Software Performance of Universal Hash Functions

Wim Nevelsteen and Bart Preneel*

Katholieke Universiteit Leuven, Dept. Electrical Engineering-ESAT
Kardinaal Mercierlaan 94, B–3001 Heverlee, Belgium
wnevelst@eps.agfa.be, bart.preneel@esat.kuleuven.ac.be

Abstract. This paper compares the parameters sizes and software performance of several recent constructions for universal hash functions: bucket hashing, polynomial hashing, Toeplitz hashing, division hashing, evaluation hashing, and MMH hashing. An objective comparison between these widely varying approaches is achieved by defining constructions that offer a comparable security level. It is also demonstrated how the security of these constructions compares favorably to existing MAC algorithms, the security of which is less understood.

1 Introduction

In many commercial applications, protecting the integrity of information is even more important than protecting its secrecy. Digital signatures, introduced in 1976 by Diffie and Hellman [13], are the main tool for protecting the integrity of information. They are essential to build a worldwide trust infrastructure. However, there are still a significant number of applications for which digital signature are *not* cost-effective:

- For applications with *short* messages, the limitation is that signing and verifying is too demanding for processors in low-cost smart cards. On a more modern processor[1], the combined time of signing and verifying a digital signature using RSA, DSA or ECDSA typically exceeds 30 milliseconds.
- For applications with *long* messages (several Megabytes), the speed of signing is limited by the speed of present-day hash functions, which is about 100 Mbit/s.
- Finally, the overhead of a digital signature varies between 25 to 128 bytes, and the keys and system parameters require between 80 and a few hundred bytes of storage.

For the reasons indicated above, many applications use conventional MAC (Message Authentication Code) algorithms to provide data integrity and data origin authentication. MACs do not provide non-repudiation of origin, unlike

* F.W.O. postdoctoral researcher, sponsored by the Fund for Scientific Research – Flanders (Belgium).

[1] Throughout this paper performance numbers will be given for a 200 MHz Pentium.

J. Stern (Ed.): EUROCRYPT'99, LNCS 1592, pp. 24–41, 1999.

digital signatures, that can be used in a setting where the parties do not trust each other. Moreover, MACs rely on shared symmetric keys, which requires additional key management functions. Banks have been using MACs since the late seventies [36,37] for message authentication. Recent applications in which MACs have been introduced include electronic purses (such as Proton and Mondex) and credit/debit applications (e.g., the EMV specifications). MACs are also being deployed for securing the Internet (e.g., IP security). For all these applications MACs are preferred over digital signatures because they are two to three orders of magnitude faster, and MAC results are shorter (typically between 4 ... 16 bytes). On present day machines, software implementations of MACs can achieve speeds from 50 ... 250 Mbit/s, and MACs require very little resources on inexpensive 8-bit smart cards and on the currently deployed Point of Sale (POS) terminals. During the last five years, our understanding of MACs has improved considerably, through development of security proofs (Bellare et al. [3,5,6]) and new attacks (Knudsen [23] and Preneel and van Oorschot [30,31]).

An important disadvantage of both digital signatures and MAC algorithms is that their security is only computational. That implies that an opponent with sufficient computing power can in principle forge a message. A second problem is that shortcut attacks might exist, which means that forging a message can be much easier than expected. This problem can partially be solved by developing security proofs; such a proof can reduce the security of a MAC or a digital signature scheme to another primitive, such as a pseudo-random function or to a problem that is believed to be difficult, such as factoring the product of two large primes. However it seems wise to anticipate further progress in cryptanalysis of specific primitives. In the nineties we have witnessed the development of differential attacks [8], linear attacks [26], and of the use of optimization techniques as in [14]. The ultimate solution to this problem is unconditional security.

The idea of unconditionally secure authentication (and the so-called *authentication codes*) dates back to the early seventies, when Simmons was developing for Sandia National Laboratories a system for the verification of treaty compliance, such as the comprehensive nuclear test-ban treaty between the USA and the USSR [37]. The motivation for his research was that apparently the NSA refused to export strong conventional cryptographic mechanisms to the USSR. The first construction of authentication codes appeared in a 1974 paper by Gilbert et al. [18]. Subsequently their theory has been developed further by Simmons, analogous to Shannon's theory of secrecy systems [34]. An overview of the theory of authentication codes can be found in the work of Simmons [36] and Stinson [38]. In the seventies and the eighties, the research on authentication codes in the cryptographic community focussed mainly on the properties of authentication codes that meet certain bounds (such as perfect authentication codes, cf. §2.1). While this work illustrates that combinatorial mathematics and information theory provides powerful tools to develop an understanding of cryptographic primitives, it was widely believed that this work was of purely academic interest only.

This is the more surprising because Carter and Wegman developed already in the late seventies efficient authentication codes under the name of *strongly universal hash functions* [12,40]. They show that this is an interesting combinatorial tool that can be applied to other problems as well (such as interactive proof systems, pseudo-random number generation, and probabilistic algorithms). Carter and Wegman make the following key observations: i) long messages can be authenticated efficiently using short keys if the number of bits in the authentication tag is increased slightly compared to 'perfect' schemes; ii) if a message is hashed to a short authentication tag, weaker properties are sufficient for the first stage of the compression; iii) under certain conditions, the hash function can remain the same for many plaintexts, provided that the hash result is encrypted using a one-time pad. Mehlhorn and Vishkin propose more efficient constructions in [28]. At Crypto'82, Brassard pointed out that combining this primitive with a pseudo-random string generator will result in efficient computationally secure message authentication with short keys [11].

In the beginning of the nineties, the two 'independent' research threads are brought together. Stinson improves the work by Wegman and Carter, and establishes an explicit link between authentication codes and strongly universal hash functions [39]. A second important development is that Johansson, Kabatianskii, and Smeets establish a relation between authentication codes and codes correcting independent errors [22]. This provides a better understanding of the existing constructions and their limitations.

During the last five years, progress has been made both in theory and practice of universal hash functions. Krawczyk has proposed universal hash functions that are linear with respect to bitwise xor [24,25]. This property makes it easier to reuse the authentication code (with the same key): one encrypts the m-bit hash result for each new message using a one-time pad. This approach leads to simple and efficient constructions based on polynomials and Linear Feedback Shift Registers (LFSRs). Other constructions based on polynomials over finite fields are proposed and analyzed by Shoup [35]. Shoup [35] and Afanassiev et al. [1] study efficient software implementations of this primitive. Another line of research has been to improve the speed at the cost of an increased key size and size of the authentication tag. Rogaway has introduced bucket hashing in [33]; a slower variant with shorter keys was proposed by Johansson in [21]. Halevi and Krawczyk have developed an extremely fast scheme (MMH) which makes optimal used of the multiply and accumulate instruction of the Pentium MMX processor [19]. Recently Black et al. have further improved the performance on high end processors with the UMAC construction [9].

While it is clear that authentication codes (or universal hash functions) have a large potential for certain applications, they are not widely known to application developers. Some of the reasons might be that the research is too new, and that it is difficult to choose among the many schemes. For example, Halevi and Krawzcyk write "An exact comparison is not possible since the data available on the most efficient implementations of other functions are based on different platforms" [19, p. 174]. The latter problem makes it more difficult to introduce

them into standards. For the time being, there is also a lack of public domain implementations, that can demonstrate the benefits of this approach.

This paper intends to solve part of these problems by providing an objective comparison of performance and parameter sizes for the most promising constructions. For three related universal hash functions, similar work has been done by Shoup [35]. Atici and Stinson [2] provide an overview of the general parameters of several schemes, but do not discuss the performance.

The remainder of this paper is organized as follows. §2 introduces the most important definitions, and §3 presents the constructions that will be compared in this paper. The comparison of implementation speeds and memory requirements of the different schemes is presented in §4, and §5 contains some concluding remarks.

2 Definitions and Background

This section presents the model for authentication without secrecy. Next universal hash functions and strongly universal hash functions are introduced, and it is explained how they can be combined.

2.1 Authentication Codes

As usually in cryptography, the main players are the sender Alice, who wants to send some information to the receiver Bob; the opponent of Alice and Bob is the active eavesdropper Eve. Here, Alice and Bob are not concerned about the secrecy of the information. In order to detect the actions of Eve, Alice attaches to the plaintext an authentication tag that is a function of a shared secret key and of the plaintext. Bob recomputes the tag and accepts the plaintext as authentic if the tag is the same. As in the Vernam scheme, the secret key can be used only once.

Eve can perform three types of attacks: (i) Eve can create a new plaintext and send it to Bob, pretending that it came from Alice (*impersonation attack*); (ii) Eve can wait until she observes a plaintext and replace it by a different plaintext (*substitution attack*); (iii) Eve can choose freely between both strategies (*deception attack*). The probability of success (when the strategy of Eve is optimal) will be denoted with P_i, P_s, and P_d respectively. A first result that follows from Kerckhoffs' assumption (namely that the strategy to choose the key is known by Eve) is that $P_d = \max(P_i, P_s)$ [27].

In the following the length (in bits) of the plaintext, authentication tag, and key is denoted with m, n, and k respectively. The combinatorial bounds state that P_i and P_s are at least $1/2^n$. In the following we will consider only schemes for which $P_i = 1/2^n$. Another important bound is the square root bound; it states that $P_d \geq 1/2^{k/2}$. This is a corollary of the 'authentication channel capacity theorem' which states that an authentication code can only be secure if the authentication tag reveals a significant amount of information on the secret key (see Massey [27] for details).

Stinson proves that if $P_i = P_s = 1/2^m$, the number of plaintexts is at most a linear function of the number of keys [39]. This shows that schemes of this type require large keys for large messages, which makes them impractical. On the other hand, Kabatianskii et al. [22] showed that if P_s exceeds P_i by an arbitrarily small amount, the number of plaintexts grows exponentially with the number of keys. This research developed from exploring connections to the rich theory of error-correcting codes, and connects to the work of Wegman and Carter [12,40]. The disadvantage of $P_s > 1/2^n$ is that for a given security level (say, $P_d = 1/2^{64}$), slightly more than 64 bits are required for the authentication tag. While [22] shows efficient constructions that require only a single extra bit, in practice one can afford to send one or more extra bytes.

2.2 Universal Hash Functions

A universal hash function is a mapping from a finite set A with size a to a finite set B with size b. For a given hash function h and for a pair (x, x') with $x \neq x'$ the following function is defined: $\delta_h(x, x') = 1$ if $h(x) = h(x')$, and 0 otherwise. For a finite set of hash functions H (in the following this will be denoted with a *family* of hash functions), $\delta_H(x, x')$ is defined as $\sum_{h \in H} \delta_h(x, x')$, or $\delta_H(x, x')$ counts the number of functions in H for which x and x' collide. When a random choice of h is made, then for any two distinct inputs x and x', the probability that these two inputs yield a collision equals $\delta_H(x, x')/|H|$. For a universal hash function, the goal is to minimize this probability together with the size of H.

Definition 1. *Let ϵ be any positive real number. An* **ϵ-almost universal** *family (or ϵ-AU family) H of hash functions from a set A to a set B is a family of functions from A to B such that for any distinct elements $x, x' \in A$*

$$| \{h \in H : h(x) = h(x')\} | = \delta_H(x, x') \leq \epsilon \cdot |H| \quad .$$

This definition states that for any two distinct inputs the probability for a collision is at most ϵ. In [12] the case $\epsilon = 1/b$ is called universal; the smallest possible value for ϵ is $(a - b)/(b(a - 1))$.

Definition 2. *Let ϵ be any positive real number. An* **ϵ-almost strongly universal** *family (or ϵ-ASU family) H of hash functions from a set A to a set B is a family of functions from A to B such that*

- *for every $x \in A$ and for every $y \in B$, $| \{h \in H : h(x) = y\} | = |H| / b$,*
- *for every $x_1, x_2 \in A$ $(x_1 \neq x_2)$ and for every $y_1, y_2 \in B$ $(y_1 \neq y_2)$,*
$$| \{h \in H : h(x_1) = y_1, h(x_2) = y_2\} | \leq \epsilon \cdot |H| / b \quad .$$

The first condition states that the probability that a given input x is mapped to a given output y equals $1/b$. The second condition implies that if x_1 is mapped to y_1, then the conditional probability that x_2 (different from x_1) is mapped to y_2 is upper bounded by ϵ. The lowest possible value for ϵ equals $1/b$ and this family has been called strongly universal functions in [40]. For this family the first condition in the definition follows from the second one [39].

If an Abelian group can be defined in the set B using the operation $\oplus$ (bitwise exclusive-or), Krawczyk defines the following variant [24] (the terminology is from [33]):

Definition 3. *Let ϵ be any positive real number. An* **ϵ-almost XOR universal** *family (or ϵ-AXU family) H of hash functions from a set A to a set B is a family of functions from A to B such that for any distinct elements $x, x' \in A$ and for any $b \in B$*

$$| \{h \in H : h(x) \oplus h(x') = b\} | \leq \epsilon \cdot |H| \quad .$$

It follows directly from the definition that ϵ-ASU families of hash functions are equivalent to authentication codes with $P_i = 1/b$ and $P_s = \epsilon$ [39,40].

Theorem 4. *There exists an ϵ-ASU family H of hash functions from A to B iff there exists an authentication code with a plaintexts, b authenticators and $k = |H|$ keys, such that $P_i = 1/b$ and $P_s \leq \epsilon$.*

A similar result has been proved by Krawczyk for ϵ-AXU families [24,40].

Theorem 5. *There exists an ϵ-AXU family H of hash functions from A to B iff there exists an authentication code with a plaintexts, b authenticators and $k = |H| \cdot b$ keys, such that $P_i = 1/b$ and $P_s \leq \epsilon$.*

The construction consists of hashing an input using a hash function from H followed by encrypting the result by xoring a random element of B (which corresponds to a one-time pad).

Rogaway [33] and Shoup [35] show how the one-time pad can be replaced by a finite pseudo-random function (respectively permutation). In addition, they develop models for the use of counters and random tags. If the keys are generated using a finite pseudo-random function, the unconditional security is lost, but one has achieved a clear separation between compression (in a combinatorial way) and the final cryptographic step. This makes it easier to analyze and understand the resulting scheme.

2.3 Composition Constructions

The following propositions show how universal hash functions can be combined in different ways in order to increase their domains, reduce ϵ, or decrease the range. Several of these results were applied by Wegman and Carter [40].

Proposition 6 (Cartesian Product [39]). *If there exists an ϵ-AU family H of hash functions from A to B, then, for any integer $i \geq 1$, there exists an ϵ-AU family H^i of hash functions from A^i to B^i with $|H^i| = |H|$.*

Proposition 7 (Concatenation [33]). *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-AU family H_2 of hash functions from A to C, then there exists an ϵ-AU family H of hash functions from A to $B \times C$, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 \epsilon_2$.*

Proposition 8 (Composition 1 [39]). *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-AU family H_2 of hash functions from B to C, then there exists an ϵ-AU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2 \leq \epsilon_1 + \epsilon_2$.*

Proposition 9 (Composition 2 [39]). *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-ASU family H_2 of hash functions from B to C, then there exists an ϵ-ASU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2 \leq \epsilon_1 + \epsilon_2$.*

Proposition 10 (Composition 3 [39]). *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-AXU family H_2 of hash functions from B to C, then there exists an ϵ-AXU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2 \leq \epsilon_1 + \epsilon_2$.*

The most important results are Proposition 9 and Proposition 10, as they allow to use more efficient (in terms of key size and computation) ϵ-AU universal hash functions in the first stage of the compression.

3 Constructions

The schemes that are discussed here are: bucket hashing, bucket hashing with a short key, fast polynomial evaluation, Toeplitz hashing, evaluation hash function, the division hash function, and MMH.

3.1 Bucket Hashing

The first hashing technique we consider is bucket hashing, which is an ϵ-AU introduced by Rogaway [33]. Fix a *word size* $w \geq 1$. For $M \geq N$ the hash functions of the family $B[w, M, N]$ are defined as mappings from $A = \{0,1\}^{wM}$ to $B = \{0,1\}^{wN}$. Each $h \in B[w, M, N]$ is specified by a list of length M, each entry of which contains three integers in the interval $[0, N-1]$. Denote this list by $h = h_1 \dots h_M$, where $h_i = \{h_i^1, h_i^2, h_i^3, \}$. The hash family $B[w, M, N]$ is the set of all possible lists h subjected to the constraints that no two of the 3-element sets in the list are the same, i.e., $h_i \neq h_j$, $\forall i \neq j$.

For a given hash function $h = h_1 \dots h_M$ and a given input $X = x_1 \dots x_M$ the hash result $h(X)$ is computed as follows. First, for each $j \in \{1, \dots, N\}$, initialize y_j to 0^w. Then for each $i \in \{1, \dots, M\}$ and $k \in h_i$, replace y_k by $y_k \oplus x_i$. When this operation is completed, set $h(X) := y_1 \| y_2 \| \dots \| y_N$.

The name bucket hashing is derived from the following interpretation of the computation. We start with N empty buckets y_1 through y_N. Each word of the input is thrown into three buckets; the ith word x_i is thrown in the buckets h_i^1, h_i^2, and h_i^3. Then, the xor of the content in each of the buckets is computed, and the hash function output is the concatenation of the final content of the buckets.

The bucket hash family is ϵ-AU with the collision probability given by a complicated expression in the number N of buckets (see Rogaway, [33, p. 35]). It is important to note that the number N of buckets increases very fast if ϵ decreases. For example, for $\epsilon = 2^{-28}$, $N = 100$ buckets are needed, but for $\epsilon = 2^{-34}$ already 197 buckets are needed.

Table 1 indicates the performance and parameter sizes for an input block of 4 Kbyte and a word length w equal to 32. The Assembly code is hand optimized, and makes optimal use of the two parallel pipes of the Pentium. Several alternatives have been compared, but it was decided not to use self-modifying code, as this poses problems in most applications. For some of these results, we have combined several bucket hash functions using the rules from §2.3 (details are omitted due to space constraints). The memory in the table below corresponds to the processed key and the hash result; the memory to store the input is not included.

Note that for this hash function only the speed was measured under DOS, while for the other schemes (that use a finite field arithmetic library), the speed was measured under Windows '95. Timing measurements under DOS tend to be a little better.

Table 1. Characteristics of bucket-hashing for a block of 4 Kbyte

ϵ	2^{-16}	2^{-32}	2^{-48}	2^{-64}
Parameters	$M = 256$	$M = 1024$	$M = 1024$	$M = 1024$
	$N = 24$	$N = 160$	$N = 62$	$N = 160$
Speed (Mbit/s)	543	341	147	138
Key (bits)	3521	22493	36582	44986
Hash result (bytes)	384	640	496	1280
Memory (bytes)	1152	3712	6640	7424

We conclude that bucket-hashing is a very fast technique, but it requires a long key and a large memory. The hash result becomes very large for small values of ϵ.

3.2 Bucket Hashing with Small Key Size

The bucket-hashing approach from §3.1 gives rise to ϵ-AU hash functions that are very fast to compute, at the cost of a very large key and a long hash result. To overcome these disadvantages Johansson proposed bucket hashing with small key size [21].

Let $N = 2^{s/L}$. Each hash function $h \in B'[w, M, N]$ is specified by a list of length M, where each entry contains L integers in the interval $[0, N-1]$. Next L arrays are introduced, each containing N buckets. Each word from the input is thrown in one bucket of each array, based on the list that describes the hash function h. Next, each array is compressed to s/L words, using a fixed primitive element $\gamma \in \mathrm{GF}(2^{s/L})$. The hash result is equal to the concatenation of the L compressed arrays, each containing s/L words.

Table 2 indicates the performance and parameter sizes for an input block of 4 Kbyte and a word length w equal to 32. Again the Assembly code is hand optimized for the Pentium. It is not possible to use exactly the same values for ϵ as for bucket hashing, because the constraints on the parameters (for example, L has to divide s). For each value of ϵ, one has to determine the optimal value for L. Too large values of L imply that the input has to be thrown in too many buckets; too small values of L imply that N becomes too large.

Table 2. Characteristics of bucket-hashing with small key for a block of 4 Kbyte

ϵ	2^{-18}	2^{-32}	2^{-46}	2^{-62}
Parameters	$s = 28$	$s = 42$	$s = 224$	$s = 72$
	$L = 4$	$L = 6$	$L = 7$	$L = 12$
	$N = 128$	$N = 128$	$N = 256$	$N = 256$
Speed (Mbit/s)	128	93	75	58
Key (bits)	28	42	56	72
Hash result (bytes)	112	168	224	288
Memory (bytes)	11264	16896	25088	43008

We conclude that bucket-hashing with small key size results indeed in very small keys, at the cost of a factor 2 to 4 in performance (depending on the value of ϵ). However, the memory requirements are still large, and the hash results are a little shorter.

3.3 Hash Family Based on Fast Polynomial Evaluation

The next family of hash functions has been proposed by Bierbrauer et al. [7]; it is based on polynomial evaluation over a finite field. Let $q = 2^r$, $Q = 2^m = 2^{r+s}$, $n = 1 + 2^s$, and π be a linear mapping from GF(Q) onto GF(q), where $Q = q_0^m$, $q = q_0^r$, and q_0 a prime power. Let $f_a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$, where $x, y, a_0, a_1, \ldots a_{n-1} \in \mathrm{GF}(Q)$, $z \in \mathrm{GF}(q)$ and

$$H = \{h_{x,y,z} : h_{x,y,z}(a) = h_{x,y,z}(a_0, a_1, \ldots, a_{n-1}) = \pi\left(y \cdot f_a(x)\right) + z\} \quad .$$

It is shown in [7] that the hash family in the construction above is ϵ-ASU with $\epsilon \leq 2/2^r$. For $q_0 = 2$, the function is also ϵ-AXU (for other values, a different group operation has to be used for the difference). The main step in the hash function construction is the evaluation of a polynomial in some point determined by the key. Afanassiev, Gehrmann and Smeets [1] have developed a very fast construction to evaluate a polynomial $f_a(x)$ in an element α (the MinWal procedure). This procedure makes use of Minimal W-nomials. Before evaluating the polynomial $f_a(x)$ in α, $f_a(x)$ is first reduced modulo the minimal W-nomial $\tau_{\alpha,w}(x)$. The minimal W-nomial $\tau_{\alpha,w}(x)$ is a multiple of the minimal polynomial of α with the lowest degree and with less than W non-zero terms.

Table 3 indicates the performance and parameter sizes for an input blocks of 4, 64, and 256 Kbyte. Most of the code has been written in C++ (compiled with

Borland C++ 5.0). The most critical step, the reduction modulo the minimal W-nomial has been written in Assembly language. Calculations are performed in GF(2^{32}). The maximal input length for one instance is equal to 256 Kbyte; of course Proposition 6 (cf. §2.3) can be used for large inputs, and ϵ can be reduced using Proposition 7. We can show that for our software, the optimal value for $W = 5$. Finding a minimal 5-nomial requires about 40 seconds using sub-optimal code. Note that this operation has be done only for the set-up phase. If one adds the (pre-computed) 5-nomials to the key, one needs about 42 bits per additional 5-nomial.

Table 3. Characteristics of hashing based on fast polynomial evaluation

ϵ	2^{-15}	2^{-30}	2^{-45}	2^{-60}
Speed 4 Kbyte (Mbit/s)	9	5	3	2
Speed 64 Kbyte (Mbit/s)	104	56	34	25
Speed 256 Kbyte (Mbit/s)	207	87	50	38
Key (bits)	80	160	240	320
Hash result (bytes)	2	4	6	8
Memory (bytes)	30	60	90	120

For large inputs (256 Kbyte or more), the polynomial evaluation hash function is rather fast and the keys sizes are reasonable. The two main advantages are the very small memory requirements, both for the computation and for the storage of the hash result.

3.4 Hash Family Using Toeplitz Matrices

The next hash family is the Toeplitz construction proposed by Krawczyk [25]. Toeplitz matrices are matrices with constant values on the left-to-right diagonals. A Toeplitz matrix of dimension $n \times m$ can be used to hash messages of length m to hash results of length n by vector-matrix multiplication. The Toeplitz construction uses matrices generated by sequences of length $n + m - 1$ drawn from δ-biased distributions. δ-biased distributions, introduced by Naor and Naor [29], are a tool for replacing truly random sequences by more compact and easier to generate sequences. The lower δ, the more random the sequence is.

Krawczyk proves that the family of hash functions associated with a family of Toeplitz-matrices corresponding to sequences selected from a δ-biased distribution is ϵ-AXU with $\epsilon = 2^{-n} + \delta$ [25]. He proposes to use the LFSR construction due to Alon et al. to construct a δ-biased distribution. This construction associates with r random bits a δ-biased sequence of length l with $\delta = l/2^{r/2}$.

Table 4 indicates the performance and parameter sizes for an input block of 4 Kbyte and a word length w equal to 32. As pointed out by Krawczyk [25], this construction is more suited for hardware, and is not very fast in software. In this case, the compiled C++ code could not be improved manually. For this version of the code, the complete matrix has been stored to improve the performance.

Table 4. Characteristics of Toeplitz hashing for a block of 4 Kbyte

ϵ	2^{-16}	2^{-32}	2^{-48}	2^{-64}
Parameters	$n = 17$	$n = 33$	$n = 44$	$n = 65$
	$r = 68$	$r = 88$	$r = 120$	$r = 142$
Speed (Mbit/s)	65	33	21	16
Key (bits)	68	88	120	142
Hash result (bytes)	68	132	176	260
Memory (bytes)	2176	4224	5632	8320

3.5 Evaluation Hash Function

The evaluation hash function was proposed by Mehlhorn and Vishkin in 1984 [28]. It is one of the variants analyzed by Shoup in [35]. The input (of length $\leq tn$) is viewed as a polynomial $M(x)$ of degree $< t$ over GF(2^n). The key is a random element $\alpha \in$ GF(2^n), and the hash result is equal to $M(\alpha) \cdot \alpha \in GF(2^n)$. This family of hash functions is ϵ-AXU with $\epsilon = t/2^n$.

We have written an implementation for $n = 64$, where GF(2^{64}) was represented as GF(2)[x]/$f(x)$, with $f(x) = x^{64} + x^4 + x^3 + x + 1$. The evaluation of the polynomial is performed using Horner's rule, and with a precomputation of the mapping $\beta \mapsto \alpha \cdot \beta$ with $\beta \in$ GF(2^n). As in [35], two options have been considered, that provide a time-memory trade-off.

For this construction ϵ grows with the number of n-bit blocks in the input. The fastest method achieves a speed of approximately 240 Mbit/s in optimized Assembly language (122 Mbit/s in C++), and requires about 16 Kbyte of memory. The second method is about a factor of 7 slower (18 Mbit/s in C++), but requires only 2 Kbyte of memory. Shoup's implementation in C is a little slower than our Assembly version, but faster than our C++ code; the latter can probably be explained by better optimization in C versus C++, and maybe by the overhead of the operating system (Linux versus Windows '95).

3.6 Division Hash Function

The division hash function was proposed by Krawczyk [24], inspired by an earlier scheme by M.O. Rabin. It represents the input as a polynomial $M(x)$ of degree less than tn over GF(2). The hash key is a random irreducible polynomial $p(x)$ of degree n over GF(2). The hash result is $m(x) \cdot x^n \bmod p(x)$. Since the total number of irreducible polynomials of degree n is roughly equal to $2^n/n$, it follows that this family of hash functions is ϵ-AXU with $\epsilon = tn/2^n$.

Again, we have written an implementation for $n = 64$. The main step is the reduction, which can be optimized by using a precomputation of the mapping $g(x) \mapsto g(x) \cdot x^{64} \bmod p(x)$, with $\deg g(x) < 64$. Again, following [35], two options were considered, that provide a time-memory trade-off. For the key generation, see [35].

For this construction $\epsilon = t/2^{58}$, with t the number of 8-byte blocks in the input (for the same value of n, the security level is 6 bits smaller compared to the

evaluation hash function). The slower implementation uses 2 Kbyte of memory and runs at 14 Mbit/s in C++. Our fastest implementation uses 8 Kbyte of memory and achieves a speed of approximately 115 Mbit/s in C++, which is still slower than the evaluation hash function (in contrast to the conclusions of Shoup [35]). Therefore it was decided not to write optimized Assembly language.

Shoup generalizes this construction to polynomials over $\mathrm{GF}(2^k)$, where k divides n [35]. The main conclusion is that for this variant the key generation is faster, but the precomputation is a little slower. For $n = 64$, $\epsilon = t/2^{58}$ (for the same value of n, the security is 3 bits better than the simple division hash, but 3 bits worse than the evaluation hash), and the performance is identical to that of the division hash.

3.7 MMH Hashing

Halevi and Krawczyk propose MMH (Multilinear Modular Hashing) in [19]. This hash function consists of a (modified) inner product between message and key modulo a prime p (close to 2^w, with w the word length; below $w = 32$.) MMH is an ϵ-AXU, but with xor replaced by subtraction modulo p. The core hash function maps 32 32-bit message words and 32 32-bit key words to a 32-bit result. The key size is 1024 bits and $\epsilon = 1.5/2^{30}$. For larger messages, a tree construction can be used based on Proposition 6 and Proposition 10; the value of ϵ and the key length have to be multiplied by the height of the tree.

This algorithm is very fast on the Pentium Pro, which has a multiply and accumulate instruction (and on other machines with this feature). On a 32-bit machine, MMH requires only 2 instructions per byte for a 32-bit result. We have not (yet) implemented MMH, but include the impressive speed given in [19] for a 200 MHz Pentium Pro (optimized Assembly language): 1.2 Gbit/s for $\epsilon = 1.5/2^{30}$, and 500 Mbit/s for $\epsilon = 1.125/2^{59}$ (for large messages, if the data resides in cache). Note that this does not take into account the final addition of the key. The memory size of the implementation is not mentioned, but 1 Kbyte is probably sufficient.

The Pentium does not have this 'multimedia' instruction, and therefore the speed is reduced to about 350 Mbit/s for $\epsilon = 1.5/2^{30}$. However, one can use the floating point co-processor; this requires that one reduces the key words from 32 bits to 27 bits to avoid overflow. This results in about 500 Mbit/s for $\epsilon = 1.5/2^{25}$, and 260 Mbit/s for the double length variant with $\epsilon \approx 1.1/2^{49}$.

4 Comparing the Hash Functions

In §3, the properties of the different constructions have been listed. However, this information does not allow to compare the different schemes. As pointed out in §2.2, for message authentication, an ϵ-ASU or an ϵ-AXU combined with an encryption are required. For this purpose, Table 5 defines six algorithms that provide a comparable functionality. Note that all these functions are ϵ-AXU

Table 5. Six schemes for message authentication and a comparison of their performance ('+' denotes composition)

Scheme	Definition
A	bucket hash(AU) + evaluation hash (AXU)
B	bucket hash/short key (AU) + evaluation hash (AXU)
C	Toeplitz hash (AXU) + evaluation hash (AXU)
D	fast polynomial evaluation (AXU)
E	evaluation hash (AXU)
F	MMH (AXU) + evaluation hash (AXU)

Scheme	A	B	C	D	E	F
ϵ	2^{-32}	2^{-32}	2^{-32}	2^{-30}	2^{-49}	$1.1 \cdot 2^{-49}$
Speed (Mbit/s)	323	89	33	87	240	250†
Key (bits)	45,114	170	216	160	128	1243
Hash result (bytes)	8	8	8	4	8	8
Memory (Kbyte)	64	26	12	0.03	8	8.5†

† estimated

(some functions need a group operation other than exor such as scheme D with $q_0 \neq 2$).

The six algorithms from Table 5 are applied to an input of 256 Kbyte with as goal $\epsilon \approx 2^{-32}$. Note that it is not possible to compare these schemes with exactly the same parameters, because the value of ϵ for the best performance is typically related to the word size of the processor. Messages of 256 Kbyte offer a fair basis of comparison, because for shorter messages the performance varies more with the message size. By introducing an unambiguous padding rule, one can also process shorter inputs with the same code. The constructions can be extended easily to larger message lengths, either by extending the basic construction or by using trees. The full version of this paper will provide an extended comparison for different values of ϵ and input sizes.

All parameters are chosen to optimize for speed (rather than for memory), and the critical part of the code has been written in Assembly language. For schemes A, B, C, and F the input is divided into blocks and Proposition 6 of §2.3 is applied. This has the advantage that the description of the hash function fits in the cache memory. The second hashing step for these schemes uses the evaluation hash with $n = 64$. The results are summarized in Table 5.

Scheme A: the input is divided into 32 blocks of 8 Kbyte; each block is hashed using the same bucket hash function with $N = 160$, which results in an intermediate string of 20 480 bytes.

Scheme B: the input is divided into 64 blocks of 4 Kbyte; each block is hashed using the same bucket hash function with short key ($s = 42$, $L = 6$, $N = 128$), which results in an intermediate string of 10 752 bytes.

Scheme C: the input is divided into 64 blocks of 4 Kbyte; each block is hashed using a 33×1024 Toeplitz matrix, based on a δ-biased sequence of length 1056 generated using an 88-bit LFSR. The length of the intermediate string is 8 448 bytes.

Scheme D: the input is hashed twice using the polynomial evaluation hash function with $\epsilon = 2^{-15}$, resulting in a combined value of 2^{-30}; the value of $W = 5$. The performance is slightly key dependent; therefore an average over a number of keys has been computed.

Scheme E: this is simply the evaluation hash function with $t = 32\,768$. Note that the resulting value of ϵ is too small. However, choosing a smaller value of n that is not a multiple of 32 induces a performance penalty.

Scheme F: the input is divided into 2048 blocks of 128 bytes; each block is hashed twice using MMH. The length of the intermediate string is 16 384 bytes. It is not possible to obtain a value of ϵ closer to 2^{-32} in an efficient way.

Note that for bucket hashing and its variant the speed was measured under DOS, while for the other schemes (that use a finite field arithmetic library), the speed was measured under Windows '95. Timing measurements under DOS tend to be a little better.

The main conclusion is that scheme A, E and F are the fastest schemes. Scheme A offers the best performance for $\epsilon = 2^{-32}$. However, if the application needs a smaller value of ϵ ($\approx 2^{-49}$), scheme E and F are faster. Moreover, the key size and memory size for scheme A are large. If the key is generated using a pseudo-random function, or if the expanded key has to be decrypted before it can be used, this will introduce a performance penalty (for example, 13.7 msec if 3-DES is used, which runs at 13.8 Mbit/s and 0.43 msec for SEAL-3, which runs at 440 Mbit/s [10]). If memory requirements (both for the hash function and for the result) are an issue, scheme D is the best solution. It is about 4 times slower than scheme A, and requires less memory than scheme B. Note however that the other schemes can reduce the memory requirement (for the hash function) at the cost of a reduced speed. Scheme E and F offer a reasonable compromise between performance and memory requirements; scheme F needs a larger key and a powerful multiplier.

Recently Black et al. [9] have proposed UMAC, that uses a different type of inner product. UMAC is faster than MMH on processors with a fast multiplication (Pentium II, PowerPC 604). They report a performance of 3.4 Gbit/s on a 233 MHz Pentium II. The value of $\epsilon = 2^{-30}$, and the key size is about 32 768 bits (but slower versions with a shorter key are possible). It is also suggested to replace the encryption at the end by a pseudo-random function that takes a nonce as second input.

We provide a comparison with MAC algorithms based on [10]. The performance of HMAC [3] and MDx-MAC [30] depends on the underlying hash function (MDx-MAC is a few percent slower than HMAC). For MD5 [32], SHA-1 [17], RIPEMD-160, and RIPEMD-128 [15] the speeds are respectively 228 Mbit/s, 122 Mbit/s, 101 Mbit/s, and 173 Mbit/s (note however that the security of MD5 as a hash-function is questionable; this has no immediate impact to its use in HMAC and MDx-MAC, but it is prudent to plan for its replacement). For CBC-MAC [6,20], the performance corresponds approximately to that of the un-

derlying block cipher. For DES [16] this is 37.5 Mbit/s; for other block ciphers, this varies between 20 and 100 Mbit/s. XOR-MAC [5] is about 25% slower.

5 Concluding Remarks

The main advantages of universal hash functions are that their security is unconditional, and that their speed is comparable to or better than that of currently used MAC algorithms. In addition, they are easy to implement and easy to parallelize. Finally, they are often incremental [4] (this means that after small updates to the input, the output can be recomputed quickly). If they are used with a pseudo-random string generator, the unconditional security is lost, but what remains is a scheme that is easy to understand (the only cryptographic requirement is concentrated in one primitive).

Applications where universal hash functions can be used are the protection of high speed telecommunication networks, video streams, and for the integrity protection of file systems.

Many banking systems currently use unique MAC keys per transaction: for each transaction a new MAC key is derived from a master key. Therefore it seems natural to replace the MAC algorithms by universal hash functions, but with the following caveats: for short messages, the performance advantage of universal hash functions is limited. Moreover, constructions based on universal hash functions often give away part of their key bits (as an example, an input consisting of zero bits is often mapped to a hash result of zero bits). This is not a problem for the authentication, because the hash result is encrypted using a one-time pad. The opponent cannot exploit this property to forge messages, but he can find easily the output bits of the pseudo-random string generator. Therefore, any cryptographic weakness in the pseudo-random string generator may compromise the master keys, and it would be advisable to invest some of the time gained by using a universal hash function in strengthening the pseudo-random string generator. The security of the MAC algorithms depends on a cryptographic assumption, and thus it might well be possible that one finds a way to forge messages. However, for none of the state-of-the art MAC algorithms, an attack is known that can recover one or more key bits by observing a single text-MAC pair. Therefore an opponent will not be able to learn the MAC keys, and mounting an attack on the pseudo-random string generator will probably be more difficult (note that there is no proof of this). In summary, universal hash functions solve in an elegant and very efficient way the authentication problem, but put a higher requirement on the pseudo-random string generator, while MAC algorithms divide the (conjectured) cryptographic strength between the MAC algorithm and the pseudo-random string generator.

Acknowledgments. We would like to thank Antoon Bosselaers, Hugo Krawczyk, and Shai Halevi for helpful discussions and the anonymous referees for their constructive comments.

References

1. V. Afanassiev, C. Gehrmann, B. Smeets, "Fast message authentication using efficient polynomial evaluation," *Fast Software Encryption, LNCS 1267*, E. Biham, Ed., Springer-Verlag, 1997, pp. 190–204.
2. M. Atici, D.R. Stinson, "Universal hashing and multiple authentication," *Proc. Crypto'96, LNCS 1109*, N. Koblitz, Ed., Springer-Verlag, 1996, pp. 16–30.
3. M. Bellare, R. Canetti, H. Krawczyk, "Keying hash functions for message authentication," *Proc. Crypto'96, LNCS 1109*, N. Koblitz, Ed., Springer-Verlag, 1996, pp. 1–15. Full version: `http:// www.research.ibm.com/security/.`
4. M. Bellare, O. Goldreich, S. Goldwasser, "Incremental cryptography: the case of hashing and signing," *Proc. Crypto'94, LNCS 839*, Y. Desmedt, Ed., Springer-Verlag, 1994, pp. 216–233.
5. M. Bellare, R. Guérin, P. Rogaway, "XOR MACs: new methods for message authentication using block ciphers," *Proc. Crypto'95, LNCS 963*, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 15–28.
6. M. Bellare, J. Kilian, P. Rogaway, "The security of cipher block chaining," *Proc. Crypto'94, LNCS 839*, Y. Desmedt, Ed., Springer-Verlag, 1994, pp. 341–358.
7. J. Bierbrauer, T. Johansson, G. Kabatianskii, B. Smeets, "On families of hash functions via geometric codes and concatenation," *Proc. Crypto'93, LNCS 773*, D. Stinson, Ed., Springer-Verlag, 1994, pp. 331–342.
8. E. Biham, A. Shamir, *"Differential Cryptanalysis of the Data Encryption Standard,"* Springer-Verlag, 1993.
9. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, P. Rogaway, "UMAC: fast and secure message authentication," preprint, 1999.
10. A. Bosselaers, "Fast implementations on the Pentium," `http://www.esat.kuleuven.ac.be/~bosselae/fast.html`.
11. G. Brassard, "On computationally secure authentication tags requiring short secret shared keys," *Proc. Crypto'82*, D. Chaum, R.L. Rivest, and A.T. Sherman, Eds., Plenum Press, New York, 1983, pp. 79–86.
12. J.L. Carter, M.N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, Vol. 18, 1979, pp. 143–154.
13. W. Diffie, M.E. Hellman, "New directions in cryptography," *IEEE Trans. on Information Theory*, Vol. IT–22, No. 6, 1976, pp. 644–654.
14. H. Dobbertin, "RIPEMD with two-round compress function is not collisionfree," *Journal of Cryptology*, Vol. 10, No. 1, 1997, pp. 51–69.
15. H. Dobbertin, A. Bosselaers, B. Preneel, "RIPEMD-160: a strengthened version of RIPEMD," *Fast Software Encryption, LNCS 1039*, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71–82.
    See also `http://www.esat.kuleuven.ac.be/~bosselae/ripemd160`.
16. FIPS 46, *"Data Encryption Standard,"* Federal Information Processing Standard, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977 (revised as FIPS 46-1:1988; FIPS 46-2:1993).
17. FIPS 180-1, *"Secure Hash Standard,"* Federal Information Processing Standard (FIPS), Publication 180-1, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., April 17, 1995.
18. E. Gilbert, F. MacWilliams, N. Sloane, "Codes which detect deception," *Bell System Technical Journal*, Vol. 53, No. 3, 1974, pp. 405–424.
19. S. Halevi, H. Krawczyk, "MMH: Software message authentication in the Gbit/second rates," *Fast Software Encryption, LNCS 1267*, E. Biham, Ed., Springer-Verlag, 1997, pp. 172–189.

20. ISO/IEC 9797, *"Information technology – Data cryptographic techniques – Data integrity mechanisms using a cryptographic check function employing a block cipher algorithm,"* ISO/IEC, 1994.
21. T. Johansson, "Bucket hashing with a small key size," *Proc. Eurocrypt'97, LNCS 1233*, W. Fumy, Ed., Springer-Verlag, 1997, pp. 149–162.
22. G.A. Kabatianskii, T. Johansson, B. Smeets, "On the cardinality of systematic A-codes via error correcting codes," *IEEE Trans. on Information Theory*, Vol. IT–42, No. 2, 1996, pp. 566–578.
23. L. Knudsen, "Chosen-text attack on CBC-MAC," *Electronics Letters*, Vol. 33, No. 1, 1997, pp. 48–49.
24. H. Krawczyk, "LFSR-based hashing and authentication," *Proc. Crypto'94, LNCS 839*, Y. Desmedt, Ed., Springer-Verlag, 1994, pp. 129–139.
25. H. Krawczyk, "New hash functions for message authentication," *Proc. Eurocrypt'95, LNCS 921*, L.C. Guillou and J.-J. Quisquater, Eds., Springer-Verlag, 1995, pp. 301–310.
26. M. Matsui, "The first experimental cryptanalysis of the Data Encryption Standard," *Proc. Crypto'94, LNCS 839*, Y. Desmedt, Ed., Springer-Verlag, 1994, pp. 1–11.
27. J.L. Massey, "An introduction to contemporary cryptology," in *"Contemporary Cryptology: The Science of Information Integrity,"* G.J. Simmons, Ed., IEEE Press, 1991, pp. 3–39.
28. K. Mehlhorn, U. Vishkin, "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories," *Acta Informatica*, Vol. 21, Fasc. 4, 1984, pp. 339–374.
29. J. Naor, M. Naor, "Small bias probability spaces: efficient construction and applications," *Siam Journal on Computing*, Vol. 22, No. 4, 1993, pp. 838–856.
30. B. Preneel, P.C. van Oorschot, "MDx-MAC and building fast MACs from hash functions," *Proc. Crypto'95, LNCS 963*, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 1–14.
31. B. Preneel, P.C. van Oorschot, "On the security of two MAC algorithms," *Proc. Eurocrypt'96, LNCS 1070*, U. Maurer, Ed., Springer-Verlag, 1996, pp. 19–32.
32. R.L. Rivest, "The MD5 message-digest algorithm," *Request for Comments (RFC) 1321*, Internet Activities Board, Internet Privacy Task Force, April 1992.
33. P. Rogaway, "Bucket hashing and its application to fast message authentication," *Proc. Crypto'95, LNCS 963*, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 29–42. Full version `http://www.cs.ucdavis.edu/~rogaway/papers`.
34. C.E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, Vol. 28, 1949, pp. 656-715.
35. V. Shoup, "On fast and provably secure message authentication based on universal hashing, *Proc. Crypto'96, LNCS 1109*, N. Koblitz, Ed., Springer-Verlag, 1996, pp. 313–328.
36. G.J. Simmons, "A survey of information authentication," in *"Contemporary Cryptology: The Science of Information Integrity,"* G.J. Simmons, Ed., IEEE Press, 1991, pp. 381–419.
37. G.J. Simmons, "How to insure that data acquired to verify treat compliance are trustworthy," in *"Contemporary Cryptology: The Science of Information Integrity,"* G.J. Simmons, Ed., IEEE Press, 1991, pp. 615–630.
38. D.R. Stinson, "The combinatorics of authentication and secrecy codes," *Journal of Cryptology*, Vol. 2, No. 1, 1990, pp. 23–49.

39. D.R. Stinson, "Universal hashing and authentication codes," *Designs, Codes, and Cryptography*, Vol. 4, No. 4, 1994, pp. 369–380.
40. M.N. Wegman, J.L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, Vol. 22, No. 3, 1981, pp. 265–279.