MMH: Software Message Authentication Towards the Gbit/second Rates

Shai Halevi* — IBM & MIT

Hugo Krawczyk† — IBM

PRELIMINARY DRAFT

September 10, 1996

Abstract

We describe a construction of almost universal hash functions suitable for very fast software implementation and applicable to the hashing of bulk data and fast cryptographic message authentication. Our construction uses fast single precision arithmetic which is increasingly supported by modern processors due to the growing needs for fast arithmetic posed by multimedia applications.

We report on hand-optimized assembly implementations on a 150 MHz PowerPC 604 and a 150 MHz Pentium-Pro, which achieve hashing speeds of 350 to 820 Mbit/second, depending on the desired level of security (or collision probability). This is a significant speed-up over current software implementations of universal hashing and other message authentication techniques (e.g., MD5-based). Moreover, our construction is designed to take advantage of emerging microprocessor technologies (such as Intel's MMX, 64-bit architectures and others), so future implementations will be able to reach well into the Gbit/second rates.

The construction is based on techniques due to Carter and Wegman for universal hashing using modular multilinear functions that we carefully modify to allow for fast software implementation. We prove the resultant construction to retain the necessary mathematical properties required for its use in hashing and message authentication.

*`shaih@theory.lcs.mit.edu`. Work was done while the author was at IBM Watson Research Center.

†`hugo@watson.ibm.com`

1 Introduction

Universal hash functions, which were first introduced by Carter and Wegman in [CW79] have a wide range of applications in many areas of computer science, including compilers, databases, parallel architectures, message authentication, complexity theory and many others. The use of universal hashing for authentication (introduced by Wegman and Carter [WC81]) received much attention lately. In particular, several recent works deal with efficient implementation of universal hashing as a tool for achieving fast message authentication (e.g., [St94, Kr94, Kr95, Ro95, AS96, Sh96]).

In this note we describe a particular construction of universal hash functions which is specially designed to admit a very fast software implementation using todays' microprocessor technology. On a very high level, this construction is obtained by implementing a well-known function and then modifying the implementation so as to eliminate costly operations. The result can be thought of a a "buggy implementation" of the original function. The new function has a very fast implementation, and moreover, we can prove that it is "almost as good" as the original one.

We report on implementations on a 150 MHz PowerPC 604 and a 150 MHz Pentium-Pro, which achieve hashing speeds of 350 to 820 Mbit/second, depending on the desired collision probability. This is a significant speed-up over current software implementations of universal hashing Moreover, our construction is designed to take advantage of emerging microprocessor technologies (e.g., Intel's MMX, Sun's VIS, 64-bit architectures and others), so future implementations will be able to reach well into the Gbit/second rates.

What's in a name. The name MMH stands for Multilinear-Modular-Hashing.

Organization. This note is organized as follows: In Section 2 we briefly go over the notions of universal hashing and the connections between those and message-authentication, and recall a well-known construction for universal hashing. In Section 3 we sketch our modifications of this well-known construction and our implementation of the resulting function, and provide some experimental results. Finally, in Section 4 we sketch the proof that the resulting function is "almost as good" as the original one.

2 Preliminaries

2.1 Notations

In the sequel we use the following notations. For two integers z, p, we denote by $z \bmod p$ the integer which is obtained by reducing z mod p. That is, $z \bmod p$ is the unique integer between 0 and $p-1$ which is congruent to z mod p. For two integers

y, z, We write $y = z$ to assert that they are equal (over the integers) and $y \equiv z \pmod{p}$ to assert that they are congruent to one another mod p.

We denote vectors by boldface small letters, e.g., $\mathbf{x} = \langle x_1 \cdots x_k \rangle$. Also we identify bit strings with binary-represented integers, and in particular we identify $\{0,1\}^{32}$ with $\{0, 1, \cdots 2^{32} - 1\}$.

2.2 Universal hashing

In the definitions below, H is a family of functions from a domain D to a range R and ϵ is a positive constant. These definitions and notations are due to Carter and Wegman [CW79], Krawczyk [Kr94], Rogaway [Ro95] and Stinson [St95].

Definition 1:

1. H is a *strongly universal family of hash functions* if for all $x \neq y \in D$ and all $a, b \in R$, $\Pr_{h \in H}[h(x) = a,\ h(y) = b] = \frac{1}{|R|^2}$
2. H is a *universal family of hash functions* if for all $x \neq y \in D$, $\Pr_{h \in H}[h(x) = h(y)] = \frac{1}{|R|}$

   H is an *ϵ-almost-universal* (ϵ-AU) family of hash functions if for all $x \neq y \in D$, $\Pr_{h \in H}[h(x) = h(y)] \leq \epsilon$
3. Assume that R is an abelian group and denote by '$-$' the group subtraction operation. H is a *Δ-universal family of hash functions* if for all $x \neq y \in D$ and all $a \in R$, $\Pr_{h \in H}[h(x) - h(y) = a] = \frac{1}{|R|}$

   H is an *ϵ-almost-Δ-universal* (ϵ-AΔU) family of hash functions if for all $x \neq y \in D$ and all $a \in R$, $\Pr_{h \in H}[h(x) - h(y) = a] \leq \epsilon$

All the probabilities are taken over the choice of $h \in H$ according to the uniform distribution over H.

2.3 Universal hashing and message authentication

Maintaining authenticated communication over untrusted links is one of the most basic goals of cryptography. In this work we consider a typical communication scenario in which two parties communicate over an unreliable link where messages can be maliciously altered by an adversary.

To authenticate the communication over this link, the communicating parties share a secret key which is unknown to the adversary. They use this secret key to compute a *message authentication code* (MAC) for every message they send on the link. A MAC is a function which takes the secret key x and the message m and returns a tag $\mu \leftarrow MAC_x(m)$. The sender send the pair (m, μ) over the untrusted link. On receipt of (m', μ'), the receiver repeat this computation and verifies that $\mu' = MAC_x(m')$.

The adversary sees a sequence $(m_1, \mu_1), (m_2, \mu_2), \cdots, (m_q, \mu_q)$ of pairs transmitted between the parties. The MAC is said to be broken if the adversary can find a message m, not included among $m_1, \cdots, m_q$, together with its corresponding valid authentication tag $\mu = \mathrm{MAC}_x(m)$. The success probability of the adversary is the probability that she breaks the MAC. For a MAC function to be "good", any reasonable adversary should have a negligible probability of success.

In the Wegman-Carter paradigm [WC81], secret key shared by the communicating parties consists of a hash function h drawn randomly from a Δ-universal family of hash functions H and a random pad d. To authenticate a message m, the sender computed $h(m)$ and add to it the next piece of the pad d. That is, we have $MAC_{h,d}(m) \stackrel{\text{def}}{=} h(m) + d_i$.

The thing to notice is that we first process the message m using a non-cryptographic operation (universal hashing), and then we apply a cryptographic operation (one-time-pad encryption) to $h(m)$, which is typically much shorter than m itself. In practical implementations, the random pad may be replaces by a pseudo-random one, so the parties only need to share the function h and the seed s to the pseudo-random generator. The reader is referred to [WC81, Br82, Kr94, Ro95, St94, Sh96] for elaborate discussions of these issues.

2.4 A Well-Known Construction

The starting point for our construction is a well known construction due to Carter and Wegman [CW79]. This construction works in the finite field Z_p for some prime integer p. The family of hash functions consists of all the multilinear functions over Z_p^k for some integer k. Namely,

Definition 2: Let p be a prime and let k be an integer $k > 0$. Define a family MMH* of functions from Z_p^k to Z_p as follows

$$\mathrm{MMH}^* \stackrel{\text{def}}{=} \{g_{\mathbf{x}} : Z_p^k \to Z_p \mid \mathbf{x} \in Z_p^k\}$$

where the functions $g_{\mathbf{x}}$ are defined for any $\mathbf{x} = \langle x_1, \cdots x_k \rangle, \mathbf{m} = \langle m_1, \cdots, m_k \rangle$,

$$g_{\mathbf{x}}(\mathbf{m}) \stackrel{\text{def}}{=} \mathbf{m} \cdot \mathbf{x} \bmod p = \sum_{i=1}^{k} m_i x_i \bmod p$$

Theorem 1: The family MMH* is Δ-universal.

Proof: Fix some $a \in Z_p$, and let $\mathbf{m}, \mathbf{m}'$ be two different messages. Assume w.l.o.g. that $m_1 \neq m_1'$. Then for any choice of $x_2, \cdots x_k$ we have

$$\begin{aligned} &\Pr_{x_1}\left[g_{\mathbf{x}}(\mathbf{m}) - g_{\mathbf{x}}(\mathbf{m}') \equiv a \pmod{p}\right] \\ &= \Pr_{x_1}\left[(m_1 - m_1')x_1 \equiv a - \textstyle\sum_{i=2}^{k}(m_i - m_i')x_i \pmod{p}\right] = \tfrac{1}{p} \end{aligned}$$

□

Reducing the collision probability. Depending on the choice of p and the application at hand, collision probability of $1/p$ may be too large. A simple way to reduce the collision probability is to hash the message twice using two independent keys. This yields a collision probability of $1/p^2$, but makes the keys twice as long and also doubles the work.

A better solution is to use a the "Toeplitz matrix construction". That is, instead of choosing two independent k-vectors $\mathbf{x}, \mathbf{x}'$, we choose $k+1$ scalars $x_1, \cdots x_{k+1}$ and set $\mathbf{x} = \langle x_1, \cdots, x_k \rangle, \mathbf{x}' = \langle x_2, \cdots, x_{k+1} \rangle$. This solution too doubles the work required for hashing.

Of course there is nothing sacred about two repetitions. The same construction can be used to reduce the collision probability to $1/p^t$ for any value of t, adding only t scalars to the key, but multiplying the work by a factor of t.

Dealing with arbitrary long messages. The above function can only be applied to a fixed-size messages (namely, to k-vectors over Z_p). The standard approach for dealing with messages of arbitrary length is to use tree-hashing. That is, we break the message into blocks of k elements and hash each block separately. We then concatenate the results and hash them again using an independent key, and so on. The drawback of this approach is that both the key-size and the collision probability grows linearly with the height of the tree. The increase in key-size is particularly annoying since the keys needed for the lower levels in the tree are hardly ever used.

A possibly better solution is to only implement the upper two levels of the tree, and to use a different hash function to hash the result. This way we only use two independent keys (one for each of the levels) and we apply the other hash to a buffer whose length is only $1/k^2$ of the message length. A reasonable choice for the other hashing scheme can be the polynomial-evaluation-hashing. The reader is referred to [Sh96] for a description and implementation details of this scheme.

3 Multilinear Modular Hashing

We now describe our modifications to the "ideal" Multilinear Modular Hashing. The main characteristics of our implementation are

- We work with 32-bit integers (although the same approach can be used in machines with different word-length).

- We work with the prime integer $p = 2^{32}+15$, so we can implement a division-less modular reduction.[1]

Therefore our starting point is a specific "very slightly modified" instance of MMH*:

$$\text{MMH}^*_{32} \stackrel{\text{def}}{=} \left\{ g_{\mathbf{x}} : \left(\{0,1\}^{32}\right)^k \to Z_p \;\middle|\; \mathbf{x} \in \left(\{0,1\}^{32}\right)^k \right\}$$

where the functions $g_{\mathbf{x}}$ are defined for any $\mathbf{x} = \langle x_1, \cdots x_k \rangle, \mathbf{m} = \langle m_1, \cdots, m_k \rangle$,

$$g_{\mathbf{x}}(\mathbf{m}) \stackrel{\text{def}}{=} \mathbf{m} \cdot \mathbf{x} \bmod (2^{32}+15) = \left[\sum_{i=1}^{k} m_i x_i\right] \bmod (2^{32}+15)$$

Similarly to Theorem 1, we have

Theorem 2: The family MMH^*_{32} is ϵ-AΔU-universal with $\epsilon = 2^{-32}$.

Notice that this function is only defined for fixed-length messages (namely, messages of $32k$ bits). To handle arbitrary-length messages we use the tree construction from Section 2.4.

3.1 Implementing the "ideal" function

The one operation in MMH^*_{32} which is by far the most expensive is the modular reduction. Even the implementation of a division-less reduction still takes about 10-15 machine instructions. Therefore, we carry the inner-product operation over the integers and then do just one modular reduction at the end. This approach forces us to deal with addition of integers of 64-bits and more. To this end we use machine instructions for addition-with-carry.

To implement the integer multiplications, we take advantage of machine instructions for multiplying two 32-bit integers and obtaining the 64-bit result. Both addition-with-carry and 32-by-32 multiplication are available in just about all the architectures of todays' computers. However, there is no syntax in any high-level programming language to access these operations, so we must write our implementation in assembly language.

Fixing the value of k. The value of k (the length of the message- and key-vectors) has two effects on the implementation.

- As we amortize the costly modular reduction over k cheaper multiply-and-add operations, increasing k should increase the speed.

[1] The idea to use such prime numbers is also due to Carter and Wegman [CW]. In fact, any prime which satisfies $2^{32} < p < 2^{32} + 2^{16}$ will do.

- Since the key $\mathbf{x}$ consists of k 32-bit integers, increasing k results in a longer key.

As a reasonable tradeoff between these conflicting objectives, we work with $k = 32$. For this value of k, the cost of modular reduction amounts to only about 10-15% of the total cost of the implementation.

3.2 Modifying the implementation

We make two modifications to the implementation of MMH_{32}^*

- We make the output of the function a 32-bit integer rather than an element in Z_p. This is done by ignoring the most-significant bit in the output of the original function, which is equivalent to reducing it mod 2^{32}.
- In the inner-product operation, we ignore any carry-bit out of the 64 bit-location. This is equivalent to computing the sum mod 2^{64}.

These two modifications together define the following family of functions

Definition 3: Set $p = 2^{32} + 15$ and $k = 32$. Define a family MMH of functions from $(\{0,1\}^{32})^k$ to $\{0,1\}^{32}$ as follows

$$\mathrm{MMH}_{32} \stackrel{\mathrm{def}}{=} \left\{ h_{\mathbf{x}} : \left(\{0,1\}^{32}\right)^k \rightarrow \{0,1\}^{32} \;\middle|\; \mathbf{x} \in \left(\{0,1\}^{32}\right)^k \right\}$$

where the functions $h_{\mathbf{x}}$ are defined for any $\mathbf{x} = \langle x_1, \cdots x_k \rangle, \mathbf{m} = \langle m_1, \cdots, m_k \rangle$,

$$h_{\mathbf{x}}(\mathbf{m}) \stackrel{\mathrm{def}}{=} \left[\left[\left[\sum_{i=1}^{k} m_i x_i \right] \bmod 2^{64} \right] \bmod (2^{32} + 15) \right] \bmod 2^{32}$$

In Section 4 we show that MMH_{32} is ϵ-AΔU with $\epsilon = 2^{-30}$.

Instruction count. To give an estimated instruction count for an implementation of MMH, we consider a machine with the following properties

- 32-bit machine integers.
- Arithmetic operations are done in registers.
- A multiplication of two 32-bit integers which yields a 64-bit result takes two machine instructions.

On such a machine, we can implement each multiply-and-add operation using

1. two instructions for loading the message-word and the key-word,

2. two instructions for the multiplication,
3. two instructions for the 64-bit integers addition, and
4. one instruction to handle the loop.

We repeat these operation $k = 32$ times, and then we need about 10-15 instructions for the modular reduction. This yields an instruction count of about $7k+15$ to handle a k-word message. That is, we have about about 7.5 instructions per-word, or less than 2 instructions per-byte.

This instruction-count can be further reduced by unrolling the loop a few times and by working on several messages at the same time (so we can load a key word just once and use it on several messages). For example, in our implementation on the PowerPC we have about 6 instructions per-word. On a 64-bit machine we may be able to get as low as 4.5 instructions per-word. Implementing the hashing-tree adds less than 10% to the instruction count, so the total count is somewhere between 5 and 8 instructions per word, depending on the actual architecture details.

Moreover, the structure of the hashing procedure (and in particular the inner-product operation) leaves plenty of room for parallelization. Emerging microprocessor technologies which are aimed at multimedia applications tend to include a good support for inner-product operations (e.g., Intel's MMX, Sun's VIS, etc.).

3.3 Experimental results

Below we describe the results of a few experimental implementations of MMH. We implemented MMH on the following machines

- A 150 MHz PowerPC 604 RISC machine running AIX.
- A 150 MHz Pentium-Pro machine running Windows NT.

For each of these machines, we implemented two variants of MMH

1. The basic MMH construction with tree-hashing. This variant has a 32-bit output and collision probability of 2^{-30} times the height of the tree.
2. A "high-security" version, where each hashing operation is repeated twice using "the Toeplitz matrix construction". This variant has a 64-bit output and collision probability of 2^{-60} times the height of the tree.

The results which we got for these variants are summarized in the following table.

150 MHz Pentium-Pro	message in memory	message in cache
64-bit output	296 Mbit/second	356 Mbit/second
32-bit output	556 Mbit/second	813 Mbit/second

150 MHz PowerPC 604	message in memory	message in cache
64-bit output	390 Mbit/second	417 Mbit/second
32-bit output	597 Mbit/second	820 Mbit/second

Table 1: Timing results for various implementations of MMH.

We also implemented MMH on a Pentium machine. However, the integer multiplication in the Pentium is so bad that the best results we got were obtained using the floating-point unit for the multiply-and-add operations. This implementation achieves a rate of about 160 Mbit/second on a 120 MHz Pentium for the 64-bit variant.

4 Analysis of the collision probability

In this section we analyze the collision probability of the Multilinear Modular Hash function. In this note we only prove that the MMH family is ϵ-AΔU for $\epsilon = 9 \cdot 2^{-32}$. It is possible, however, to refine the analysis below to show $\epsilon = 2^{-30}$. The same analysis can also be applied to similar constructions using word-length other than 32-bits.

We start the analysis by analyzing the collision probability of a hybrid construction which is half-way between the ideal family MMH^*_{32} and the actual construction MMH_{32}.

Definition 4: Set $p = 2^{32} + 15$ and $k = 32$. Define a family H_{32} of functions from $(\{0,1\}^{32})^k$ to Z_p as follows

$$H_{32} \stackrel{\text{def}}{=} \left\{ \tilde{h}_{\mathbf{x}} : \left(\{0,1\}^{32}\right)^k \to Z_p \;\middle|\; \mathbf{x} \in \left(\{0,1\}^{32}\right)^k \right\}$$

where the function $\tilde{h}_{\mathbf{x}}$ is defined for any $\mathbf{x} = \langle x_1, \cdots x_k \rangle, \mathbf{m} = \langle m_1, \cdots, m_k \rangle$,

$$\tilde{h}_{\mathbf{x}}(\mathbf{m}) \stackrel{\text{def}}{=} \left[\left[\sum_{i=1}^{k} m_i x_i \right] \bmod 2^{64} \right] \bmod (2^{32} + 15)$$

Note that H_{32} is defined like MMH_{32}, but without the reduction mod 2^{32} at the end.

Lemma 4.1: H_{32} is an ϵ-AΔU family of hash function with $\epsilon \le 3 \cdot 2^{-32}$.

Proof: Fix any $a \in Z_p$, and any two different message-vectors $\mathbf{m} \neq \mathbf{m}'$, and assume w.l.o.g. that $m_1 \neq m'_1$. We prove that for any choice of $x_2, \cdots, x_k$, $\Pr_{x_1}[\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \equiv a \pmod p] \le 3/2^{32}$, which implies the lemma.

Since $x_1 m_1 < 2^{64}$ for any value of x_1, then for any choice of the addition of $x_2 \cdots x_k$ the term $x_1 m_1$ adds at most one carry bit to the sum. Fix some choice of $x_2 \cdots x_k$ and denote $s \stackrel{\text{def}}{=} [\sum_{i=2}^{k} x_i m_i] \bmod 2^{64}$. We conclude that

$$\left[\sum_{i=1}^{k} x_i m_i \right] \bmod 2^{64} = x_1 m_1 + s - 2^{64} b \qquad \text{for some } b \in \{0,1\}$$

Similarly we denote $s' \stackrel{\text{def}}{=} \sum_{i=2}^{k} x_i m'_i \bmod 2^{64}$, and we get

$$\left[\sum_{i=1}^{k} x_i m'_i\right] \bmod 2^{64} = x_1 m'_1 + s' - 2^{64} b' \qquad \text{for some } b' \in \{0,1\}$$

Notice that s, s' do not depend on the choice of x_1, but b, b' may depend on it. We stress this below by writing $b(x_1), b'(x_1)$. We can now write $\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}^*_{\mathbf{x}}(\mathbf{m}')$ as

$$\begin{aligned}
&\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \\
&= \left[\sum_{i=1}^{k} x_i m_i \bmod 2^{64}\right] - \left[\sum_{i=1}^{k} x_i m'_i \bmod 2^{64}\right] \\
&= (m_1 - m'_1)x_1 - 2^{64}[b(x_1) - b'(x_1)] + s - s'
\end{aligned}$$

Since $b(x_1) - b'(x_1) \in \{-1, 0, 1\}$ for all x_1, then

$$\begin{aligned}
&\Pr_{x_1}\left[\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \equiv a \pmod{p}\right] \\
&= \Pr_{x_1}\left[(m_1 - m'_1)x_1 - 2^{64}[b(x_1) - b'(x_1)] + s - s' \equiv a \pmod{p}\right] \\
&\le \Pr_{x_1}\left[(m_1 - m'_1)x_1 + s - s' \equiv a - s + s' - 2^{64} \pmod{p}\right] \\
&\quad + \Pr_{x_1}\left[(m_1 - m'_1)x_1 + s - s' \equiv a - s + s' \pmod{p}\right] \\
&\quad + \Pr_{x_1}\left[(m_1 - m'_1)x_1 + s - s' \equiv a - s + s' + 2^{64} \pmod{p}\right] \\
&\le 3 \cdot 2^{-32} \qquad \square
\end{aligned}$$

We now proceed to analyze the collision probability of MMH_{32}.

Theorem 3: MMH_{32} is an ϵ-AΔU family of hash function with $\epsilon \le 9 \cdot 2^{-32}$.

Proof: Below we only prove that MMH_{32} is ϵ-AU. The proof of ϵ-AΔU follows from the same ideas and a somewhat tedious case analysis. Recall that MMH_{32} is obtained from H_{32} by reducing the result mod 2^{32}. That is, for any $\mathbf{x}, \mathbf{m}$, $h_{\mathbf{x}}(\mathbf{m}) = \tilde{h}_{\mathbf{x}}(\mathbf{m}) \bmod 2^{32}$.

Fix any two different message-vectors $\mathbf{m}, \mathbf{m}'$, and let $\mathbf{x}$ be a key-vector so that $h_{\mathbf{x}}(\mathbf{m}) = h_{\mathbf{x}}(\mathbf{m}')$. This means that we must have

- Either $\tilde{h}_{\mathbf{x}}(\mathbf{m}) = \tilde{h}_{\mathbf{x}}(\mathbf{m}')$,
- or $\tilde{h}_{\mathbf{x}}(\mathbf{m}) = \tilde{h}_{\mathbf{x}}(\mathbf{m}') + 2^{32}$,
- or $\tilde{h}_{\mathbf{x}}(\mathbf{m}) = \tilde{h}_{\mathbf{x}}(\mathbf{m}') - 2^{32}$.

Therefore, this key $\mathbf{x}$ satisfies $[\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \bmod (2^{32}+15)] \in \{0, 15, 2^{32}\}$. Thus we get

$$\begin{aligned}
&\Pr_{\mathbf{x}}\left[h_{\mathbf{x}}(\mathbf{m}) = h_{\mathbf{x}}(\mathbf{m}')\right] \\
&\le \Pr_{\mathbf{x}}\left[[\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \bmod (2^{32}+15)] \in \{0, 15, 2^{32}\}\right] \quad \le \quad \frac{9}{2^{32}} \quad \square
\end{aligned}$$

References

[AS96] M. Atici and D. Stinson. Universal Hashing and Multiple Authentication to appear in CRYPTO '96

[Br82] G. Brassard. On computationally secure authentication tags requiring short secret shared keys, *Advances in Cryptology – CRYPTO '82 Proceedings*, Springer-Verlag, 1983, pp. 79–86.

[CW79] L. Carter and M. Wegman. Universal Hash Functions. J. of Computer and System Science 18, 1979, pp. 143-154.

[CW] L. Carter and M. Wegman. Private Communications.

[WC81] M. Wegman. and L. Carter. New hash functions and their use in authentication and set equality. *J. of Computer and System Sciences,* vol. 22, 1981, pp. 265-279.

[GMR88] S. Goldwasser, S. Micali and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing,* vol. 17, no. 2 (April 1988), pp. 281-308.

[Ro95] P. Rogaway. Bucket Hashing and its application to Fast Message Authentication. Proceedings of CRYPTO '95, Lecture Notes in Computer Science, vol. 963, Springer-Verlag, 1995, pp. 15-25.

[Kr94] H. Krawczyk. LFSR-based Hashing and Authentication. Proceedings of CRYPTO '94, Lecture Notes in Computer Science, vol. 839, Springer-Verlag, 1994, pp. 129-139.

[Kr95] H. Krawczyk. New Hash Functions for Message Authentication. Proceedings of EUROCRYPT '95, Lecture Notes in Computer Science, vol. 921, Springer-Verlag, 1995, pp. 301-310.

[Sh96] V. Shoup. On Fast and Provably Secure Message Authentication Based on Universal Hashing to appear in CRYPTO '96

[St94] D. Stinson. Universal Hashing and Authentication Codes. Designs, Codes and Cryptography, vol. 4, 1994, pp. 369-380.

[St95] D. Stinson. On the Connection Between Universal Hashing, Combinatorial Designs and Error-Correcting Codes. TR95-052, Electronic Colloquium on Computational Complexity, 1995.