MMH: Software Message Authentication in the Gbit/second Rates*

Shai Halevi[†] Hugo Krawczyk[‡]

Extended Abstract

March, 1997

Abstract

We describe a construction of almost universal hash functions suitable for very fast software implementation and applicable to the hashing of variable size 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/sec, depending on the desired level of security (or collision probability), and a rate of more than 1 Gbit/sec on a 200 MHz Pentium-Pro. This represents a significant speed-up over current software implementations of universal hashing and other message authentication techniques (e.g., MD5-based). Moreover, our construction is specifically designed to take advantage of emerging microprocessor technologies (such as Intel's MMX, 64-bit architectures and others) and then best suited to accommodate the growing performance needs of cryptographic (and other universal hashing) applications.

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.

*Appeared in *Proceedings of the 4th Workshop on Fast Software Encryption*, LNCS vol. 1267, Springer, 1997. Pages 172-189.

[†] Lab. of Computer Science, Massachusetts Institute of Technology, 545 Tech Square, Cambridge, MA 02139, USA. Email: shaih@theory.lcs.mit.edu. Work was done while the author was at IBM Watson Research Center.

[‡] IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, New York 10598, USA. Email: 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, search engines, parallel architectures, complexity theory, cryptography, and many others. The use of universal hashing for message authentication (introduced by Wegman and Carter [WC81] as well) received much attention lately. In particular, many recent works deal with efficient implementation of universal hashing as a tool for achieving fast and secure message authentication (e.g., [St94, Kr94, Kr95, Ro95, AS96, HJ96, Sh96, AGS97]). This is also the motivation for our work; however, the construction presented here applies to the other (non-cryptographic) uses of universal hashing as well.

Roughly speaking, universal hash functions are collections of hash functions that map strings (or messages) into short outputs such that the probability of any given pair of messages to collide (i.e., have the same hash value) is small. This probability does not depend on any particular distribution of the input data but only on the random choice of the particular function used to hash the data from the set of all hash functions in the universal family. A stronger version of universal hash functions guarantees that elements are mapped into their images in a pairwise independent way.

These properties make universal hashing a prime tool for data storage and retrieval. In addition, as originally observed in [WC81], universal hashing can be used for building secure message authentication schemes where the adversary's ability to forge messages is bounded by the collision probability of the hash family. In such a scheme, a message is authenticated by first hashing it and then encrypting the hash value with a one-time pad (where the pad is of the length of the hash output rather than of the length of the message). The resultant encrypted hash is transmitted together with the message as an authentication tag that is recomputed and validated by the receiver.

In this setting the communicating parties share a secret and random index to a particular function in the hash family as well as the random one-time pads used for encryption. The security of such a message authentication scheme is *unconditional*, namely, *no adversary* (not even a computationally unlimited one) can forge a message with probability better than the collision probability of the universal hash family. In practice, one time pads are usually replaced with pseudorandom pads (or pseudorandom functions) and the security conditioned on the strength of this encryption.

The attractiveness of using universal hashing in the context of message authentication comes then from two sources. First, it allows decoupling the cryptographic work (reduced to the encryption of the hash value) from the bulk work on the data. Second, given the simplicity of the requirements from a universal hash function it allows for potentially efficient constructions.

In this paper we strongly demonstrate the validity of these properties. We show how to build very fast universal hash families with good (and controllable) levels of security. Our emphasis is on high speed implementation using software only. (For hardware optimized universal hashing see [Kr94].) To this end, we exploit todays' microprocessor technology as well as the current trends in microprocessor design.

On a very high level, this construction is obtained by implementing a well-known family of universal hash functions and then modifying the implementation so as to eliminate costly software operations. The result can be thought of as a "buggy implementation" of the original functions, but with a much faster software implementation. Most importantly, we can prove that the obtained construction is "almost as good" (for its collision probability and security) as the original function.

We report on a 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 collision probability. The same implementation in a 200 MHz Pentium-Pro exceeds the 1 Gbit/second speed (for a 32 bit hash value).

This represents a significant speed-up over current software implementations of universal hashing, or any other secure message authentication technique. An exact comparison is not possible since the data available on the most efficient implementations of other functions are based on different platforms. The reader is referred to [Sh96] for results on the implementation of division hash (or cryptographic CRC [Ra79, Kr94]), and to [BGV96] for results on the implementation of MD5 and SHA-1 which are currently the most popular bases for software implementation of message authentication codes (e.g, [BCK96]). The best reported time on these functions is 114 Mbit/sec for a hand-optimized assembly implementation of MD5 in a Pentium 90 MHz, and half of it for SHA[1]. (See also [To95] for an analysis of the inherent performance limits of MD5 and for motivation of the needs for faster message authentication techniques.)

Very importantly, our construction is specifically designed to take further advantage of emerging microprocessor technologies (such as Intel's MMX, 64 bit architectures and others) in order to accommodate the growing performance needs of cryptographic applications. (In particular, *single precision* scalar-products are increasingly supported in these new architectures as a means to accelerate multimedia and graphic applications; MMH takes direct advantage of that acceleration by using single precision scalar-products as its most basic operation.) It is worth remarking that the need for performance in the Gbit/sec range is not just for Gbit networks; the goal is that machines will spend only a small portion of their power (say less than 10%) in cryptographic operations while they can use most of that power to do other operations (e.g. doing something "useful" with the authenticated data, like playing a multimedia title).

What's in the name. The name MMH stands for Multilinear-Modular-Hashing. It is also intended to hint to MultiMedia applications, which serve both as motivation for the need of fast software message authentication (for example, to verify the integrity of an on-line multimedia title), and as the motivation for the improved support of integer scalar-products in modern microprocessors, which is a crucial factor for MMH high performance.

Organization. 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 describe our modifications of this well-known construction and our implementation of the resulting function, and provide some experimental results. In Section 4 we show that the resulting function is "almost as good" as the original one, and thus suitable for secure message authentication applications as well as other universal hashing uses. Finally, a related and alternative construction is discussed in Section 5. In Appendix A we present sample C code for the implementation of the core routine in MMH.

2 Preliminaries

2.1 Notation

For integers y, z, p, 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 mod p. By $z \bmod p$ we denote the residue of the division of z by 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 $\{0, 1, \cdots 2^{32} - 1\}$.

[1] By extrapolating these figures one could expect a maximal rate of about 300 Mbit/sec for MD5 in a Pentium-Pro 200 MHz, though no such actual implementation is known to us.

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 constant $1/|R| \leq \epsilon \leq 1$. The probabilities below, denoted by $\Pr_{h \in H}[\cdot]$, are taken over the choice of $h \in H$ according to a given probability distribution on H (usually, the uniform distribution). These definitions and terminology are due to Carter and Wegman [CW79, WC81], Krawczyk [Kr94], Rogaway [Ro95] and Stinson [St95].

Definition 1:

1. 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 $x \neq y \in D$, $\Pr_{h \in H}[h(x) = h(y)] \leq \epsilon$

2. 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 $a \in R$, $\Pr_{h \in H}[h(x) - h(y) = a] \leq \epsilon$

   (We stress that Δ-universality is relative to a given group operation in the set R.)

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

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

2.3 Universal hashing and message authentication

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 communications over this link, the legitimate parties share a secret key which is unknown to the adversary. They use this secret key to compute a *message authentication code* (MAC) on 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 sends the pair (m, μ) over the untrusted link. On receipt of (m', μ'), the receiver repeats this computation and verifies that $\mu' = MAC_x(m')$.

We evaluate the security of a MAC function in the usual model which was introduced in [GMR88]. The adversary A, which is not given the shared secret key, has as a goal to *forge* the MAC value for a message not sent between the legitimate parties. In order to do so, A can eavesdrop the communication between these parties, choose the messages to be sent between them (i.e., to see the output of MAC_x computed on messages of its own choice), and can also modify the message and tags in their way between sender and receiver. In the later case A gets to see whether the replaced values are accepted or not by the receiver; we call these attempts "verification queries". If any of these verification queries uses a message not previously sent between the legitimate parties and is accepted by the receiver (i.e., the right MAC_x was computed by the adversary) then the MAC is broken. For a MAC function to be "good", any adversary with "reasonable" resources (time, memory, number of queries, etc.) should have only a negligible probability of breaking the MAC. We refer to [BKR94] for a formal definition of security of MAC functions.

In the Wegman-Carter paradigm [WC81], the secret key shared by the communicating parties consists of a hash function h drawn randomly from a family of hash functions H and a sequence of random pads $d_1, d_2, \ldots$. To authenticate the i-th message m_i, the sender computes the authentication tag $h(m_i) + d_i$, that is, $MAC_{h,d}(m_i) \stackrel{\text{def}}{=} h(m_i)+d_i$. If h is drawn from an ϵ-AΔU-universal family the probability of an adversary (even one with unlimited computational power) to forge a single message is bounded by ϵ [WC81, Kr94]. If the attacker is allowed to perform q verification queries then its probability to successfully forge a MAC is at most $q\epsilon$. (Notice that in this case passive gathering of information does not buy anything to the attacker, only active tampering with messages and authentication tags can help him, thus making the attack harder to mount and easier to detect.) Hence, universal hashing provides with a simple paradigm for achieving secure cryptographic message authentication.

One important thing to notice is that in this approach one first processes the message m using a non-cryptographic operation (universal hashing), and then applies a cryptographic operation (one-time-pad encryption) on $h(m)$, which is typically much shorter than m itself. In practical implementations, the random pad may be replaced by a pseudo-random one, so the parties only need to share the function h and the seed s to the pseudo-random generator (s can also be a key to a pseudorandom function). One can also directly apply a pseudorandom function to the output of the hash function concatenated with a counter. The reader is referred to [WC81, Br82, St94, Kr94, Ro95, Sh96] for more elaborate discussions of these issues.

In the reminder of this paper we concentrate on the construction of an efficient ϵ-AΔU-universal family of hash functions for small ϵ.

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

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

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

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

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 insufficient (i.e., 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 $1/p^2$, at the expense of doubling the computational work and length of output. It also requires double-size keys.

The last aspect (key size) can be resolved by recurring to a "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 $\mathbf{x} = \langle x_1, \cdots, x_k \rangle, \mathbf{x}' = \langle x_2, \cdots, x_{k+1} \rangle$. It is a well-known fact that such a choice of keys will still result in a reduced collision probability of $1/p^2$ while increasing the key size by a single scalar x_{k+1}. The same methodology can be applied to further reduce the collision probability to $1/p^n$ for any integer value n. In this case the computational work and output are increased by a factor of n, while keys are only increased with n scalars.

Dealing with arbitrary long messages. The above function can only be applied to a fixed-size messages (namely, to vectors in ${Z_p}^k$). The standard approach for dealing with messages of arbitrary length is to use tree-hashing as already suggested by Carter and Wegman [WC81]. That is, we break the message into blocks of k elements (over Z_p) each and hash each block separately (using the *same* hash function). We then concatenate the hash results of all these blocks 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 grow linearly with the height of the tree. A suggestion to counter this problem appears in section 3.4.)

3 Multilinear Modular Hashing

The MMH family described here is a variant of the construction MMH* described in section 2.4 designed to achieve fast software implementations while preserving the low collision probability. For the sake of simplicity, we describe the function for a specific set of parameters (particularly suited for 32-bit word architectures); however, the approach is general and can be used with different parameters according to application needs and hardware platform.

The main characteristics of our implementation are

- We work with 32-bit integers (as said 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.[2]

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

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

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 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.

[2] The idea is adopted from a suggestion by Carter and Wegman [CW] to use the primes $2^{16} + 1$ or $2^{31} - 1$. In our approach, any prime which satisfies $2^{32} < p < 2^{32} + 2^{16}$ will do; $2^{32} + 15$ is the smallest among those primes.

3.1 Implementing the "ideal" function

Modular reduction. The one operation in MMH_{32}^* which is by far the most expensive is the modular reduction. However, since we work with the prime $p = 2^{32} + 15$, we can implement a division-less modular reduction as follows.

Let x be an (unsigned) 64-bit integer, and denote $x = 2^{32}a + b$, where a, b are both unsigned 32-bit integers. Then we note that

$$2^{32}a + b \equiv (2^{32}a + b) - a \cdot (2^{32} + 15) = b - 15a \pmod{2^{32} + 15}$$

Moreover, since $a, b \in [0, 2^{32} - 1]$, then $b - 15a \in [-15 \cdot (2^{32} - 1),\ 2^{32} - 1]$. Thus, if we denote $y = b - 15a$, then $y \equiv x \pmod{2^{32} + 15}$, and y can be represented as a signed 64-bit integer (in two's complement) $y = c \cdot 2^{32} + d$, where $c \in \{-15, \ldots, 0\}$ and d is an unsigned 32-bit integer.

We can now repeat this process once more and compute $z = d - 15c$. Then $z \equiv y \equiv x \pmod{2^{32} + 15}$, and $z \in \{0, \cdots 2^{32} + 15^2\}$. Finally, we test to see if z is still larger than $2^{32} + 15$. If not we return z, otherwise we return $z - (2^{32} + 15)$.

Inner-product. Even the above implementation of a division-less reduction still takes about 10-15 machine instructions, which is very expensive if we need to execute it too often. Therefore, we carry the whole 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. This, in particular, involves the use of 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 (in 32-bit word architectures the 64 bit result of the multiplication is returned in two 32-bit registers). However, there is no (official) syntax in high-level programming language to access these operations, so we write our implementation in assembly language.[3]

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

- Since we amortize the costly modular reduction over k (cheaper) multiply-and-add operations, increasing k should increase the speed.
- 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 modulo 2^{32}.

[3]Even if not part of the language definition, some C compilers support 64 bit types (*long long integers*). See Appendix A.

- 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

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

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

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

In Section 4 we show that MMH_{32} is ϵ-AΔU with $\epsilon = \frac{1.5}{2^{30}}$.

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.

A pseudo-code for MMH on such machine may be as follows

```
MMH(msg, key)
    SumHigh = SumLow = 0
    For i = 1 to k
        load msg[i]
        load key[i]
        <ProdHigh, ProdLow> = msg[i] * key[i]
        SumLow = SumLow + ProdLow
        SumHigh = SumHigh + ProdHigh + carry
    Reduce <SumHigh, SumLow> mod 2^32 + 15 and then mod 2^32
```

Each multiply-and-add operation takes total of about 7 instructions: 2 for loading the message- and key-words to registers, 2 for the multiplication, 2 for the addition, and 1 more 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 (more precisely, several k-word message blocks) 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 instructions per 32 bits of input. (An even faster implementation in a 64-bit machine can be achieved by working on 64-bit words using a prime modulus which is slightly larger than 2^{64} – e.g. $2^{64} + 13$ – as long as the architecture supports the integer multiplication of two 64 bit words.) The implementation of the hashing-tree adds less than 10% to the total work of the function.

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.) even for standard processors; therefore, we can expect even faster implementations of MMH in the near future.

3.3 Experimental results

Below we describe the results of a few experimental implementations of MMH. We implemented MMH on PowerPC and Intel x86 architectures. The basic MMH function itself was hand-optimized in assembly language on each machine and the tree structure and various initializations were implemented in C. For each of these architectures 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 $\frac{1.5}{2^{30}}$ 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 $\frac{2.25}{2^{60}}$ times the height of the tree.

For each version we performed two different tests: First we tested what happens when the message is long and resides in a memory buffer. We evaluated the hash function on a 4 Mbyte buffer and repeated it 64 times. This yields total length of 256 Mbyte = 2 Gbit. Below we refer to this test as the "message in memory" test. Then we performed another test to find how much of the running time is spent on cache misses. For that we modified the code so that whenever it access data, it always takes it from the same memory buffer (of size a few Kbytes). We refer to this test as the "message in cache" test.

To get a good assessment of the performance potential of our construction in different architectures, we tested our implementation on the following platforms

- A 150 MHz PowerPC 604 RISC machine running AIX.
- A 150 MHz Pentium-Pro machine running Windows NT.
- A 200 MHz Pentium-Pro machine running Linux.

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

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

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

200 MHz Pentium-Pro	message in memory	message in cache
64-bit output	380 Mbit/second	500 Mbit/second
32-bit output	645 Mbit/second	1080 Mbit/second

Table 1: Timing results for various implementations of MMH.

We also tested MMH on a Pentium machine. However, the integer multiplication in the Pentium is slow and therefore we obtained our best results by using the floating-point unit for the multiply-and-add operations.[4] This implementation achieves a rate of about 160 Mbit/second on a 120 MHz Pentium for the 64-bit variant (message in memory). This is somewhat faster than the performance reported in [Sh96] for the polynomial division function and in [BGV96] for MD5, but not as impressive as the other speeds reported above.

It is important to note that the above results are for bulk data processing. For particular applications, the actual effect of these faster functions depends on the details of the application, the length of authenticated (or hashed) data, etc. In particular, when used for message authentication the operation of encrypting the hash value (e.g., the generation of a pseudorandom one-time pad) can be a significant overhead for very short messages. However, we remark that MMH does not need of particularly long messages in order to achieve its superior performance relative to other universal hash functions (this is to be contrasted, for example, with bucket hashing [Ro95]).

3.4 Further issues and variants

Variants. Some further optimizations to our implementation can be achieved by introducing some changes to the definition of MMH. In particular, by exploiting some architecture-specific optimizations we have achieved performance improvements of about 10% over the above reported figures. In the Pentium-Pro 200 MHz, for example, this brings the 32-bit function speed to 1.2 Gigabit/second (at the cost of a slight increase in the collision probability of the function).

Mixing hash functions. Most of our description above concentrated on the basic MMH function as applied to fixed length messages (e.g., k-word long). The techniques described in Section 2.4 were used to implement the function for arbitrary length messages (the experimental results of section 3.3 correspond to this general construction). A drawback of this implementation is that the length of the keys not only depends on the parameter k but also on the height of the hash-tree (a different key is needed for each level in the tree even if that key is seldom used by the function). One approach to overcome this drawback is to apply the proposed function MMH only to the top levels of the tree (one or two levels) and then to use a different hash family to hash the result from these levels. The idea is to use for this second hashing a hash family that requires shorter keys even if it is slower than MMH. Since the data hashed by this function is much shorter than the original message (e.g., by a factor of $1/k^2$) the inferior performance of the second function would not be noticeable. A reasonable choice for the second hashing scheme can be the polynomial-evaluation-hashing. The reader is referred to [Sh96, AGS97] for a description and implementation details of this scheme.

Hashing short data items. Our hash functions are particularly flexible as for the way they deal with information of different sizes. While long streams of data can be processed as explained above, short strings of data can also be hashed very efficiently by just choosing a key which is not shorter than any such string. For example, a compiler that hashes a symbol table where no such symbol exceeds 1 Kbit in length can choose a 1 Kbit long key and hash all the symbols using that single key. In this case, there is no need for the

[4] This requires some modification in the definition of MMH.

hash tree technique. (Notice that shorter than 1 Kbit strings will just use the part of the key corresponding to their length.)

Padding to block boundaries. Another issue related to dealing with variable length messages is the need to pad data to some block boundary. This can be easily handled by appending some prescribed pad to the end of the data. In the case of message authentication it is particularly important that the padding will be unambiguous, namely, two different messages are mapped into different padded strings. (For example, a pad formed by concatenating a '1' followed by a suitable number of '0's could be appended to every message.)

Generation and sharing of long keys. Depending on the variant and application of MMH one may need long keys (e.g., a few Kbits). These keys can be generated using a strong pseudorandom generator. In particular, in the case of such a key being shared by two parties, only the seed to the pseudorandom generator needs to be exchanged (such a seed will be considerably shorter than the key, e.g. 100-200 bit long).

Byte ordering. For the purpose of hashing we identify streams of data with a sequence of integer numbers (e.g., 32-bit integers). However, different computer architectures load data bytes into words in different orders. Thus, when interested in inter-operability between different machines one needs to specify a particular loading order (little-endian or big-endian). Any such specification will favor one architecture or the other. We do not provide such a specification at this point. One thing to notice is that while all architectures provide instructions for switching their default order this conversion can cause a degradation in the performance of the function in these architectures. (See [BGV96, To95] for a related discussion.)

4 Analysis of the collision probability

In this section we analyze the collision probability of the Multilinear Modular Hash function. For simplicity, we concentrate on the parameters of MMH_{32}, however, the same analysis works for similar constructions using word-length other than 32 bits.

We start 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 $\epsilon \leq 2 \cdot 2^{-32}$.

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 $x_2, \cdots, x_k$, $\Pr_{x_1}[\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \equiv a \pmod p] \leq 2/2^{32}$, which implies the lemma.

Since $x_1 m_1 < 2^{64}$ for any value of x_1, then for any choice 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

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

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

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

This bound can be improved to $2 \cdot 2^{-32}$ by noticing that the difference $b(x_1) - b'(x_1)$ cannot assume simultaneously the values 1 and -1. Namely, if there exists a value of x_1 for which $b(x_1) = 1$ and $b'(x_1) = 0$ then there cannot be another value x_1' for which $b(x_1') = 0$ and $b'(x_1') = 1$, and vice-versa. Indeed, having $b(x_1) = 1, b'(x_1) = 0$ for a given x_1 means that $x_1 m_1 + s \ge 2^{64}$ while $x_1 m_1' + s' < 2^{64}$. Since the expressions $x_1 m_1 + s$ and $x_1 m_1' + s'$ are monotonic increasing in x_1, then there cannot be another value x_1' for which $x_1' m_1 + s < 2^{64}$ while $x_1' m_1' + s' \ge 2^{64}$. That is, there is no x_1' for which $b(x_1') = 0$ and $b'(x_1') = 1$. □

We now show MMH_{32} to be a good -AΔU universal family (relative to the mod2^{32} subtraction).

Theorem 3: MMH_{32} is an ϵ-AΔU family of hash functions with $\epsilon \le 6 \cdot 2^{-32}$.

Proof: Recall that MMH_{32} is obtained from H_{32} by reducing the result modulo 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 value $v, 0 \le v < 2^{32}$ and two different message-vectors $\mathbf{m}, \mathbf{m}'$, and let $\mathbf{x}$ be a key-vector so that $h_{\mathbf{x}}(\mathbf{m}) - h_{\mathbf{x}}(\mathbf{m}') \equiv v \pmod{2^{32}}$. Equivalently, we have $\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \equiv v \pmod{2^{32}}$. Since the values $\tilde{h}_{\mathbf{x}}(\mathbf{m})$ and $\tilde{h}_{\mathbf{x}}(\mathbf{m}')$ are both between 0 and $p-1$ we get that their difference (over the integers) lies between $-p+1$ and $p-1$. As $p = 2^{32} + 15$, we get

$$\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \in \begin{cases} \{v - 2^{32}, v, v + 2^{32}\} & 0 \le v < 15 \\ \{v - 2^{32}, v\} & 15 \le v \le 2^{32} - 15 \\ \{v - 2 \cdot 2^{32}, v - 2^{32}, v\} & 2^{32} - 15 < v < 2^{32} \end{cases}$$

That is, if $h_{\mathbf{x}}(\mathbf{m}) - h_{\mathbf{x}}(\mathbf{m}') \equiv v \pmod{2^{32}}$ then over the integers the difference $\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}')$ can assume at most 3 values. But then this is also true for this difference when taken mod p. Lemma 4.1 tells us that for any value v' the probability (over the choice of $\mathbf{x}$) that $\tilde{h}_{\mathbf{x}}(\mathbf{m}) - \tilde{h}_{\mathbf{x}}(\mathbf{m}') \equiv v' \pmod{p}$ is at most $2/2^{32}$. And then the probability for any value of v that $\mathbf{x}$ solves the equation $h_{\mathbf{x}}(\mathbf{m}) - h_{\mathbf{x}}(\mathbf{m}') \equiv v \pmod{2^{32}}$ is at most 3 times larger. In other words,

$$\Pr_{\mathbf{x}}\left[h_{\mathbf{x}}(\mathbf{m}) - h_{\mathbf{x}}(\mathbf{m}') \equiv v \pmod{2^{32}}\right] \leq 3 \cdot 2 \cdot 2^{-32}.$$

$\square$

5 Further Work

Recently, Mark Wegman has suggested to us the use of an unpublished universal hash function invented by Larry Carter and himself many years ago. This function is related to the construction, by the same authors, that we presented in section 2.4 and which we called MMH*. This "new" function is not linear and then we denote it by NMH* (for Non-linear).

Definition 5: Let p be a prime and let k be an even positive integer. Define a family NMH* of functions from Z_p^k to Z_p as follows

$$\mathrm{NMH}^* \stackrel{\mathrm{def}}{=} \{g_{\mathbf{x}} : Z_p^k \rightarrow Z_p \mid \mathbf{x} \in Z_p^k\}$$

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

$$g_{\mathbf{x}}(\mathbf{m}) \stackrel{\mathrm{def}}{=} \sum_{i=1}^{k/2} (m_{2i-1} + x_{2i-1})(m_{2i} + x_{2i}) \bmod p$$

It is not hard to see that NMH* is Δ-universal. This function uses the same number of arithmetic operations as MMH* but requires half of the number of multiplications (at the expense of more additions). At least in machines where multiplication is significantly slower than addition its performance should be expected to be better than that of MMH*. We modify NMH* as we did with MMH* for improved performance and define NMH_{32} as follows.

Definition 6: Set $p = 2^{32} + 15$ and $k = 32$. Define a family NMH_{32} of functions from $(\{0,1\}^{32})^k$ to $\{0,1\}^{32}$ as follows

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

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

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

(the symbol $+_{32}$ denotes addition modulo 2^{32}):

We can show in a similar way as we did for MMH_{32} that NMH_{32} is ϵ-AΔU for ϵ close to 2^{-32}.

We have not yet implemented this function. Report on such an implementation will be presented in the future.

Acknowledgments.

We thank Mark Wegman for invaluable comments and suggestions on this work, and Robert Geva for his help with the Pentium-Pro implementation. We also wish to thank Matteo Frigo for suggesting the 'C' implementation in Appendix A and Dave Wagner for helping us to get the Pentium-Pro 200 performance measurements.

References

[AGS97] V. Afanassiev, C. Gehrmann and B. Smeets. Fast Message Authentication using Efficient Polynomial Evaluation Appeares in these proceedings.

[AS96] M. Atici and D. Stinson. Universal Hashing and Multiple Authentication *Advances in Cryptology – CRYPTO '96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz, ed., Springer-Verlag, 1996. pp. 16-30.

[BCK96] M. Bellare, R. Canetti and H. Krawczyk. Keying hash functions for message authentication. *Advances in Cryptology – CRYPTO '96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz, ed., Springer-Verlag, 1996. pp. 1-15.

[BKR94] M. Bellare, J. Kilian and P. Rogaway. The security of cipher block chaining. *Advances in Cryptology – CRYPTO '94 Proceedings*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt, ed., Springer-Verlag, 1994. pp. 341-358.

[BGV96] A. Bosselaers, R. Govaerts, J. Vandewalle. Fast Hashing on the Pentium, *Advances in Cryptology – CRYPTO '96 Proceedings* Lecture Notes in Computer Science Vol. 1109, N. Koblitz, ed., Springer-Verlag, 1996. pp. 298- 312.

[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 Communication.

[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.

[HJ96] T. Helleseth and T. Johansson. Universal Hash Functions from Exponential Sums over Finite Fields *Advances in Cryptology – CRYPTO '96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz, ed., Springer-Verlag, 1996. pp. 31-44.

[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.

[Ra79] Rabin, M.O., "Fingerprinting by Random Polynomials", Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass., 1981.

[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.

[Sh96] V. Shoup. On Fast and Provably Secure Message Authentication Based on Universal Hashing *Advances in Cryptology – CRYPTO '96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz, ed., Springer-Verlag, 1996. pp. 313-328.

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

[To95] J. Touch. Performance Analysis of MD5. Proc. Sigcomm '95, Boston, pp. 77-86.

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

[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.

A A 'C' Implementation of MMH

Below we describe a sample 'C' implementation of MMH, using the `long long` data type of gcc to handle 64-bit integers. Note that the code below does not include an implementation of the hashing tree, nor does it implement the reduced collision probability from Section 2.4 (Page 4). Rather, it is just a straightforward implementation of the basis MMH_{32} function, as defined in Definition 3.

In the following code, the 32-word message is stored in the `msg[]` buffer and the 32-word key is stored in the `key[]` buffer. The following is a straight line code rather than in a loop, to take maximum advantage of the optimizing capabilities of the compiler.

```
#define DO(i)  sum += key[i] * (unsigned long long) msg[i]

unsigned long basic_mmh(unsigned long *key, unsigned long *msg)
{
     signed long long stmp;          /* temporary variables */
     unsigned long long utmp;

     unsigned long long sum = 0LL;    /* running sum */

     unsigned long ret;              /* return value */

     DO(0);     DO(1);     DO(2);     DO(3);
     DO(4);     DO(5);     DO(6);     DO(7);
     DO(8);     DO(9);     DO(10);    DO(11);
     DO(12);    DO(13);    DO(14);    DO(15);
     DO(16);    DO(17);    DO(18);    DO(19);
     DO(20);    DO(21);    DO(22);    DO(23);
     DO(24);    DO(25);    DO(26);    DO(27);
     DO(28);    DO(29);    DO(30);    DO(31);

     /********** return (sum % 0x10000000fLL); **********/

     stmp = (sum  & 0xffffffffLL) - ((sum  >> 32) * 15);  /* lo - hi * 15 */
     utmp = (stmp & 0xffffffffLL) - ((stmp >> 32) * 15);  /* lo - hi * 15 */

     ret = utmp & 0xffffffff;
     if (utmp > 0x10000000fLL) /* if larger than p - subtract 15 again */
         ret -= 15;

     return ret;
}
```