On Fast and Provably Secure Message Authentication Based on Universal Hashing*

Victor Shoup

Bellcore, 445 South St., Morristown, NJ 07960

shoup@bellcore.com

December 4, 1996

Abstract

There are well-known techniques for message authentication using universal hash functions. This approach seems very promising, as it provides schemes that are both efficient and provably secure under reasonable assumptions. This paper contributes to this line of research in two ways. First, it analyzes the basic construction and some variants under more realistic and practical assumptions. Second, it shows how these schemes can be efficiently implemented, and it reports on the results of empirical performance tests that demonstrate that these schemes are competitive with other commonly employed schemes whose security is less well-established.

1 Introduction

Message Authentication. Message authentication schemes are an important security tool. As more and more data is being transmitted over networks, the need for secure, high-speed, software-based message authentication is becoming more acute.

The setting for message authentication is the following. Two parties A and B agree on a secret key a. A message authentication scheme consists of two algorithms S and V. If A wants to send a message x to B, then A first computes the message authentication code, or MAC, $\alpha = S_a(x)$, and sends the pair (x, α) to B. When B receives a pair (x, α), B evaluates $V_a(x, \alpha)$, which returns 1 if the MAC is valid, and 0 otherwise.

Security for message authentication schemes can be formally defined, as in Bellare *et al.* [4], essentially along the same lines as for digital signatures [8]: we say that an adversary forges a MAC if, when given oracle access to S_a and V_a, it obtains $V_a(x, \alpha) = 1$ for some message x that was never given to the oracle for S_a; a message authentication scheme is secure if it is computationally infeasible to forge a MAC.

Common Approaches to Message Authentication. One of the most widely used message authentication schemes is built using a block cipher, typically the Data Encryption Standard (DES), and applying it to the message in Cipher Block Chaining (CBC) mode. Only recently has this scheme been shown to be secure [4], under a reasonable assumption about DES, although the level of security provided by this scheme degrades quite quickly as the number of queries or the message length increases. Moreover, as DES is applied to every block of the message, this scheme is quite slow, especially in software.

Another common practice today is to use a cryptographic hash function h, such as MD5, and set $S_a(x) = h(a \cdot x \cdot a)$, where "$\cdot$" denotes concatenation. Many variations on this scheme have been proposed as well (see [16])). These schemes are typically much faster than the CBC-DES scheme; unfortunately, the security of these schemes is not well-established; to obtain much confidence in the security of this approach, one must

*A preliminary version of this paper appears in *Proc. Crypto '96*, pp. 313-328, 1996.

assume a good deal more about the properties of h than seems warranted (but see [2] for some progress in this area).

The Universal-Hash Construction. The problem of message authentication was studied early on in an information-theoretic setting, first by Gilbert *et al.* [7], and later by Wegman and Carter [18]. Wegman and Carter's *universal-hash construction* was later placed in a cryptographic setting by Brassard [6], Krawczyk [12], and Rogaway [17]. This construction uses a 2-universal family H of hash functions, and a pseudo-random family F of functions. Assume that the outputs of both types of functions are bit strings of the same length, say l. The secret key for such a scheme consists of a pair (h, f), where $h \in H$ and $f \in F$ are chosen at random. The MAC for a message x is $(r, f(r) \oplus h(x))$, where the "tag" value r is a counter that is incremented with each application of algorithm S.

Actually, one does not need a 2-universal family of hash functions, but rather, a family of hash functions satisfying the following property for suitably small $\epsilon \geq 2^{-l}$: for any pair of inputs $x_1 \neq x_2$ and for any l-bit string z, for a random $h \in H$, the probability that $h(x_1) \oplus h(x_2) = z$ is no more than ϵ. In this case, we say H is an ϵ-AXU (almost exclusive-or universal) family of hash functions.

The main theorem concerning the security of the basic universal-hash construction is the following (see [17] and [12] for more details and references).

Theorem 1 *Assume H is ϵ-AXU, and that F is replaced by the truly random family R of functions. In this case, if an adversary makes q_1 queries to S and q_2 queries to V, the probability of forging a MAC is at most $q_2\epsilon$.*

If in passing from R to F the forgery probability should significantly increase, this would give us a statistical test to distinguish F from R that makes $q_1 + q_2$ queries to the test function.

Our Contributions. We contribute to this line of research in two ways. In the first part, §§2-3, we analyze the basic construction and some variants under more realistic and practical assumptions. In the second part, §§4-7, we show how schemes based on universal hashing can be efficiently implemented, and we report on the performance of these implementations.

New Analysis and Constructions. Consider the choice of the family F of pseudo-random functions F. Since f is evaluated at just a single counter value per message, one can usually afford to employ a function with strong security properties, but which may be somewhat slow to evaluate. A block cipher such as DES seems like a very good choice.

There is, however, an irritating problem with using DES in conjunction with Theorem 1: namely, DES is a permutation (on 64-bit strings). The level of security implied by Theorem 1 decreases quadratically with q_1+q_2, and as q_1+q_2 nears 2^{32}, Theorem 1 says *nothing at all* about the security of the message authentication scheme. This is because with close to 2^{32} queries to a test function, we *can already* distinguish DES from a random function, since DES will not yield any collisions, unlike a random function.

There are several cryptographic constructions in the literature (e.g., [3, 1]) that suffer from the same problem.

In §2, we analyze the security of the universal-hash construction using pseudo-random permutations, and show that it is in fact more secure than implied by the above theorem. We also give a small modification to the universal-hash construction with even better security properties.

Another potential problem with the basic universal-hash construction is that algorithm S is not stateless. This might be inconvenient in certain situations where reliably maintaining state is difficult, or where many parties are authenticating with the same key. In §3, we show a modification to the basic construction that is stateless and efficient, while still being just as secure as the basic universal-hash construction.

Fast Implementations. The most critical aspect of the universal-hash construction in terms of performance is the family H of hash functions. We need to be able to generate random elements of H reasonably quickly, and more importantly, we need to be able to apply functions in H to messages very quickly.

We discuss three types of hash functions based on polynomials over finite fields. We show how these three types of hash functions can be efficiently implemented in software, and we report on the performance of these implementations. In §4 we present the three hash functions under consideration, and summarize our empirical results. In §5-7 we discuss our implementations of these functions, as well as some possible alternative implementations. Our results indicate that on typical workstations and personal computers, the performance of these hash functions is competitive with that of other commonly employed authentication schemes whose security is less well established.

Some of our techniques may be useful in other contexts as well, such as our method for constructing a random irreducible polynomial of given degree over GF(2).

2 Using a Pseudo-Random Family of Permutations

As mentioned in the introduction, the established theory on the universal-hash construction is not adequate to explain what happens when pseudo-random permutations are used instead of pseudo-random functions. The following theorem is useful in that regard.

Theorem 2 *In the basic universal-hash construction, suppose H is ϵ-AXU, and that F is replaced by the truly random family P of permutations on l-bit strings. Suppose that the adversary makes q_1 queries to S and q_2 queries to V. Then provided $q_1^2 \leq 1/\epsilon$, the probability that the adversary forges a MAC is at most $2q_2\epsilon$.*

This theorem is proved in the Appendix A.

As usual, if in passing from P to a pseudo-random family F of permutations, the forgery probability increases significantly, we get a statistical test distinguishing F from P.

The usefulness of this theorem depends on the ϵ; for long messages, there is usually a trade-off between the efficiency of the hash function and $1/\epsilon$. This motivates the following construction.

Let F be a family pseudo-random permutations on l bits. Let H_1 be an ϵ_1-AXU family of hash functions, and H_2 an ϵ_2-AXU family of hash functions. Assume these functions have l-bit outputs and that functions in H_1 have l-bit inputs.

As in the basic universal-hash construction we use a tag value r that is a counter incremented with each invocation of S. The secret key for the MAC consists of $f \in F$, $h_1 \in H_1$, and $h_2 \in H_2$, chosen randomly. The MAC for a message x is $(r, f(r) \oplus h_1(r) \oplus h_2(x))$.

Theorem 3 *Suppose that F is replaced by the truly random family P of permutations, and that an adversary makes q_1 queries to S and q_2 queries to V. Then provided $q_1^2 \leq 1/\epsilon_1$, the probability that the adversary forges a MAC is at most $2q_2\epsilon_2$.*

This theorem is also proved in appendix A.

As an example, suppose we are using DES and $l = 64$. Since h_1 is applied to a short string, we can afford to use a family H_1 with $\epsilon_1 = 1/2^{64}$. The theorem says we should use algorithm S no more than 2^{32} times, at which point we should switch the MAC key. But note that until this point is reached (if ever), the security degrades only very little.

3 Using a Random Tag

Consider the basic universal-hash construction. Let H be an ϵ-AXU family of hash functions, and F a pseudo-random family of functions, all functions mapping to l-bit strings, and that the functions in F have l-bit inputs. To make S stateless, instead of a counter, we might use a random l-bit tag. However, the security in this case can degrade very rapidly. After $O(l^{1/2})$ queries to the S-oracle, it is likely that two tag values collide. Depending on the family of hash functions, this event can compromise the scheme completely (this is certainly true for the hash functions discussed in this paper).

One solution is to double the length of the random tag. However, we then need a pseudo-random function from $2l$ to l bits. If we want to base the security on DES, with $l = 64$, we could use the general construction of Aiello and Venkatesan [1] to build a pseudo-random function from $2l$ to l bits. However, that would require 6 DES applications. For the particular situation at hand, it turns out that two DES applications are sufficient. We outline this construction.

The secret key consists of two functions f and g chosen at random from a pseudo-random family F of functions on l-bit strings, a random $\alpha \in \mathrm{GF}(2^l)$, and a hash function h chosen at random from an ϵ-AXU family of hash functions with l-bit outputs. To compute a MAC for a message x, the signing algorithm generates $r, s \in \mathrm{GF}(2^l)$ at random; the MAC is $(r, s, f(r) \oplus g(s) \oplus \alpha rs \oplus h(x))$.

Theorem 4 *Suppose that F is replaced by the family R of truly random functions on l-bit strings, and that the adversary makes q_1 queries to the signing algorithm, and q_2 queries to the verifying algorithm, where $q_1 < 2^{l-1}$. Then the probability that the adversary forges a MAC is at most $1.5q_1^2/2^{2l} + q_2\epsilon$.*

As usual, if in passing from R to F we get a significant increase the forgery probability, we get a statistical test to distinguish F from R.

It still remains to prove an analogous theorem for permutations; nevertheless, DES, or some simple construction based on it, still seems like a good candidate for F.

4 Three Types of Hash Functions

In the remainder of this paper, we deal with the choice and implementation of an ϵ-AXU family of hash functions.

In this section, we present the three types of hash functions under consideration. We assume that messages are broken up into n blocks, each containing l bits. The output of the hash functions is l bits.

The Evaluation Hash. The *evaluation hash* views the input as a polynomial $M(t)$ of degree less than n over $\mathrm{GF}(2^l)$. The hash key is a random element α in $\mathrm{GF}(2^l)$. The hash value is $M(\alpha) \cdot \alpha \in \mathrm{GF}(2^l)$. This family of hash functions is ϵ-AXU with $\epsilon \approx n/2^l$.

The Division Hash. The *division hash* views the input as a polynomial $m(x)$ of degree less than nl over $\mathrm{GF}(2)$. The hash key is a random irreducible polynomial $p(x)$ of degree l over $\mathrm{GF}(2)$. The hash value is $m(x) \cdot x^l \bmod p(x)$. Since the total number of irreducible polynomials of degree l is $\approx 2^l/l$, it is easy to see that this family of hash functions is ϵ-AXU with $\epsilon \approx nl/2^l$.

The Generalized Division Hash. The third hash function actually includes each of the first two as special cases. Suppose that $k \mid l$. The *generalized division hash* views the input as a polynomial $m(x)$ over $\mathrm{GF}(2^k)$ of degree less than nl/k. The key is a random monic irreducible polynomial $p(x)$ of degree l/k over $\mathrm{GF}(2^k)$. The hash value is $m(x)x^{l/k} \bmod p(x)$. It is easy to show that this is ϵ-AXU with $\epsilon \approx nl/k2^l$.

The division hash was first suggested for use in message authentication by Krawczyk [12]. The other two are obvious variants, but have somewhat different performance and security properties.

An output length of $l = 64$ should provide an adequate level of security for the above three hash functions. Note that from the point of view of message authentication, MD5's output length of 128 is really "overkill"— this output length was chosen to make finding collisions hard, another problem entirely.

We have implemented the evaluation and division hashes with $l = 64$. One disadvantage of the division hash is that we have to generate a random irreducible polynomial of degree 64 over GF(2) whenever we generate a hash function. This can be somewhat time consuming. Moreover, with the division hash, one effectively has 6 bits less security than with the evaluation hash (i.e., ϵ increases by a factor of 2^6). However, the division hash runs somewhat faster than the evaluation hash. We have also implemented the generalized division hash with $l = 64$ and $k = 8$. We have found that with this method, hash function generation is much faster than with the division hash, while hashing speed is identical to that of the division hash. Also, one has only 3 bits less security than with the evaluation hash.

We briefly summarize some of our empirical results; more details can be found later in the paper. The timings are based on a `C` implementation using `gcc` on a Sun Sparc-10 workstation with a 70MHz clock. The Sparc-10 has a very typical 32-bit RISC architecture.

One implementation of the generalized division hash uses one 8KB table for each hash function. The set-up time (the time to generate the hash function and pre-compute the associated table) is about 255μs. The hash function itself achieves a bit rate of 50-75Mbps (10^6 bits per second).

Cache Behavior. Because of the relatively large table size, cache behavior can heavily influence the speed of the hash function. We performed a number of experiments to try to measure this influence, and where the speed seemed to rely heavily on cache behavior, we report this speed as an interval. The highest speed in this interval represents an ideal situation, where a huge amount of data is hashed before pushing the table out of cache. The lower speed represents a situation where only 2KB of data are hashed before pushing the table out of cache. We still need to gain more practical experience with cache behavior.

Using a table of just 2KB, the evaluation hash can be implemented so that it has a set-up time of just 30μs, and runs at 34-36Mbps. Note the much smaller variance in running time due to cache effects.

We have not included in the above the cost of the pseudo-random function. Using one of the faster DES implementations, built by How [10], the set-up time is about 75μs, and the time for one DES operation is about 10.5μs.

We compare the above with a standard `C` implementation of MD5 on our machine, for which `gcc` produces quite good code. MD5 achieves a top speed of 41Mbps. This measures the speed of the internal compression function; dealing with word-alignment and byte-ordering problems can reduce MD5's speed somewhat. Cache effects do not seem to affect the speed of MD5 significantly.

It is clear from the above running times that CBC-DES is very slow, running at only 6Mbps.

As another example, we compiled our code for the generalized division hash on a 90 MHz Pentium, running `linux` and using `gcc`. Because of the very small register set on the Pentium, the `gcc` compiler was not able to generate very good code, and so we hand optimized the assembly code. The set-up time was was 220μs, and the hash function runs at 85-100Mbps.

We compare this to the hand-optimized assembly implementation of MD5 by Bosselaers, Govaerts, and Vandewalle [5]. This runs at 113Mbps.

Also, How's implementation of DES on our Pentium has a set-up time of 94μs, and one application takes 11.5μs. This implies a rate of about 6Mbps for CBC-DES.

5 The Evaluation Hash

To implement the evaluation hash for $\mathrm{GF}(2^{64})$, we select an irreducible polynomial $f(x) \in \mathrm{GF}(2)[x]$ of degree 64, and represent $\mathrm{GF}(2^{64})$ as $\mathrm{GF}(2)[x]/(f(x))$. It is convenient, especially on 32-bit machines, to select $f(x)$ of the form $x^{64} + f_0(x)$, where $\deg f_0(x)$ is small, for example $f(x) = x^{64} + x^4 + x^3 + x + 1$.

To evaluate a polynomial in $\mathrm{GF}(2^{64})[t]$ at a point $\alpha \in \mathrm{GF}(2^{64})$, we use Horner's rule. Thus, the critical operation is the map $\beta \mapsto \alpha \cdot \beta$ ($\beta \in \mathrm{GF}(2^{64})$). Since α remains fixed for many such multiplications, we can speed things up considerably by performing a pre-computation.

Suppose $\alpha = a(x) \bmod f(x)$, where $a(x) \in \mathrm{GF}(2)[x]$, with $\deg a(x) < 64$. For a given $b(x) \in \mathrm{GF}(2)[x]$, with $\deg b(x) < 64$, we want to compute $a(x) \cdot b(x) \bmod f(x)$. We discuss two methods to do this. In both of these methods, we assume that we have a table that allows us to compute the map $v(x) \mapsto v(x) \cdot x^{64} \bmod f(x)$ ($\deg(v(x) < 8$) by table-lookup. This table will have 256 entries, and because of the special form of $f(x)$, each entry will be only 16-bits wide, for a total of 0.5KB.

Method 1. Without any pre-computation, we can compute $a(x) \cdot b(x) \bmod f(x)$ for given $a(x)$ and $b(x)$ as follows. First, we compute $x^i a(x) \bmod f(x)$ for $0 \le i < 8$. Second, we write $b(x) = \sum_{i=0}^{7} b_i(x) x^{8i}$, initialize

$r(x)$ to zero, and do the following:

$$\text{for } i \leftarrow 7 \text{ down to } 0 \text{ do } \; r(x) \leftarrow r(x)x^8 + b_i(x)a(x) \bmod f(x).$$

The pre-computed tables facilitate the computation.

Timing results. In our Sparc-10 implementation, each multiplication mod $f(x)$ takes about 6μs, which yields a hash rate of about 11Mbps. The values $x^i a(x) \bmod f(x)$ $(0 \le i < 8)$ were allocated to registers by our compiler, and the number of instructions executed per byte is about 32.

Method 2. This method is the same as Method 1, except that given $a(x)$ we perform a pre-computation that allows us to compute the map $v(x) \mapsto v(x) \cdot a(x) \bmod f(x)$ $(\deg v(x) < 8)$ by table-lookup. This table will have 256 entries, but each entry will be 64-bits wide, for a total of 2KB.

Timing results. In our Sparc-10 implementation, the pre-computation step for a given $a(x)$ takes 30μs. The hash function then runs at about about 34-36Mbps. The number of machine instructions executed per byte is about 14.

For both of these methods, to achieve these hash rates one must process the message word-by-word, and not byte-by-byte; that is, each word of the message is read from memory as a whole, and then exclusive-ored into a register. Any byte-ordering problems can be dealt with at virtually no cost.

6 The Division Hash

We now consider the division hash. There are two problems that need to be dealt with: how to apply the hash function given the polynomial $p(x) \in \mathrm{GF}(2)[x]$ of degree 64 that defines it, and how to generate a random irreducible polynomial over GF(2) of degree 64. We deal with these problems in turn.

6.1 Hash Function Application

Assume we have the polynomial $p(x)$ defining the hash function. If the input to the function is $m(x) = \sum_{i=0}^{n-1} m_i(x)x^{64i}$, we initialize $r(x)$ to zero, and do the following:

$$\text{for } i \leftarrow n-1 \text{ down to } -1 \text{ do } \; r(x) \leftarrow r(x)x^{64} + m_i(x) \bmod p(x),$$

where $m_{-1}(x)$ is defined to be zero.

The critical operation is the 64-bit reduction map $v(x) \mapsto v(x)x^{64} \bmod p(x)$ $(\deg v(x) < 64)$. We describe two methods to implement this map.

Method 1. In this method, we perform a pre-computation that allows us to compute $v(x) \mapsto v(x)x^{64} \bmod p(x)$ $(\deg v(x) < 8)$ by table look-up. This will require a table of 256 64-bit entries, for a total 2KB. Given this table for 8-bit reduction, we can easily compute the 64-bit reduction using 8 table lookups, shifts, and exclusive-ors.

Timing results. In our Sparc-10 implementation of this method, the pre-computation step takes about 30μs, and achieves a rate of 35-38Mbps. The number of machine instructions executed per byte is about 10.

Method 2. The double-word shifts required in the above method are quite costly on 32-bit machines. On such machines, the following avoids these shifts, and yields better pipeline utilization as well. In this method, we perform a pre-computation that allows us to compute, for $0 \le i < 4$, the maps $v(x) \mapsto v(x)x^{64+8i} \bmod p(x)$ $(\deg v(x) < 8)$. This requires 4 tables, each with 256 64-bit entries, for a total of 8KB. With these tables, we can perform a 32-bit reduction with just 4 table look-ups and exclusive-ors. We repeat this twice to get a 64-bit reduction.

Timing results. For this method, the pre-computation step takes 120μs, and achieves a rate of 50-75Mbps. The number of machine instructions executed per byte is about 6.

As in the evaluation hash, for reasons of efficiency, the message should be processed word-by-word, instead of byte-by-byte.

6.2 Generating an Irreducible Polynomial

We now consider the problem of generating a random irreducible polynomial of degree 64 over GF(2). One way is to generate polynomials at random and test for irreducibility. This is quite time consuming, and requires a lot of random bits.

A much better way to proceed is the following. We can assume that we already have one irreducible polynomial of degree 64, defining the extension field $\mathrm{GF}(2^{64})$. Given this, we generate a random element in $\mathrm{GF}(2^{64})$ and then compute the minimal polynomial of this element. This procedure is also nice since we only need 64 random bits.

With this procedure, the probability that we get a polynomial whose degree is less than 64 is $1/2^{32}$ (the probability of choosing an element in $\mathrm{GF}(2^{32})$). While this is small, it cannot be ignored. If this happens, one could repeat the above procedure. However, it is actually better from both an efficiency and security standpoint to do the following: if we get an irreducible $q(x)$ of degree less than 64, then simply define the hash function by the polynomial $p(x) = q(x)x^{64-\deg q(x)}$. Although perhaps counter-intuitive, it is not difficult to show that the security of this hash function is just as good as that of the original (we leave this to the reader to verify).

So we have reduced our problem to the following, which we state in more general terms. Let K be a field and $f(x) \in K[x]$ a monic, irreducible polynomial of degree d. We are given a polynomial $g(x) \in K[x]$ of degree less than d, and we want to compute its minimal polynomial modulo $f(x)$, i.e., the monic polynomial $h(x) \in K[x]$ of least degree such that $h(g(x)) \equiv 0 \bmod f(x)$.

We describe three ways to solve this problem.

Method 1. One of the most obvious and well-known methods is to compute powers of $g(x)$ modulo $f(x)$, and then find a linear relation using elimination techniques. This will in general take $O(d^3)$ arithmetic operations in K.

Consider the situation where $K = \mathrm{GF}(2)$ and $d = 64$. To compute the sequence of powers of $g(x)$ modulo $f(x)$, we first build a table to make multiplication by $g(x)$ modulo $f(x)$ fast. For this, we use the technique of method 2 in §5. Now we have a matrix $M \in \mathrm{GF}(2)^{65\times 64}$, and we want to find a vector $v \in \mathrm{GF}(2)^{1\times 65}$ satisfying $vM = 0$. One way to do this is standard Gaussian elimination; however, when we build the matrix, the *rows* are represented as word-pairs, but to perform Gaussian elimination, we need to perform *column* operations. Converting this matrix to a form that makes Gaussian elimination efficient is quite time consuming. A much better approach is that of Parkinson and Wunderlich [15] (see also Lenstra and Manasse [13]) which finds a solution using *row* operations.

Timing results. In our Sparc-10 implementation, this method requires about 570μs: 30μs to build the multiplication look-up table; 125μs to compute the powers of $g(x)$; and 415μs to perform the Parkinson-Wunderlich algorithm.

Method 2. This method, due to Gordon [9], applies only to a *finite* field $K = \mathrm{GF}(q)$. We compute the sequence of polynomials $g(x)^{q^i} \bmod f(x)$ for $0 \le i \le m$, where m is the smallest positive integer such that $g(x)^{q^m} \equiv g(x) \bmod f(x)$. Note that $m \mid d$. We then compute $h(x) = \prod_{i=0}^{m-1}(x - g(x)^{q^i}) \bmod f(x)$. When $m = d$, we replace $h(x)$ with $h(x) + f(x)$. This method uses $O(d^3 \log q)$ arithmetic operations in K.

Now consider the situation where $K = \mathrm{GF}(2)$ and $d = 64$. We have to do 63 squarings and multiplies modulo $f(x)$. There are a variety of ways to make the squarings fast with a pre-computed table. However, since the operands in the multiplies are different every time, we cannot perform a pre-computation to speed this up, making these multiplications quite slow. To perform these multiplications, we use the technique of method 1 in §5.

Timing results. In our Sparc-10 implementation, this method takes about 410μs: 35μs for the squarings, and 375μs to do the multiplications.

Method 3. Consider the sequence of polynomials $g_0(x), g_1(x), \ldots$, where $g_i(x) = g(x)^i \bmod f(x)$. This is a linearly generated sequence over K with minimal polynomial $h(x)$, i.e., it satisfies a linear recurrence whose coefficients are those of $h(x)$. Borrowing a simple idea from Wiedemann [19], we consider the projected sequence $a_0 = g_0(0), a_1 = g_1(0), \ldots$, i.e., we simply take the constant terms of the polynomial sequence to get a sequence over K. This latter sequence is also linearly generated over K; in general its minimal polynomial will divide $h(x)$, but since $h(x)$ is irreducible, and since the projected sequence is nonzero, the minimal polynomial of the projected sequence is also $h(x)$.

So now we have the following problem. We have a sequence of elements $a_0, a_1, \ldots$ in K that is linearly generated over K with minimal polynomial of degree at most d. The first $2d$ elements of this sequence fully determine its minimal polynomial, and this can be very efficiently computed using the Berlekamp-Massey algorithm (see Massey [14] and also Kaltofen and Saunders [11]), which uses $O(d^2)$ arithmetic operations in K.

Consider now the situation where $K = \mathrm{GF}(2)$ and $d = 64$. We compute the powers of $g(x)$ as in method 1, and pack the constant-term bits into 4 machine words. By keeping elements of GF(2) packed into words, with some care the Berlekamp-Massey algorithm can be implemented so as to be quite efficient.

Timing results. In our Sparc-10 implementation, the total time to compute a minimal polynomial with this method is about 360μs: 30μs to build the multiplication look-up table; 250μs to compute the sequence of powers; and 80μs to perform the Berlekamp-Massey algorithm.

7 The Generalized Division Hash

The generalized division hash achieves a bit-rate identical to that of the division hash, but has the advantage that the required irreducible polynomial can be generated much faster.

The generalized division hash works over the field $K = \mathrm{GF}(2^8)$. To generate the hash function and required tables, we have to perform arithmetic in K. To do this, we use the standard technique of using exponentiation and logarithm tables so that a multiplication in K takes one addition and three table look-ups. To avoid special cases involving multiplication by 0, we set the logarithm of 0 to -255, and the exponentiation table is then indexed from -510 to 508. The total size of these tables is 2KB.

7.1 Hash Function Application

Suppose we have a polynomial $p(x) \in K[x]$ defining the hash function. We can carry out division with remainder in much the same way as in §6. In fact, once we pre-compute the necessary tables, the algorithms for division with remainder are identical to those in §6. One difference is that constructing the tables takes just a little more time: 35μs (instead of 30μs) in the 1-table method, and 140μs (instead of 120μs) in the 4-table method.

7.2 Generating an Irreducible Polynomial

We generate a random irreducible polynomial over K as follows. We fix an irreducible polynomial $f(x) \in K[x]$ of degree 8. For efficiency purposes, $f(x)$ is chosen to be of the form $x^8 + f_0(x)$, where $\deg f_0(x) < 4$. We choose a random polynomial $g(x) \in K[x]$ of degree less than 8, and compute its minimal polynomial. This is done using the Berlekamp-Massey algorithm, as in the last section. This requires that we compute $g(x)^i \bmod f(x)$ for $0 \leq i < 16$. These multiplications are done by a method analogous to method 2 in §5. Again, the special form of $f(x)$ makes these multiplications more efficient. Also as in §6, if we get an irreducible polynomial of degree less than 8, we use it anyway.

Timing results. In our Sparc-10 implementation, the total time required to generate a random irreducible polynomial is 115μs: 35μs to build the multiplication look-up table; 30μs to compute the sequence of powers; and 55μs to perform the Berlekamp-Massey algorithm.

8 Conclusion

Our experience indicates that a message authentication scheme based on either the generalized division hash or the evaluation hash, along with DES, is an attractive alternative to schemes based on MD5, or similar cryptographic hash functions: one can obtain a much higher degree of provable security, while attaining reasonable performance.

We summarize our empirical results here. Details of how these estimates were obtained are contained in the body of the paper. The scheme based on the generalized division hash requires 120 random bits to generate an instance of the scheme. It uses one 8KB table per instance. Given the 120 bits defining the instance, there is a set-up cost. On a 70MHz Sparc-10, the total set-up time is 330μs, and on a 90MHz Pentium, 315μs. As to speed, it runs at 50-75Mbps on a Sparc-10, and 85-100Mbps on a Pentium. There is also the cost of one DES application per message: about 11μs on both machines.

In contrast, consider a scheme based on the evaluation hash. It also requires 120 random bits to generate an instance of the scheme. One variant of this scheme uses no tables, has no set-up time, but runs at 11Mbps on a Sparc-10. Another variant uses a table of 2KB per instance, and on a Sparc-10 has a set-up time is just 105μs, and runs at 34-36μs. We have not implemented this on the Pentium. Because of the smaller set-up time, and because the smaller table places less pressure on the cache, this scheme could be preferable to the generalized hash scheme in some situations.

We compare the above to MD5 and CBC-DES.

MD5 has no significant set-up time or storage requirements. It runs at 41Mbps on a Sparc-10, and at 113Mbps on a Pentium.

CBC-DES has a set-up time of 75μs on the Sparc-10, and 94μs on the Pentium. The storage requirements are not significant. It runs at about 6Mbps on both machines.

We note that our hash techniques complement the bucket-hash technique developed by Rogaway [17] very nicely. For high-speed authentication of *very* large files, one would reduce the input size by a factor of, say, 10 using a bucket hash, and then apply, say, a generalized division hash to this shorter string.

Acknowledgement

The author gratefully acknowledges Mihir Bellare for several discussions that led to the discovery and repair of an error in Theorem 4 in an earlier version of this paper, and also for pointing out the unpublished work of Bellare, Goldreich, and Krawczyk analyzing the related construction "$f_1(r_1) \oplus f_2(r_2) \oplus f_3(r_3) \oplus h(x)$."

Appendix A: Proof of Theorems 2 and 3

To prove Theorem 2, without loss of generality, we assume that the adversary is deterministic, and that all S-queries are made before all V-queries. We are assuming that f is a random permutation. For $1 \le i \le q_1$, the adversary obtains strings $w_i = f(i) \oplus h(x_i)$, where each message x_i is some function of $w_1, \ldots, w_{i-1}$. Let $\vec{w} = (w_1, \ldots, w_{q_1})$.

Lemma 1 *Let $h \in H$ be an arbitrary hash function, and let $\vec{w}$ be an arbitrary sequence of strings that can appear as outputs from the S-oracle with nonzero probability. Then we have* $\Pr[h|\vec{w}] \le 2\Pr[h]$.

Theorem 2 follows trivially from this lemma, using the standard argument for the security of the basic universal-hash construction with the fact that H is ϵ-AXU.

To prove Lemma 1, we use Bayes' theorem:

$$\Pr[h|\vec{w}] = \frac{\Pr[h]\Pr[\vec{w}|h]}{\sum_{g \in H} \Pr[g]\Pr[\vec{w}|g]}.$$

We want to bound the quantity
$$T = \sum_{g \in H} \Pr[g] \Pr[\vec{w}|g].$$
from below.

Fix $g \in H$, and let $v_i = w_i \oplus g(x_i)$ for $1 \leq i \leq q_1$. Then we have
$$\Pr[\vec{w}|g] = \begin{cases} \prod_{i=0}^{q_1-1}(2^l - i)^{-1} & \text{if } v_i \neq v_j \text{ for all } 1 \leq i < j \leq q_1, \\ 0 & \text{otherwise.} \end{cases}$$

It follows that T is just $\prod_{i=0}^{q_1-1}(2^l - i)^{-1}$ times the probability that for a random $g \in H$, the sequence $\vec{v} = (v_1, \ldots, v_{q_1})$ contains no duplicates. Now, fix i and j with $1 \leq i < j \leq q_1$. If, on the one hand, $x_i = x_j$, then by the assumption that $\vec{w}$ appears with nonzero probability, and the fact that f is a permutation, it follows that $w_i \neq w_j$, and so $v_i \neq v_j$. On the other hand, if $x_i \neq x_j$, then by the fact that H is ϵ-AXU, it follows that $v_i = v_j$ with probability at most ϵ. Thus, the sequence $\vec{v}$ contains duplicates with probability at most $q_1^2\epsilon/2$, and so
$$T \geq \prod_{i=0}^{q_1-1}(2^l - i)^{-1}(1 - q_1^2\epsilon/2).$$

From this it follows that $\Pr[h|\vec{w}] \leq \Pr[h]/(1 - q_1^2\epsilon/2)$. The lemma then follows from the assumption that $q_1^2 \leq 1/\epsilon$.

That proves Theorem 2. For Theorem 3, the key lemma is the following.

Lemma 2 *Let $h_1, h_2 \in H_1 \times H_2$ be an arbitrary pair of hash functions, and let $\vec{w}$ be an arbitrary sequence of strings that can appear as outputs from the S-oracle with nonzero probability. Then we have* $\Pr[h_1, h_2|\vec{w}] \leq 2\Pr[h_1, h_2]$.

The proof of this lemma is quite similar to the proof of Lemma 1, and we leave the details to the reader.

Appendix B: Proof of Theorem 4

We sketch the proof of this theorem. As usual, we assume that all signing queries are made before all verifying queries. Let $r_1, s_1, \ldots, r_{q_1}, s_{q_1}$ be the random values created by the signer.

For $i \geq 1$, define a *$2i$-cycle* as a sequence of distinct indices $(j(1), \ldots, j(2i)) \in \{1, \ldots, q_1\}^{2i}$ such that
$$r_{j(1)} = r_{j(2)}, s_{j(2)} = s_{j(3)}, \ldots, r_{j(2i-1)} = r_{j(2i)}, s_{j(2i)} = s_{j(1)}.$$

For $i \geq 2$, define a *partial $2i$-cycle* as a sequence of distinct indices $(j(1), \ldots, j(2i-1)) \in \{1, \ldots, q_1\}^{2i-1}$ such that
$$r_{j(1)} = r_{j(2)}, s_{j(2)} = s_{j(3)}, \ldots, r_{j(2i-3)} = r_{j(2i-2)}, s_{j(2i-2)} = s_{j(2i-1)}.$$
We will call this partial $2i$-cycle *bad* if additionally,
$$r_{j(1)}s_{j(1)} \oplus \cdots \oplus r_{j(2i-1)}s_{j(2i-1)} = r_{j(2i-1)}s_{j(1)}.$$

Claim 1. Conditioning on the event that for all $i \geq 1$ there are no $2i$-cycles and for all $i \geq 2$ there are no bad partial $2i$-cycles, the probability of forging a MAC is at most $q_2\epsilon$.

To see this, note that on any cycle-free set $\{(r_i, s_i)\}$ of inputs, the set $\{f(r_i) \oplus g(s_i)\}$ are are independent random strings (see Aiello and Venkatesan, Eurocrypt '96). So if there are no cycles, the signer has not leaked any information about α or h. Suppose the adversary attempts a MAC forgery for a message x of the form (r, s, γ). There are three cases. First, suppose this (r, s) pair completes no cycle. Then by the the independence of the value $f(r) \oplus g(s)$, the probability of success is 2^{-l}. Second, if this (r, s) pair was previously output by the signer, creating a 2-cycle, then by the AXU property of h, the probability of success is at most ϵ. Third, suppose that for some $i \geq 2$, this (r, s) pair completes a partial $2i$-cycle to form a $2i$-cycle. Since this partial cycle was not bad, and the value of α was not leaked, the probability of success is at most 2^{-l}.

Claim 2. For any $i \geq 1$, the probability that there is a $2i$-cycle is at most $(q_1/2^l)^{2i}/(2i)$.

This is a simple counting argument (see Venkatesan and Aiello, Eurocrypt '96).

Claim 3. For any $i \geq 2$, the probability that there is a a bad partial $2i$-cycle is at most $2(q_1/2^l)^{2i-1}$.

To show this, one first considers a fixed sequence of $2i-1$ indices. This will form a partial $2i$-cycle with probability $2^{-l(2i-2)}$. Then conditioning on the event that this is a partial $2i$-cycle, the probability that it is bad is simply the probability that a certain nonzero polynomial in i variables of total degree 2 vanishes at a random point. This happens with probability $2 \cdot 2^{-l}$. The claim then follows, as there are at most q_1^{2i-1} such sequences to consider.

The theorem now follows by a simple calculation.

References

[1] W. Aiello and R. Venkatesan. Foiling birthday attacks in output-doubling transformations. In *Advances in Cryptology—Eurocrypt '96*, 1996. To appear.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology—Crypto '96*, 1996.

[3] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology—Crypto '95*, pages 15–28, 1995.

[4] M. Bellare, J. Kilian, and P. Rogaway. On the security of cipher block chaining. In *Advances in Cryptology—Crypto '94*, pages 341–358, 1994.

[5] A. Bosselaers, R. Govaerts, and J. Vandewalle. Fast hashing on the Pentium. In *Advances in Cryptology—Crypto '96*, 1996.

[6] G. Brassard. On computationally secure authentication tags requiring short secret shared keys. In *Advances in Cryptology–Crypto '82*, pages 79–86, 1982.

[7] E. Gilbert, F. M. Williams, and N. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974.

[8] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17:281–308, 1988.

[9] J. Gordon. Very simple method to find the minimal polynomial of an arbitrary non-zero element of a finite field. *Electronic Letters*, 12:663–664, 1976.

[10] D. How. Fast and portable DES encryption and decryption, 1992. Available from `how@isl.stanford.edu`.

[11] E. Kaltofen and B. Saunders. On Wiedeman's method of solving sparse linear systems. In *Symp. Applied Algebra, Algebraic Algorithms, Error-Correcting Codes (Lecture Notes in Computer Science no. 539)*, pages 29–38, 1991.

[12] H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology—Crypto '94*, pages 129–139, 1994.

[13] A. K. Lenstra and M. S. Manasse. Compact incremental Gaussian elimination over $\mathbf{Z}/2\mathbf{Z}$. Technical Report 88-16, University of Chicago—Dept. of Computer Science, 1988.

[14] J. Massey. Shift-register synthesis and BCH coding. *IEEE Trans. Inf. Theory*, IT-15:122–127, 1969.

[15] D. Parkinson and M. Wunderlich. A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers. *Parallel Computing*, pages 65–73, 1984.

[16] B. Preneel and P. van Oorschot. MDx-MAC and building fast MACs from hash functions. In *Advances in Cryptology–Crypto '95*, pages 1–14, 1995.

[17] P. Rogaway. Bucket hashing and its application to fast message authentication. In *Advances in Cryptology–Crypto '95*, pages 29–42, 1995.

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

[19] D. Wiedemann. Solving sparse linear systems over finite fields. *IEEE Trans. Inf. Theory*, IT-32:54–62, 1986.