Volume xx (200y), Number z, pp. 1–11

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 believes 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 [Opea] 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 are to assess the current MAC implementations in OpenSSH. Each of these algorithms are based on HMAC [Opea] and we will show the HMAC construction briefly [BCK96b]. 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]. We will consider the resulting attack scenarios and whether they have impact on OpenSSH. 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.

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, with a brief comparison to HMAC. Section 4 deals with the implementation and benchmarking results in OpenSSH, as well as possible attack scenarios. 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 [BCK96b], 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 [BCK96b].

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 was 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)

Definition A hash function is a function h which has, as a minimum, the following properties [MvOV96]:

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.

Hash functions are *many-to-one* by definition, as they compress an arbitrarily large input to a small, fixed-size output. As a result, the existence of collisions (pairs of inputs with identical hash output) is unavoidable. In order to be useful for cryptographic uses, hash functions 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 $h(m) = h(m')$. This is because in the application of cryptographic hash functions, the hash value of a string is viewed as uniquely identifiable with that string, and security is based on that property.

In practice [CY06], hash functions are constructed by iterating a *compression function* $f(cv, m)$ which takes fixed length inputs: cv is a chaining variable of n bits and m is a message block of b bits. The hash function F is defined as follows: First we divide the input message M into blocks $x_1, x_2, ..., x_s$ of length b. Then set the first chaining variable cv_0 as the fixed *initialization vector* (IV) and compute the rest as $cv_i = f(cv_{i-1}, x_i)$ where $i = 1, 2, ..., s$. The output of the hash function H is the resulting value for cv_s.

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 [BCK96b].

In the normally intended use of hash functions, anyone can compute a hash of a message without secrecy. The approach to *key* hash functions for the purpose of message authentication in [BCK96b, BCK96a] is to modify their *initial vector* (IV) which is usually fixed. The modified IV is kept secret and becomes the key. One problem is that this requires changes to the hash function implementation in order to key the IV. While this is usually simple, one would still like to avoid even those minimal changes, and use the existing code (or hardware implementation) as is. In [BCK96a] it is shown that keying the IV can be "simulated" through the use of keys padded and prepended to data, and this allows the use of existing hash functions as a black box.

Definition Assuming that 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 fixed padding constants. The m above is one block long for the purpose of the simple definition, but a slightly expanded construct accounts for arbitrary length messages.

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. The adversary breaks the MAC if she can find a new message m together with its corresponding valid tag $a = MAC_k(m)$, given that she does 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 she can choose the sequence of messages $m_1, ..., m_q$, and it is then called a *chosen message attack* [BCK96a].

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 **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 ≈ 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 only consider them practical if they do better than the birthday attack.

Before we continue, we will define pseudorandom functions, as their qualities play an important role in the security of HMAC.

Definition Pseudorandom functions (PRFs) are functions whose input-output functionality appears like that of a random function (one with completely random outputs), to any polynomial-time algorithm [BCK96c].

PRFs 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*. The properties of pseudorandom functions guarantee that the security is preserved.

In the original paper [BCK96b], 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* (it is computationally infeasible to find a collision),
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

Generic forgery attacks were already discussed in the previous section. A **distinguishing attack** on a type of 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 $2^{m/2}$ randomly chosen messages M_i and obtain their MAC values, denoted C_i. We find message pairs M_j and M_k such that $C_j = C_k$. For each of these message pairs, we obtain a MAC pair for $M_j||P$ and $M_k||P$, where P is a non-empty string. If there is at least one MAC pair that collides in this last step, conclude that the algorithm used is HMAC. By the birthday paradox, this attack requires $O(2^{m/2})$ messages. Any further distinguishing and forgery attacks of this type that we consider must have better complexity than this, in order for us to consider them useful.

Recall that in the original paper, HMAC was proved to be a PRF assuming that the underlying hash function was weakly collision resistant. When the MD5 and SHA-1 attacks [WYY05, WY05] were presented, this assumption was shown false for these functions. Then Bellare refined his proof [Bel06] to show that HMAC is a PRF under the sole assumption that the underlying compression function is a PRF. This meant that HMAC is resistant to attacks even when implemented with hash functions whose (weak) collision resistance is compromised. However, questions about the remaining assumptions in the HMAC proof still exist. After the proof was refined, it was still not clear whether existing collision attacks on hash functions compromised the PRF assumption, and whether that could lead to possible attacks.

Independently, [CY06, KBPH06] presented distinguishing and forgery attacks on HMAC based on MD5, SHA-1 and other MDx-type functions. In this paper, we studied both of these works, with our primary focus on [KBPH06]. The attacks in [CY06] are similar (the work was done in parallel independently), however these authors considered partial key recovery attacks as well. Here we will only focus on distinguishing and forgery attacks. The first type of attack, referred to as **distinguishing-R** in [KBPH06], allows distinguishing HMAC from a random function. This attack is used to develop a **forgery** attack on HMAC which works with a data complexity of less than $O(2^{m/2})$, where m is the tag size. The second type of attack, **distinguishing-H**, distinguishes between HMAC based on a specific hash function (MD5, SHA-1, or another), versus HMAC based on a random function. In this case, an analogous birthday attack does not exist, so this attack is considered useful regardless of its complexity. All of these attacks work by picking random messages and modifying them to create small differences, in an attempt to find internal collisions [KBPH06].

There are two types of distinguishers discussed, *differential* and *rectangle*. In [KBPH06], the HMAC construct is broken down into four compression functions that operate on the inner and outer parts of the iterated construction we defined earlier. The focus is on finding internal collisions in one of those functions, which operates directly on the input - here we will refer to it as h_c.

The differential distinguisher works as follows.

1. We do a chosen message attack by randomly picking $2 \cdot q^{-1}$ messages M_i and for each of them computing another message $M_i' = M_i \oplus \alpha$ where α has the same length as M_i.
2. Compute the MAC values $C_i = MAC_K(M_i)$ and $C_i' = MAC_K(M_i')$.
3. Check if $C_i \oplus C_i' = 0$ - if it's true, we output HMAC.

Here, **q** is the probability that we find a collision in the compression function h_c. In order for this to be useful, q should be larger than $2^{-m/2}$ (for m-bit tags), which makes it possible for these attacks to work with less than $2^{m/2}$ message queries. The data complexity is $4 \cdot q^{-1}$ chosen messages [KBPH06].

The **forgery attack** follows from the distinguishing attack and the first two steps are the same. We try to collect a large number of messages with a small difference α in the hopes to find a collision in h_c. This is shown to work with good probability if the underlying hash function has slow difference propagation (small differences in inputs fed to the hash functions do not propagate into large differences in the resulting hash output). We then do a chosen message attack by asking for the MAC value of $M_i'||P||P'$ of our choice.

1. Collect $2 \cdot q^{-1}$ messages pairs (M_i, M_i') with difference α, where all the M_i and M_i' have the same bit-length.
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 (here we assume that the MAC algorithm is an instantiated HMAC).
3. Check if $C_i \oplus C_i' = 0$ and ask for the HMAC pairs of $M_i||P$ and $M_i'||P$, where M_i and M_i' have a MAC value that is the same, and P is a non-empty string. If the obtained MAC pair collides, ask again for the HMAC value of $M_i||P||P'$ where P' is another non-empty string. We denote this obtained MAC value 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').

The data complexity of this forgery attack is the same as the distinguishing attack: $4 \cdot q^{-1}$. Using this method on a reduced version of SHA-1 (34 rounds instead of 80), a forgery attack requires at most 2^{53} messages. This is much better than the 2^{80} messages a birthday attack would require (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 α ($\neq 0$). 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 as an example, 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. This distinguisher is useful for distinguishing-H attacks as there is no alternative method for these attacks with better bounds. However, it cannot be used for distinguishing-R and forgery attacks, since its required data complexity is always larger than $2^{m/2}$ messages (for m-bit tags), which we get with the birthday attack, as previously discussed.

In conclusion, we have looked at differential and rectangle distinguishers on HMAC, which are derived from its structure [KBPH06]. They allow distinguishing and forgery attacks on HMAC when the hash functions have slow difference propagation. These distinguishers can be used to launch attacks (with better complexity than that of the previously known attacks) on HMAC based on existing functions such as SHA-1 with reduced number of rounds. All of these attacks do not contradict the security proof of HMAC and do not cause immediate concern, but they improve the general understanding of how weaknesses in existing cryptographic hash functions can be used to attack HMAC.

3. UMAC

UMAC is a relatively new message authentication code described in [Ted06, BHK*99, Kro00]. It was 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 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 ε-universal if for any pair of distinctive messages, the probability that they give a hash collision is at most ε (the probability is with respect to a randomly chosen hash function).

The concept of universal hash functions was developed by Wegman and Carter, who suggested their use in authenticating messages in 1981 [WC81]. In a Wegman-Carter MAC, a message is hashed with a secretly chosen universal hash function, and the resulting hash is encrypted. Since the amount of cryptography (encryption step) is small, the key to making this type of MAC fast is to choose a fast universal hash function. UMAC is a Wegman-Carter MAC which uses a family of universal hash functions called NH, built with the goal of being fast and secure. In the general Wegman-Carter construct, it is shown that when the hash function is strongly universal (the strongest notion of universality) 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]. However, the notion of a *strongly universal* function (stronger property than that needed by UMAC, which is described in the next section) usually leads to inefficient implementations and the one time pad is impractical to use in real implementations. A lesser, sufficient notion of universality, as well as a practical approach to the encryption step are presented in [Kro00] together with security proofs. They show that 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 begin with the necessary background and define universal hash functions. Then we will define the NH family of functions that is the heart of UMAC, and discuss the extensions to NH. We will put the pieces together to create the UMAC scheme. The security proofs associated with NH and the UMAC construct will be discussed. Lastly, we will compare the results to HMAC in the previous section.

3.1. Preliminaries

As we mentioned, universal hash functions are the foundation of UMAC, so we will 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.

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-almost universal***, denoted by 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 **two** distinct messages M, M', we know the probability that $Pr_{h \leftarrow H}[h(M) = h(M')] \leq \varepsilon$.

Note that ε-AU refers to "almost universal". This concept is weaker than that of *strong universality* required by the original Wegman-Carter approach, in which for **all** distinct message pairs M, M', the probability that $Pr_{h \leftarrow H}[h(M) = h(M')] \leq \varepsilon$. This weaker notion of ε-AU is shown to be sufficient to obtain the desired security [Kro00]. In the next section, we will discuss the particular family of such functions that UMAC uses.

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 possessing the best collision probabilities.

We will describe the family of functions **NH[**n, w**]** defined by [Kro00]. The parameters *blocksize* (an even $n \geq 2$, $n \in \mathbb{Z}$) and a *wordsize* ($w \geq 1$, $w \in \mathbb{Z}$) are not pre-specified, since their values are dictated by the hardware features. Therefore the values of n and w are left as parameters for the implementation. The family of functions NH$[n, w]$ is defined as operating on the message space, with a fixed output. The key is a nw-bit string K which specifies the function picked from the family. The function indicated by K is written as $NH_K()$.

Let $U_w = \{0, \ldots, 2^w - 1\}$, $U_{2w} = \{0, \ldots, 2^{2w} - 1\}$. 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

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

The function $NH_K(M)$ operates on arbitrary length messages M and is computed as follows. First, split M into word-size blocks so that $M = M_1 \ldots M_l$, where $|M_1| = \ldots = |M_l| = w$, the wordsize. As above, define the key $K \in \{0,1\}^{nw}$ and denote $K = K_1 \ldots K_n$, where $|K_1| = \ldots = |K_n| = w$. 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 bit 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}. It is desirable to reduce this probability, and there are several ways to approach this. Extending the word size w gives an improvement, but this value is dictated by the architectural characteristics of the hardware in order to sustain efficiency. Another technique is to apply several independent functions from the hash function family (with different keys) and concatenate the results. For example, concatenating the results from four NH instances would reduce the collision probability from 2^{-w} to 2^{-4w}. However, this is expensive and requires four times as much key material.

A well-known method to reduce the collision probability without much cost is referred to as the Toeplitz construction. Given one key, we "left shift" ($\ll$) to derive another key, and we hash again. As an example, consider $NH[n, 16]$ which has collision probability 2^{-16}: to reduce this to 2^{-64} we can choose a key $K = K_1 || \ldots || K_{n+6}$ and from it, derive the four keys: $K_1 || \ldots || K_n$, $K_3 || \ldots || K_{n+2}$, $K_5 || \ldots || K_{n+4}$ and $K_7 || \ldots || K_{n+6}$. Then we hash by applying NH four times with each of these keys. In essense, this is shown to be the same as applying four independent instances of NH. The four keys are derived from the same source, so it is unintuitive why this achieves the desired result. The result is shown in Theorem 3.1. The original work on the Toeplitz construction only applied 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 [Kro00], 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, Kro00]. 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 $K = K_1 || \ldots || K_{n+2(t-1)}$ so it has $w(n + 2(t-1))$ bits. Accept the notation $K_{i..j} = K_i || \ldots || K_j$. Then for a message m we define

$$NH_K^T(m) = NH_{K_{1..n}}(m) || NH_{K_{3..n+2}}(m) || \ldots || NH_{K_{(2t-1)..(n+2t-2)}}(m)$$

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

Proof The proof is quite involved and we refer to [Kro00]. $\square$

Now we can manipulate NH^T to achieve a better collision probability, but we still have a restriction that the inputs have to be of equal length. In the next section, we show how this is solved in the implementation.

3.4. The UMAC construction

By defining NH^T (we will refer to it as simply NH from now on) we now have a ε-AU family of hash functions with a small ε. Note that the inputs are still restricted to a fixed size. UMAC uses a function referred to as **UHASH** which wraps NH as necessary. UHASH takes arbitrary input and produces 8-byte output (for 64-bit UMAC tags, otherwise 4, 12, 16-byte outputs are supported). UHASH works in three layers/stages [Ted06]. A message is first hashed by the level 1 hash function, the output fed to the level 2 function, whose output is then hashed with the level 3 function.

- Level 1: The input is broken into 1024-byte chunks (word size). These are individually hashed with NH. The results are concatenated to form a string, which is up to 128 times shorter than the initial message.
- Level 2: This is only used if the message is over 16MB in size, and it uses a polynomial hash algorithm to further reduce the size of the output of the previous function.
- Level 3: Takes the output from the previous stage and hashes it to the final output length.

Recall that we defined pseudorandom functions in Section 2.2. Pseudorandom bits are needed internally by UHASH as well as at the time of tag generation. For this, the UMAC specification [Ted06] uses a block cipher - AES with 128-bit keys, but this is flexible. There are two functions we will discuss.

The key derivation function (**KDF**) generates pseudorandom bits used to generate the keys for the hash functions. The pad derivation function (**PDF**) takes a *key* and a *nonce* and returns a pseudorandom pad to be used for the tag generation as the final encryption step. This pad is of the same length as the UHASH output: 4, 8, 12, or 16 bytes depending on the desired tag size.

To generate a UMAC tag, the algorithm first uses UHASH to hash the message, then the PDF is applied to the nonce, and the resulting strings XOR'ed.

```
UMAC(Key, Message, Nonce)
    HashedMessage = UHASH(K, M)
    Pad = PDF(K, Nonce)
    Tag = Pad XOR HashedMessage
    return Tag
```

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 from the 2^{32} otherwise provided.

In the next section we will show the collision probability bounds for NH, and an outline of UMAC's security proof.

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 The proof comes from [Kro00], where it is shown that it is sufficient to prove the case $m = 0$. 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 first equation from the second, 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 $\mathbb{Z}_{2^{2w}}$. □

□

To analyze the security of UMAC, we will begin with some definitions from [Kro00]. Suppose that an adversary F is attacking a MAC scheme Σ and has access to two oracles. The first oracle **Tag** generates an authentication tag $Tag_K(M, Nonce)$ when given a query $(M, Nonce)$. The second oracle **VF** verifies a given a query $(M, Nonce, Tag)$, and outputs 1 if $Tag_K(M, Nonce) = Tag$, and returns 0 otherwise. An adversary is said to *forge*, or *succeed*, if she asks the VF oracle a query $(M, Nonce, Tag)$ which returns 1, even though $(M, Nonce)$ was not an earlier query to the Tag oracle.

We must note that this notion of security is very strong, as the adversary can manipulate nonces. Additionally, we regarded the adversary as successful if she could forge any new message (or an old one with a new nonce). In practice, the nonce will be a counter and not at the adversary's control. Also, it assumes the scenario of a *chosen message* attack. In a real life scenario, an adversary is most likely to only be able to eavesdrop on messages and their tags being transmitted over the network, in which she only has access to *known message* tags.

The success of an adversary F attacking Σ is denoted $Succ_{\Sigma}^{mac}(F)$, the probability that F succeeds with a forgery. A MAC scheme Σ is considered "good" if $Succ_{\Sigma}^{mac}(F)$ is small for any adversary. The information theoretic proof is quite involved and given in [Kro00]. It is shown that if the hash function family is ε-AU (which NH is designed to be) and no reasonable adversary can distinguish the pseudorandom function from a truly random function with advantage exceeding δ, $Succ_{UMAC}^{mac}(F) \leq \varepsilon + \delta$. That is, no reasonable adversary can break the resulting UMAC scheme with probability exceeding $\varepsilon + \delta$. Again, the size of δ comes from the security of pseudorandom function used (AES as specified in [Ted06]).

There is some concern in the community with regard to the above proof in [Kro00]. In the original Wegman-Carter construction, two independent keys were used: one for the family of hash functions, and one for the encryption step. In UMAC, the same key is used for both steps, and this does not seem to be taken into account in the current proofs. It remains for an audit of the proofs to be done, to fill in the gaps.

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]. In our own tests on typical SSH packets described in Section 4.3, it was ten to twenty times faster.
- 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 takes advantage of modern processor parallelization, such as MMX instructions on the Pentium [Kro00].

In the next section, we will look at the SSH protocol and OpenSSH implementation of UMAC. We will also determine what functionality the MAC provides, and where it may be vulnerable to attacks.

4. OpenSSH

OpenSSH supports HMAC algorithms based on the MD5, SHA-1 and RIPEMD-160 hash functions [Opea]. In section 2, we have discussed some HMAC attacks based on weaknesses in these hash functions. In order to determine the impact that these and the more generic MAC attacks have on OpenSSH, we will first study the protocol, and discuss what an adversary might try to accomplish.

In essense, the SSH protocol specifies a client/server protocol for securely logging into remote machines. Through public key cryptography, the client/server exchange a secret key which is valid for the session. Host authentication is also provided through the public key algorithms [T. 06b]. The rest of the session data is encrypted with the shared key using a symmetric cipher (eg. AES). Session integrity is protected by including a MAC in each packet, which is computed before encryption, using a negotiated secret key, as follows:

$$MAC_{key}(sequence\ number \mid\mid unencrypted\ packet)$$

The "unencrypted packet" contains the payload, as well as host/destination/length fields and any padding. The MAC value is appended to the encrypted packet before transmission, and must be unencrypted.

4.1. General SSH attacks

First, we should note that the sequence number mentioned above is implemented as a 32-bit integer. Therefore, if the session stays active long enough, this sequence number will *wrap* (and the nonces will repeat). This may provide an attacker with an opportunity to do a **replay attack** and replay a previously recorded packet with an identical sequence number. This assumes that the same key is used. To counteract this attack, the SSH protocol specifies that the parties must rekey before a wrap of sequence numbers [T. 06a]. If they have rekeyed, when the attacker attempts the replay attack, the MAC check will fail and the packet is discarded. Otherwise, if the attacker attempts to replay a captured packet before rekeying, the receiver of the duplicate packet will still fail the MAC check and discard the packet. This is because they will calculate the MAC based on the expected sequence number, and it will not match the one received.

A **man-in-the-middle attack** (MITM) is an attack in which a third party is able to observe and intercept messages exchanged between two parties. In this attack, the third party is able to read, insert and modify messages in the communication link, without the two parties suspecting that the link is compromised. As an example, the original Diffie-Hellman key exchange algorithm was vulnerable to such an attack when used without authentication. In order to prevent these attacks, the communicating hosts need to be able to verify their identities when connecting. If the identities have been verified, the connection is encrypted end-to-end, and no intermediate machines on the link have access to the connection. In SSH, this verification is done through the **public host keys** of each server.

Unlike other protocols such as HTTPS, SSH does not provide a "Trusted Authority" which maintains a collection of the public keys of hosts. Instead, this "authority" is a configuration file (`$HOME/.ssh/known_hosts` and `/etc/ssh/ssh_known_hosts`), which is essentially a database of the known host keys. Users must verify whether the host key is correct before initiating a connection. Realistically, it is expected that people will sometimes use the protocol without first verifying the server's host key. This usage is vulnerable to *man-in-the-middle attacks*. We must stress the importance of **verifying the association of a host with their host key before a session is initiated**. If an attacker has intercepted and proxied the SSH connection, they can trick the client to accept their key, not the actual host key, and then gain complete control over the communication channel. To verify the host key, users can use a variety of methods. One possibility is for the administrator to publish the key fingerprint on an SSL-protected webpage, or even a piece of paper distributed to clients. The next best thing to verifying this information in advance is to check the fingerprint of the host key on the machine after login, against the fingerprint that was accepted on the wire - if these do not match, there is an ongoing attack. More information on how these attack work is provided in [T. 06b]. With social engineering and users often ignoring changing host keys, MITM attacks are very realistic in practice.

4.2. SSH packet insertion (MAC forgery)

The interesting attack for this paper is the case where attackers attempt to **insert packets in transit between peers after the session has been established** - this is what the MAC safeguards against. The HMAC attacks in Section 2.3 are chosen text attacks and not immediately applicable in this context. First, an attacker needs to figure out how to inject a packet into an SSH connection. The most common way to do this would be through a man-in-the-middle attack as described in the previous section, or by TCP spoofing. For the purpose of this discussion, let's assume that they can carry on such an attack and inject packets. We will look at what they need to do to generate a legitimate packet that will be accepted by the target host.

Let's first consider the format of an SSH packet. Recall that in Section 4 we showed that the MAC is generated from the unencrypted packet concatenated with the sequence number. Then the packet is encrypted with the secret key, and the MAC appended to it. It becomes apparent that for an attacker to generate a packet that will be accepted they need the following things:

1. To get the secret key right, in order to generate a packet which is not garbage when decrypted (and matches the MAC).
2. To get a valid MAC by finding a collision (or finding the key).

Most importantly, an attacker only has **one chance** to attempt a forgery, because a session is automatically aborted when a packet with an invalid MAC is received.

Suppose that an attacker has broken the symmetric cipher key and has the ability to encrypt/decrypt packets that are in transmit - perhaps by breaking AES. The distinguishing and forgery attacks discussed in Section 2 require a chosen message attack scenario. In the context of SSH, even with an adversary as a man-in-the-middle, the most she gets is a collection of known messages. If we entertain the idea that she can collect a large number of these messages in the hopes of finding a collision, the best attack in [KBPH06] requires a data complexity of 2^{53} (for reduced round SHA-1). Recall that the sequence number we have been talking about is implemented as a 32-bit integer, and the parties re-key before it wraps. Therefore an attack would need to have data complexity less than 2^{32}, and the above mentioned attacks are still not feasible.

To conclude, there are no realistic attacks on SSH based on the current usage of HMAC and the best known attacks. In the next section, we will discuss our implementation of UMAC in OpenSSH.

4.3. UMAC implementation

The description of UMAC in [Kro00] is accompanied with an implementation in C [Kro]. This implementation uses AES at the encryption step. We integrated this implementation in OpenSSH [Opea]. The implementation is available upon demand and is posted on the project website [web].

OpenSSL [Opeb] is an open source toolkit that implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) as well as being a full-strength general purpose cryptographic library. It includes implementations of AES as well as HMAC and the common hash functions (MD5, SHA-1, RIPEMD-160, etc.) OpenSSH uses it to do much of the cryptographic work. We are focusing on the MAC algorithms here, so we note that OpenSSL has an **EVP** interface for creating *message digests* (the outputs of a hash function). First we will note that OpenSSH currently implements HMAC using this interface.

We tried to take the implementation of UMAC in [Kro] and adapt it to the OpenSSL EVP API, however the two designs are not compatible. UMAC wants the user to provide the key before the first message block is fed (in umac_update(), see [Kro]), while the EVP API only takes the key when at final digest generation time. Due to this obstacle, we implemented UMAC directly in OpenSSH without the use of OpenSSL. The only real difference is that we created a function which creates an instance of a UMAC ctx (context) as soon as we have negotiated the shared key. Later on, when in the function to compute a packet's MAC, the code is similar, and the details are mechanical in nature.

For more details, we refer readers to our code [web]. The resulting algorithm is named `umac@openssh.com` and is negotiated at connection time. If the server does not support `umac@openssh.com`, the hosts negotiate the next available algorithm, perhaps falling back to the previous default of `hmac-md5`. Our implementation is being considered for inclusion in the OpenSSH codebase.

4.4. Performance results

The overall time spent on MAC computations in OpenSSH is negligible for the typical interactive sessions, since the rate of packets exchanged per second is low. One application of SSH with a high rate of packets per second is X11 forwarding (running X applications through an SSH tunnel) of graphically intensive applications. We used this as a part of our experiments.

To benchmark our implementation of UMAC, we measured the real time that 1000 computations with each algorithm took, in milliseconds. The results are presented in the table below, for two different packet sizes - 32 and 512 bytes. While the default maximum packet size in the SSH protocol is 32KB [Opea], in our tests we never observed packets larger than 4096 bytes (4KB). We collected results from more varying packet sizes (up to 4KB) and the results are presented in Figure 1 attached at the end of this paper. **Most** of the packets in a typical session are under 512 bytes in size. We ran the tests on a Pentium 4, as well as a MIPS RS12000 machine to collect our data.

Time to compute 1000 iterations of mac_compute() with the given MAC algorithm

packet length (in bytes)	Pentium 4		MIPS RS12000	
	32	512	32	512
HMAC-MD5	15 ms	20 ms	60 ms	70 ms
HMAC-SHA1	17 ms	22 ms	50 ms	80 ms
H-RIPEMD160	20 ms	30 ms	271 ms	690 ms
UMAC (64-bit)	1 ms	3 ms	3 ms	6 ms
UMAC (128-bit)	1 ms	5 ms	3 ms	9 ms

Note that we implemented UMAC with 64-bit output tags, as recommended in [Kro00], which additionally specifies 32-bit, 96-bit and 128-bit implementations. In comparison, MD5 has 128-bit output, and SHA-1/RIPEMD-160 have 160-bit output. For the purpose of comparing functions with similar-sized output, we also benchmarked UMAC with 128-bit tags (UMAC-128) and included it in our results.

5. Conclusion

We have studied the most current attacks on HMAC and concluded that there are no immediate concerns with its security in real world implementations. However, the current results will influence further research, and it will be wise to keep an eye on future results. We researched UMAC as an alternative algorithm to consider, and implemented it in OpenSSH. Significant performance improvement was observed, and the changes are being considered for the official OpenSSH codebase. In addition to its high performance, UMAC has strong security properties. Some concern exists in the community about the validity and accuracy of UMAC's security proofs, and an audit of the proofs would perhaps help its wider use and acceptance.

6. Acknowledgements

Thanks to Damien Miller for the idea of considering UMAC and pointers about OpenSSH.

References

[BCK96a] BELLARE M., CANETTI R., KRAWCZYK H.: Keying hash functions for message authentication. *Lecture Notes in CS 1109* (1996), 1–15.

[BCK96b] BELLARE M., CANETTI R., KRAWCZYK H.: Message authentication using hash functions: the HMAC construction. *CryptoBytes 2*, 1 (Spring 1996), 12–15.

[BCK96c] 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.

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

[Opeb] OpenSSL Project. ⟨http://www.OpenSSL.org/⟩.

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

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

[T. 06b] T. YLONEN, C. LONVICK: The Secure Shell (SSH) Protocol Architecture, January 2006. RFC 4251, ⟨http://www.ietf.org/rfc/rfc4251.txt⟩.

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

[WC81] WEGMAN, M. N., CARTER, J. L.: New hash functions and their use in authentication and set equality. *J. Computer and System Sciences 22* (1981), 265–279.

[web] CPSC503 project website. ⟨http://sightly.net/peter/cpsc503/⟩.

[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–??

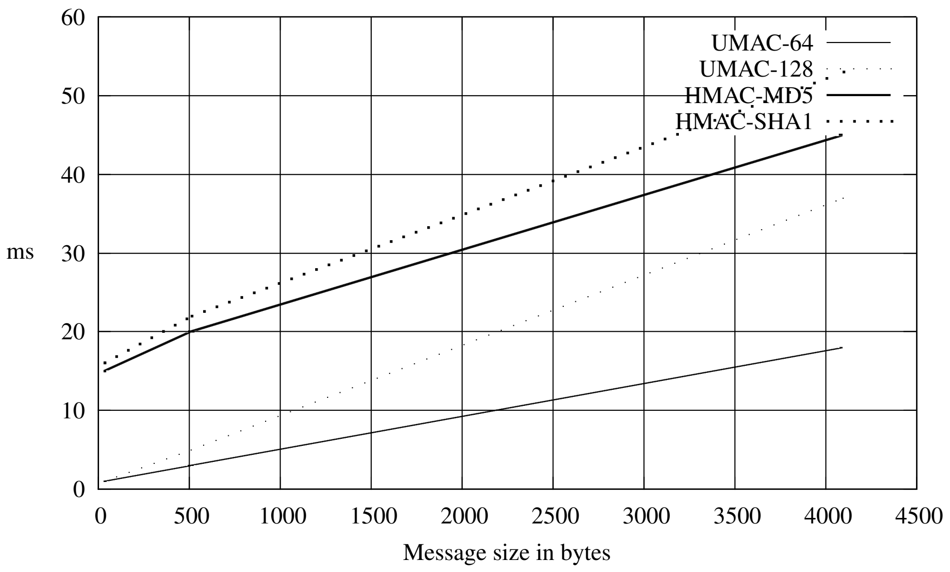

Figure 1: *Performance of the mac_compute() function in OpenSSH using the newly implemented UMAC algorithm in comparison to the existing HMAC-MD5 and HMAC-SHA1 implementations. Data is provided for typical SSH packet sizes observed in our tests. (The default maximum packet size is 32768 bytes) Timing in milliseconds is given for 1000 computations on a Pentium 4.*