4
#elif (UMAC_PREFIX_LEN == 4)
#define nh_aux   nh_aux_8
#elif (UMAC_PREFIX_LEN == 8)
#define nh_aux   nh_aux_16
#elif (UMAC_PREFIX_LEN == 12)
#define nh_aux   nh_aux_24
#elif (UMAC_PREFIX_LEN == 16)
```

```
#define nh_aux   nh_aux_32
#endif

#define L1_KEY_SHIFT         16     /* Toeplitz key shift between streams */
#define L1_PAD_BOUNDARY      32     /* pad message to boundary multiple   */
#define ALLOC_BOUNDARY       32     /* Keep buffers aligned to this       */
#define HASH_BUF_BYTES      128     /* nh_aux_hb buffer multiple          */

/* How many extra bytes are needed for Toeplitz shift? */
#define TOEPLITZ_EXTRA       ((PREFIX_STREAMS - 1) * L1_KEY_SHIFT)

typedef struct {
    UINT8  nh_key [L1_KEY_LEN + TOEPLITZ_EXTRA]; /* NH Key */
    UINT8  data   [HASH_BUF_BYTES];    /* Incomming data buffer           */
    int next_data_empty;    /* Bookeeping variable for data buffer.       */
    int bytes_hashed;        /* Bytes (out of L1_KEY_LEN) incorperated.   */
    LARGE_UWORD state[PREFIX_STREAMS];               /* on-line state     */
} nh_ctx;


/* ------------------------------------------------------------------------ */
#if (WORD_LEN == 4)
/* ------------------------------------------------------------------------ */

/* ------------------------------------------------------------------------ */
#if (USE_C_ONLY || ! ARCH_NH32)
/* ------------------------------------------------------------------------ */

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
/* NH hashing primitive. Previous (partial) hash result is loaded and
 * then stored via hp pointer. The length of the data pointed at by "dp",
 * "dlen", is guaranteed to be divisible by L1_PAD_BOUNDARY (32).  Key
 * is expected to be endian compensated in memory at key setup.
 */
{
  UINT64 h;
  UWORD c = dlen / 32;
  UINT32 *k = (UINT32 *)kp;
  UINT32 *d = (UINT32 *)dp;
  UINT32 d0,d1,d2,d3,d4,d5,d6,d7;
  UINT32 k0,k1,k2,k3,k4,k5,k6,k7;

  h = *((UINT64 *)hp);
  do {
    d0 = LOAD_UINT32_LITTLE(d+0); d1 = LOAD_UINT32_LITTLE(d+1);
    d2 = LOAD_UINT32_LITTLE(d+2); d3 = LOAD_UINT32_LITTLE(d+3);
    d4 = LOAD_UINT32_LITTLE(d+4); d5 = LOAD_UINT32_LITTLE(d+5);
    d6 = LOAD_UINT32_LITTLE(d+6); d7 = LOAD_UINT32_LITTLE(d+7);
    k0 = *(k+0); k1 = *(k+1); k2 = *(k+2); k3 = *(k+3);
    k4 = *(k+4); k5 = *(k+5); k6 = *(k+6); k7 = *(k+7);
    h += MUL64((k0 + d0), (k4 + d4));
    h += MUL64((k1 + d1), (k5 + d5));
    h += MUL64((k2 + d2), (k6 + d6));
    h += MUL64((k3 + d3), (k7 + d7));
```

```
    d += 8;
    k += 8;
  } while (--c);
  *((UINT64 *)hp) = h;
}

/* ---------------------------------------------------------------------- */

static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
/* Same as nh_aux_8, but two streams are handled in one pass,
 * reading and writing 16 bytes of hash-state per call.
 */
{
  UINT64 h1,h2;
  UWORD c = dlen / 32;
  UINT32 *k = (UINT32 *)kp;
  UINT32 *d = (UINT32 *)dp;
  UINT32 d0,d1,d2,d3,d4,d5,d6,d7;
  UINT32 k0,k1,k2,k3,k4,k5,k6,k7,
         k8,k9,k10,k11;

  h1 = *((UINT64 *)hp);
  h2 = *((UINT64 *)hp + 1);
  k0 = *(k+0); k1 = *(k+1); k2 = *(k+2); k3 = *(k+3);
  do {
    d0 = LOAD_UINT32_LITTLE(d+0); d1 = LOAD_UINT32_LITTLE(d+1);
    d2 = LOAD_UINT32_LITTLE(d+2); d3 = LOAD_UINT32_LITTLE(d+3);
    d4 = LOAD_UINT32_LITTLE(d+4); d5 = LOAD_UINT32_LITTLE(d+5);
    d6 = LOAD_UINT32_LITTLE(d+6); d7 = LOAD_UINT32_LITTLE(d+7);
    k4 = *(k+4); k5 = *(k+5); k6 = *(k+6); k7 = *(k+7);
    k8 = *(k+8); k9 = *(k+9); k10 = *(k+10); k11 = *(k+11);

    h1 += MUL64((k0 + d0), (k4 + d4));
    h2 += MUL64((k4 + d0), (k8 + d4));

    h1 += MUL64((k1 + d1), (k5 + d5));
    h2 += MUL64((k5 + d1), (k9 + d5));

    h1 += MUL64((k2 + d2), (k6 + d6));
    h2 += MUL64((k6 + d2), (k10 + d6));

    h1 += MUL64((k3 + d3), (k7 + d7));
    h2 += MUL64((k7 + d3), (k11 + d7));

    k0 = k8; k1 = k9; k2 = k10; k3 = k11;

    d += 8;
    k += 8;
  } while (--c);
  ((UINT64 *)hp)[0] = h1;
  ((UINT64 *)hp)[1] = h2;
}
```

```
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
#endif /* (USE_C_ONLY || ! ARCH_NH32) */
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
/* ----- NH16 Universal Hash ------------------------------------------- */
/* ---------------------------------------------------------------------- */

#else /* WORD_LEN == 2 */


/* ---------------------------------------------------------------------- */
#if (USE_C_ONLY || ! ARCH_NH16)
/* ---------------------------------------------------------------------- */

static void nh_aux_4(void *kp, void *dp, void *hp, UINT32 dlen)
/* NH hashing primitive. Previous (partial) hash result is loaded and
 * then stored via hp pointer. The length of the data pointed at by "dp",
 * "dlen", is guaranteed to be divisible by L1_PAD_BOUNDARY (32).  Key
 * is expected to be endian compensated in memory at key setup.
 */
{
    UINT32 h;
    UINT32 c = dlen / 32;
    UINT16 *k = (UINT16 *)kp;
    UINT16 *d = (UINT16 *)dp;

    h = *(UINT32 *)hp;
    do {
     /* Cast to signed integers to forced signed multiplication */
        h += (INT32)(INT16)(*(k+0)  + LOAD_UINT16_LITTLE(d+0)) *
             (INT32)(INT16)(*(k+8)  + LOAD_UINT16_LITTLE(d+8));
        h += (INT32)(INT16)(*(k+1)  + LOAD_UINT16_LITTLE(d+1)) *
             (INT32)(INT16)(*(k+9)  + LOAD_UINT16_LITTLE(d+9));
        h += (INT32)(INT16)(*(k+2)  + LOAD_UINT16_LITTLE(d+2)) *
             (INT32)(INT16)(*(k+10) + LOAD_UINT16_LITTLE(d+10));
        h += (INT32)(INT16)(*(k+3)  + LOAD_UINT16_LITTLE(d+3)) *
             (INT32)(INT16)(*(k+11) + LOAD_UINT16_LITTLE(d+11));
        h += (INT32)(INT16)(*(k+4)  + LOAD_UINT16_LITTLE(d+4)) *
             (INT32)(INT16)(*(k+12) + LOAD_UINT16_LITTLE(d+12));
        h += (INT32)(INT16)(*(k+5)  + LOAD_UINT16_LITTLE(d+5)) *
             (INT32)(INT16)(*(k+13) + LOAD_UINT16_LITTLE(d+13));
        h += (INT32)(INT16)(*(k+6)  + LOAD_UINT16_LITTLE(d+6)) *
             (INT32)(INT16)(*(k+14) + LOAD_UINT16_LITTLE(d+14));
        h += (INT32)(INT16)(*(k+7)  + LOAD_UINT16_LITTLE(d+7)) *
             (INT32)(INT16)(*(k+15) + LOAD_UINT16_LITTLE(d+15));
        d += 16;
        k += 16;
    } while (--c);
    *(UINT32 *)hp = h;
}
```

```
/* ---------------------------------------------------------------------- */

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
/* Same as nh_aux_4, but two streams are handled in one pass,
 * reading and writing 8 bytes of hash-state per call.
 */
{
    nh_aux_4(kp,dp,hp,dlen);
    nh_aux_4((UINT8 *)kp+16,dp,(UINT8 *)hp+4,dlen);
}

/* ---------------------------------------------------------------------- */

static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
/* Same as nh_aux_8, but four streams are handled in one pass,
 * reading and writing 16 bytes of hash-state per call.
 */
{
    nh_aux_4(kp,dp,hp,dlen);
    nh_aux_4((UINT8 *)kp+16,dp,(UINT8 *)hp+4,dlen);
    nh_aux_4((UINT8 *)kp+32,dp,(UINT8 *)hp+8,dlen);
    nh_aux_4((UINT8 *)kp+48,dp,(UINT8 *)hp+12,dlen);
}

/* ---------------------------------------------------------------------- */



/* ---------------------------------------------------------------------- */
#endif  /* (USE_C_ONLY || ! ARCH_NH16)        */
/* ---------------------------------------------------------------------- */
#endif  /* WORD_LEN                           */
/* ---------------------------------------------------------------------- */

/* The following two routines use previously defined ones to build up longer
 * outputs of 24 or 32 bytes.
 */


/* ---------------------------------------------------------------------- */

static void nh_aux_24(void *kp, void *dp, void *hp, UINT32 dlen)
{
    nh_aux_16(kp,dp,hp,dlen);
    nh_aux_8((UINT8 *)kp+((8/WORD_LEN)*L1_KEY_SHIFT),dp,
                                          (UINT8 *)hp+16,dlen);
}

/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */

static void nh_aux_32(void *kp, void *dp, void *hp, UINT32 dlen)
{
```

```
    nh_aux_16(kp,dp,hp,dlen);
    nh_aux_16((UINT8 *)kp+((8/WORD_LEN)*L1_KEY_SHIFT),
                                    dp,(UINT8 *)hp+16,dlen);
}

/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */

static void nh_transform(nh_ctx *hc, UINT8 *buf, UINT32 nbytes)
/* This function is a wrapper for the primitive NH hash functions. It takes
 * as argument "hc" the current hash context and a buffer which must be a
 * multiple of L1_PAD_BOUNDARY. The key passed to nh_aux is offset
 * appropriately according to how much message has been hashed already.
 */
{
    UINT8 *key;

    key = hc->nh_key + hc->bytes_hashed;
    nh_aux(key, buf, hc->state, nbytes);
}

/* ---------------------------------------------------------------------- */

static void endian_convert(void *buf, UWORD bpw, UINT32 num_bytes)
/* We endian convert the keys on little-endian computers to               */
/* compensate for the lack of big-endian memory reads during hashing.     */
{
    UWORD iters = num_bytes / bpw;
    if (bpw == 2) {
        UINT16 *p = (UINT16 *)buf;
        do {
            *p = ((UINT16)*p >> 8) | (*p << 8);
            p++;
        } while (--iters);
    } else if (bpw == 4) {
        UINT32 *p = (UINT32 *)buf;
        do {
            *p = LOAD_UINT32_REVERSED(p);
            p++;
        } while (--iters);
    } else if (bpw == 8) {
        UINT32 *p = (UINT32 *)buf;
        UINT32 t;
        do {
            t = LOAD_UINT32_REVERSED(p+1);
            p[1] = LOAD_UINT32_REVERSED(p);
            p[0] = t;
            p += 2;
        } while (--iters);
    }
}
#if (__LITTLE_ENDIAN__)
```

```
#define endian_convert_if_le(x,y,z) endian_convert((x),(y),(z))
#else
#define endian_convert_if_le(x,y,z) do{}while(0)  /* Do nothing */
#endif

/* ------------------------------------------------------------------------ */

static void nh_reset(nh_ctx *hc)
/* Reset nh_ctx to ready for hashing of new data */
{
    hc->bytes_hashed = 0;
    hc->next_data_empty = 0;
    hc->state[0] = 0;
    #if (PREFIX_STREAMS > 1)
    hc->state[1] = 0;
    #if (PREFIX_STREAMS > 2)
    hc->state[2] = 0;
    #if (PREFIX_STREAMS > 3)
    hc->state[3] = 0;
    #if (PREFIX_STREAMS > 4)
    hc->state[4] = 0;
    hc->state[5] = 0;
    #if (PREFIX_STREAMS > 6)
    hc->state[6] = 0;
    hc->state[7] = 0;
    #endif
    #endif
    #endif
    #endif
    #endif
}

/* ------------------------------------------------------------------------ */

static void nh_init(nh_ctx *hc, UINT8 *prf_key)
/* Generate nh_key, endian convert and reset to be ready for hashing.   */
{

    kdf(hc->nh_key, prf_key, 0, sizeof(hc->nh_key));
    endian_convert_if_le(hc->nh_key, WORD_LEN, sizeof(hc->nh_key));
    #if (ARCH_KEY_MODIFICATION)
    /* Some specialized code may need the key in an altered format.
     * They will define ARCH_KEY_MODIFICATION == 1 and provide a
     * arch_key_modification(UINT8 *buf, int buflen) function
     */
    arch_key_modification(hc->nh_key, sizeof(hc->nh_key));
    #endif
    nh_reset(hc);
}

/* ------------------------------------------------------------------------ */

static void nh_update(nh_ctx *hc, UINT8 *buf, UINT32 nbytes)
/* Incorporate nbytes of data into a nh_ctx, buffer whatever is not an    */
```

```
/* even multiple of HASH_BUF_BYTES.                                    */
{
    UINT32 i,j;

    j = hc->next_data_empty;
    if ((j + nbytes) >= HASH_BUF_BYTES) {
        if (j) {
            i = HASH_BUF_BYTES - j;
            memcpy(hc->data+j, buf, i);
            nh_transform(hc,hc->data,HASH_BUF_BYTES);
            nbytes -= i;
            buf += i;
            hc->bytes_hashed += HASH_BUF_BYTES;
        }
        if (nbytes >= HASH_BUF_BYTES) {
            /* i = nbytes - (nbytes % HASH_BUF_BYTES); */
            i = nbytes & ~(HASH_BUF_BYTES - 1);
            nh_transform(hc, buf, i); /* buf could be poorly aligned */
            nbytes -= i;
            buf += i;
            hc->bytes_hashed += i;
        }
        j = 0;
    }
    memcpy(hc->data + j, buf, nbytes);
    hc->next_data_empty = j + nbytes;
}

/* ------------------------------------------------------------------- */

static void zero_pad(UINT8 *p, int nbytes)
{
/* Write "nbytes" of zeroes, beginning at "p" */
    if (nbytes >= (int)sizeof(UWORD)) {
        while ((int)p % sizeof(UWORD)) {
            *p = 0;
            nbytes--;
            p++;
        }
        while (nbytes >= (int)sizeof(UWORD)) {
            *(UWORD *)p = 0;
            nbytes -= sizeof(UWORD);
            p += sizeof(UWORD);
        }
    }
    while (nbytes) {
        *p = 0;
        nbytes--;
        p++;
    }
}

/* ------------------------------------------------------------------- */
```

```
static void nh_final(nh_ctx *hc, UINT8 *result)
/* After passing some number of data buffers to nh_update() for integration
 * into an NH context, nh_final is called to produce a hash result. If any
 * bytes are in the buffer hc->data, incorporate them into the
 * NH context. Finally, add into the NH accumulation "state" the total number
 * of bits hashed. The resulting numbers are written to the buffer "result".
 */
{
    int nh_len, nbits;

    if (hc->next_data_empty) {
        nh_len = ((hc->next_data_empty + (L1_PAD_BOUNDARY - 1)) &
                                                ~(L1_PAD_BOUNDARY - 1));
        zero_pad(hc->data + hc->next_data_empty,
                                          nh_len - hc->next_data_empty);
        nh_transform(hc, hc->data, nh_len);
        hc->bytes_hashed += hc->next_data_empty;
    }
    nbits = (hc->bytes_hashed << 3);
    ((LARGE_UWORD *)result)[0] = ((LARGE_UWORD *)hc->state)[0] + nbits;
    #if (PREFIX_STREAMS > 1)
    ((LARGE_UWORD *)result)[1] = ((LARGE_UWORD *)hc->state)[1] + nbits;
    #if (PREFIX_STREAMS > 2)
    ((LARGE_UWORD *)result)[2] = ((LARGE_UWORD *)hc->state)[2] + nbits;
    #if (PREFIX_STREAMS > 3)
    ((LARGE_UWORD *)result)[3] = ((LARGE_UWORD *)hc->state)[3] + nbits;
    #if (PREFIX_STREAMS > 4)
    ((LARGE_UWORD *)result)[4] = ((LARGE_UWORD *)hc->state)[4] + nbits;
    ((LARGE_UWORD *)result)[5] = ((LARGE_UWORD *)hc->state)[5] + nbits;
    #if (PREFIX_STREAMS > 6)
    ((LARGE_UWORD *)result)[6] = ((LARGE_UWORD *)hc->state)[6] + nbits;
    ((LARGE_UWORD *)result)[7] = ((LARGE_UWORD *)hc->state)[7] + nbits;
    #endif
    #endif
    #endif
    #endif
    #endif
    nh_reset(hc);
}

/* ------------------------------------------------------------------------ */

static void nh(nh_ctx *hc, UINT8 *buf, UINT32 padded_len,
               UINT32 unpadded_len, UINT8 *result)
/* All-in-one nh_update() and nh_final() equivalent.
 * Assumes that padded_len is divisible by L1_PAD_BOUNDARY and result is
 * well aligned
 */
{
    UINT32 nbits;

    /* Initialize the hash state */
    nbits = (unpadded_len << 3);
```

```
    ((LARGE_UWORD *)result)[0] = nbits;
    #if (PREFIX_STREAMS > 1)
    ((LARGE_UWORD *)result)[1] = nbits;
    #if (PREFIX_STREAMS > 2)
    ((LARGE_UWORD *)result)[2] = nbits;
    #if (PREFIX_STREAMS > 3)
    ((LARGE_UWORD *)result)[3] = nbits;
    #if (PREFIX_STREAMS > 4)
    ((LARGE_UWORD *)result)[4] = nbits;
    ((LARGE_UWORD *)result)[5] = nbits;
    #if (PREFIX_STREAMS > 6)
    ((LARGE_UWORD *)result)[6] = nbits;
    ((LARGE_UWORD *)result)[7] = nbits;
    #endif
    #endif
    #endif
    #endif
    #endif

    nh_aux(hc->nh_key, buf, result, padded_len);
}

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Begin UHASH Section ---------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* UHASH is a multi-layered algorithm. Data presented to UHASH is first
 * hashed by NH. The NH output is then hashed by a polynomial-hash layer
 * unless the initial data to be hashed is short. After the polynomial-
 * layer, an inner-product hash is used to produce the final UHASH output.
 *
 * UHASH provides two interfaces, one all-at-once and another where data
 * buffers are presented sequentially. In the sequential interface, the
 * UHASH client calls the routine uhash_update() as many times as necessary.
 * When there is no more data to be fed to UHASH, the client calls
 * uhash_final() which
 * calculates the UHASH output. Before beginning another UHASH calculation
 * the uhash_reset() routine must be called. The all-at-once UHASH routine,
 * uhash(), is equivalent to the sequence of calls uhash_update() and
 * uhash_final(); however it is optimized and should be
 * used whenever the sequential interface is not necessary.
 *
 * The routine uhash_init() initializes the uhash_ctx data structure and
 * must be called once, before any other UHASH routine.
 */

/* ---------------------------------------------------------------------- */
/* ----- PDF Constants and uhash_ctx ------------------------------------- */
/* ---------------------------------------------------------------------- */
```

```
/* ---------------------------------------------------------------------- */

typedef struct uhash_ctx {
    nh_ctx hash;                         /* Hash context for L1 NH hash    */
    /* Extra stuff for the WORD_LEN == 2 case, where a polyhash tansition
     * may occur between p32 and p64
     */
    #if (WORD_LEN == 2)
    UINT32 poly_key_4[PREFIX_STREAMS]; /* p32 Poly keys                    */
    UINT64 poly_store[PREFIX_STREAMS]; /* To buffer NH-16 output for p64   */
    int poly_store_full;               /* Flag for poly_store              */
    UINT32 poly_invocations;           /* Number of p32 words hashed       */
    #endif
    UINT64 poly_key_8[PREFIX_STREAMS];    /* p64 poly keys                 */
    UINT64 poly_accum[PREFIX_STREAMS];    /* poly hash result              */
    LARGE_UWORD ip_keys[PREFIX_STREAMS*4];/* Inner-product keys            */
    SMALL_UWORD ip_trans[PREFIX_STREAMS]; /* Inner-product translation     */
    UINT32 msg_len;               /* Total length of data passed to uhash */
} uhash_ctx;

/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */
#if (USE_C_ONLY || ! ARCH_POLY)
/* ---------------------------------------------------------------------- */

/* The polynomial hashes use Horner's rule to evaluate a polynomial one
 * word at a time. As described in the specification, poly32 and poly64
 * require keys from special domains. The following impelementations exploit
 * the special domains to avoid overflow. The results are not guaranteed to
 * be within Z_p32 and Z_p64, but the Inner-Product hash implementation
 * patches any errant values.
 */
static UINT32 poly32(UINT32 cur, UINT32 key, UINT32 data)
/* requires 29 bit keys */
{
    UINT64 t;
    UINT32 hi, lo;

    t = cur * (UINT64)key;
    hi = (UINT32)(t >> 32);
    lo = (UINT32)t;
    hi *= 5;
    lo += hi;
    if (lo < hi)
        lo += 5;
    lo += data;
    if (lo < data)
        lo += 5;
    return lo;
}

static UINT64 poly64(UINT64 cur, UINT64 key, UINT64 data)
```

```
{
    UINT32 key_hi = (UINT32)(key >> 32),
           key_lo = (UINT32)key,
           cur_hi = (UINT32)(cur >> 32),
           cur_lo = (UINT32)cur,
           x_lo,
           x_hi;
    UINT64 X,T,res;

    X =  MUL64(key_hi, cur_lo) + MUL64(cur_hi, key_lo);
    x_lo = (UINT32)X;
    x_hi = (UINT32)(X >> 32);

    res = (MUL64(key_hi, cur_hi) + x_hi) * 59 + MUL64(key_lo, cur_lo);

    T = ((UINT64)x_lo << 32);
    res += T;
    if (res < T)
        res += 59;

    res += data;
    if (res < data)
        res += 59;

    return res;
}

#endif



#if (WORD_LEN == 2)

/* Although UMAC is specified to use a ramped polynomial hash scheme, this
 * impelemtation does not handle all ramp levels. When WORD_LEN is 2, we only
 * handle the p32 and p64 modulus polynomial calculations. Because we don't
 * handle the ramp up to p128 modulus in this implementation, we are limited
 * to 2^31 poly_hash() invocations per stream (for a total capacity of 2^41
 * bytes per tag input to UMAC).
 */
const UINT32 poly_crossover = (1ul << 9);

static void poly_hash(uhash_ctx_t hc, UINT32 data[])
{
    int i;

    if (hc->poly_invocations < poly_crossover) { /* Use poly32 */
        for (i = 0; i < PREFIX_STREAMS; i++) {
            /* If the data passed in is out of range, we hash a marker
             * and then hash the data offset to be in range.
             */
            if (data[i] >= (p32-1)) {
                hc->poly_accum[i] = poly32((UINT32)hc->poly_accum[i],
                                           hc->poly_key_4[i], p32-1);
```

```
                hc->poly_accum[i] = poly32((UINT32)hc->poly_accum[i],
                                           hc->poly_key_4[i], (data[i] - 5));
            } else
                hc->poly_accum[i] = poly32((UINT32)hc->poly_accum[i],
                                           hc->poly_key_4[i], data[i]);
        }
    } else if (hc->poly_invocations > poly_crossover) {      /* Use poly64 */
        /* We must buffer every other 32-bit word to build up a 64-bit one */
        if ( ! hc->poly_store_full) {
            for (i = 0; i < PREFIX_STREAMS; i++) {
                hc->poly_store[i] = ((UINT64)data[i]) << 32;
            }
            hc->poly_store_full = 1;
        } else {
            for (i = 0; i < PREFIX_STREAMS; i++) {
                /* If the data passed in is out of range, we hash a marker
                 * and then hash the data offset to be in range.
                 */
                if ((UINT32)(hc->poly_store[i] >> 32) == 0xffffffffu) {
                    hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                               hc->poly_key_8[i], p64 - 1);
                    hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                           hc->poly_key_8[i],
                                           (hc->poly_store[i] + data[i] - 59));
                } else {
                    hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                           hc->poly_key_8[i],
                                           hc->poly_store[i] + data[i]);
                }
                hc->poly_store_full = 0;
            }
        }
    } else { /* (hc->poly_invocations == poly_crossover) */
        /* Implement the ramp from p32 to p64 hashing    */
        for (i = 0; i < PREFIX_STREAMS; i++) {
            hc->poly_accum[i] = poly64(1, hc->poly_key_8[i],
                                       hc->poly_accum[i]);
            hc->poly_store[i] = ((UINT64)data[i]) << 32;
        }
        hc->poly_store_full = 1;
    }
    hc->poly_invocations += 1;
}

#else /* WORD_LEN == 4 */

/* Although UMAC is specified to use a ramped polynomial hash scheme, this
 * impelemtation does not handle all ramp levels. When WORD_LEN is 4, we only
 * handle the p64 modulus polynomial calculations. Because we don't handle
 * the ramp up to p128 modulus in this implementation, we are limited to
 * 2^14 poly_hash() invocations per stream (for a total capacity of 2^24
 * bytes per tag input to UMAC).
 */
static void poly_hash(uhash_ctx_t hc, UINT32 data_in[])
```

```
{
/* This routine is simpler than that above because there is no ramping. */
    int i;
    UINT64 *data=(UINT64*)data_in;

    for (i = 0; i < PREFIX_STREAMS; i++) {
        if ((UINT32)(data[i] >> 32) == 0xffffffffu) {
            hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                       hc->poly_key_8[i], p64 - 1);
            hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                       hc->poly_key_8[i], (data[i] - 59));
        } else {
            hc->poly_accum[i] = poly64(hc->poly_accum[i],
                                       hc->poly_key_8[i], data[i]);
        }
    }
}

#endif

/* ------------------------------------------------------------------------ */

#if (WORD_LEN == 4)

/* ------------------------------------------------------------------------ */
#if (USE_C_ONLY || ! ARCH_IP)
/* ------------------------------------------------------------------------ */

/* The final step in UHASH is an inner-product hash. The poly hash
 * produces a result not neccesarily WORD_LEN bytes long. The inner-
 * product hash breaks the polyhash output into 16-bit chunks and
 * multiplies each with a 36 bit key.
 */
static UINT64 ip_aux(UINT64 t, UINT64 *ipkp, UINT64 data)
{
    t = t + ipkp[0] * (UINT64)(UINT16)(data >> 48);
    t = t + ipkp[1] * (UINT64)(UINT16)(data >> 32);
    t = t + ipkp[2] * (UINT64)(UINT16)(data >> 16);
    t = t + ipkp[3] * (UINT64)(UINT16)(data);

    return t;
}

static UINT32 ip_reduce_p36(UINT64 t)
{
/* Divisionless modular reduction */
    UINT64 ret;

    ret = (t & m36) + 5 * (t >> 36);
    if (ret >= p36)
        ret -= p36;

    /* return least significant 32 bits */
    return (UINT32)(ret);
```

```
}

#endif

/* If the data being hashed by UHASH is no longer than L1_KEY_LEN, then
 * the polyhash stage is skipped and ip_short is applied directly to the
 * NH output.
 */
static void ip_short(uhash_ctx_t ahc, UINT8 *nh_res, char *res)
{
    UINT64 t;
    UINT64 *nhp = (UINT64 *)nh_res;

    t  = ip_aux(0,ahc->ip_keys, nhp[0]);
    STORE_UINT32_BIG((UINT32 *)res+0, ip_reduce_p36(t) ^ ahc->ip_trans[0]);
    #if (PREFIX_STREAMS > 1)
    t  = ip_aux(0,ahc->ip_keys+1, nhp[1]);
    STORE_UINT32_BIG((UINT32 *)res+1, ip_reduce_p36(t) ^ ahc->ip_trans[1]);
    #if (PREFIX_STREAMS > 2)
    t  = ip_aux(0,ahc->ip_keys+2, nhp[2]);
    STORE_UINT32_BIG((UINT32 *)res+2, ip_reduce_p36(t) ^ ahc->ip_trans[2]);
    #if (PREFIX_STREAMS > 3)
    t  = ip_aux(0,ahc->ip_keys+3, nhp[3]);
    STORE_UINT32_BIG((UINT32 *)res+3, ip_reduce_p36(t) ^ ahc->ip_trans[3]);
    #if (PREFIX_STREAMS > 4)
    t  = ip_aux(0,ahc->ip_keys+4, nhp[4]);
    STORE_UINT32_BIG((UINT32 *)res+4, ip_reduce_p36(t) ^ ahc->ip_trans[4]);
    t  = ip_aux(0,ahc->ip_keys+5, nhp[5]);
    STORE_UINT32_BIG((UINT32 *)res+5, ip_reduce_p36(t) ^ ahc->ip_trans[5]);
    #if (PREFIX_STREAMS > 6)
    t  = ip_aux(0,ahc->ip_keys+6, nhp[6]);
    STORE_UINT32_BIG((UINT32 *)res+6, ip_reduce_p36(t) ^ ahc->ip_trans[6]);
    t  = ip_aux(0,ahc->ip_keys+7, nhp[7]);
    STORE_UINT32_BIG((UINT32 *)res+7, ip_reduce_p36(t) ^ ahc->ip_trans[7]);
    #endif
    #endif
    #endif
    #endif
    #endif
}

/* If the data being hashed by UHASH is longer than L1_KEY_LEN, then
 * the polyhash stage is not skipped and ip_long is applied to the
 * polyhash output.
 */
static void ip_long(uhash_ctx_t ahc, char *res)
{
    int i;
    UINT64 t;

    for (i = 0; i < PREFIX_STREAMS; i++) {
        /* fix polyhash output not in Z_p64 */
        if (ahc->poly_accum[i] >= p64)
            ahc->poly_accum[i] -= p64;
```

```
        t  = ip_aux(0,ahc->ip_keys+i, ahc->poly_accum[i]);
        STORE_UINT32_BIG((UINT32 *)res+i,
                         ip_reduce_p36(t) ^ ahc->ip_trans[i]);
    }
}


/* ---------------------------------------------------------------------- */
#elif (WORD_LEN == 2)
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
#if (USE_C_ONLY || ! ARCH_IP)
/* ---------------------------------------------------------------------- */

/* The final step in UHASH is an inner-product hash. The poly hash
 * produces a result not neccesarily WORD_LEN bytes long. The inner-
 * product hash breaks the polyhash output into 16-bit chunks and
 * multiplies each with a 19 bit key.
 */
static UINT64 ip_aux(UINT64 t, UINT32 *ipkp, UINT32 data)
{
    t = t + MUL64(ipkp[0], (data >> 16));
    t = t + MUL64(ipkp[1], (UINT16)(data));

    return t;
}

static UINT16 ip_reduce_p19(UINT64 t)
{
/* Divisionless modular reduction */
    UINT32 ret;

    ret = ((UINT32)t & m19) + (UINT32)(t >> 19);
    if (ret >= p19)
        ret -= p19;

    /* return least significant 16 bits */
    return (UINT16)(ret);
}


#endif


/* If the data being hashed by UHASH is no longer than L1_KEY_LEN, then
 * the polyhash stage is skipped and ip_short is applied directly to the
 * NH output.
 */
static void ip_short(uhash_ctx_t ahc, UINT8 *nh_res, char *res)
{
    UINT64 t;
    UINT32 *nhp = (UINT32 *)nh_res;
```

```
    t  = ip_aux(0,ahc->ip_keys+2, nhp[0]);
    STORE_UINT16_BIG((UINT16 *)res+0,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[0]));
    #if (PREFIX_STREAMS > 1)
    t  = ip_aux(0,ahc->ip_keys+6, nhp[1]);
    STORE_UINT16_BIG((UINT16 *)res+1,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[1]));
    #if (PREFIX_STREAMS > 2)
    t  = ip_aux(0,ahc->ip_keys+10, nhp[2]);
    STORE_UINT16_BIG((UINT16 *)res+2,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[2]));
    #if (PREFIX_STREAMS > 3)
    t  = ip_aux(0,ahc->ip_keys+14, nhp[3]);
    STORE_UINT16_BIG((UINT16 *)res+3,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[3]));
    #if (PREFIX_STREAMS > 4)
    t  = ip_aux(0,ahc->ip_keys+18, nhp[4]);
    STORE_UINT16_BIG((UINT16 *)res+4,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[4]));
    t  = ip_aux(0,ahc->ip_keys+22, nhp[5]);
    STORE_UINT16_BIG((UINT16 *)res+5,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[5]));
    #if (PREFIX_STREAMS > 6)
    t  = ip_aux(0,ahc->ip_keys+26, nhp[6]);
    STORE_UINT16_BIG((UINT16 *)res+6,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[6]));
    t  = ip_aux(0,ahc->ip_keys+30, nhp[7]);
    STORE_UINT16_BIG((UINT16 *)res+7,
            (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[7]));
    #endif
    #endif
    #endif
    #endif
    #endif
}

/* If the data being hashed by UHASH is longer than L1_KEY_LEN, then
 * the polyhash stage is not skipped and ip_long is applied to the
 * polyhash output.
 */
static void ip_long(uhash_ctx_t ahc, char *res)
{
    int i;
    UINT64 t;

    if (ahc->poly_invocations > poly_crossover) { /* hash 64 bits */
        for (i = 0; i < PREFIX_STREAMS; i++) {
            if (ahc->poly_accum[i] >= p64)
                ahc->poly_accum[i] -= p64;
            t = ip_aux(0,ahc->ip_keys+i*4,(UINT32)(ahc->poly_accum[i] >> 32));
            t = ip_aux(t,ahc->ip_keys+i*4+2,(UINT32)ahc->poly_accum[i]);
            /* Store result big endian for sonsistancy across architectures */
            STORE_UINT16_BIG((UINT16 *)res+i,
                             (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[i]));
```

```
        }
    } else {                                    /* hash 32 bits */
        for (i = 0; i < PREFIX_STREAMS; i++) {
            if (ahc->poly_accum[i] >= p32)
                ahc->poly_accum[i] -= p32;
            t  = ip_aux(0,ahc->ip_keys+i*4+2,(UINT32)ahc->poly_accum[i]);
            STORE_UINT16_BIG((UINT16 *)res+i,
                             (UINT16)(ip_reduce_p19(t) ^ ahc->ip_trans[i]));
        }
    }
}

#endif

/* ------------------------------------------------------------------------ */

/* Reset uhash context for next hash session */
int uhash_reset(uhash_ctx_t pc)
{
    nh_reset(&pc->hash);
    pc->msg_len = 0;
    #if (WORD_LEN == 2)
    pc->poly_invocations = 0;
    pc->poly_store_full = 0;
    #endif
    pc->poly_accum[0] = 1;
    #if (PREFIX_STREAMS > 1)
    pc->poly_accum[1] = 1;
    #if (PREFIX_STREAMS > 2)
    pc->poly_accum[2] = 1;
    #if (PREFIX_STREAMS > 3)
    pc->poly_accum[3] = 1;
    #if (PREFIX_STREAMS > 4)
    pc->poly_accum[4] = 1;
    pc->poly_accum[5] = 1;
    #if (PREFIX_STREAMS > 6)
    pc->poly_accum[6] = 1;
    pc->poly_accum[7] = 1;
    #endif
    #endif
    #endif
    #endif
    #endif
    #endif
    return 1;
}

/* ------------------------------------------------------------------------ */

/* Given a pointer to the internal key needed by kdf() and a uhash context,
 * initialize the NH context and generate keys needed for poly and inner-
 * product hashing. All keys are endian adjusted in memory so that native
 * loads cause correct keys to be in registers during calculation.
 */
static void uhash_init(uhash_ctx_t ahc, UINT8 *prf_key)
```

```
{
    int i;
    UINT8 buf[(8*PREFIX_STREAMS+4)*sizeof(LARGE_UWORD)];

    /* Zero the entire uhash context */
    memset(ahc, 0, sizeof(uhash_ctx));

    /* Initialize the L1 hash */
    nh_init(&ahc->hash, prf_key);

    /* Setup L2 hash variables */
    kdf(buf, prf_key, 1, sizeof(buf));    /* Fill buffer with index 1 key */
    for (i = 0; i < PREFIX_STREAMS; i++) {
        /* Fill keys from the buffer, skipping bytes in the buffer not
         * used by this implementation. Endian reverse the keys if on a
         * little-endian computer.
         */
        #if (WORD_LEN == 2)
        memcpy(ahc->poly_key_4+i, buf+28*i, 4);
        memcpy(ahc->poly_key_8+i, buf+28*i+4, 8);
        endian_convert_if_le(ahc->poly_key_4+i, 4, 4);
        ahc->poly_key_4[i] &= 0x1fffffff;  /* Mask to special domain */
        #elif (WORD_LEN == 4)
        memcpy(ahc->poly_key_8+i, buf+24*i, 8);
        #endif
        endian_convert_if_le(ahc->poly_key_8+i, 8, 8);
        /* Mask the 64-bit keys to their special domain */
        ahc->poly_key_8[i] &= ((UINT64)0x01ffffffu << 32) + 0x01ffffffu;
        ahc->poly_accum[i] = 1;  /* Our polyhash prepends a non-zero word */
    }

    /* Setup L3-1 hash variables */
    kdf(buf, prf_key, 2, sizeof(buf)); /* Fill buffer with index 2 key */
    for (i = 0; i < PREFIX_STREAMS; i++)
          memcpy(ahc->ip_keys+4*i, buf+(8*i+4)*sizeof(LARGE_UWORD),
                                                 4*sizeof(LARGE_UWORD));
    endian_convert_if_le(ahc->ip_keys, sizeof(LARGE_UWORD),
                                                  sizeof(ahc->ip_keys));
    for (i = 0; i < PREFIX_STREAMS*4; i++)
        #if (WORD_LEN == 2)
        ahc->ip_keys[i] %= p19;  /* Bring into Z_p19 */
        #elif (WORD_LEN == 4)
        ahc->ip_keys[i] %= p36;  /* Bring into Z_p36 */
        #endif

    /* Setup L3-2 hash variables    */
    /* Fill buffer with index 3 key */
    kdf(ahc->ip_trans, prf_key, 3, PREFIX_STREAMS * sizeof(SMALL_UWORD));
    endian_convert_if_le(ahc->ip_trans, sizeof(SMALL_UWORD),
                         PREFIX_STREAMS * sizeof(SMALL_UWORD));
}

/* ---------------------------------------------------------------------- */
```

```
uhash_ctx_t uhash_alloc(char key[])
{
/* Allocate memory and force to a 16-byte boundary. */
    uhash_ctx_t ctx;
    char bytes_to_add;
    UINT32 prf_key[RC6_TABLE_WORDS];

    ctx = (uhash_ctx_t)malloc(sizeof(uhash_ctx)+ALLOC_BOUNDARY);
    if (ctx) {
        if (ALLOC_BOUNDARY) {
            bytes_to_add = ALLOC_BOUNDARY - ((int)ctx & (ALLOC_BOUNDARY -1));
            ctx = (uhash_ctx_t)((char *)ctx + bytes_to_add);
            *((char *)ctx - 1) = bytes_to_add;
        }
        RC6_SETUP((UINT8 *)key, prf_key);  /* Intitialize the block-cipher */
        uhash_init(ctx, (UINT8 *)prf_key);
    }
    return (ctx);
}

/* ---------------------------------------------------------------------- */

int uhash_free(uhash_ctx_t ctx)
{
/* Free memory allocated by uhash_alloc */
    char bytes_to_sub;

    if (ctx) {
        if (ALLOC_BOUNDARY) {
            bytes_to_sub = *((char *)ctx - 1);
            ctx = (uhash_ctx_t)((char *)ctx - bytes_to_sub);
        }
        free(ctx);
    }
    return (1);
}

/* ---------------------------------------------------------------------- */

int uhash_update(uhash_ctx_t ctx, char *input, long len)
/* Given len bytes of data, we parse it into L1_KEY_LEN chunks and
 * hash each one with NH, calling the polyhash on each NH output.
 */
{
    UWORD bytes_hashed, bytes_remaining;
    UINT8 nh_result[PREFIX_STREAMS*sizeof(LARGE_UWORD)];

    if (ctx->msg_len + len <= L1_KEY_LEN) {
        nh_update(&ctx->hash, (UINT8 *)input, len);
        ctx->msg_len += len;
    } else {

         bytes_hashed = ctx->msg_len % L1_KEY_LEN;
         if (ctx->msg_len == L1_KEY_LEN)
```

```
            bytes_hashed = L1_KEY_LEN;

        if (bytes_hashed + len >= L1_KEY_LEN) {

            /* If some bytes have been passed to the hash function      */
            /* then we want to pass at most (L1_KEY_LEN - bytes_hashed) */
            /* bytes to complete the current nh_block.                  */
            if (bytes_hashed) {
                bytes_remaining = (L1_KEY_LEN - bytes_hashed);
                nh_update(&ctx->hash, (UINT8 *)input, bytes_remaining);
                nh_final(&ctx->hash, nh_result);
                ctx->msg_len += bytes_remaining;
                poly_hash(ctx,(UINT32 *)nh_result);
                len -= bytes_remaining;
                input += bytes_remaining;
            }

            /* Hash directly from input stream if enough bytes */
            while (len >= L1_KEY_LEN) {
                nh(&ctx->hash, (UINT8 *)input, L1_KEY_LEN,
                                    L1_KEY_LEN, nh_result);
                ctx->msg_len += L1_KEY_LEN;
                len -= L1_KEY_LEN;
                input += L1_KEY_LEN;
                poly_hash(ctx,(UINT32 *)nh_result);
            }
        }

        /* pass remaining < L1_KEY_LEN bytes of input data to NH */
        if (len) {
            nh_update(&ctx->hash, (UINT8 *)input, len);
            ctx->msg_len += len;
        }
     }

    return (1);
}

/* ------------------------------------------------------------------------ */

int uhash_final(uhash_ctx_t ctx, char *res)
/* Incorporate any pending data, pad, and generate tag */
{
    UINT8 nh_result[PREFIX_STREAMS*sizeof(LARGE_UWORD)];

    if (ctx->msg_len > L1_KEY_LEN) {
        if (ctx->msg_len % L1_KEY_LEN) {
            nh_final(&ctx->hash, nh_result);
            poly_hash(ctx,(UINT32 *)nh_result);
        }
        /* If WORD_LEN == 2 and we have ramped-up to p64 in the polyhash,
         * then we must pad the data passed to poly64 with a 1 bit and then
         * zero bits up to the next multiple of 64 bits.
         */
```

```
        #if (WORD_LEN == 2)
        if (ctx->poly_invocations > poly_crossover) {
            UINT32 tmp[PREFIX_STREAMS];
            int i;
            for (i = 0; i < PREFIX_STREAMS; i++)
                tmp[i] = 0x80000000u;
            poly_hash(ctx,tmp);
            if (ctx->poly_store_full) {
                for (i = 0; i < PREFIX_STREAMS; i++)
                    tmp[i] = 0;
                poly_hash(ctx,tmp);
            }
        }
        #endif
        ip_long(ctx, res);
    } else {
        nh_final(&ctx->hash, nh_result);
        ip_short(ctx,nh_result, res);
    }
    uhash_reset(ctx);
    return (1);
}

/* ---------------------------------------------------------------------- */

int uhash(uhash_ctx_t ahc, char *msg, long len, char *res)
/* assumes that msg is in a writable buffer of length divisible by */
/* L1_PAD_BOUNDARY. Bytes beyond msg[len] may be zeroed.           */
/* Does not handle zero length message                             */
{
    UINT8 nh_result[PREFIX_STREAMS*sizeof(LARGE_UWORD)];
    UINT32 nh_len;
    int extra_zeroes_needed;

    /* If the message to be hashed is no longer than L1_HASH_LEN, we skip
     * the polyhash.
     */
    if (len <= L1_KEY_LEN) {
        nh_len = ((len + (L1_PAD_BOUNDARY - 1)) & ~(L1_PAD_BOUNDARY - 1));
        extra_zeroes_needed = nh_len - len;
        zero_pad((UINT8 *)msg + len, extra_zeroes_needed);
        nh(&ahc->hash, (UINT8 *)msg, nh_len, len, nh_result);
        ip_short(ahc,nh_result, res);
    } else {
        /* Otherwise, we hash each L1_KEY_LEN chunk with NH, passing the NH
         * output to poly_hash().
         */
        do {
            nh(&ahc->hash, (UINT8 *)msg, L1_KEY_LEN, L1_KEY_LEN, nh_result);
            poly_hash(ahc,(UINT32 *)nh_result);
            len -= L1_KEY_LEN;
            msg += L1_KEY_LEN;
        } while (len >= L1_KEY_LEN);
        if (len) {
```

```
            nh_len = ((len + (L1_PAD_BOUNDARY - 1)) & ~(L1_PAD_BOUNDARY - 1));
            extra_zeroes_needed = nh_len - len;
            zero_pad((UINT8 *)msg + len, extra_zeroes_needed);
            nh(&ahc->hash, (UINT8 *)msg, nh_len, len, nh_result);
            poly_hash(ahc,(UINT32 *)nh_result);
        }
        /* If WORD_LEN == 2 and we have ramped-up to p64 in the polyhash,
         * then we must pad the data passed to poly64 with a 1 bit and then
         * zero bits up to the next multiple of 64 bits.
         */
        #if (WORD_LEN == 2)
        if (ahc->poly_invocations > poly_crossover) {
            UINT32 tmp[PREFIX_STREAMS];
            int i;
            for (i = 0; i < PREFIX_STREAMS; i++)
                tmp[i] = 0x80000000u;
            poly_hash(ahc,tmp);
            if (ahc->poly_store_full) {
                for (i = 0; i < PREFIX_STREAMS; i++)
                    tmp[i] = 0;
                poly_hash(ahc,tmp);
            }
        }
        #endif
        ip_long(ahc, res);
    }

    uhash_reset(ahc);
    return 1;
}

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- Begin UMAC Section ----------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* The UMAC interface has two interfaces, an all-at-once interface where
 * the entire message to be authenticated is passed to UMAC in one buffer,
 * and a sequential interface where the message is presented a bit at a time.
 * The all-at-once is more optimaized than the sequential version and should
 * be preferred when the sequential interface is not required.
 */
typedef struct umac_ctx {
    uhash_ctx hash;          /* Hash function for message compression    */
    pdf_ctx pdf;             /* PDF for hashed output                    */
} umac_ctx;

/* ---------------------------------------------------------------------- */

int umac_reset(umac_ctx_t ctx)
/* Reset the hash function to begin a new authentication.        */
{
    uhash_reset(&ctx->hash);
```

```
    return (1);
}

/* ---------------------------------------------------------------------- */

int umac_delete(umac_ctx_t ctx)
/* Deallocate the ctx structure */
{
    char bytes_to_sub;

    if (ctx) {
        if (ALLOC_BOUNDARY) {
            bytes_to_sub = *((char *)ctx - 1);
            ctx = (umac_ctx_t)((char *)ctx - bytes_to_sub);
        }
        free(ctx);
    }
    return (1);
}

/* ---------------------------------------------------------------------- */

umac_ctx_t umac_new(char key[])
/* Dynamically allocate a umac_ctx struct, initialize variables,
 * generate subkeys from key. Align to 16-byte boundary.
 */
{
    umac_ctx_t ctx;
    char bytes_to_add;
    UINT32 prf_key[RC6_TABLE_WORDS];

    ctx = (umac_ctx_t)malloc(sizeof(umac_ctx)+ALLOC_BOUNDARY);
    if (ctx) {
        if (ALLOC_BOUNDARY) {
            bytes_to_add = ALLOC_BOUNDARY - ((int)ctx & (ALLOC_BOUNDARY - 1));
            ctx = (umac_ctx_t)((char *)ctx + bytes_to_add);
            *((char *)ctx - 1) = bytes_to_add;
        }
        RC6_SETUP((UINT8 *)key, prf_key);
        pdf_init(&ctx->pdf, (UINT8 *)prf_key);
        uhash_init(&ctx->hash, (UINT8 *)prf_key);
    }

    return (ctx);
}

/* ---------------------------------------------------------------------- */

int umac_final(umac_ctx_t ctx, char tag[], char nonce[8])
/* Incorporate any pending data, pad, and generate tag */
{
    /* pdf_gen_xor writes OUTPUT_STREAMS * WORD_LEN bytes to its output
     * buffer, so if PREFIX_STREAMS == OUTPUT_STREAMS, we write directly
     * to the buffer supplied by the client. Otherwise we use a temporary
```

```
     * buffer.
     */
    #if ((PREFIX_STREAMS == OUTPUT_STREAMS) || HASH_ONLY)
    UINT8 *uhash_result = (UINT8 *)tag;
    #else
    UINT8 uhash_result[UMAC_OUTPUT_LEN];
    #endif

    uhash_final(&ctx->hash, (char *)uhash_result);
    #if ( ! HASH_ONLY)
    pdf_gen_xor(&ctx->pdf, (UINT8 *)nonce, uhash_result);
    #endif

    #if ((PREFIX_STREAMS != OUTPUT_STREAMS) && ! HASH_ONLY)
    memcpy(tag,uhash_result,UMAC_PREFIX_LEN);
    #endif

    return (1);
}

/* ---------------------------------------------------------------------- */

int umac_update(umac_ctx_t ctx, char *input, long len)
/* Given len bytes of data, we parse it into L1_KEY_LEN chunks and   */
/* hash each one, calling the PDF on the hashed output whenever the hash- */
/* output buffer is full.                                            */
{
    uhash_update(&ctx->hash, input, len);
    return (1);
}

/* ---------------------------------------------------------------------- */

int umac(umac_ctx_t ctx, char *input,
         long len, char tag[],
         char nonce[8])
/* All-in-one version simply calls umac_update() and umac_final().        */
{
    #if ((PREFIX_STREAMS == OUTPUT_STREAMS) || HASH_ONLY)
    UINT8 *uhash_result = (UINT8 *)tag;
    #else
    UINT8 uhash_result[UMAC_OUTPUT_LEN];
    #endif

    uhash(&ctx->hash, input, len, (char *)uhash_result);
    #if ( ! HASH_ONLY)
    pdf_gen_xor(&ctx->pdf, (UINT8 *)nonce, uhash_result);
    #endif

    #if ((PREFIX_STREAMS != OUTPUT_STREAMS) && ! HASH_ONLY)
    memcpy(tag,uhash_result,UMAC_PREFIX_LEN);
    #endif

    return (1);
```

```
}

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ----- End UMAC Section ------------------------------------------------ */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/* If RUN_TESTS is defined non-zero, then we define a main() function and */
/* run some verification and speed tests.                                 */

#if RUN_TESTS

#include <stdio.h>
#include <time.h>

static void pbuf(void *buf, UWORD n, char *s)
{
    UWORD i;
    UINT8 *cp = (UINT8 *)buf;

    if (n <= 0 || n >= 30)
        n = 30;

    if (s)
        printf("%s: ", s);

    for (i = 0; i < n; i++)
        printf("%2X", (unsigned char)cp[i]);
    printf("\n");
}

static void primitive_verify(void)
{
    #if (UMAC_KEY_LEN == 16)
    char key[] = "\x01\x23\x45\x67\x89\xAB\xCD\xEF"
                 "\x01\x12\x23\x34\x45\x56\x67\x78";
    char res[] = "524E192F4715C6231F51F6367EA43F18";
    #elif (UMAC_KEY_LEN == 32)
    char key[] = "\x01\x23\x45\x67\x89\xAB\xCD\xEF"
                 "\x01\x12\x23\x34\x45\x56\x67\x78"
                 "\x89\x9A\xAB\xBC\xCD\xDE\xEF\xF0"
                 "\x10\x32\x54\x76\x98\xba\xdc\xfe";
    char res[] = "C8241816F0D7E48920AD16A1674E5D48";
    #endif
    char pt[]  = "\x02\x13\x24\x35\x46\x57\x68\x79"
                 "\x8A\x9B\xAC\xBD\xCE\xDF\xE0\xF1";
    UINT32 k1[RC6_TABLE_WORDS];

    RC6_SETUP((UINT8 *)key, k1);
    RC6(k1, pt, pt);
    printf("\nRC6 Test\n");
    pbuf(pt, 16, "Digest is       ");
    printf("Digest should be: %s\n", res);
```

```
}

static void umac_verify(void)
{
    umac_ctx_t ctx;
    char *data_ptr;
    int data_len = 4 * 1024;
    char nonce[8] = {0};
    char tag[21] = {0};
    char tag2[21] = {0};
    int bytes_over_boundary, i, j;
    int inc[] = {1,99,512};

    /* Initialize Memory and UMAC */
    nonce[7] = 1;
    data_ptr = (char *)malloc(data_len + 16);
    bytes_over_boundary = (int)data_ptr & (16 - 1);
    if (bytes_over_boundary != 0)
        data_ptr += (16 - bytes_over_boundary);
    for (i = 0; i < data_len; i++)
        data_ptr[i] = (i%127) * (i%123) % 127;
    ctx = umac_new("abcdefghijklmnopqrstuvwxyz");

    umac(ctx, data_ptr, data_len, tag, nonce);
    umac_reset(ctx);

    #if ((WORD_LEN == 2) && (UMAC_OUTPUT_LEN == 8) && \
         (L1_KEY_LEN == 1024) && (UMAC_KEY_LEN == 16))
    printf("UMAC-2/8/1024/16/LITTLE/SIGNED Test\n");
    pbuf(tag, PREFIX_STREAMS*WORD_LEN, "Tag is                    ");
    printf("Tag should be a prefix of: %s\n", "E658CB58FAC91FB7");
    #elif ((WORD_LEN == 4) && (UMAC_OUTPUT_LEN == 8) && \
         (L1_KEY_LEN == 1024) && (UMAC_KEY_LEN == 16))
    printf("UMAC-4/8/1024/16/LITTLE/UNSIGNED Test\n");
    pbuf(tag, PREFIX_STREAMS*WORD_LEN, "Tag is                    ");
    printf("Tag should be a prefix of: %s\n", "EE7DA51D16 CE7C2");
    #endif



    printf("\nVerifying consistancy of single- and"
           " multiple-call interfaces.\n");
    for (i = 1; i < (int)(sizeof(inc)/sizeof(inc[0])); i++) {
            for (j = 0; j <= data_len-inc[i]; j+=inc[i])
                umac_update(ctx, data_ptr+j, inc[i]);
            umac_final(ctx, tag, nonce);
            umac_reset(ctx);

            umac(ctx, data_ptr, (data_len/inc[i])*inc[i], tag2, nonce);
            umac_reset(ctx);
            nonce[7]++;

            if (memcmp(tag,tag2,sizeof(tag)))
                printf("\ninc = %d data_len = %d failed!\n",
```

```
                        inc[i], data_len);
    }
    printf("Done.\n");
    umac_delete(ctx);
}


static double run_cpb_test(umac_ctx_t ctx, int nbytes, char *data_ptr,
                           int data_len, double hz)
{
    clock_t ticks;
    double secs;
    char nonce[8] = {0};
    char tag[UMAC_PREFIX_LEN+1] = {0}; /* extra char for null terminator */
    long total_mbs;
    long iters_per_tag, remaining;
    long tag_iters, i, j;

    if (nbytes < 16)
        total_mbs = 2;
    if (nbytes < 32)
        total_mbs = 10;
    else if (nbytes < 64)
        total_mbs = 50;
    else if (nbytes < 256)
        total_mbs = 150;
    else if (nbytes < 1024)
        total_mbs = 250;
    else
        total_mbs = 500;

    tag_iters = (total_mbs * 1024 * 1024) / (nbytes) + 1;

    if (nbytes <= data_len) {

        i = tag_iters;
        umac(ctx, data_ptr, nbytes, tag, nonce);
        ticks = clock();
        do {
            umac(ctx, data_ptr, nbytes, tag, nonce);
            nonce[7] += 1;
        } while (--i);
        ticks = clock() - ticks;

    } else {

        i = tag_iters;
        iters_per_tag = nbytes / data_len;
        remaining = nbytes % data_len;
        umac_update(ctx, data_ptr, data_len);
        umac_final(ctx, tag, nonce);
        ticks = clock();
        do {
            j = iters_per_tag;
```

```
            do {
                umac_update(ctx, data_ptr, data_len);
            } while (--j);
            if (remaining)
                umac_update(ctx, data_ptr, remaining);
            umac_final(ctx, tag, nonce);
            nonce[7] += 1;
        } while (--i);
        ticks = clock() - ticks;

    }

    secs = (double)ticks / CLOCKS_PER_SEC;
    return (secs * (hz/(tag_iters*nbytes)));
}

static void speed_test(void)
{
    umac_ctx_t ctx;
    char *data_ptr;
    int data_len;
    double hz;
    double cpb;
    int bytes_over_boundary, i;
    int length_range_low = 1;
    int length_range_high = 0;
    int length_pts[] = {43,64,256,1024,1500,256*1024};

    /* hz and data_len must be set appropriately for your system
     * for optimal results.
     */
    #if  (__GNUC__ && __i386__)
    hz = ((double)7.0e8);
    data_len = 4096;
    #elif  (_M_IX86)
    hz = ((double)4.5e8);
    data_len = 4096;
    #elif ((__MRC__ || __MWERKS__) && __POWERPC__)
    hz = ((double)4.33e8);
    data_len = 8192;
    #else
    #error -- unsupported platform
    #endif

    /* Allocate memory and align to 16-byte multiple */
    data_ptr = (char *)malloc(data_len + 16);
    bytes_over_boundary = (int)data_ptr & (16 - 1);
    if (bytes_over_boundary != 0)
        data_ptr += (16 - bytes_over_boundary);
    for (i = 0; i < data_len; i++)
        data_ptr[i] = (i*i) % 128;
    ctx = umac_new("abcdefghijklmnopqrstuvwxyz");

    printf("\n");
```

```
    if (length_range_low < length_range_high) {
        for (i = length_range_low; i <= length_range_high; i++) {
            cpb = run_cpb_test(ctx, i, data_ptr, data_len, hz);
            printf("Authenticating %8d byte messages: %5.2f cpb.\n", i, cpb);
        }
    }

    if (sizeof(length_pts) > 0) {
        for (i = 0; i < (int)(sizeof(length_pts)/sizeof(int)); i++) {
            cpb = run_cpb_test(ctx, length_pts[i], data_ptr, data_len, hz);
            printf("Authenticating %8d byte messages: %5.2f cpb.\n",
                                length_pts[i], cpb);
        }
    }
    umac_delete(ctx);
}

int main(void)
{
    umac_verify();
    primitive_verify();
    speed_test();
    /* printf("Push return to continue\n"); getchar(); */
    return (1);
}

#endif
```

B.3 UMAC Accelleration File for Intel IA-32 Architectures

```
/* ---------------------------------------------------------------------
 *
 * umac.c -- C Implementation UMAC Message Authentication
 *
 * Version 0.04 of draft-krovetz-umac-00.txt -- 2000 August
 *
 * For a full description of UMAC message authentication see the UMAC
 * world-wide-web page at http://www.cs.ucdavis.edu/~rogaway/umac
 * Please report bugs and suggestions to the UMAC webpage.
 *
 * Copyright (c) 1999-2000 Ted Krovetz (tdk@acm.org)
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee, is hereby granted,
 * provided that the above copyright notice appears in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation, and that the names of the University of
 * California and Ted Krovetz not be used in advertising or publicity
 * pertaining to distribution of the software without specific,
 * written prior permission.
 *
 * The Regents of the University of California and Ted Krovetz disclaim
 * all warranties with regard to this software, including all implied
 * warranties of merchantability and fitness.  In no event shall the
 * University of California or Ted Krovetz be liable for any special,
 * indirect or consequential damages or any damages whatsoever resulting
 * from loss of use, data or profits, whether in an action of contract,
 * negligence or other tortious action, arising out of or in connection
 * with the use or performance of this software.
 *
 * ------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
 * Rules for writing an architecture specific include file
 *
 * - For any "class" of functions written here (eg. ARCH_ROTL), all
 *   functions in the class must be written here.
 * - For each "class" written, define the class macro as 1
 *   (eg. #define ARCH_ROTL 1).
 * - This file is included because we are using extensions to ANSI C,
 *   but you must distinguish between "intrinsic" and "intrinsic+asm"
 *   extensions. This is easily done by writing this file in three
 *   sections: (1) intrinsic only functions, (2) intrinsic functions for
 *   which there exists an assembly equivalent and (3) assembly functions.
 *   If we are to do "intrinsic" extensions, then (1) and (2) should be
 *   compiled, otherwise if we are "intrinsic+asm", then (1) and (3).
 *   The assumption is that for speed, C < C+intrinsics < C+assembly.
 * ------------------------------------------------------------------- */

#define SSE2               0  /* Streaming SIMD 2 available on P4         */
#define INTEL_INTRINSICS   0  /* Intel's MMX and SSE intrinsics           */
```

```
#pragma warning(disable: 4731)  /* Turn off "ebp manipulation" warning    */

/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* First define routines which are only written using compiler intrinsics */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */
#define ARCH_ROTL  1
#if     ARCH_ROTL
/* ---------------------------------------------------------------------- */
#define ROTL32_VAR(r,n)   _rotl(r,n)
#define ROTL32_CONST(r,n) _rotl(r,n)
/* ---------------------------------------------------------------------- */
#endif /* ARCH_ROTL */
/* ---------------------------------------------------------------------- */


/* ----------------------------------------------------------------------
 * ----------------------------------------------------------------------
 * ----------------------------------------------------------------------
 * Second define routines which are written using compiler intrinsics but
 * which have assembly equivalents in the third section.
 * ----------------------------------------------------------------------
 * ---------------------------------------------------------------------- */
#if ( ! USE_C_AND_ASSEMBLY)  /* Intrinsics only allowed */
/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */
#define ARCH_ENDIAN_LS  1
#if     ARCH_ENDIAN_LS
/* ---------------------------------------------------------------------- */

static UINT32 LOAD_UINT32_REVERSED(void *ptr)
{
    UINT32 temp = *(UINT32 *)ptr;
    temp = (ROTL32_CONST(temp,8 ) & 0x00FF00FF) |
           (ROTL32_CONST(temp,24) & 0xFF00FF00);
    return temp;
}

static void STORE_UINT32_REVERSED(void *ptr, UINT32 x)
{
    UINT32 temp = x;
    temp = (ROTL32_CONST(temp,8 ) & 0x00FF00FF) |
           (ROTL32_CONST(temp,24) & 0xFF00FF00);
    *(UINT32 *)ptr = temp;
}
```

```
static UINT16 LOAD_UINT16_REVERSED(void *ptr)
{
    UINT16 temp = *(UINT16 *)ptr;
    temp = (temp >> 8) | (temp << 8);
    return temp;
}

static void STORE_UINT16_REVERSED(void *ptr, UINT16 x)
{
    UINT16 temp = x;
    temp = (temp >> 8) | (temp << 8);
    *(UINT16 *)ptr = temp;
}

/* ---------------------------------------------------------------------- */
#endif /* ARCH_ENDIAN_LS */
/* ---------------------------------------------------------------------- */


#if ((WORD_LEN == 2) && INTEL_INTRINSICS)

/* ---------------------------------------------------------------------- */
#define ARCH_NH16  1
#if     ARCH_NH16
/* ---------------------------------------------------------------------- */

#if (SSE2)  /* 128-bit vector registers */

#include "emmintrin.h"

#define NH_STEP(k0,k1,acc1) \
    m0 = _mm_add_epi16(k0,d0); \
    m1 = _mm_add_epi16(k1,d1); \
    m2 = _mm_madd_epi16(m0,m1); \
    acc1 = _mm_add_epi32(acc1, m2)

/* ---------------------------------------------------------------------- */

static void nh_aux_4(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[2];
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

    __m128i *key = (__m128i *)kp;
    __m128i *data = (__m128i *)dp;
    __m128i acc;
    __m128i d0,d1,k0,k1,m0,m1,m2;

    acc = _mm_setzero_si128();

    do {
        k0 = _mm_load_si128(key + 0);
        k1 = _mm_load_si128(key + 1);
```

```
        d0 = _mm_load_si128(data + 0);
        d1 = _mm_load_si128(data + 1);

        NH_STEP(k0,k1,acc);

        key += 2;
        data += 2;
    } while (--iters);

    acc = _mm_add_epi32(acc, _mm_srli_si128(acc, 8));
    _mm_storel_epi64((__m128i *)t,acc);
    p[0] += t[0] + t[1];
}

/* ---------------------------------------------------------------------- */

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[4];
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

    __m128i *key = (__m128i *)kp;
    __m128i *data = (__m128i *)dp;
    __m128i acc1,acc2;
    __m128i d0,d1,k0,k1,k2,m0,m1,m2;

    acc1 = _mm_setzero_si128();
    acc2 = _mm_setzero_si128();

    do {
        k0 = _mm_load_si128(key + 0);
        k1 = _mm_load_si128(key + 1);
        k2 = _mm_load_si128(key + 2);
        d0 = _mm_load_si128(data + 0);
        d1 = _mm_load_si128(data + 1);

        NH_STEP(k0,k1,acc1);
        NH_STEP(k1,k2,acc2);

        key += 2;
        data += 2;
    } while (--iters);

    acc1 = _mm_add_epi32(acc1, _mm_srli_si128(acc1, 8));
    _mm_storel_epi64((__m128i *)t,acc1);
    p[0] += t[0] + t[1];
    acc2 = _mm_add_epi32(acc2, _mm_srli_si128(acc2, 8));
    _mm_storel_epi64((__m128i *)(t+2),acc2);
    p[1] += t[2] + t[3];
}

/* ---------------------------------------------------------------------- */
```

```
static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[8];
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

    __m128i *key = (__m128i *)kp;
    __m128i *data = (__m128i *)dp;
    __m128i acc1,acc2,acc3,acc4;
    __m128i d0,d1,k0,k1,k2,k3,k4,m0,m1,m2;

    acc1 = _mm_setzero_si128();
    acc2 = _mm_setzero_si128();
    acc3 = _mm_setzero_si128();
    acc4 = _mm_setzero_si128();

    do {
        k0 = _mm_load_si128(key + 0);
        k1 = _mm_load_si128(key + 1);
        k2 = _mm_load_si128(key + 2);
        k3 = _mm_load_si128(key + 3);
        k4 = _mm_load_si128(key + 4);
        d0 = _mm_load_si128(data + 0);
        d1 = _mm_load_si128(data + 1);

        NH_STEP(k0,k1,acc1);
        NH_STEP(k1,k2,acc2);
        NH_STEP(k2,k3,acc3);
        NH_STEP(k3,k4,acc4);

        key += 2;
        data += 2;
    } while (--iters);

    acc1 = _mm_add_epi32(acc1, _mm_srli_si128(acc1, 8));
    _mm_storel_epi64((__m128i *)t,acc1);
    p[0] += t[0] + t[1];
    acc2 = _mm_add_epi32(acc2, _mm_srli_si128(acc2, 8));
    _mm_storel_epi64((__m128i *)(t+2),acc2);
    p[1] += t[2] + t[3];
    acc3 = _mm_add_epi32(acc3, _mm_srli_si128(acc3, 8));
    _mm_storel_epi64((__m128i *)(t+4),acc3);
    p[2] += t[4] + t[5];
    acc4 = _mm_add_epi32(acc4, _mm_srli_si128(acc4, 8));
    _mm_storel_epi64((__m128i *)(t+6),acc4);
    p[3] += t[6] + t[7];
}

/* ------------------------------------------------------------------------ */

#else /* No SSE2 */

#include <mmintrin.h>
```

```
/* ---------------------------------------------------------------------- */
static void nh_aux_4(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

  __m64 *key = (__m64 *)kp;
  __m64 *data = (__m64 *)dp;
  __m64 acc = 0;

  do {

    acc = _m_paddd(acc, _m_pmaddwd(_m_paddw(*(key + 0), *(data + 0)),
                                   _m_paddw(*(key + 2), *(data + 2))));
    acc = _m_paddd(acc, _m_pmaddwd(_m_paddw(*(key + 1), *(data + 1)),
                                   _m_paddw(*(key + 3), *(data + 3))));

    key += 4;
    data += 4;
  } while (--iters);

    p[0] += _m_to_int(_m_paddd(acc, _m_psrlqi(acc, 32)));

    _m_empty();
}

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

  __m64 *key = (__m64 *)kp;
  __m64 *data = (__m64 *)dp;
  __m64 acc1 = 0, acc2 = 0;

  do {
    acc1 = _m_paddd(acc1, _m_pmaddwd(_m_paddw(*(key + 0), *(data + 0)),
                                     _m_paddw(*(key + 2), *(data + 2))));
    acc2 = _m_paddd(acc2, _m_pmaddwd(_m_paddw(*(key + 2), *(data + 0)),
                                     _m_paddw(*(key + 4), *(data + 2))));

    acc1 = _m_paddd(acc1, _m_pmaddwd(_m_paddw(*(key + 1), *(data + 1)),
                                     _m_paddw(*(key + 3), *(data + 3))));
    acc2 = _m_paddd(acc2, _m_pmaddwd(_m_paddw(*(key + 3), *(data + 1)),
                                     _m_paddw(*(key + 5), *(data + 3))));

    key += 4;
    data += 4;
  } while (--iters);

    p[0] += _m_to_int(_m_paddd(acc1, _m_psrlqi(acc1, 32)));
    p[1] += _m_to_int(_m_paddd(acc2, _m_psrlqi(acc2, 32)));

    _m_empty();
```

```
}

static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 *p = (UINT32 *)hp;
    UINT32 iters = dlen/32;

  __m64 *key = (__m64 *)kp;
  __m64 *data = (__m64 *)dp;
  __m64 acc1 = 0, acc2 = 0, acc3 = 0, acc4 = 0;

  do {
    acc1 = _m_paddd(acc1, _m_pmaddwd(_m_paddw(*(key + 0), *(data + 0)),
                                     _m_paddw(*(key + 2), *(data + 2))));
    acc2 = _m_paddd(acc2, _m_pmaddwd(_m_paddw(*(key + 2), *(data + 0)),
                                     _m_paddw(*(key + 4), *(data + 2))));
    acc3 = _m_paddd(acc3, _m_pmaddwd(_m_paddw(*(key + 4), *(data + 0)),
                                     _m_paddw(*(key + 6), *(data + 2))));
    acc4 = _m_paddd(acc4, _m_pmaddwd(_m_paddw(*(key + 6), *(data + 0)),
                                     _m_paddw(*(key + 8), *(data + 2))));

    acc1 = _m_paddd(acc1, _m_pmaddwd(_m_paddw(*(key + 1), *(data + 1)),
                                     _m_paddw(*(key + 3), *(data + 3))));
    acc2 = _m_paddd(acc2, _m_pmaddwd(_m_paddw(*(key + 3), *(data + 1)),
                                     _m_paddw(*(key + 5), *(data + 3))));
    acc3 = _m_paddd(acc3, _m_pmaddwd(_m_paddw(*(key + 5), *(data + 1)),
                                     _m_paddw(*(key + 7), *(data + 3))));
    acc4 = _m_paddd(acc4, _m_pmaddwd(_m_paddw(*(key + 7), *(data + 1)),
                                     _m_paddw(*(key + 9), *(data + 3))));

    key += 4;
    data += 4;
  } while (--iters);

    p[0] += _m_to_int(_m_paddd(acc1, _m_psrlqi(acc1, 32)));
    p[1] += _m_to_int(_m_paddd(acc2, _m_psrlqi(acc2, 32)));
    p[2] += _m_to_int(_m_paddd(acc3, _m_psrlqi(acc3, 32)));
    p[3] += _m_to_int(_m_paddd(acc4, _m_psrlqi(acc4, 32)));

    _m_empty();
}


#endif /* SSE2 */
#endif /* ARCH_NH16 */
#endif /* ((WORD_LEN == 2) && INTEL_INTRINSICS) */

#if ((WORD_LEN == 4) && INTEL_INTRINSICS && SSE2)

/* ------------------------------------------------------------------------ */
#define ARCH_NH32  1
#if     ARCH_NH32
/* ------------------------------------------------------------------------ */
```

```
#include "emmintrin.h"

#define NH_STEP(k1,k2,acc) \
    m1 = _mm_add_epi32(k1, d1); \
    m2 = _mm_add_epi32(k2, d2); \
    m3 = _mm_mul_epu32(m1, m2); \
    acc = _mm_add_epi64(acc, m3); \
    m1 = _mm_srli_si128(m1,4); \
    m2 = _mm_srli_si128(m2,4); \
    m3 = _mm_mul_epu32(m1, m2); \
    acc = _mm_add_epi64(acc, m3)


static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT64 *p = (UINT64 *)hp;
    UINT32 len = dlen;

  __m128i *key = (__m128i *)kp;
  __m128i *data = (__m128i *)dp;
  __m128i acc = _mm_loadl_epi64((__m128i const*)p);
  __m128i m1,m2,m3,d1,d2,k1,k2;

  while (len >= 128) {
    d1 = _mm_load_si128(data);
    d2 = _mm_load_si128(data+1);
    k1 = _mm_load_si128(key);
    k2 = _mm_load_si128(key+1);
    NH_STEP(k1,k2,acc);
    d1 = _mm_load_si128(data+2);
    d2 = _mm_load_si128(data+3);
    k1 = _mm_load_si128(key+2);
    k2 = _mm_load_si128(key+3);
    NH_STEP(k1,k2,acc);
    d1 = _mm_load_si128(data+4);
    d2 = _mm_load_si128(data+5);
    k1 = _mm_load_si128(key+4);
    k2 = _mm_load_si128(key+5);
    NH_STEP(k1,k2,acc);
    d1 = _mm_load_si128(data+6);
    d2 = _mm_load_si128(data+7);
    k1 = _mm_load_si128(key+6);
    k2 = _mm_load_si128(key+7);
    NH_STEP(k1,k2,acc);
    key += 8;
    data += 8;
    len -= 128;
  }
  while (len >= 32) {
    d1 = _mm_load_si128(data);
    d2 = _mm_load_si128(data+1);
    k1 = _mm_load_si128(key);
    k2 = _mm_load_si128(key+1);
    NH_STEP(k1,k2,acc);
```

```
    key += 2;
    data += 2;
    len -= 32;
  }

    acc = _mm_add_epi64(acc, _mm_srli_si128(acc, 8));
    _mm_storel_epi64((__m128i *)(p),acc);
}


static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT64 *p = (UINT64 *)hp;
    UINT32 len = dlen;

  __m128i *key = (__m128i *)kp;
  __m128i *data = (__m128i *)dp;
  __m128i acc = _mm_loadl_epi64((__m128i const*)p),
          acc2 = _mm_loadl_epi64((__m128i const*)(p+1));
  __m128i m1,m2,m3,d1,d2,k1,k2,k3;

  while (len >= 128) {
    d1 = _mm_load_si128(data);
    d2 = _mm_load_si128(data+1);
    k1 = _mm_load_si128(key);
    k2 = _mm_load_si128(key+1);
    k3 = _mm_load_si128(key+2);
    NH_STEP(k1,k2,acc);
    NH_STEP(k2,k3,acc2);
    d1 = _mm_load_si128(data+2);
    d2 = _mm_load_si128(data+3);
    k1 = _mm_load_si128(key+2);
    k2 = _mm_load_si128(key+3);
    k3 = _mm_load_si128(key+4);
    NH_STEP(k1,k2,acc);
    NH_STEP(k2,k3,acc2);
    d1 = _mm_load_si128(data+4);
    d2 = _mm_load_si128(data+5);
    k1 = _mm_load_si128(key+4);
    k2 = _mm_load_si128(key+5);
    k3 = _mm_load_si128(key+6);
    NH_STEP(k1,k2,acc);
    NH_STEP(k2,k3,acc2);
    d1 = _mm_load_si128(data+6);
    d2 = _mm_load_si128(data+7);
    k1 = _mm_load_si128(key+6);
    k2 = _mm_load_si128(key+7);
    k3 = _mm_load_si128(key+8);
    NH_STEP(k1,k2,acc);
    NH_STEP(k2,k3,acc2);
    key += 8;
    data += 8;
    len -= 128;
  }
```

```
  while (len >= 32) {
    d1 = _mm_load_si128(data);
    d2 = _mm_load_si128(data+1);
    k1 = _mm_load_si128(key);
    k2 = _mm_load_si128(key+1);
    k3 = _mm_load_si128(key+2);
    NH_STEP(k1,k2,acc);
    NH_STEP(k2,k3,acc2);
    key += 2;
    data += 2;
    len -= 32;
  }

    acc = _mm_add_epi64(acc, _mm_srli_si128(acc, 8));
    _mm_storel_epi64((__m128i *)(p),acc);

    acc2 = _mm_add_epi64(acc2, _mm_srli_si128(acc2, 8));
    _mm_storel_epi64((__m128i *)(p+1),acc2);
}


#endif /* ARCH_NH32 */
#endif /* ((WORD_LEN == 4) && INTEL_INTRINSICS && SSE2) */

/* ---------------------------------------------------------------------
 * ---------------------------------------------------------------------
 * ---------------------------------------------------------------------
 * Third define routines which are written using inline assembly.
 * ---------------------------------------------------------------------
 * ---------------------------------------------------------------------
 * --------------------------------------------------------------------- */
#else /* (USE_C_AND_ASSEMBLY) */
/* --------------------------------------------------------------------- */


/* --------------------------------------------------------------------- */
#define ARCH_ENDIAN_LS  1
/* --------------------------------------------------------------------- */

static UINT32 LOAD_UINT32_REVERSED(void *p)
{
    __asm {
        mov eax, p
        mov eax, [eax]
        bswap eax
    }
}

static void STORE_UINT32_REVERSED(void *p, UINT32 x)
{
    __asm {
        mov eax,x
        bswap eax
        mov ecx, p
```

```
        mov [ecx], eax
    }
}

static UINT16 LOAD_UINT16_REVERSED(void *p)
{
    __asm {
        mov eax, p
        mov ax, [eax]
        rol ax,8
    }
}

static void STORE_UINT16_REVERSED(void *p, UINT16 x)
{
    __asm {
        mov ax,x
        rol ax,8
        mov ecx, p
        mov [ecx], ax
    }
}

/* ---------------------------------------------------------------------- */
#define ARCH_RC6  1
/* ---------------------------------------------------------------------- */
#define RC6_BLOCK(a,b,c,d,n) \
    __asm lea eax, [b+b+1] \
    __asm imul eax,b \
    __asm rol eax, 5 \
    __asm lea ecx, [d+d+1] \
    __asm imul ecx,d \
    __asm rol ecx, 5 \
    __asm xor a,eax \
    __asm rol a,cl \
    __asm add a,n[esi] \
    __asm xor c,ecx \
    __asm mov ecx,eax \
    __asm rol c,cl \
    __asm add c,n+4[esi]

static void RC6(UINT32 S[], void *pt, void *ct)
{
    __asm {
        push ebp
        mov esi, S
        mov ecx, pt
        mov edi,   [ecx]
        mov ebx,  4[ecx]
        mov ebp,  8[ecx]
        mov edx, 12[ecx]
        add ebx,   [esi]
        add edx,  4[esi]
```

```
        RC6_BLOCK(edi,ebx,ebp,edx,8)
        RC6_BLOCK(ebx,ebp,edx,edi,16)
        RC6_BLOCK(ebp,edx,edi,ebx,24)
        RC6_BLOCK(edx,edi,ebx,ebp,32)

        RC6_BLOCK(edi,ebx,ebp,edx,40)
        RC6_BLOCK(ebx,ebp,edx,edi,48)
        RC6_BLOCK(ebp,edx,edi,ebx,56)
        RC6_BLOCK(edx,edi,ebx,ebp,64)

        RC6_BLOCK(edi,ebx,ebp,edx,72)
        RC6_BLOCK(ebx,ebp,edx,edi,80)
        RC6_BLOCK(ebp,edx,edi,ebx,88)
        RC6_BLOCK(edx,edi,ebx,ebp,96)

        RC6_BLOCK(edi,ebx,ebp,edx,104)
        RC6_BLOCK(ebx,ebp,edx,edi,112)
        RC6_BLOCK(ebp,edx,edi,ebx,120)
        RC6_BLOCK(edx,edi,ebx,ebp,128)

        RC6_BLOCK(edi,ebx,ebp,edx,136)
        RC6_BLOCK(ebx,ebp,edx,edi,144)
        RC6_BLOCK(ebp,edx,edi,ebx,152)
        RC6_BLOCK(edx,edi,ebx,ebp,160)

        mov eax, ebp
        pop ebp
        add edi, 168[esi]
        add eax, 172[esi]
        mov ecx, ct
        mov   [ecx], edi
        mov  4[ecx], ebx
        mov  8[ecx], eax
        mov 12[ecx], edx
    }
}



#if (WORD_LEN == 4)
/* ------------------------------------------------------------------------ */
#define ARCH_NH32  1
/* ------------------------------------------------------------------------ */

#if (SSE2)  /* 128-bit vector registers */

/* These macros uses movdqa which requires 16-byte aligned data
 * and key.
 */
#define NH_STEP_128(n) \
    __asm movdqa xmm2, n[ecx] \
    __asm movdqa xmm3, n+16[ecx] \
    __asm movdqa xmm0, n[eax] \
    __asm movdqa xmm1, n+16[eax] \
```

```
    __asm movdqa xmm4, xmm3 \
    __asm paddd xmm2, xmm0 \
    __asm paddd xmm3, xmm1 \
    __asm movdqa xmm5, xmm2 \
    __asm pmuludq xmm2, xmm3 \
    __asm psrldq xmm5, 4 \
    __asm psrldq xmm3, 4 \
    __asm paddq xmm6, xmm2 \
    __asm pmuludq xmm3, xmm5 \
    __asm movdqa xmm5, n+32[ecx] \
    __asm paddd xmm4, xmm0 \
    __asm paddq xmm6, xmm3 \
    __asm paddd xmm5, xmm1 \
    __asm movdqa xmm3, xmm4 \
    __asm pmuludq xmm4, xmm5 \
    __asm psrldq xmm5, 4 \
    __asm psrldq xmm3, 4 \
    __asm pmuludq xmm5, xmm3 \
    __asm paddq xmm7, xmm4 \
    __asm paddq xmm7, xmm5

#define NH_STEP_64(n) \
    __asm movdqa xmm2, n[ecx] \
    __asm movdqa xmm0, n[eax] \
    __asm movdqa xmm3, n+16[ecx] \
    __asm movdqa xmm1, n+16[eax] \
    __asm paddd xmm2, xmm0 \
    __asm paddd xmm3, xmm1 \
    __asm movdqa xmm5, xmm2 \
    __asm pmuludq xmm2, xmm3 \
    __asm psrldq xmm5, 4 \
    __asm psrldq xmm3, 4 \
    __asm paddq xmm6, xmm2 \
    __asm pmuludq xmm3, xmm5 \
    __asm paddq xmm6, xmm3


static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    __asm{
        mov edx,dlen
        mov eax,hp
        sub edx, 128
        movq xmm6, [eax]
        mov eax, dp
        mov ecx, kp
        jb label2
label1:
        NH_STEP_64(0)
        NH_STEP_64(32)
        NH_STEP_64(64)
        NH_STEP_64(96)
        add eax, 128
        add ecx, 128
```

```
        sub edx, 128
        jnb label1
label2:
        add edx,128
        je label4
label3:
        NH_STEP_64(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
        mov eax,hp
        movdqa xmm0,xmm6
        psrldq xmm0, 8
        paddq xmm6, xmm0
        movq [eax], xmm6
    }
}

static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    __asm{
        mov edx,dlen
        mov eax,hp
        sub edx, 128
        movq xmm6, [eax]
        movq xmm7, 8[eax]
        mov eax, dp
        mov ecx, kp
        jb label2
label1:
        NH_STEP_128(0)
        NH_STEP_128(32)
        NH_STEP_128(64)
        NH_STEP_128(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
        add edx,128
        je label4
label3:
        NH_STEP_128(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
        mov eax,hp
        movdqa xmm0,xmm6
        movdqa xmm1,xmm7
        psrldq xmm0, 8
```

```
        psrldq xmm1, 8
        paddq xmm6, xmm0
        paddq xmm7, xmm1
        movq [eax], xmm6
        movq 8[eax], xmm7
    }
}

#else /* no SSE2 */

#define NH_STEP(n) \
    __asm mov eax,n[ebx]   \
    __asm mov edx,n+16[ebx] \
    __asm add eax,n[ecx]   \
    __asm add edx,n+16[ecx] \
    __asm mul edx         \
    __asm add esi,eax      \
    __asm adc edi,edx


static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
  __asm{
      push ebp
      mov ecx,kp
      mov ebx,dp
      mov eax,hp
      mov ebp,dlen
      sub ebp,128
      mov esi,[eax]
      mov edi,4[eax]
      jb label2     /* if 0 */
label1:
    NH_STEP(0)
    NH_STEP(4)
    NH_STEP(8)
    NH_STEP(12)
    NH_STEP(32)
    NH_STEP(36)
    NH_STEP(40)
    NH_STEP(44)
    NH_STEP(64)
    NH_STEP(68)
    NH_STEP(72)
    NH_STEP(76)
    NH_STEP(96)
    NH_STEP(100)
    NH_STEP(104)
    NH_STEP(108)
      add ecx,128
      add ebx,128
      sub ebp,128
      jnb label1
label2:
```

```
      add ebp,128
      je label4
label3:
    NH_STEP(0)
    NH_STEP(4)
    NH_STEP(8)
    NH_STEP(12)
      add ecx,32
      add ebx,32
      sub ebp,32
      jne label3
label4:
      pop ebp
      mov eax,hp
      mov [eax],esi
      mov 4[eax],edi
   }
}

/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */


static void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    nh_aux_8(kp,dp,hp,dlen);
    nh_aux_8((UINT8 *)kp+16,dp,(UINT8 *)hp+8,dlen);
}

#endif
#endif


/* ---------------------------------------------------------------------- */
#if (WORD_LEN == 2)
/* ---------------------------------------------------------------------- */
#define ARCH_NH16  1
/* ---------------------------------------------------------------------- */

#if (SSE2)  /* 128-bit vector registers */

#define NH_STEP_128(n)  \
    __asm movdqa xmm0,n+0[eax] \
    __asm movdqa xmm1,n+16[eax] \
    __asm movdqa xmm2,n+0[ecx] \
    __asm movdqa xmm3,n+16[ecx] \
    __asm paddw xmm2,xmm0 \
    __asm paddw xmm3,xmm1 \
    __asm pmaddwd xmm2,xmm3 \
    __asm psubw xmm3,xmm1         \
    __asm paddd xmm4,xmm2 \
    __asm movdqa xmm2,n+32[ecx] \
    __asm paddw xmm3,xmm0 \
```

```
    __asm paddw xmm2,xmm1 \
    __asm pmaddwd xmm3,xmm2 \
    __asm psubw xmm2,xmm1         \
    __asm paddd xmm5,xmm3 \
    __asm movdqa xmm3,n+48[ecx] \
    __asm paddw xmm2,xmm0 \
    __asm paddw xmm3,xmm1 \
    __asm pmaddwd xmm2,xmm3 \
    __asm psubw xmm3,xmm1         \
    __asm paddd xmm6,xmm2 \
    __asm movdqa xmm2,n+64[ecx] \
    __asm paddw xmm3,xmm0 \
    __asm paddw xmm2,xmm1 \
    __asm pmaddwd xmm3,xmm2 \
    __asm paddd xmm7,xmm3

#define NH_STEP_64(n)          \
    __asm movdqa xmm0,n+0[eax]  \
    __asm movdqa xmm1,n+16[eax] \
    __asm movdqa xmm2,n+0[ecx]  \
    __asm movdqa xmm3,n+16[ecx] \
    __asm paddw xmm2,xmm0         \
    __asm paddw xmm3,xmm1         \
    __asm pmaddwd xmm2,xmm3       \
    __asm paddd xmm4,xmm2         \
    __asm psubw xmm3,xmm1         \
    __asm movdqa xmm2,n+32[ecx] \
    __asm paddw xmm3,xmm0         \
    __asm paddw xmm2,xmm1         \
    __asm pmaddwd xmm2,xmm3       \
    __asm paddd xmm5,xmm2

#define NH_STEP_32(n)     \
    __asm movdqa xmm0,n+0[ecx]     \
    __asm movdqa xmm1,n+16[ecx]    \
    __asm movdqa xmm2,n+0[eax]     \
    __asm movdqa xmm3,n+16[eax]    \
    __asm paddw xmm2,xmm0         \
    __asm paddw xmm3,xmm1         \
    __asm pmaddwd xmm2,xmm3       \
    __asm paddd xmm7,xmm2

static void nh_aux_4(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[2];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor xmm7, xmm7
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
```

```
label1:
        NH_STEP_32(0)
        NH_STEP_32(32)
        NH_STEP_32(64)
        NH_STEP_32(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
      add edx,128
      je label4
label3:
       NH_STEP_32(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movdqa xmm0, xmm7
      psrldq xmm7, 8
      paddd xmm7, xmm0
      movq t, xmm7
  }

  p[0] += (t[0] + t[1]);
}


/* ---------------------------------------------------------------------- */

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[4];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor xmm4, xmm4
      pxor xmm5, xmm5
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
label1:
        NH_STEP_64(0)
        NH_STEP_64(32)
        NH_STEP_64(64)
        NH_STEP_64(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
```

```
      add edx,128
      je label4
label3:
       NH_STEP_64(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movdqa xmm0, xmm4
      movdqa xmm1, xmm5
      psrldq xmm4, 8
      psrldq xmm5, 8
      paddd xmm0, xmm4
      paddd xmm1, xmm5
      movq t, xmm0
      movq t+8, xmm1
  }

  p[0] += (t[0] + t[1]);
  p[1] += (t[2] + t[3]);
}

/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */

 void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[8];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor xmm4, xmm4
      pxor xmm5, xmm5
      pxor xmm6, xmm6
      pxor xmm7, xmm7
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
label1:
        NH_STEP_128(0)
        NH_STEP_128(32)
        NH_STEP_128(64)
        NH_STEP_128(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
      add edx,128
```

```
      je label4
label3:
       NH_STEP_128(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movdqa xmm0, xmm4
      movdqa xmm1, xmm5
      movdqa xmm2, xmm6
      movdqa xmm3, xmm7
      psrldq xmm4, 8
      psrldq xmm5, 8
      psrldq xmm6, 8
      psrldq xmm7, 8
      paddd xmm4, xmm0
      paddd xmm5, xmm1
      paddd xmm6, xmm2
      paddd xmm7, xmm3
      movq t   , xmm4
      movq t+ 8, xmm5
      movq t+16, xmm6
      movq t+24, xmm7
  }

  p[0] += (t[0] + t[1]);
  p[1] += (t[2] + t[3]);
  p[2] += (t[4] + t[5]);
  p[3] += (t[6] + t[7]);
}

#else /* no SSE */

#define NH_STEP_128(n)  \
    __asm movq mm0,n+0[eax] \
    __asm movq mm1,n+16[eax] \
    __asm movq mm2,n+0[ecx] \
    __asm movq mm3,n+16[ecx] \
    __asm paddw mm2,mm0 \
    __asm paddw mm3,mm1 \
    __asm pmaddwd mm2,mm3 \
    __asm paddd mm4,mm2 \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+32[ecx] \
    __asm paddw mm3,mm0 \
    __asm paddw mm2,mm1 \
    __asm pmaddwd mm3,mm2 \
    __asm paddd mm5,mm3 \
    __asm psubw mm2,mm1         \
    __asm movq mm3,n+48[ecx] \
    __asm paddw mm2,mm0 \
    __asm paddw mm3,mm1 \
    __asm pmaddwd mm2,mm3 \
```

```
    __asm paddd mm6,mm2 \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+64[ecx] \
    __asm paddw mm3,mm0 \
    __asm paddw mm2,mm1 \
    __asm pmaddwd mm3,mm2 \
    __asm paddd mm7,mm3 \
    __asm movq mm0,n+8[eax] \
    __asm movq mm1,n+24[eax] \
    __asm movq mm2,n+8[ecx] \
    __asm movq mm3,n+24[ecx] \
    __asm paddw mm2,mm0 \
    __asm paddw mm3,mm1 \
    __asm pmaddwd mm2,mm3 \
    __asm paddd mm4,mm2 \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+40[ecx] \
    __asm paddw mm3,mm0 \
    __asm paddw mm2,mm1 \
    __asm pmaddwd mm3,mm2 \
    __asm paddd mm5,mm3 \
    __asm psubw mm2,mm1         \
    __asm movq mm3,n+56[ecx] \
    __asm paddw mm2,mm0 \
    __asm paddw mm3,mm1 \
    __asm pmaddwd mm2,mm3 \
    __asm paddd mm6,mm2 \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+72[ecx] \
    __asm paddw mm3,mm0 \
    __asm paddw mm2,mm1 \
    __asm pmaddwd mm3,mm2 \
    __asm paddd mm7,mm3

#define NH_STEP_64(n)          \
    __asm movq mm0,n+0[eax]  \
    __asm movq mm1,n+16[eax] \
    __asm movq mm2,n+0[ecx]  \
    __asm movq mm3,n+16[ecx] \
    __asm paddw mm2,mm0         \
    __asm paddw mm3,mm1         \
    __asm pmaddwd mm2,mm3       \
    __asm paddd mm4,mm2         \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+32[ecx] \
    __asm paddw mm3,mm0         \
    __asm paddw mm2,mm1         \
    __asm pmaddwd mm2,mm3       \
    __asm paddd mm5,mm2         \
    __asm movq mm0,n+8[eax]  \
    __asm movq mm1,n+24[eax] \
    __asm movq mm2,n+8[ecx]  \
    __asm movq mm3,n+24[ecx] \
    __asm paddw mm2,mm0         \
```

```
    __asm paddw mm3,mm1         \
    __asm pmaddwd mm2,mm3       \
    __asm paddd mm4,mm2         \
    __asm psubw mm3,mm1         \
    __asm movq mm2,n+40[ecx] \
    __asm paddw mm3,mm0         \
    __asm paddw mm2,mm1         \
    __asm pmaddwd mm2,mm3       \
    __asm paddd mm5,mm2

#define NH_STEP_32(n)     \
    __asm movq mm0,n+0[ecx]     \
    __asm movq mm1,n+16[ecx]    \
    __asm movq mm2,n+0[eax]     \
    __asm movq mm3,n+16[eax]    \
    __asm paddw mm2,mm0         \
    __asm paddw mm3,mm1         \
    __asm movq mm4,n+8[ecx]     \
    __asm movq mm6,n+24[ecx]    \
    __asm pmaddwd mm2,mm3       \
    __asm movq mm5,n+8[eax]     \
    __asm movq mm3,n+24[eax]    \
    __asm paddd mm7,mm2         \
    __asm paddw mm5,mm4         \
    __asm paddw mm3,mm6         \
    __asm pmaddwd mm5,mm3       \
    __asm paddd mm7,mm5

static void nh_aux_4(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[2];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor mm7, mm7
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
label1:
        NH_STEP_32(0)
        NH_STEP_32(32)
        NH_STEP_32(64)
        NH_STEP_32(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
      add edx,128
      je label4
label3:
       NH_STEP_32(0)
```

```
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movq t   , mm7
      emms
  }

  p[0] += (t[0] + t[1]);
}


/* ---------------------------------------------------------------------- */

static void nh_aux_8(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[4];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor mm4, mm4
      pxor mm5, mm5
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
label1:
        NH_STEP_64(0)
        NH_STEP_64(32)
        NH_STEP_64(64)
        NH_STEP_64(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
      add edx,128
      je label4
label3:
       NH_STEP_64(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movq t   , mm4
      movq t+ 8, mm5
      emms
  }

  p[0] += (t[0] + t[1]);
  p[1] += (t[2] + t[3]);
```

```
}

/* ---------------------------------------------------------------------- */


/* ---------------------------------------------------------------------- */

 void nh_aux_16(void *kp, void *dp, void *hp, UINT32 dlen)
{
    UINT32 t[8];
    UINT32 *p = (UINT32 *)hp;

  __asm{
      mov edx,dlen
      pxor mm4, mm4
      pxor mm5, mm5
      pxor mm6, mm6
      pxor mm7, mm7
      sub edx, 128
      mov eax, dp
      mov ecx, kp
      jb label2
label1:
        NH_STEP_128(0)
        NH_STEP_128(32)
        NH_STEP_128(64)
        NH_STEP_128(96)
        add eax, 128
        add ecx, 128
        sub edx, 128
        jnb label1
label2:
      add edx,128
      je label4
label3:
       NH_STEP_128(0)
        add eax, 32
        add ecx, 32
        sub edx, 32
        jne label3
label4:
      movq t   , mm4
      movq t+ 8, mm5
      movq t+16, mm6
      movq t+24, mm7
      emms
  }

  p[0] += (t[0] + t[1]);
  p[1] += (t[2] + t[3]);
  p[2] += (t[4] + t[5]);
  p[3] += (t[6] + t[7]);
}
```

```
#endif
#endif


#if (WORD_LEN == 4)
/* ------------------------------------------------------------------------ */
#define ARCH_IP  1
/* ------------------------------------------------------------------------ */

static UINT64 ip_aux(UINT64 t, UINT64 *ipkp, UINT64 data)
{
    UINT32 data_hi = (UINT32)(data >> 32),
           data_lo = (UINT32)(data),
           t_hi = (UINT32)(t >> 32),
           t_lo = (UINT32)(t);
    __asm{
        mov edi, ipkp
        mov ebx,data_hi
        mov ecx,data_lo
        mov esi, t_lo
        mov edx, t_hi
        push ebp
        mov ebp,edx
        mov eax,ebx
        shr eax,16
        mul DWORD PTR 0[edi]
        add esi,eax
        adc ebp,edx
        mov eax,ebx
        shr eax,16
        mul DWORD PTR 4[edi]
        add ebp,eax

        movzx eax,bx
        mul DWORD PTR 8[edi]
        add esi,eax
        adc ebp,edx
        movzx eax,bx
        mul DWORD PTR 12[edi]
        add ebp,eax

        mov eax,ecx
        shr eax,16
        mul DWORD PTR 16[edi]
        add esi,eax
        adc ebp,edx
        mov eax,ecx
        shr eax,16
        mul DWORD PTR 20[edi]
        add ebp,eax

        movzx eax,cx
        mul DWORD PTR 24[edi]
```

```
        add esi,eax
        adc ebp,edx
        movzx eax,cx
        mul DWORD PTR 28[edi]
        lea edx,[eax+ebp]
        mov eax,esi
        pop ebp
        /* MSVC returns UINT64 in edx:eax */
    }
}

static UINT32 ip_reduce_p36(UINT64 t)
{
    UINT32 t_hi = (UINT32)(t >> 32),
           t_lo = (UINT32)(t);
    __asm{
        mov edx,t_hi
        mov eax,t_lo
        mov edi,edx
        and edx,15
        shr edi,4
        lea edi,[edi+edi*4]
        add eax,edi
        adc edx,0
        cmp edx,0xf
        jb skip_sub
        ja do_sub
        cmp eax,0xfffffffb
        jb skip_sub
do_sub:
        sub eax, 0xfffffffb
        /* sbb  edx, 0xf We don't return the high word */
skip_sub:
    }
}

#else  /* WORD_LEN == 2 */

/* ---------------------------------------------------------------------- */
#define ARCH_IP  1
/* ---------------------------------------------------------------------- */

static UINT64 ip_aux(UINT64 t, UINT32 *ipkp, UINT32 data)
{
    UINT32 t_hi = (UINT32)(t >> 32),
           t_lo = (UINT32)(t);
    __asm{
        mov ecx, data
        mov ebx, ipkp
        mov esi,t_lo
        mov edi,t_hi
        mov eax,ecx
        shr eax,16
        mul DWORD PTR [ebx]
```

```
        add esi,eax
        adc edi,edx
        movzx eax,cx
        mul DWORD PTR 4[ebx]
        add eax,esi
        adc edx,edi
        /* MSVC returns UINT64 in edx:eax */
    }
}

 UINT16 ip_reduce_p19(UINT64 t)
{
    UINT32 t_hi = (UINT32)(t >> 32),
           t_lo = (UINT32)(t);
    __asm{
        mov edx,t_hi
        mov eax,t_lo
        shld edx,eax,13
        and eax,0x7ffff
        add eax,edx
        mov edx,eax
        sub eax,0x7ffff
        cmovc eax,edx
    }
}

#endif
#endif
```

Appendix C

UMAC (1999) Specification

In 1999 we presented a paper and posted a specification and working code to the UMAC website [10]. The specification was to be revised into an IETF Internet-Draft and is conatined in this appendix. Instead of pursuing the standardization of the MAC specified in this document, we revisd the algorithms into those specified in Appendix A.

==========

UMAC -- Message Authentication Code using Universal Hashing

Version 0.25 (July 8, 1999) -- DRAFT

Black, Halevi, Hevia, Krawczyk, Krovetz, Rogaway

==========

Contents

==========

1. Introduction

This specification describes how to generate an authentication tag (also called a "MAC" or "tag") using the UMAC message authentication code. Typically the authentication tag will be transmitted along with a message

and a nonce to allow the receiver of the message to verify the message's authenticity. Generation and verification of the authentication tag depends, in addition to the message and nonce, on a secret key (typically, shared by sender and receiver).

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

For many applications, sufficient speed is already obtained from an algorithm such as HMAC-SHA1 [RFC-2104, BCK] or the CBC-MAC of a block cipher [ANSI-9.9, ISO-9797]. But for the most speed-demanding applications, UMAC may be a better choice: An optimized implementation of UMAC can achieve peak performance which is about an order of magnitude faster than what can be achieved with HMAC or CBC-MAC. Moreover, UMAC offers a tradeoff between forging probability and speed, by which it is possible to trade forging probability for speed. Such a tradeoff is not present in other constructions. Finally, UMAC also enjoys better analytical security properties than other constructions.

Closely associated to this specification is the paper [BHKKR]. See that paper for a description of the ideas which underlie this algorithm, for performance data, and for a proof of the correctness and maximal forging probability of UMAC.

UMAC is a "parameterized" algorithm. This means that various low-level choices, like the endian convention and the underlying cryptographic primitive, have not been fixed. One must choose values for these parameters before the authentication tag generated by UMAC (for a given message, key, and nonce) becomes well-defined. The advantage of giving a parameterized algorithm is that we have not been forced to make "generic" compromises. Instead, an application that uses UMAC can choose the parameters which best suit its requirements or implementation environment. Examples of specific parameter choices are provided in this document for common environments; in particular, this can help the adoption of the algorithm by standardized applications and protocols. A concrete specification, including reference code is provided in [TO-BE-WRITTEN].

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

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

* The key for a pseudorandom function. This is where one needs a cryptographic hardness assumption. The pseudorandom function may be obtained (for example) from a block cipher or cryptographic hash function.

To authenticate a message m, one first applies the non-cryptographic hash function, resulting in a string Hm which is typically much shorter than the

original message m. The pseudorandom function is then applied to a string which includes Hm. The result is the authentication tag. For a general discussion of the speed and assurance advantages of this approach see, for example, [BHKKR, HK, R].

One features of UMAC is that tag generation depends on a nonce. The authentication tag is not only a function of the message, but also of a nonce which is selected by the sender. It is imperative that the nonce not be re-used when generating authentication tags of different messages under the same key. Thus the nonce will normally be implemented by a counter, though any other way to achieve a non-repeating value is also acceptable.

To authenticate messages over a duplex channel (where two parties send messages to each others), a different key could be used for each direction. If the same key is used in both directions, then it is crucial that the nonces be all distinct. For example, in this case one party can use even nonces while the other party uses odd ones.

The exact encoding by which the message, nonce, and authentication tag are combined to form an authenticated message is outside the scope of this document. Specifying such an encoding is left to individual or standardized applications.

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

Public-domain reference code for UMAC is available from the UMAC homepage: http://www.cs.ucdavis.edu/~rogaway/umac/

1.1 UMAC overview

As was described above, the two main components of the UMAC construction are universal hashing and pseudorandom functions. In a nutshell, the authentication tag is computed from the message and the nonce by setting

```
Tag = prf( hash(Msg), Nonce )
```

Two other components are a key-setup procedure, by which an initial key is stretched to provide enough key material for both the universal hashing and the pseudorandom function, and an encoding scheme which is used to "pack together" the hashing result and the nonce before applying to them the pseudorandom function.

The rest of this document is organized as follows: In Section 2 we introduce the basic notations used throughout the rest of the document. In the next four sections we describe the four components of the UMAC construction: Section 3 describes the key-setup function, Section 4 describes the pseudorandom function, Section 5 describes the universal hash function, and Section 6 describes the encoding scheme. Finally, in Section 7 we describe how all these components fit together in the UMAC construction. In the appendix we suggest a programming interface and name four sets of UMAC parameters.

```
==============================================================================

2. Notation and Basic Operations

In this specification, we view the messages to be authenticated (as well
as the keys used for authentication) as strings of bytes.  We use the
following notation to manipulate these strings.

+ Let "length(S)" be the length of string S in bytes.
+ Let "zeroes(n)" be the string of n zero-bytes.
+ Let "S[a]" be the a-th byte of the string S (indices begin at 1).
+ Let "S[a..b]" be the substring of bytes a through b of S.
+ Let "S[a..]" be the substring of bytes a through length(S) of S.
+ Let "||" be string concatenation.

zero_pad(S,n): appends the minimum number of zero-bytes to the string S
   such that the resulting string has a length which is a multiple of
   the integer n. Formally, if we let ln = length(S) mod n, then if
   ln > 0 then zero_pad(S,n) = S || zeroes(n - ln). If ln == 0 then
   zero_pad(S,n) = S.


2.1  Strings and Integers

The construction of UMAC often uses arithmetic operations which are
applied to integers of various bit-lengths. This necessitates encoding
integers as byte-strings and vice-versa.  We use the following notations
for this encoding:

str2uint(S): is the non-negative integer whose binary representation
   is the string S. Here we use the standard convention that the first
   bit in S is the most-significant bit and the last bit in S is the
   least-significant bit (ie, big-endian convention).
   Formally, if the j-th bit of S is S_j (S = S_1 ... S_t) then
   str2uint(S) ::= 2**{t-1} * S_1
                 + 2**{t-2} * S_2
                 ...
                 + 2 * S_{t-1}
                 + S_t
   (where 2**x denotes 2 raised to the x'th power)

uint2str(n, i): is the i-byte string S such that str2uint(S) = n.
   If no such string exists then uint2str(n,i) is undefined.

str2sint(S): is the integer whose binary representation in two's-complement
   is the string S.
   Formally, if the j-th bit of S is S_j (S = S_1 ... S_t) then
   str2sint(S) ::= - 2**{t-1} * S_1
                 + 2**{t-2} * S_2
                 ...
                 + 2 * S_{t-1}
                 + S_t
```

```
sint2str(n,i): is the i-byte string S such that str2sint(S) = n.
   If no such string exists then sint2str(n,i) is undefined.

================================================================================
```

3. Key setup

A mechanism is required to stretch the user-supplied MAC key into the keys used internally. This is done with a "key setup" function. It is expected that the key setup be defined using some underlying block cipher, BLOCK_CIPHER, or some cryptographic hash function, HASH_FN. Example block ciphers are DES, RC6 and MARS. Example hash functions are MD5, SHA-1, and RIPEMD-160. Although not specified here, other key setup functions are possible (e.g. stream cipher).

3.1 External Interface

The interface for any key setup function used in this MAC is

```
   KEY_SETUP
   ---------
   EXPORTS: - SETUP_KEY_LENGTH ::= length of key in bytes (before expansion)
            - KEY_SETUP        ::= name of key setup function to use
   INPUT:   - K, string of length SETUP_KEY_LENGTH
            - index, a non-negative integer < 2**31
            - len, positive integer < 2**31
   OUTPUT:  - Y, string of length len bytes
```

Here K is a user supplied random key, index is a number used to select one of 2**31 different key setup functions associated with key K, len is the number of bytes of output required, and SETUP_KEY_LENGTH is a parameter associated with the underlying key setup function.

Any key setup function which conforms to this external interface can be used in the MAC. Below we describe two implementations of key setup functions, one using a block cipher, and the other using a cryptographic hash function.

3.2 Key setup based on a block cipher.

Given a block cipher BLOCK_CIPHER with a block length of BC_BLOCK_LENGTH bytes and a key length of BC_KEY_LENGTH bytes, we define BC_KEY_SETUP, using the cipher in a feedback mode, as:

```
BC_KEY_SETUP
------------
EXPORTS: - SETUP_KEY_LENGTH ::= BC_KEY_LENGTH
         - KEY_SETUP        ::= BC_KEY_SETUP
INPUT:   - K, string of length BC_KEY_LENGTH
         - index, a non-negative integer < 2**31
         - len, positive integer < 2**31
OUTPUT:  - Y, string of length len
```

```
t = ceiling(len/BC_BLOCK_LENGTH)

Y_0 = uint2str(index, BC_BLOCK_LENGTH)
for i = 1 to t do
  Y_i = BLOCK_CIPHER(K, Y_(i-1))

Y = (Y_1 || Y_2 || ... || Y_t)[1..len]
```

3.3 Key setup based on a cryptographic hash function.

Given a cryptographic hash function HASH_FN which produces output
of length HF_OUTPUT_LENGTH bytes, we define HF_KEY_SETUP as:

```
HF_KEY_SETUP
------------
EXPORTS: - SETUP_KEY_LENGTH ::= HF_OUTPUT_LENGTH
         - KEY_SETUP        ::= HF_KEY_SETUP
INPUT:   - K, string of length HF_OUTPUT_LENGTH
         - index, a non-negative integer < 2**31
         - len, positive integer < 2**31
OUTPUT:  - Y, string of length len

t = ceiling(len/HF_OUTPUT_LENGTH)

Y_0 = uint2str(index, HF_OUTPUT_LENGTH)
for i = 1 to t do
  Y_i = HASH_FN(K || Y_(i-1))

Y = (Y_1 || Y_2 || ... || Y_t)[1..len]
```

==========

4. Pseudorandom Function (PRF)

A method for turning the variable-length output from the hash function
(Section 5) into a fixed length MAC tag is required. A "pseudorandom
function" (PRF) is defined for this purpose. It is expected that the
PRF be defined using some underlying block cipher, BLOCK_CIPHER, or
some cryptographic hash function, HASH_FN.

4.1 External Interface

The interface for any PRF used in this MAC is

```
   PRF
   ---
   EXPORTS: - PRF_KEY_LENGTH    ::= Key length associated with PRF in bytes
            - PRF_OUTPUT_LENGTH ::= Output length of PRF in bytes
            - PRF               ::= Name of PRF function to use
   INPUT:   - K, string of length PRF_KEY_LENGTH
            - index, a non-negative integer < 2**31
```

```
            - M, string of arbitrary length

   OUTPUT:  - Y, string of length PRF_OUTPUT_LENGTH
```

Here K is a random key, index is number used to select one of 2**31 different PRFs associated with key K, M is the input message, and PRF_KEY_LENGTH and PRF_OUTPUT_LENGTH are parameters associated with the underlying PRF.

Any PRF which conforms to this external interface can be used in the MAC. We describe two implementations of PRF, one using a block cipher, and the other using a cryptographic hash function.

4.2 PRF based on a block cipher.

Given a block cipher BLOCK_CIPHER with a block length of BC_BLOCK_LENGTH bytes and a key length of BC_KEY_LENGTH bytes, we define BC_PRF as:

```
BC_PRF
------
EXPORTS: - PRF_KEY_LENGTH      ::= 3 * BC_KEY_LENGTH
         - PRF_OUTPUT_LENGTH   ::= BC_BLOCK_LENGTH
         - PRF                 ::= BC_PRF
INPUT:   - K, string of length 3 * BC_KEY_LENGTH
         - index, a non-negative integer < 2**31
         - M, string of arbitrary length
OUTPUT:  - Y, string of length BC_BLOCK_LENGTH

// If the length of M is a multiple of BC_BLOCK_LENGTH, we use the
// cipher in CBC mode on M, using a different key for the final block.
// Otherwise, we first pad M and then use the cipher in CBC mode, but
// using yet another key for the last block. Prepend 'index' to
// differentiate input classes.

M = uint2str(index, BC_BLOCK_LENGTH) || M

K1 = K[1 .. BC_KEY_LENGTH]
if (length(M) mod BC_BLOCK_LENGTH != 0) then
  M = M || uint2str(128,1)
  M = zero_pad(M, BC_BLOCK_LENGTH)
  K2 = K[BC_KEY_LENGTH + 1 .. 2 * BC_KEY_LENGTH]
else
  K2 = K[2 * BC_KEY_LENGTH + 1 .. 3 * BC_KEY_LENGTH]

t = length(M) / BC_BLOCK_LENGTH

Let M_1, M_2, .., M_t be strings of length BC_BLOCK_LENGTH such
that M_1 || M_2 || .. || M_t = M.

Chain = zeroes(BC_BLOCK_LENGTH)
for i = 1 to (t - 1) do
  Chain = BLOCK_CIPHER(K1, Chain XOR M_i)
Y = BLOCK_CIPHER(K2, Chain XOR M_t)
```

4.3 PRF based on a cryptographic hash function.

Given a cryptographic hash function HASH_FN which produces output of length HF_OUTPUT_LENGTH bytes and has a block length of HF_BLOCK_LENGTH, we define HF_PRF as follows.

Notice that HF_PRF is based on the HMAC construction of [RFC-2104, BCK].

```
HF_PRF
------
EXPORTS: - PRF_KEY_LENGTH      ::= HF_BLOCK_LENGTH
         - PRF_OUTPUT_LENGTH   ::= HF_OUTPUT_LENGTH
         - PRF                 ::= HF_PRF
INPUT:   - K, string of length HF_BLOCK_LENGTH
         - index, a non-negative integer < 2**31
         - M, string of arbitrary length
OUTPUT:  - Y, string of length HF_OUTPUT_LENGTH

K1 = K XOR 0x363636...
K2 = K XOR 0x5c5c5c...

Y = HASH_FN( K1 || HASH_FN(K2 || uint2str(index, HF_BLOCK_LENGTH) || M) )
```

==========

5. Universal Hash Function

The MAC uses a keyed universal hash function, to hash and shorten the length of the data acted on by the PRF.

5.1 External Interface

The interface for any hash function used in this MAC is

```
   UHASH
   -----
   EXPORTS: - UHASH_KEY_LENGTH    ::= length of universal hashing key in bytes
            - UHASH               ::= name of universal hash function to use
   INPUT:   - Hk, string of length HASH_KEY_LENGTH
            - M, string of arbitrary length

   OUTPUT:  - Hm, string of variable length related to length(M)
```

Here Hk is a random key, M is the message being hashed, and HASH_KEY_LENGTH is a parameter associated with the underlying hash family. The length of the output string depends on the length of the input string, and on the hash function itself. Any universal hash function which conforms to this external interface can be used in UMAC. We describe one (parameterized) implementation below, based on the NH hashing primitive.

5.2 Hashing based on the NH primitive

5.2.1 Parameters

Since the universal hash function handles the (potentially long) data string, this is where the bulk of the authentication work is likely to occur. It is therefore this function that is most important to optimize to obtain fast authentication. Accordingly, in this document we identify several parameters that can be adjusted to increase the performance of UMAC in different environments. To fully specify the implementation below, one needs to specify the following parameters:

```
BYTES_PER_WORD ::= 2 | 4 | 8
  // Specifies the size (in bytes) of a "word" in this implementation.
  // This need not be equal to the native word size of any specific machine.

L1_RATIO       ::= 16 | 32 | 64 | 128 | 256 | 512 | 1024 | ...
L2_RATIO       ::=  1 | 16 | 32 |  64 | 128 | 256 |  512 | ...
  // Specifies the compression ratio of the NH hashing primitive. The
  // implementation can use two levels of compression, and the compression
  // ratio in each one can be adjusted separately.

OPERATIONS     ::= SIGNED | UNSIGNED
  // Are the arithmetic operations applied to signed or unsigned integers

SECURITY       ::= 32 | 64 | 96 | ...
  // The maximum forging probability that an adversary can achieve is
  // approximately 2**{-SECURITY} for each attempt. Halving SECURITY
  // causes UMAC to execute nearly twice as fast. The choice of SECURITY
  // must be divisible by (8 * BYTES_PER_WORD).

L1_WORD_STRIDE ::= 1 | 2 | 4 | 8 | 16
L2_WORD_STRIDE ::= 1 | 2 | 4 | 8 | 16
  // The implementation below takes a vector of integers and multiplies
  // its elements in pairs.  The WORD_STRIDE parameter determines which
  // words are multiplied.  For example, with WORD_STRIDE = 1, we multiply
  // v[1]*v[2], v[3]*v[4], v[5]*v[6], etc.  With WORD_STRIDE = 2, instead,
  // we multiply v[1]*v[3], v[2]*v[4], v[5]*v[7], etc.
  // See Subsections 5.2.3 and 5.2.4.

ENDIANESS      ::= BIG_ENDIAN | LITTLE_ENDIAN
  // When a message initially resides in memory, then the loading
  // of the message from memory to registers is done differently
  // depending on the endian orientation of the host computer. This
  // parameter specifies which endian convention should be used
  // for the loading of message from memory.
```

The options selected determine the following derived parameters

```
OUTPUT_LEN = 2 * BYTES_PER_WORD
  // Each application of the NH primitive outputs two words.

L1_PAD_BOUNDARY = 2 * L1_WORD_STRIDE * BYTES_PER_WORD
L2_PAD_BOUNDARY = 2 * L2_WORD_STRIDE * BYTES_PER_WORD
```

```
  // The input message may have to be padded to some boundary

NUM_HASH_KEYS = SECURITY / (BYTES_PER_WORD * 8)
  // To reduce the forging probability of an adversary, we apply the NH
  // hashing primitive several times, each time with a different key.

L1_KEY_SHIFT = max(2, L1_WORD_STRIDE) * BYTES_PER_WORD
L2_KEY_SHIFT = max(2, L2_WORD_STRIDE) * BYTES_PER_WORD
  // The keys used in the different applications of the "basic NH hashing"
  // are not disjoint.  Rather, they are shifted versions of each other
  // (the Toeplitz construction).

L1_KEY_LENGTH = OUTPUT_LEN * L1_RATIO
  // How much key material is needed for one application of the first
  // NH compression level.

TOTAL_L1_KEY_LENGTH = L1_KEY_LENGTH + (NUM_HASH_KEYS-1) * L1_KEY_SHIFT
  // How much key material is needed for all the applications of the
  // first NH compression level.

if (L2_RATIO > 1) then
  L2_KEY_LENGTH = OUTPUT_LEN * L2_RATIO
  TOTAL_L2_KEY_LENGTH = L2_KEY_LENGTH + (NUM_HASH_KEYS-1) * L2_KEY_SHIFT
else
  L2_KEY_LENGTH = 0
  TOTAL_L2_KEY_LENGTH = 0
  // How much key material needed for one application/all applications
  // of the second NH compression level.

HASH_KEY_LENGTH = TOTAL_L1_KEY_LENGTH + TOTAL_L2_KEY_LENGTH
  // The key length (in bytes) that is needed for the entire hashing scheme.

BYTES_PER_HASHBLOCK = OUTPUT_LEN * L1_RATIO * L2_RATIO

BITS_PER_HASHBLOCK = 8 * BYTES_PER_HASHBLOCK
  // The NH-based hashing scheme hashes the message in "blocks",
  // each of length BYTES_PER_HASHBLOCK

MLEN_BYTES = ceiling(log_2(BITS_PER_HASHBLOCK) / 8)
  // The number of bytes needed to represent the length (in bits) of a block
```

5.2.2 Mathematical Operations

The primary operations in the hashing of data are repeated applications of multiplication and addition. Below, we use '+_n' and '*_n' to denote the string operations which are equivalent to addition and multiplication modulo 2**n, respectively. Note that in this specification, *_n is only applied to operands of length n/2 bits and +_n is only applied to operands of length n-bits.
Formally, for a number n (divisible by 8), and strings S and T, we denote

```
   if (OPERATIONS == UNSIGNED) then
     S *_n T ::= uint2str(str2uint(S) * str2uint(T) mod 2**n, n/8)
```

```
   else
     S *_n T ::= uint2str(str2sint(S) * str2sint(T) mod 2**n, n/8)

+_n is defined as above except all occurrences of '*' are replaced with '+'.
```

5.2.3 Endian Re-Orientation

The following reverses the endian-orientation of a string. This operation is needed if a string is assembled by reading a message from memory and the endian-orientation of the host computer is different from the orientation specified for the UMAC.

```
BSWAP
-----
INPUT:  - n, positive integer
        - X, string of length divisible by n
OUTPUT: - Y, string of length length(X)

M = length(x) / n

Let X_1, X_2, ..., X_m be strings of length n such that
X_1 || X_2 || .. || X_m = X.

for i = 1 to M do
  Y_i = X_i[n] || X_i[n-1] || .. || X_i[1]

Y = Y_1 || ... || Y_m
```

5.2.4 The NH hashing primitive

Given the parameters from above, the NH hashing primitive is defined as follows:

```
NH
--
INPUT:   - K, string of length at least length(M)
         - M, string of length divisible by 2 * BYTES_PER_WORD * stride
         - stride, positive integer
OUTPUT:  - Y, string of length OUTPUT_LEN

num_words = length(M) / BYTES_PER_WORD

Let M_1, M_2, ..., M_{num_words} be strings of length BYTES_PER_WORD
such that M_1 || M_2 || .. || M_{num_words} = M.

Let K_1, K_2, ..., K_{num_words} be strings of length BYTES_PER_WORD
such that K_1 || K_2 || ... || K_{num_words} is a prefix of K

Compute Y as follows:

  acc = zeroes(OUTPUT_LEN)
  i = 1
```

```
  while (i <= num_words) do
      for c = i to i+stride-1 do
          case (BYTES_PER_WORD) of
            8:  tmp1 = M_c           +_64  K_c
                tmp2 = M_(c+stride)  +_64  K_(c+stride)
                 acc = acc  +_128  (tmp1 *_128 tmp2)

            4:  tmp1 = M_c           +_32  K_c
                tmp2 = M_(c+stride)  +_32  K_(c+stride)
                 acc = acc  +_64  (tmp1 *_64 tmp2)

            2:  tmp1 = M_c           +_16  K_c
                tmp2 = M_(c+stride)  +_16  K_(c+stride)
                 acc = acc  +_32  (tmp1 *_32 tmp2)
      i = i + 2 * stride
  Y = acc
```

5.2.5 The NHx hashing primitive

```
NHx
---
INPUT:   - K, string of length HASH_KEY_LENGTH bytes
         - M, string of length divisible by L1_PAD_BOUNDARY
              and no longer than BYTES_PER_HASHBLOCK bytes
OUTPUT:  - Y, string of length OUTPUT_LEN * NUM_HASH_KEYS

Compute Y as follows:

  Y = <empty string>
  for c = 1 to NUM_HASH_KEYS do
    K' = K[(c-1)*L1_KEY_SHIFT + 1 .. ]
    Y = Y || NH(K', M, L1_WORD_STRIDE)
```

5.2.6 The 2NHx hashing primitive

```
2NHx
----
INPUT:   - K, string of length HASH_KEY_LENGTH bytes
         - M, string of length divisible by L1_PAD_BOUNDARY
              and no longer than BYTES_PER_HASHBLOCK bytes
OUTPUT:  - Y, string of length OUTPUT_LEN * NUM_HASH_KEYS

t = ceiling(length(M) / L1_KEY_LENGTH)

Let M_1, M_2, .., M_t be strings such that M_1 || M_2 || .. || M_t = M
and length(M_j) = L1_KEY_LENGTH for all 1 <= j < t.

Compute Y as follows:

  Y = <empty string>
  for a = 1 to NUM_HASH_KEYS do
    M' = <empty string>
```

```
    K1 = K[(a-1)*L1_KEY_SHIFT + 1 .. ]
    K2 = K[(a-1)*L2_KEY_SHIFT + TOTAL_L1_KEY_LENGTH + 1 .. ]
    for b = 1 to t do
      M' = M' || NH(K1, M_b, L1_WORD_STRIDE)
    M' = zero_pad(M', L2_PAD_BOUNDARY)
    Y = Y || NH(K2, M', L2_WORD_STRIDE)
```

5.2.7 Universal Hash Function Based on NH

The top-level hashing scheme consists of breaking the message into appropriately sized blocks, hashing each block with either NHx or 2NHx, concatenating the results and encoding the concatenation along with the original message length modulo the hashing block-length. Formally, we define:

```
NH_UHASH
--------
EXPORTS: - UHASH_KEY_LENGTH    ::= HASH_KEY_LENGTH  // see section 5.2.1
         - UHASH               ::= NH_UHASH
INPUT:   - Hk, string of length HASH_KEY_LENGTH
         - M, string of arbitrary length
OUTPUT:  - Hm, string of length ceil(length(M) / BYTES_PER_HASHBLOCK) *
                                OUTPUT_LEN * NUM_HASH_KEYS + MLEN_BYTES


Mlen = uint2str(8 * (length(M) mod BYTES_PER_HASHBLOCK), MLEN_BYTES)

M = zero_pad(M, L1_PAD_BOUNDARY)       // Pad to L1_PAD_BOUNDARY multiple
t = ceiling(length(M) / BYTES_PER_HASHBLOCK)
if (ENDIANESS == LITTLE_ENDIAN) then      // Re-orient bytes if needed
  M = BSWAP(BYTES_PER_WORD, M)

Let M_1, M_2, .., M_t be strings such that M_1 || M_2 || .. || M_t = M
and length(M_j) = BYTES_PER_HASHBLOCK for all 1 <= j < t.

Out = <empty string>                 // NH-hash each block M_1, ... , M_t
for i = 1 to t do
  if (L2_RATIO > 1)
    Out = Out || 2NHx(Hk, M_i)
  else
    Out = Out || NHx(Hk, M_i)

Hm = Out || Mlen
```

==========

6. Encoding

The output of the hash function along with the nonce value must be packaged into an argument to the PRF function, using a reversible encoding.

```
ENCODE
------
INPUT:  - Hm, string of arbitrary length
        - Nonce, string of length 8
OUTPUT: - Ehm, string of length n, where length(Hm) + 2 <= n <= length(Hm) + 9

Compute string Ehm as follows:

  // eliminate leading zeros
  i = 1;
  while ((Nonce[i] = zeroes(1)) AND (i < 8)) do
    i = i + 1
  Nonce = Nonce[i..8]

  nonce_len = length(Nonce)
  Nonce = Nonce || uint2str(nonce_len, 1)

  Ehm = Hm || Nonce.
```

```
================================================================================
```

7. UMAC Tag Computation

The UMAC message authentication scheme uses a key-setup function (as described in Section 3), a PRF (as described in Section 4), a universal hash function (as described in Section 5) and an encoding function (as described in Section 6) to produce a tag. Given all these functions (with the parameters that they export), UMAC is defined as follows:

```
Parameter: - MIN_LEN_TO_HASH ::= 1 | 2 | 3 | ...

 // MIN_LEN_TO_HASH allows for optimization of the performance of UMAC on
 // messages of small length: In some implementations, it may be faster
 // to apply the PRF function directly to short messages, rather than
 // hashing them first with the UHASH function.


UMAC
----
EXPORTS: - MAC_KEY_LENGTH ::= SETUP_KEY_LENGTH
         - TAG_LENGTH     ::= PRF_OUTPUT_LENGTH
INPUT:   - M, string of arbitrary length
         - K, string of length SETUP_KEY_LENGTH
         - Nonce, string of length 8 bytes
OUTPUT:  - Tag, string of length TAG_LENGTH

PRF_Key  = KEY_SETUP(K, 0, PRF_KEY_LENGTH)
Hash_Key = KEY_SETUP(K, 1, UHASH_KEY_LENGTH)

if (length(M) >= MIN_LEN_TO_HASH) then
  Hm  = UHASH(Hash_Key, M)
  Ehm = ENCODE(Hm, Nonce)
  Tag = PRF(PRF_Key, 0, Ehm)
else
```

```
  Tag = PRF(PRF_Key, 1, M)
```

==========

8. Acknowledgments

Phillip Rogaway, John Black, and Ted Krovetz were supported in this work
under Rogaway'S NSF CAREER Award CCR-962540, and under MICRO grants 97-150
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, hosted by the Computer Service Center, under
Prof. Krisorn Jittorntrum and Prof. Darunee Smawatakul.

==========

9. References

[ANSI-9.9] ANSI X9.9, American National Standard for Financial Institution
Message Authentication (Wholesale),'' American Bankers Association,
1981. Revised 1986.

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

[BHKKR] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway,
"UMAC: Fast and Provably Secure Message Authentication".
Manuscript associated to this spec, 1998.

[FIPS-180] FIPS PUB 180-1, "Secure Hash Standard," April 1995.

[HK] S. Halevi and H. Krawczyk, "MMH: Software Message Authentication
in the Gbit/second Rates", Fast Software Encryption, LNCS Vol.1267,
Springer-Verlag, pp. 172-189, 1997.

[ISO-9797] ISO/IEC 9797, "Information technology - Security techniques -
Data integrity mechanism using a cryptographic check function
employing a block cipher algorithm", International Organization for
Standardization, 1994 (second edition).

[RFC-2104] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for
Message Authentication", RFC-2104, February 1997.

[RFC-1321] Ronald Rivest, "The MD5 Message-Digest Algorithm",
RFC-1321, April 1992.

[R] P. Rogaway, "Bucket Hashing and it Application
to Fast Message Authentication." Advances in Cryptology -
CRYPTO '95, LNCS vol. 963, pp. 29-42, 1995. Full version
availble from http://www.cs.ucdavis.edu/~rogaway/

[VW] P. van Oorschot and M. Wiener, "Parallel Collision

Search with Applications to Hash Functions and Discrete Logarithms", Proceedings of the 2nd ACM Conf. Computer and Communications Security, Fairfax, VA, November 1994.

[WC] M. Wegman and L. Carter, "New Hash Functions and their Use in Authentication and Set Equality", Journal of Computer and System Sciences, 22 (1981), pp. 265-279.

==========

10. Author Contact Information

Authors' Addresses

John Black
Department of Computer Science
University of California
Davis CA 95616
USA

EMail: blackj@cs.ucdavis.edu

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

EMail: shaih@watson.ibm.com

Alejandro Hevia
Department of Computer Science (DCC)
University of Chile
Av. Blanco Encalada 2120, Casilla 2777, Santiago
CHILE

EMail: ahevia@dcc.uchile.cl

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

EMail: hugo@watson.ibm.com

Ted Krovetz
Department of Computer Science
University of California
Davis CA 95616
USA

EMail: tdk@acm.org

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

EMail: rogaway@cs.ucdavis.edu

==========

A. Suggested Application Programming Interface (API)

```
/* umac.h */

typedef struct UMAC_CTX *umac_ctx_t;

umac_ctx_t umac_alloc(char key[]); /* Dynamically allocate MAC_CTX struct */
                                   /* initialize variables and generate   */
                                   /* subkeys for default parameters.     */

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

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

int umac_reset(umac_ctx_t ctx);    /* Reset for new message without       */
                                   /* calculating a tag.                  */

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

int umac_final(umac_ctx_t ctx,     /* Incorporate nonce value and return  */
               char tag[],         /* tag. Reset ctx for next message.    */
               char nonce[8]);

int umac(umac_ctx_t ctx,           /* All-in-one (non-incremental)        */
         char *input,              /* implementation of the functions     */
         long len,                 /* umac_update() and umac_final().     */
         char tag[],
         char nonce[8]);
```

Each routine returns zero if unsuccessful.

==========

B. Parameter Specifications

Although many combinations of parameter settings are possible, we have selected the following four sets as candidate standards. The UMAC-MMX variants offer maximum performance but only on processors equipped

```
   with SIMD vector units (e.g. Intel MMX and Motorola AltiVec). The UMAC-STD
   variants offer good performance on a wide variety of non-vector
   processors.

   UMAC-MMX-15
     KEY_SETUP        ::= BC_KEY_SETUP
     PRF              ::= BC_PRF
     BLOCK_CIPHER     ::= RC6-32/20/16
     UHASH            ::= NH_UHASH
     BYTES_PER_WORD   ::= 2
     L1_RATIO         ::= 1024
     L2_RATIO         ::= 1
     OPERATIONS       ::= SIGNED
     SECURITY         ::= 16
     L1_WORD_STRIDE   ::= 8
     L2_WORD_STRIDE   ::= n/a
     ENDIANESS        ::= LITTLE_ENDIAN
     MIN_LEN_TO_HASH  ::= 33

   UMAC-MMX-30
     Same as UMAC-MMX-15 except
     SECURITY         ::= 32

   UMAC-MMX-60
     Same as UMAC-MMX-15 except
     SECURITY         ::= 64

   UMAC-STD-30
     KEY_SETUP        ::= HF_KEY_SETUP
     PRF              ::= HF_PRF
     HASH_FN          ::= SHA-1
     UHASH            ::= NH_UHASH
     BYTES_PER_WORD   ::= 4
     L1_RATIO         ::= 32
     L2_RATIO         ::= 16
     OPERATIONS       ::= SIGNED
     SECURITY         ::= 32
     L1_WORD_STRIDE   ::= 1
     L2_WORD_STRIDE   ::= 1
     ENDIANESS        ::= LITTLE_ENDIAN
     MIN_LEN_TO_HASH  ::= 56

   UMAC-STD-60
     Same as UMAC-STD-30 except
     SECURITY         ::= 64

================================================================================

C. Reference Code

   See the UMAC World Wide Web homepage for implementations of the
   versions of UMAC specified in Appendix B.

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

==========

D. Test Vectors

   To be written once implementation teams have interoperable versions,
   and we have convinced ourselves that the implementations match the spec.