*Volume xx (200y), Number z, pp. 1–7*

# Analysis and improvement of message authentication codes in OpenSSH

Peter Valchev

University of Calgary, Alberta, Canada

**Abstract**

*In this paper, we will analyze the current MAC algorithms used in OpenSSH in light of recent attacks to HMAC functions. We will evaluate the suitability of UMAC as an alternative to the existing algorithms, in terms of both security and efficiency.*

Categories and Subject Descriptors (according to ACM CCS): D.4.6 [Security]: Authentication, Verification

## 1. Introduction

A message authentication code (MAC) is a keyed cryptographic hash function designed to ensure data integrity (message has not been altered) and source authentication (message originated from the purported sender). Suppose Alice and Bob share a secret key. When Bob sends a message to Alice, he computes the MAC of the message using the key, and sends the message and MAC to her using the communication channel. Alice then uses the shared key to compute the MAC on the message received from Bob and compares the computed MAC with the MAC received. If these match, then Alice is assured that the message was not altered and that it came from Bob, because only someone possessing the shared key can compute a valid MAC for the message. MAC algorithms are widely used in Internet protocols (SSH, SSL/TLS, IPsec) for providing data integrity and source authentication.

OpenSSH is a free implementation of the SSH protocol [Ope] which allows the establishment of a secure communication channel between two users. The first version of the protocol did not use MAC algorithms [SSH]. The SSH-2 protocol, which has been in wide use since 1996, and has been proposed as an Internet standard in 2006, uses public key cryptography to authenticate the remote user and provides a secure and reliable way to exchange data using encryption and message authentication codes.

The goals of this project will be to assess the current MAC implementations in OpenSSH. Each of these algorithms are based on HMAC [Ope] and we will show the HMAC construction briefly [BCK96a]. The HMAC functionality and security is based on a cryptographically secure hash function. OpenSSH uses HMAC based on the MD5, SHA-1 and RIPEMD-160 hash functions. Recent results in cryptographic research have presented theoretical limitations of HMAC [CY06, KBPH06, Bel06] that we will consider. We will then assess the suitability of UMAC as an alternative. We will research the current state-of-the-art of the security of UMAC and the methods for implementing it in comparison to HMAC. In order to compare the performance of UMAC with HMAC, we will implement UMAC in OpenSSH. We will benchmark the performance of the UMAC implementation and compare it with the existing HMAC implementations, and time permitting, include benchmarking results on the VIA C7 with hardware AES.

The remainder of this paper is organized as follows. Section 2 describes HMAC and the best attacks currently known. Section 3 describes UMAC with some of its security proofs and briefly compares it to HMAC. Section 4 deals with the implementation and benchmarking results in OpenSSH. Section 5 concludes the paper.

## 2. HMAC

Originally, MACs were constructed from block ciphers. People started considering the construction of MACs from cryptographic hash functions for a variety of reasons. For one thing, they are much faster than block ciphers when implemented in software. Additionally, there were tight cryptographic export restrictions in the US and other countries at

the time which affected block ciphers, but not hash functions. The HMAC (and related NMAC) design, first introduced in 1996 [BCK96a], describes a MAC scheme based on cryptographic hash functions such as MD5 and SHA-1. These hash functions were not originally designed for message authentication, and it was studied how to turn them into keyed primitives. There were various proposed designs to address the problems encountered, until a construction was presented which was backed by a rigorous security analysis [BCK96a].

The resulting NMAC and HMAC schemes could utilize any cryptographic hash function as a "black box" and had many attractive features. The results showed that the security of these constructs were directly related to the security of the underlying hash function in use. It was proven that if any significant weaknesses are found in these MAC schemes, not only does the underlying hash function need to be dropped from these particular usages, but it also must be dropped from all other uses.

### 2.1. MACs based on Hash Functions (HMAC)

In order to construct MACs, we will first explain what a hash function is.

**Definition** A hash function is a function $h$ which has, as a minimum, the following properties:

1. compression - $H$ maps an input $x$ of arbitrary length to an output of fixed length $n$.
2. it is easy to compute - given $h$ and an input $x$, $h(x)$ is easy to compute.

This definition implies an unkeyed hash function. [MvOV96]

Hash functions are *many-to-one* by definition, as they compress an arbitrarily large input to a small, fixed-size output. In order to be useful for cryptographic uses, they need to have some additional properties.

**Definition** Cryptographic hash functions are hash functions designed with some additional goals [BCK96a]. Primarily, they need to be *collision resistant*: if our hash function is $h$, then it should be infeasible for an adversary to find two unique strings $m, m'$ such that $h(m) = h(m')$.

As we mentioned, one of main functions of a hash is to *compress*. Thus we define a *compression function*.

**Definition** A compression function processes short fixed-length inputs, and is iterated in a particular way (in an iterated hash function) to hash arbitrarily long inputs. If we denote the compression function as $f$, it will accept two inputs: a *chaining variable* (length $l$) and a block of data (length $b$), and produces output of length $l$ [BCK96a].

And now we formally define what a message authentication code is.

**Definition** A message authentication code (MAC) is a function which takes a secret key $k$ and a message $m$ as input, and returns an authentication tag $MAC_k(m)$ as output.

In order for a cryptographic hash function to be useful in the context of message authentication, it needs to have the ability to incorporate a secret key. In their normally intended use, anyone can compute a hash of a message without secrecy. The approach in [BCK96a] is to key these hash functions through modifying their *initial vector* (IV) which is usually fixed. The modified IV is kept secret and becomes the key.

**Definition** Assuming $K$ is the secret key, $h$ is the cryptographic hash function and $m$ is the message to be authenticated, **HMAC** is defined as:

$$HMAC_K(m) = h((K \oplus opad)||h((K \oplus ipad)||m)),$$

where $\oplus$ is XOR, $||$ is the concatenation symbol and $ipad, opad$ are padding constants. The $m$ above is one block long. The message can have an arbitrary length, and it is split up in blocks of fixed size.

### 2.2. Security properties

In the context of a MAC, security usually exclusively refers to resistance to **forgery**, the ability of an attacker to produce a new message and to compute a correct authentication tag for it under the secret key.

The adversary may see a sequence $(m_1, a_1), (m_2, a_2), ..., (m_q, a_q)$ of pairs of messages and their corresponding tags ($a_i = MAC_k(m_i)$) transmitted between the communicating parties. They break the MAC if they can find a new message $m$ together with its corresponding valid tag $a = MAC_k(m)$, given that they do not possess the key $k$. When the adversary has no way of influencing the messages exchanged by the parties, but is simply eavesdropping on the wire, this is called a *known message attack*. In some cases they can choose the sequence of messages $m_1, ..., m_q$, and it is then called a *chosen message attack*.

There are some generic attacks that apply to MAC algorithms that need to be considered. Most simply, an adversary can attempt a naive exhaustive search (*brute force*) on all possible keys - this will require $O(2^k)$ operations for a $k$-bit key. The problem is that verifying such an attack (determining whether the MAC is correct) requires a total of $(k, m)$ pairs of $(key, MAC)$, where the hash function has an $m$-bit output. If $m < k$, the adversary may prefer to simply guess the MAC corresponding to a chosen message, without the need of recovering the key - the probability of success then is $1/2^m$, but this attack is not verifiable. The **birthday attack** is discussed in [MvOV96]. It is named after a classic probability problem, in which if we have 23 people in a room, the probability that at least 2 of them have the same birthday is $\approx 0.507$, which is surprisingly large. For an $m - bit$

tag, this attack allows us to forge given the ability to perform about $2^{m/2}$ MAC queries, which is a big improvement over the brute force attack. When considering other attacks on MAC functions, we can only consider them practical if they do better than the birthday attack.

Before we continue, we will define pseudorandom function families and functions.

**Definition** A pseudorandom function family is a family where every member function is specified by a random key, and can be easily computed given the key. The main property is that each function behaves like a random one, in the sense that if you are not given the key, the output of the function looks like that of a random function [BCK96b].

**Definition** Pseudorandom functions (PRFs) are functions whose input-output functionality appears like that of a random function, to any polynomial-time algorithm. They enable an important paradigm in the design of private key cryptography: First design a protocol assuming the parties share a truly random function, and prove it secure. Then substitute the random function with a *pseudorandom function* indexed by the shared (private) key $a$. The properties of *pseudorandom function families* guarantee that the security is preserved [BCK96b].

In the original paper [BCK96a], HMAC was proven to be secure based on three assumptions:

1. The compression function of the underlying hash function is a pseudorandom function (PRF),
2. The underlying hash function is *weakly collision-resistant*,
3. The key derivation function is a PRF.

These security properties were a main factor for driving HMAC into its wide use. In software today, we most commonly see HMAC-MD5 and HMAC-SHA1 being implemented, which rely on the MD5 and SHA-1 cryptographic hash functions, respectively. Recently there have been described collision attacks on these functions [WYY05, WY05] that show the 2nd assumption to be false when HMAC-MD5 and HMAC-SHA1 are considered. As a result, Bellare refined his proof and showed that the security of HMAC only depends on the first and third assumptions [Bel06].

### 2.3. Distinguishing and Forgery attacks on HMAC

Forgery attacks were already discussed above. A **distinguishing attack** on a MAC function is a method an adversary uses to tell apart this function from a random one. The birthday attack can be modified to serve as a general distinguishing attack on HMAC as follows: we collect $2^{l/2}$ message/MAC pairs and find a colliding pair $M_j, M_k$ such that $h(M_j) = h(M_k)$ ($l$ is the size of hash output in bits). Then for each of these pairs we ask for $h(M_j||P)$ and $h(M_k||P)$ where $P$ is a non-empty string. If there is at least one collision in this last step, the algorithm used is HMAC [KBPH06]. By the birthday paradox, this attack requires $O(2^{l/2})$ messages. Any further distinguishing and forgery attacks we consider must have better complexity than this, in order for us to consider them useful.

When the MD5 and SHA-1 attacks [WYY05, WY05] were first presented and the original HMAC proof refined [Bel06], it was widely assumed that they had no impact on the security of MAC algorithms based on these functions. However, the question still remained whether the other two assumptions in the HMAC proof held.

Independently, [CY06, KBPH06] answered this question and presented distinguishing and forgery attacks on HMAC based on MD5, SHA-1 and other MDx-type functions. They introduce two distinguishers of the HMAC structure - a *differential distinguisher* and a *rectangle distinguisher*, and that is used as the basis of the attacks. A *distinguisher* is a method used to distinguish (in this case) between HMAC based on a specific hash function, versus HMAC based on a random function. They are referred to as *distinguishing-R* and *distinguishing-H* attacks [KBPH06]. Since the general attacks on HMAC require $2^{n/2}$ messages, this makes other distinguishing and forgery attacks on HMAC which require more than $2^{n/2}$ messages not practical. When the adversary wants to determine whether HMAC or another form of MAC is used, the distinguishing-R attack applies, and their distinguisher requires fewer messages than the birthday approach. If they know that HMAC is used, but want to determine which hash function is used as a basis for HMAC (MD5, SHA-1, or another), they mount a distinguishing-H attack. In that case, an analogous birthday attack does not exist, so this attack is considered useful regardless of its complexity. Both of these attacks work by finding internal collisions [KBPH06].

The first type of attack in [KBPH06] distinguishes HMAC from a random function. This would be useful when an attacker does not know what MAC function is used, and would be a prerequisite for forgery attacks. There are two methods presented: a **differential distinguisher** and a **rectangle distinguisher** [KBPH06], they both work by picking random messages and modifying them to create small differences in an attempt to find a collision.

The differential distinguisher works as follows. We do a chosen message attack by randomly picking a message $M_i$ and compute another message $M_i' = M_i \oplus \alpha$ where $\alpha$ has the same length as $M_i$. Then we compute the MAC values $C_i = HMAC_K(M_i)$ and $C_i' = HMAC_K(M_i')$ and check if $C_i \oplus C_i' = 0$ - if it's true, we output HMAC. The data complexity is discussed in [KBPH06]. Using this distinguisher on a reduced version of SHA-1 (34 rounds instead of 80), they show a forgery attack requires at most $2^{53}$ messages. This is much better than the $2^{80}$ messages a birthday attack would require (this is because SHA-1 has 160-bit output).

The rectangle distinguisher works slightly differently. We

do a chosen message attack as well, and randomly pick two messages $M_i$ and $M_j$, then compute two other messages $M_i' = M_i \oplus \alpha$ and $M_j' = M_j \oplus \alpha$ where $M_i$ and $M_j$ both have the same length as $\alpha$. Then we obtain the MAC values $C_i, C_i', C_j, C_j'$ as above, and check if $C_i \oplus C_j = C_i' \oplus C_j' = 0$ or $C_i \oplus C_j' = C_i' \oplus C_j = 0$ and output HMAC if it holds. The data complexity is discussed in [KBPH06], and using this distinguisher they show a successful distinguishing attack on a reduced SHA-1 (43 rounds) requires $2^{154.9}$ chosen messages to succeed. As mentioned, SHA-1 has 160-bit output and this complexity is quite great, therefore this distinguisher cannot be used in distinguishing-R and forgery attacks in practice.

The **forgery attack** on HMAC presented in [KBPH06] works as follows. It tries to collect a large number of messages with a small difference $\alpha$ in the hopes to find a hash collision (if the underlying hash function has slow difference propagation). It then does a chosen message attack by asking for the HMAC value of $M_i||P||P'$ which he manipulates to get a desired message/MAC pair.

1. Collect $2q^{-1}$ messages pairs $(M_i, M_i')$ with difference $\alpha$, where all the $M_i$ and $M_i'$ have the same bit-length. $q$ is defined as the differential distinguisher probability in [KBPH06].
2. With a chosen message attack scenario in mind, ask for corresponding HMAC pairs $(C_i, C_i')$ of all message pairs $(M_i, M_i')$ we collected.
3. Check if $C_i \oplus C_i' = 0$ and ask for the HMAC pair of $M_i||P$, where $M_i$ and $M_i'$ have a MAC value that is the same, and P is a padding string. If the obtained pair collides, ask again for the HMAC value of $M_i||P||P'$ where $P'$ is another non-empty string. We denote this result by $C$ and output $C$ as the resulting authentication tag for $M_i'||P||P'$ - our forgery is completed. Otherwise, repeat this step until we check all HMAC pairs $(C_i, C_i')$.

### 2.4. Impact on OpenSSH

OpenSSH supports HMAC algorithms based on the MD5, SHA-1 and RIPEMD-160 hash functions [Ope]. We have discussed some theoretical limitations to HMAC used with these functions. In order to determine the impact on OpenSSH, let's first look at how an adversary can attempt an attack on the protocol.

In essense, the SSH protocol specifies a client/server protocol for securely logging into remote machines. Each host has a host-specific key used to identify it. For protocol 2, forward security is provided through a Diffie-Hellman key agreement. From this, the two hosts share a private key, valid for the session. The rest of the session is encrypted using this (private) key, using a symmetric cipher (eg. AES). The role of **HMAC** comes only to provide *session integrity*, and not authentication, which is guaranteed [T. 06].

Session integrity is protected by including a MAC in each packet. The MAC is computed from a shared secret, the packet sequence number and the contents of the packet. The MAC algorithm and key are negotiated during the Diffie-Hellman key exchange, and the tag is computed before encryption each packet as follows:

$$HMAC_{key}(sequence\ number\ ||\ unencrypted\ packet)$$

where "unencrypted packet" is the whole packet (without the MAC field) - in other words, the payload, host/destination/length fields, any padding, etc. Note that the MAC algorithms in each direction work independently [T. 06]. The MAC value is transmitted as the last part of the packet, and must be unencrypted.

Information on attacks will be in the final report.

## 3. UMAC

UMAC is a relatively new message authentication code described in [Ted06, BHK*99, Kro00]. It is developed with the goal of being very fast - on a Pentium II PC used to perform the tests, it requires 0.98 cycles/byte, compared to a 12.6 cycles/byte for HMAC-SHA1 [BHK*99]. It is also provably secure - it is proven that forging a UMAC authenticated message requires breaking the cryptographic primitive used in its construct [BHK*99]. The "U" in UMAC stands for *universal hashing*, since the algorithm uses a special family of hash functions called *universal*. A hash function is $\varepsilon$-universal if for any pair of distinctive messages, the probability that they give a hash collision is at most $\varepsilon$ (the probability is over the random choice of hash function). A UMAC is generated by hashing the message with a secretly chosen universal hash function, and then encrypting the resulting hash. It is shown that when the hash function is strongly universal and the encryption is done by a one-time pad, a third party cannot forge the message with a probability better than that by randomly chosing a string for the MAC [BHK*99]. In real world usage any symmetric key cryptosystem can be used instead of the one-time pad for a high level of security. For example, it has been shown that 96-bit UMAC tags cannot be forged with probability greater than $1/2^{90}$, assuming that the probability of a successful attack against the encryption function used is low [Ted06].

We will first begin with some background below. We will describe what universal hash functions are, and then define the NH family of functions that is at the heart of UMAC. We will then show some of the extensions to NH and the security proofs that make UMAC secure. After we have this knowledge, we will talk about putting the pieces together and creating the UMAC scheme. Lastly, we will compare the results to HMAC in the previous section.

### 3.1. Preliminaries

As we mentioned, at the foundation of UMAC is a special family of functions called *universal*, so we should formalize the definitions.

**Definition** A *family of functions* (with domain $A \in \{0,1\}^*$ and range $B \in \{0,1\}^*$ is a set of functions $H = h : A \to B$ endowed with some distribution. Specifically, $h \to H$ means that we choose a random function $h \in H$ according to this distribution.

**Definition** We are interested in families of hash functions in which collisions (where $h(M) = h(M')$ for distinct messages $M, M'$) are improbable and infrequent enough for our needs. Particularly, we are interested in functions that are **ε-*universal***, and we denote them as ε-AU. Let $H = h : A \to B$ be a family of hash functions and $\varepsilon \geq 0$, where $\varepsilon \in \mathbb{R}$. Then $H$ is ε-AU if, for all distinct messages $M, M' \in A$, we know the probability that $Pr_{h \leftarrow H}[h(M) = h(M')] \leq \varepsilon$.

Now that we know what these functions are, we can begin to talk about how to create a MAC from them.

### 3.2. NH family of functions

As we mentioned, UMAC relies on a special family of hash functions. Through several iterations described in [Kro00], the NH family was defined as most appropriate and posessing the best collision probabilities.

We will first describe the family of functions **NH[$n,w$]** defined by [Kro00]. We define a *blocksize* (an even $n \geq 2$, $n \in \mathbb{Z}$) and a *wordsize* ($w \geq 1$, $w \in \mathbb{Z}$). Then the family of functions NH[$n,w$] is defined as having a domain $A = \{0,1\}^{2w} \bigcup \{0,1\}^{4w} \bigcup ... \bigcup \{0,1\}^{nw}$ and range $B = \{0,1\}^{2w}$. Then each function in that family is specified by a $nw$-bit string $K$. The function indicated by $K$ is written as $NH_K()$.

Let $U_w = 0,...,2^w - 1$, $U_{2w} = 0,...,2^{2w} - 1$. Note that when we do arithmetic modulo $2^w$, the result is in $U_w$, and when we work modulo $2^{2w}$, we are in $U_{2w}$. With this, define

$$x +_w y = (x + y) \pmod{2^w}.$$

Then if $M$ is an arbitrary length message, $M \in A$, split it into wordsize blocks so that $M = M_1 ... M_l$, where $|M_1| = ... = |M_l| = w$, the wordsize. As above, define $K \in 0,1^{nw}$ and denote $K = K_1 ... K_n$, where $|K_1| = ... = |K_n| = w$. Then the function $NH_K(M)$ is defined as:

$$NH_K(M) = \sum_{i=1}^{l/2} (k_{2i-1} +_w m_{2i-1})(k_{2i} +_w m_{2i}) \pmod{2^{2w}}$$

where $m_i \in U_w$, $k_i \in U_w$ are the integers that the strings $M_i$ and $K_i$ are represented as, respectively. The result of the equation is in $U_{2w}$.

The collision probability for NH as described here is $2^{-w}$, this is shown in Theorem 3.2 in Section 3.5. In order to do better, an extension is described in the next section.

### 3.3. NH-Toeplitz

As described, the collision probability for NH is $2^{-w}$. There is an easy and well-known method to reduce the error probability without much cost, which is referred to as the Toeplitz construction. It consists of shifting the key left ($\ll$) to get the "next" key and hashing again. As an example, consider $NH[n,16]$ which has collision probability $2^{-16}$: to reduce it to $2^{-64}$ we have a single key, which we can write as $K = (K_1, ..., K_{n+6})$. We hash the message with the *four* derived (by shifting) keys: $(K_{1+wi}, ..., K_{n+2i})$ where $i = 0,1,2,3$. It is not intuitively clear why this reduces the collision probability, since the same key is manipulated 4 times - this is described in more detail in [BHK*99]. One immediate problem is that the Toeplitz construction, as originally described, only applies to linear functions that work over *fields*. However, NH is non-linear and operates on the two *rings* $\mathbb{Z}_{2^w}, \mathbb{Z}_{2^{2w}}$. In [BHK*99], the original results are extended to show that this construction achieves the desired bound for NH, too.

**Definition** The Toeplitz extension to NH ($NH^T[n,w,t]$) is defined as follows ( [BHK*99]). Introduce a new variable $t \geq 1$ (the iteration count), and $n, w$ remain as before. The domain is the same as NH, but the range changes to $B = \{0,1\}^{2wt}$. A function in $NH^T[n,w,t]$ is named by a key $K$ - this is the central key, which we will shift. Let $K = K_1 || ... || K_{n+2(t-1)}$ so it has $w(n + 2(t-1))$ bits. Accept the notation $K_{i..j} = K_i || ... || K_j$. Then for a message $m$ we define

$$NH^T_K(m) = NH_{K_{1..n}}(m) || NH_{K_{3..n+2}}(m) || ... || NH_{K_{(2t-1)..(n+2t-2)}}(m)$$

**Theorem 3.1** For any parameters $n, w, t$, $NH^T[n,w,t]$ is $2^{-wt}$-AU on equal-length input strings.

*Proof* In [Kro00]. □

Now we can manipulate $\text{NH}^T$ to achieve a better collision probability, but we still have a restriction that the inputs have to be of equal length. Thankfully, this is easily solved in [Kro00] by padding, concatenation and length annotation.

### 3.4. The UMAC construction

By defining $\text{NH}^T$ we now have a ε-AU family of hash functions with a small ε. We have already defined what a pseudorandom function (PRF) is and we will now show how turn them into UMAC. We will describe two pieces: a *key generator* `Key()` and a *tag generator* `Tag()`. There are 4 sets we will be working with: "key", the set of all possible keys, "message", "nonce", and "tag", which are sets of strings.

The *key generator* takes no arguments and produces a key (K ← `Key()`). The *tag generator* takes a key $K$, a message $M$, and a nonce *Nonce* and produces an authentication tag. We will write this as $\text{Tag}_K$(M,Nonce).

If we denote the family of hash functions $H = \{h : \{0,1\}^* \to \{0,1\}^*\}$ and the family of PRFs $F = \{f : \{0,1\}^* \to \{0,1\}^T\}$ as our parameters for the MAC scheme [BHK*99, Kro00], we define it as

$$UMAC[H,F] = (Key, Tag) :$$

```
function KEY()
    f <- F
    h <- H
    return (f,h)

function Tag_(f,h) (M,Nonce)
    return f(<h(M),Nonce>)
```

A question arises as to the purpose of the nonce in this scheme. In [Kro00] it is demonstrated that in the absence of a nonce, the security bounds are significantly worse. As an example, without the nonce, using a $2^{-32}$-AU hash function lets us authenticate fewer than $2^{16}$ messages, which is a significant reduction.

In the next section we will show the collision probability bounds for NH.

### 3.5. Security results/proofs

Theorem 3.2 bounds the collision probability for the function NH, it is taken from [BHK*99, Kro00].

**Theorem 3.2** For any even $n \geq 2$ and $w \geq 1$, $NH[n,w]$ is $2^{-w}$-AU (universal) on equal-length strings. In other words, the probability of collision in that case is $1/2^w$.

*Proof* Let $M, M' \in A$ (messages) and $|M| = |M'|$. By the definition for strongly universal hash functions, we want to show that the probability $Pr_{K \leftarrow NH}[NH_K(M) = NH_K(M')] \leq 2^{-w}$. Invoking the definition of $NH$ we can rewrite the left hand side as:

$$Pr[\sum_{i=1}^{l/2}(k_{2i-1} +_w m_{2i-1})(k_{2i} +_w m_{2i}) =$$

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

We want to show that this probability is less than or equal to $2^{-w}$. Note that all of the arithmetic is carried in $\mathbb{Z}_{2^{2w}}$, the *commutative ring* of integers modulo $2^{2w}$.
Recall rings are defined with two binary operations: addition and multiplication. Note that this is *not* a field (while $\mathbb{Z}_p$ is). In a commutative ring, both operations are commutative and we can assume that $m_2 \neq m'_2$ without loss of generality. The probability we are looking at, is over the uniform choices $(k_1, ..., k_n)$ with each $k_i \in U_w$. For any choice of $k_i$, the aim is to prove that $Pr_{k_1 \in U_w}[(m_1 +_w k_1)(m_2 +_w k_2) + \sum_{i=2}^{l/2}(m_{2i-1} +_w k_{2i-1})(m_{2i} +_w k_{2i}) = (m'_1 +_w k_1)(m'_2 +_w k_2) + \sum_{i=2}^{l/2}(m'_{2i-1} +_w k_{2i-1})(m'_{2i} +_w k_{2i})] \leq 2^{-w}$, which will imply the theorem.
Getting the summations on the same side, let's define

$$y = \sum_{i=2}^{l/2}(m_{2i-1} +_w k_{2i-1})(m_{2i} +_w k_{2i}) -$$

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

and let $c = (m_2 +_w k_2)$, $c' = m'_2 +_w k_2)$. Then rewrite the original equation as:

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

Now with Lemma 3.3 we will show that there can be at most one $k_1 \in U_w$ satisfying $c(m_1 +_w k_1) - c'(m'_1 +_w k_1) + y = 0$, yielding the desired result.

**Lemma 3.3** Let $c, c' \in U_w$ (distinct). Then for any messages $m, m' \in U_w$ 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 $\mathbb{Z}_{2^{2w}}$.

*Proof* It is sufficient to prove the case $m = 0$ and this proof is taken from [Kro00]. To rephrase the above, we want to 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 $\mathbb{Z}_{2^{2w}}$. There are two cases to consider.

1. If $k < 2^w - m'$, then $k(c - c') = m'c' + y$,
2. If $k \geq 2^w - m'$, then $k(c - c') = (m' - 2^w)c' + y$.

We need Lemma 3.4 now - it shows that there is at most one solution to each of these equations (1 and 2).
Once that is done, we want to show there cannot exist $k = k_1 \in U_w$ satisfying equation (1) and $k = k_2 \in U_w$ satisfying equation (2), keeping in mind that we are working in $\mathbb{Z}_{2^{2w}}$. We will again prove this *by contradiction*: assume there exist such $k = k_1 \in U_w$ satisfying (1) and $k = k_2 \in U_w$ satisfying (2). Substituting in the main equations we get

$$if\ k_1 < 2^w - m'\ then\ k_1(c - c') = m'c' + y$$

$$if\ k_2 \geq 2^w - m'\ then\ k_2(c - c') = (m' - 2^w)c' + y$$

Substracting the 2nd from the first, we get

$$(k_2 - k_1)(c' - c) = 2^w c'$$

and we want to show it has no solutions in $\mathbb{Z}_{2^{2w}}$. We will only consider the first case: $c' > c$ ($c' < c$ is analogous). The way we have chosen them, $(k_2 - k_1)$ and $(c' - c)$ are positive and also smaller than $2^{2w}$, in other words $(k_2 - k_1) < 2^w$ and $(c' - c) \leq c'$. Clearly from that, $(k_2 - k_1)(c' - c) < 2^w c'$ which contradicts our assumption that $(k_2 - k_1)(c' - c) = 2^w c'$. Therefore, there is at most one $k \in U_w$ such that $c(k +_w m) = c'(k +_w m') + y$ in $\mathbb{Z}_{2^{2w}}$. □

**Lemma 3.4** Suppose $D_w = \{-2^w + 1, ..., 2^w - 1\}$ are the results from a difference of any two elements in $U_w$. Let $x \in D_w$, $x \neq 0$. Then for any $y \in U_{2w}$, there exists at most one $a \in U_w$ such that $ax = y$ in $\mathbb{Z}_{2^{2w}}$.

*Proof* We will prove this by contradiction. Assume there exist two distinct elements $a, a' \in U_w$ such that $ax = y$ and $a'x = y$. From that it follows that $ax = a'x \Rightarrow x(a - a') = 0$. But in the beginning we assumed $x \neq 0$ and in addition $a, a'$ are distinct, hence $x(a - a') = 2^{2w}k$ for $k \neq 0$. Since $x \in D_w$ and $(a - a') \in D_w$, $x(a - a') \in \{-2^{2w} + 2^{w+1} - 1, ..., 2^{2w} - 2^{w+1} + 1\}$ and the only multiple of $2^{2w}$ is 0. Therefore we've reached a contradiction, and in fact there exists at most one $a \in U_w$ such that $ax = y$ in $\mathbb{Z}_{2^{2w}}$. □

□

### 3.6. Comparison with HMAC

There are several reasons why UMAC may be preferred instead of HMAC, or why it may be provided as an alternative:

- UMAC is much faster than HMAC. On average sized messages, it is five to ten times faster than HMAC-SHA1 as determined in results by [Kro].
- UMAC is provably secure: it is up to the final encryption step with a block cipher to be secure. In comparison, HMAC's proof relies on a strong cryptographic hash function, and with continual advances in cryptanalysis, the most common hash functions are being broken.
- UMAC is stateful for the sender, which is different than HMAC - the *nonce* must never be reused, so this may be more expensive in an implementation. This could be either an advantage or a disadvantage.
- UMAC takes advantage of modern processor parallelization, such as MMX instructions on the Pentium [Kro00].

This section will be revisited in the final paper.

## 4. Implementation Results

The description of UMAC in [Kro00] is accompanied with an implementation in C [Kro]. This implementation uses AES at the encryption step. It will be integrated into OpenSSH next and this described in the final paper.

The VIA C7 CPU which includes the VIA PadLock Security Engine can do AES encryption and decryption operations in hardware at a sustained rate of 12.8 gigabits per second [VIA]. This could lead to a significant performance improvement for UMAC operations and we will check into this for the final paper, pending time constraints.

## 5. Conclusion

After exploring the HMAC construct and the known attacks, we will justify why UMAC is an alternative to consider. In addition to having strong security properties, it is designed for very high performance, as will be shown by implementing it in the OpenSSH codebase.

## References

[BCK96a] BELLARE M., CANETTI R., KRAWCZYK H.: Message authentication using hash functions: the HMAC construction. *CryptoBytes 2*, 1 (Spring 1996), 12–15.

[BCK96b] BELLARE M., CANETTI R., KRAWCZYK H.: Pseudorandom functions revisited: The cascade construction and its concrete security. In *FOCS* (1996), pp. 514–523.

[Bel06] BELLARE M.: New Proofs for NMAC and HMAC: Security Without Collision-Resistance. Cryptology ePrint Archive, Report 2006/043, 2006.

[BHK*99] BLACK J., HALEVI S., KRAWCZYK H., KROVETZ T., ROGAWAY P.: UMAC: Fast and secure message authentication. *CRYPTO '99: Proceedings of Crypto, 19th annual international cryptology conference & Lecture Notes in Computer Science 1666* (1999), 216–233.

[CY06] CONTINI S., YIN Y. L.: Forgery and partial key-recovery attacks on HMAC and NMAC using hash collisions. In *ASIACRYPT 2006* (2006), Lai X., Chen K., (Eds.), vol. 4284 of *Lecture Notes in Computer Science*, Springer, pp. 37–53.

[KBPH06] KIM J., BIRYUKOV A., PRENEEL B., HONG S.: On the security of HMAC and NMAC based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (extended abstract). In *SCN 2006* (2006), Prisco R. D., Yung M., (Eds.), vol. 4116 of *Lecture Notes in Computer Science*, Springer, pp. 242–256.

[Kro] KROVETZ T.: UMAC C Source Code. Web site. ⟨http://fastcrypto.org/umac/2004/code.html⟩.

[Kro00] KROVETZ T. D.: *Software-optimized universal hashing and message authentication*. PhD thesis, University of California, Davis, 2000.

[MvOV96] MENEZES A. J., VAN OORSCHOT P. C., VANSTONE S. A.: *Handbook of Applied Cryptography*. CRC Press, New York, 1996.

[Ope] OpenSSH SSH client. ⟨http://www.OpenSSH.com/⟩.

[SSH] SSH Frequently Asked Questions. ⟨http://www.snailbook.com/faq/ssh-1-vs-2.auto.html⟩.

[T. 06] T. YLONEN: The Secure Shell (SSH) Transport Layer Protocol, January 2006. RFC 4253, ⟨http://www.ietf.org/rfc/rfc4253.txt⟩.

[Ted06] TED KROVETZ: UMAC: Message Authentication Code using Universal Hashing, March 2006. RFC 4418, ⟨http://fastcrypto.org/umac/rfc4418.txt⟩.

[VIA] VIA PadLock Security Engine. http://www.via.com.tw/en/initiatives/padlock/hardware.jsp.

[WY05] WANG X., YU H.: How to break MD5 and other hash functions. In *EUROCRYPT* (2005), Cramer R., (Ed.), vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 19–35.

[WYY05] WANG X., YIN Y. L., YU H.: Finding collisions in the full SHA-1. *Lecture Notes in Computer Science 3621* (2005), 17–??