Bucket Hashing with a Small Key Size

Thomas Johansson

Department of Information Technology, Lund University,
PO Box 118, S-221 00 Lund, Sweden
Email:thomas@it.lth.se

Abstract. In this paper we consider very fast evaluation of strongly universal hash functions, or equivalently, authentication codes. We show how it is possible to modify some known families of hash functions into a form such that the evaluation is similar to "bucket hashing", a technique for very fast hashing introduced by Rogaway. Rogaway's bucket hash family has a huge key size, which for common parameter choices can be more than a hundred thousand bits. The proposed hash families have a key size that is close to the key size of the theoretically best known constructions, typically a few hundred bits, and the evaluation has a time complexity that is similar to bucket hashing.

Keywords. Universal hash functions, message authentication, authentication codes, bucket hashing, software implementations.

1 Introduction

Universal hashing is a concept that was introduced by Carter and Wegman [8] in 1979. Since then, many results in theoretical computer science use different kinds of universal hashing. One of the main topics in universal hashing is called *strongly universal hashing*, and has a large amount of applications in computer science. The most widely known application in cryptography is the construction of unconditionally secure authentication codes. The model for unconditionally secure authentication codes was originally developed by Simmons [25, 26], see also [10]. One of the most important aspect of strongly universal hash functions is that the constructions should be simple to implement in software and/or hardware. Such implementation aspects have recently been in focus, and there are several papers addressing this topic [13, 16, 17, 22, 24, 11, 1].

Message authentication is one of the most common cryptographic settings today. In this setting a transmitter and a receiver share a secret key e. When the transmitter wants to send the receiver a message s, he computes a socalled *message authentication code*[1] (MAC), MAC $= f_e(s)$, and sends the pair (s, MAC). Here $f_e()$ denotes the function producing the MAC using key e. Receiving a pair (s', MAC') the receiver checks that $\mathrm{MAC}' = f_e(s')$. If this is the case, the message is accepted as authentic, otherwise it is rejected.

[1] In the theory of universal hashing, this is usually referred to as a *tag* (or an *authenticator*).

W. Fumy (Ed.): Advances in Cryptology - EUROCRYPT '97, LNCS 1233, pp. 149-162, 1997.

The fastest software MACs in common use today are based on software efficient cryptographic hash functions, such as MD5 [21, 7]. We refer to such an approach as the *MAC scheme approach*. For an overview, see [18, 19, 20]. Since we are computing one of the fastest types of cryptographic primitives[2] on a string essentially identical to the message, one might think that it is not possible to do much better. However, as was shown by Wegman and Carter already in 1981 [28], this is not the case. It was noted that one does not need to work with a "cryptographically strong primitive". A "cryptographically strong primitive" needs some complexity to resist attacks (e.g. many rounds), and this complexity is also time consuming. Through Wegman and Carter's universal hashing, one can instead work with a very simple function (the universal hash function) to produce a MAC. We refer to this approach as the ***universal hash approach***. The details of such an approach are given in the last section of this paper. We review some advantages of using the universal hash approach instead of the usual MAC scheme approach.

- *Speed:* The universal hash function can be very simple to implement, and experimental implementations (e.g. [11]) indicate that producing the MAC using universal hash functions is faster than for example MD5 based techniques.
- *Parallelizable:* For this self-explaining property to hold, it is sufficient that (a part of) the universal hash function is a linear function, which is usually the case.
- *Incremental:* If a small part of the message is modified or a part is added to the message, we do not need to perform the new MAC calculation over the whole message but only over the small part that was modified/added. This is again a consequence of the linearity of (a part of) the universal hash function.
- *Unconditional security/Provable security:* Universal hashing is "unconditionally secure", i.e., the probability of success in an attack is independent of computational resources. The universal hash approach sometimes includes usage of some cryptographic primitive to provide multiple use. This usage can be done in the form of provable security, i.e., an adversary who can break the scheme can also break the underlying cryptographic primitive [22].

Note that MAC schemes are highly nonlinear, hence usually neither parallelizable nor incremental[3]. Also, reductions for MAC schemes to show provable security are not at all as tight as for the universal hash approach, for details see [4, 22, 2].

This paper studies very fast software implementations of strongly universal hash functions. One of the most important steps in this direction was taken by Rogaway when he introduced a technique for hashing called "bucket hashing" [22]. It is a very efficient way of producing a MAC, ideally requiring only 6 - 10 simple instruction per word to be authenticated. The drawback of this approach

[2] MD5 can probably not be considered to be a "cryptographically strong primitive", due to an attack by Dobbertin [9].

[3] In [3], a MAC scheme (XOR-MAC) was presented, which is incremental.

is the huge key size that is included, which for common parameter choices can be more than a hundred thousand bits. This requires the key to be generated through a pseudo-random number generator.

As mentioned before, there have been some previous work on software efficiency of universal hash functions, [17, 24, 11, 1]. The recent paper [11] considers evaluation of universal hash functions on processors supporting very fast integer multiplication. On such processors, they get an extremely high speed. Another recent paper [1] is more in the line of our work, focusing on evaluation in hash families with a small key size.

Our contribution is to show how it is possible to modify some known families of hash functions into a form such that the evaluation is similar to "bucket hashing". The proposed hash functions have a key size that is close to the key size of the theoretically best known constructions, which for common parameter choices can be around a hundred bits for a single use. Furthermore, the evaluation has a time complexity that is similar to bucket hashing and use the same simple instructions.

The paper is organized as follows. In Section 2 the basic definitions in universal hashing and authentication theory are given, as well as connections between them. Section 3 reviews bucket hashing, and in Section 4 we introduce our new approach to bucket hashing. In Section 5 we discuss implementation and parameter choices and finally, in Section 6, we review how the proposed hash families are used to produce a MAC.

2 Universal hash functions and authentication codes

In universal hashing, we consider a hash family $\mathcal{G}$, which is a set $\mathcal{G}$ of $|\mathcal{G}|$ functions such that $g : X \rightarrow Y$ for each $g \in \mathcal{G}$. Interesting cardinality parameters for a hash family are $|\mathcal{G}|$, $|X|$, and $|Y|$. Two relevant definitions are the following.

Definition 1. A hash family $\mathcal{G}$ is called *ϵ-almost universal$_2$* if for any two distinct elements $x_1, x_2 \in X$, there are at most $\epsilon|\mathcal{G}|$ functions $g \in \mathcal{G}$ such that $g(x_1) = g(x_2)$. We use the abbreviation ϵ-AU$_2$ for the family.

Definition 2. A hash family $\mathcal{G}$ is called *ϵ-almost strongly universal$_2$* if

i) for any $x \in X$ and any $y \in Y$, there are exactly $|\mathcal{G}|/|Y|$ functions $g \in \mathcal{G}$ such that $g(x) = y$.

ii) for any two distinct elements $x_1, x_2 \in X$, and for any two elements $y_1, y_2 \in Y$, there are at most $\epsilon|\mathcal{G}|/|Y|$ functions $g \in \mathcal{G}$ such that $g(x_1) = y_1$, and $g(x_2) = y_2$.

We here use the abbreviation ϵ-ASU$_2$.

For a more thorough treatment of universal hashing, we refer to [27], where these concepts are derived further. We will instead consider the known equivalences between strongly universal hashing and authentication codes.

Authentication theory as originally described by Simmons [25], [26], see also [10], considers the problem of two trusting parties, who want to send information from the transmitter to the receiver in the presence of an adversary. The adversary may introduce false messages to the receiver or replace a legal message with a false one. To protect against these threats, the sender and the receiver share a secret key. The key is then used in an authentication code (A-code).

A *systematic* (or Cartesian) A-code is a code where the information to be transmitted appears in plaintext in the transmitted message. Such a code is a triple $(\mathcal{S}, \mathcal{E}, \mathcal{Z})$ of finite sets and a map $f : \mathcal{S} \times \mathcal{E} \to \mathcal{Z}$. Here $\mathcal{S}$ is the set of source states, i.e., the information that is to be transmitted, $\mathcal{E}$ is the set of keys, and $\mathcal{Z}$ is the tag alphabet. When the transmitter wants to send the information $s \in \mathcal{S}$ using his secret key $e \in \mathcal{E}$, he transmits the message $m = (s, z)$, where $z = f(s, e)$, and $m \in \mathcal{M} = \mathcal{S} \times \mathcal{Z}$. When the receiver receives a message $m' = (s', z')$, he checks the authenticity by calculating whether $z' = f(s', e)$ or not. If equality holds, the message m is called valid. The adversary has two different attacks to choose between. He might introduce a false message $m = (s, z)$, and hence impersonating the transmitter, called the ***impersonation attack***. He can also choose to observe a transmitted message $m = (s, z)$, and then replace this message with another message $m' = (s', z')$, where $s' \neq s$. This is called the ***substitution attack***. The probability of success for the adversary when trying either of the two attacks, denoted by P_I and P_S respectively, are formally defined by $P_I = \max_{s,z} P(m = (s,z) \text{ valid})$ and $P_S = \max_{s,z} \max_{s' \neq s, z'} P(m' = (s', z') \text{ valid} | m = (s, z) \text{ observed})$. We assume that the keys are uniformly distributed. Then these probabilities can be written as

$$P_I = \max_{s,z} \frac{|\{e \in \mathcal{E} : z = f(s,e)\}|}{|\{e \in \mathcal{E}\}|}, \tag{1}$$

$$P_S = \max_{s,z} \max_{s' \neq s, z'} \frac{|\{e \in \mathcal{E} : z = f(s,e), z' = f(s',e)\}|}{|\{e \in \mathcal{E} : z = f(s,e)\}|}. \tag{2}$$

For a review of different bounds and constructions of A-codes, we refer to [15]. The main result on the equivalence between strongly universal hashing and authentication/coding theory is the following.

Theorem 3 [5, 28, 27].

i) If there exists a q-ary code with codeword length n, cardinality M, and minimum Hamming distance d, then there exists an ϵ-AU_2 family of hash functions where $\epsilon = 1 - d/n$, $|\mathcal{G}| = n$, $|X| = M$, and $|Y| = q$. Conversely, if there exists an ϵ-AU_2 family of hash functions, then there exists a code with parameters as above.

ii) If there exists an A-code with parameters $|\mathcal{S}|$, $|\mathcal{E}|$, $P_I = 1/|\mathcal{Z}|$, and P_S, then there exists an ϵ-ASU_2 family of hash functions where $\epsilon = P_S$, $|\mathcal{G}| = |\mathcal{E}|$, $|X| = \mathcal{S}$, and $|Y| = |\mathcal{Z}|$. Conversely, if there exists an ϵ-ASU_2 family of hash functions, then there exists an A-code with parameters as above.

We review the equivalence ii) above. Each key $e \in \mathcal{E}$ in the A-code corresponds to a unique function g_e in $\mathcal{G}$, and $\mathcal{S} = X$. The tag z in the authentication code is then obtained as

$$z = g_e(s).$$

The significance of ϵ-AU$_2$ families in strongly universal hashing lies in the fact that they are very useful when constructing strongly universal hash families. This is due to the following result by Stinson.

Lemma 4 [27]. *Let $\mathcal{G}_1$ be ϵ_1-AU_2 from X_1 to Y_1 and let $\mathcal{G}_2$ be ϵ_2-ASU_2 from Y_1 to Y_2. Then $\mathcal{G} = \{g_2(g_1(x)) : g_1 \in \mathcal{G}_1, g_2 \in \mathcal{G}_2\}$ is ϵ-ASU_2 with $\epsilon = \epsilon_1 + \epsilon_2$.*

Most constructions of ϵ-ASU$_2$ families of hash functions for large $|X|$ use this composition construction. The constructions giving best performance in terms of key size [5] (see also [12]) uses Reed-Solomon codes as the ϵ-AU$_2$ family in the above composition construction. Another useful result, originally used in the Wegman-Carter construction [28], is obtained through the Cartesian product.

Lemma 5 [28, 27]. *Let $\mathcal{G}$ be ϵ-AU_2 from X to Y. Let $\mathcal{G}^m = \{g^m(x_1, x_2, \ldots, x_m) = (g(x_1), g(x_2), \ldots, g(x_m)) : g \in \mathcal{G}_1\}$ be a set of hash functions from X^m to Y^m. Then $\mathcal{G}^m = \{g^m\}$ is ϵ-AU_2.*

3 Bucket hashing

The bucket hashing technique was introduced by Rogaway in [22]. It gave rise to ϵ-AU$_2$ families that are extremely fast to compute, at the cost of a very large key. Rogaway's arguments was to produce this long key through a pseudo-random number generator. We review some details of the bucket hashing technique.

Fix a "word size" $w \geq 1$. For $n \geq N$ the hash function is defined to map from $X = \{0,1\}^{wn}$ to $Y = \{0,1\}^{wN}$. The number N is referred to as "the number of buckets". It is further required that $N(N-1)(N-2) \geq 6n$.

Let $\mathcal{H}_B[w,n,N]$ denote the hash family. Then each $h \in \mathcal{H}_B[w,n,N]$ is specified by a length n list where each entry contains 3 integer numbers in the interval $[0, N-1]$. Denote this list by $h = (h_0, h_1, \ldots, h_{n-1})$, where $h_i = (h_i^1, h_i^2, h_i^3)$. The hash family $\mathcal{H}_B[w,n,N]$ is given by the hash functions taken over the set of all possible lists h subject to the constraint that no two of the 3-element sets in the list are the same, i.e., $h_i \neq h_j, \forall i \neq j$.

With a given hash function $h = (h_0, h_1, \ldots, h_{n-1})$, the output value $h(x)$ is defined as follows. Let $x = x_0 x_1 \cdots x_{n-1}$, where each x_i is a bit vector of length w. Initialize y_j to 0^w for $0 \leq j \leq N-1$. Then, for each i, replace $y_{h_i^1}$ with $y_{h_i^1} \oplus x_i$, $y_{h_i^2}$ with $y_{h_i^2} \oplus x_i$, and $y_{h_i^3}$ with $y_{h_i^3} \oplus x_i$. Then set the output to be

$h(x) = y_0y_1 \cdots y_{n-1}$. In pseudocode, we can write the algorithm as follows.

```
for $j = 0$ to $N - 1$ do
    $y[j] = 0^w$
for $i = 0$ to $n - 1$ do
    $y[h_i^1] = y[h_i^1] \oplus x_i$
    $y[h_i^2] = y[h_i^2] \oplus x_i$
    $y[h_i^3] = y[h_i^3] \oplus x_i$
return $y[0]y[1] \cdots y[n-1]$
```

The computation of $h(x)$ gives rise to the name "bucket hashing", since it can be envisioned in the following way. We have N initially empty buckets. The first word of x is then thrown into three buckets, specified by h_0. Then the second word of x is thrown into three buckets, specified by h_1, and so on. Finally, 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. This is shown in Figure 1.

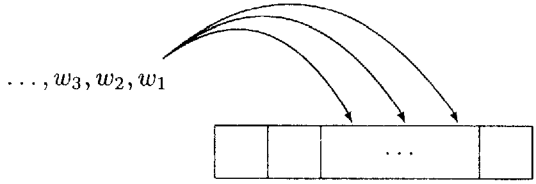

Fig. 1. A word is thrown into three buckets in Rogaway's bucket hashing.

The collision probability ϵ is given by a complicated expression [22] and instead of giving it here, we will just transfer some numerical values from [22] whenever needed. For example, for $n = 1024$ and $N = 100$, the collision probability is approximately 2^{-28}, i.e., $\mathcal{H}_B[w, n, N]$ is an ϵ-AU$_2$ hash family where $\epsilon = 2^{-28}$.

The bucket hashing approach gives a very fast implementation, since it only requires simple word operations as `load`, `store` and `xor`. Rogaway estimates that one word can be processed using only 6 – 10 such simple instructions. Usually such simple instructions require only one clock cycle each, and can even be executed in parallel on many processors.

The drawback of the bucket hashing approach is the long key that is used. The key size is approximately $3n \log_2 N$, which is huge. For $n = 1024$ and $N = 100$, this is already more than 20000 bits, whereas a theoretically good construction [6, 14] for the same ϵ would require 76 key bits. Hence, the key bits in the bucket hashing construction must be generated by a pseudo-random number generator. This might be time consuming and the hash families are no longer unconditionally secure.

4 Bucket hashing with a small key size

The purpose of this section is to slightly modify some existing constructions of ϵ-AU$_2$ families of hash functions and then show that they can be implemented in a way that resembles the bucket hashing technique. The approach taken here is based on evaluation of polynomials similar to [6, 14]. The difference is essentially that we only consider polynomials over $GF(2)$, whereas the previous approaches consider polynomials over a larger field.

The following is a description of an ϵ-AU$_2$ family of hash functions. Let $\mathcal{P}_D$ be the set of all polynomials over $GF(2)$ without constant term and with degree at most D , i.e.,

$$\mathcal{P}_D = \{p(x) : p(x) = p_1x + p_2x^2 + \cdots + p_Dx^D, p_i \in GF(2), 1 \leq i \leq D\}.$$

The hash family $\mathcal{G}_1$ is defined as follows. Let the functions in $\mathcal{G}_1$ map from $X = \mathcal{P}_D$ to $Y = GF(2^m)$, let $p \in \mathcal{P}_D = X$, $\alpha \in GF(2^m)$, and define

$$g_\alpha(p) = p(\alpha).$$

Theorem 6. *The family*

$$\mathcal{G}_1 = \{g_\alpha(p) : \alpha \in GF(2^m)\},$$

is an ϵ-AU_2 family of hash functions where

$$|\mathcal{G}_1| = 2^m, \ |X| = 2^D, \ |Y| = 2^m, \ \epsilon = \frac{D}{2^m}.$$

Proof.

$$\begin{aligned}
\epsilon &= \max_{x_1 \neq x_2} \frac{|\{g \in \mathcal{G}_1 : g(x_1) = g(x_2)\}|}{|\mathcal{G}_1|} \\
&= \max_{x_1 \neq x_2} \frac{|\{\alpha \in GF(2^m) : p_{x_1}(\alpha) = p_{x_2}(\alpha)\}|}{2^m} \\
&= \max_{x_1 \neq x_2} \frac{|\{\alpha \in GF(2^m) : p_{x_1 - x_2}(\alpha) = 0\}|}{2^m} \\
&\leq \frac{D}{2^m},
\end{aligned}$$

since any nonzero polynomial of degree D has at most D zeros. □

Note that this is a slightly weaker result than in [6], where the polynomials have coefficients from $GF(2^m)$ and this does not change ϵ. However, as we will see, our approach will give a very efficient evaluation.

A generalization of the above construction is the hash family $\mathcal{G}_2$, constructed as follows. Let the functions in $\mathcal{G}_2$ map from $X = \mathcal{P}_D^n$ to $Y = GF(2^m)$, let $p = (p_1, p_2, \cdots, p_n) \in \mathcal{P}_D^n = X$ and define

$$g_{\alpha_1, \ldots, \alpha_n}(p) = p_1(\alpha_1) + \cdots + p_n(\alpha_n).$$

Theorem 7. *The family*

$$\mathcal{G}_2 = \{g_{\alpha_1,\ldots,\alpha_n}(p) : \alpha_1, \ldots, \alpha_n \in GF(2^m), \}$$

is an ϵ-AU_2 family of hash functions where

$$|\mathcal{G}_2| = 2^{nm}, \ |X| = 2^{nD}, \ |Y| = 2^m, \ \epsilon = \frac{D}{2^m}.$$

Proof. Similar to Theorem 6. □

The central topic is to have a fast evaluation. We will now describe a hash family, denoted $\mathcal{G}_B[w, n, N]$, which has a fast evaluation. Then we show that this hash family is an implementation of $\mathcal{G}_1$ or $\mathcal{G}_2$, depending on a parameter choice in $\mathcal{G}_B[w, n, N]$.

Description of $\mathcal{G}_B[w, n, N]$: Fix w as the "word size" and let $N = 2^{m/L}$. For $n \geq N$ the hash function is defined to map from $X = \{0,1\}^{wn}$ to $Y = \{0,1\}^{wm}$. In the implementation there is an intermediate level using L arrays with $N = 2^{m/L}$ words in each, so the hash function can be described to map

$$X = \{0,1\}^{wn} \to \{0,1\}^{wNL} \to \{0,1\}^{wm} = Y.$$

The number N can be interpreted as "the number of buckets" and the number L can be interpreted as "the number of rows of buckets".

Each $h \in \mathcal{G}_B[w, n, N]$ is specified by a length n list where each entry contains L integer numbers in $[0, N-1]$. Denote this list by $h = (h_0, h_1, \ldots, h_{n-1})$, where $h_i = (h_i^0, \ldots, h_i^{L-1})$. The hash family $\mathcal{G}_B[w, n, N]$ is given by a set of such lists, which we call the set of all allowed lists. Different choices of this set will give different hash families.

With a given hash function $h = (h_0, h_1, \ldots, h_{n-1})$, the output value $h(x)$ is defined as follows. Let $x = x_0 x_1 \cdots x_{n-1}$, where each x_i is a bit vector of length w. Introduce L arrays of length N, called y_k, $0 \leq k \leq L-1$. Initialize $y_k[j]$ to 0^w for $0 \leq k \leq L-1$ and $0 \leq j \leq N-1$. Then, for each i, replace $y_0[h_i^0]$ with $y_0[h_i^0] \oplus x_i$, $y_1[h_i^1]$ with $y_1[h_i^1] \oplus x_i$, continuing in this way, and finally replacing $y_{L-1}[h_i^{L-1}]$ with $y_{L-1}[h_i^{L-1}] \oplus x_i$. This first step has hashed the input to the intermediate level of L rows of buckets, each containing N words. The procedure for $L = 2$ is shown in Figure 2.

Next, for each array, we compress the array in the following way. In $GF(2^{m/L})$ we have a primitive element γ which satisfies $\gamma^{m/L} = g_{m/L-1}\gamma^{m/L-1} + \cdots + g_1\gamma + g_0$, where $g_i \in GF(2)$. From $j = N-2$ down to m/L we add (xor) $y_k[j]$ to $y_k[j-i]$ for all i such that $g_i = 1$. Finally, set the output to be $h(x) = y_0 \cdots y_{L-1}$, where y_i denotes the content of the array $(y_i[0] \cdots y_i[m/L-1])$.

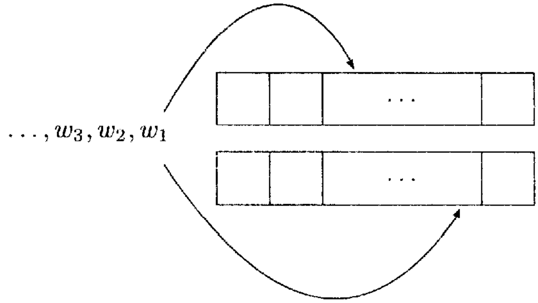

Fig. 2. A word is thrown into one bucket in each "row of buckets", here $L = 2$.

Assuming a generated list h, we can give a pseudocode for the case $L = 2$ with $\gamma^{m/2} = \gamma^{m/2-b} + 1$, for some integer b with $1 \leq b \leq m/2 - 1$, as follows.

```
for $j = 0$ to $N - 2$ do
    $y_0[j] = 0^w$, $y_1[j] = 0^w$
for $i = 0$ to $n - 1$ do
    $y_0[h_i^0] = y_0[h_i^0] \oplus x_i$
    $y_1[h_i^1] = y_1[h_i^1] \oplus x_i$
for $j = N - 2$ to $m/2$ do
    $y_0[j - b] = y_0[j - b] \oplus y_0[j]$, $y_0[j - m/2] = y_0[j - m/2] \oplus y_0[j]$
    $y_1[j - b] = y_1[j - b] \oplus y_1[j]$, $y_1[j - m/2] = y_1[j - m/2] \oplus y_1[j]$
return $y_0[0]y_0[1]\cdots y_0[m/2-1]y_1[0]y_1[1]\cdots y_1[m/2-1]$
```

Observe that $\mathcal{G}_B[w, n, N]$ can be evaluated efficiently using only simple instructions as `load`, `store` and `xor`. Next we prove equivalences between $\mathcal{G}_B[w, n, N]$ and the hash families $\mathcal{G}_1$ and $\mathcal{G}_2$. Let $[z]$ denote the vector $(z_0, \dots, z_{L-1})$, where $\gamma^{z_i} \in GF(2^{m/L})$, and by convention $\gamma^{N-1} = 0$, such that $z = \gamma^{z_0} + \gamma^{z_1}\beta + \cdots + \gamma^{z_{L-1}}\beta^{L-1} \in GF(2^m)$. Here $\beta \in GF(2^m)$ and $h(\beta) = 0$ for some irreducible polynomial $h(x)$ of degree L over $GF(2^{m/L})$.

Theorem 8. *Let the set of allowed lists be*

$$\{([\alpha], [\alpha^2], \dots, [\alpha^n]), \forall \alpha \in GF(2^m)\}.$$

Then the hash family $\mathcal{G}_B[w, n, N]$ is equivalent to $\mathcal{G}_1^m$, i.e., the Cartesian product of w hash families $\mathcal{G}_1$ as in Lemma 5.

Proof. The proof is in two steps.

1. The ith bit of each output word is only dependent on the ith bit of each input word x_j and independent of all the other bit positions in the input words. Hence we can view the hash family $\mathcal{G}_B[w, n, N]$ as a Cartesian product of w hash families each having an input word size of one, as in Lemma 5. So w.l.o.g we assume $w = 1$.

2. Regard each array (of length $2^{m/L}$) as corresponding to an enumeration of elements in $GF(2^{m/L})$, i.e., $GF(2^{m/L}) = \{\gamma^0, \gamma^1, \gamma^2, \ldots, \gamma^{2^{m/L}-2}, 0\}$ and entry z_i corresponds to element γ^{z_i}. View $GF(2^m)$ as a direct product of such subfields, i.e.,

$$GF(2^m) = \underbrace{GF(2^{m/L}) \otimes \cdots \otimes GF(2^{m/L})}_{L},$$

where each subfield is represented by one array. An element $z \in GF(2^m)$ is represented by the vector $z = (z_0, \ldots, z_{L-1})$, where $\gamma^{z_i} \in GF(2^{m/L})$. Putting an input word x_i $(w = 1)$ in bucket z_i means adding γ^{z_i} in the subfield, and hence, putting an input word x_i in buckets represented by $[z] = (z_0, \ldots, z_{L-1})$ means adding $x_i z$ to the previous content of the buckets. Hence, the list $([\alpha], [\alpha^2], \ldots, [\alpha^n])$, for $\alpha \in GF(2^m)$ means adding $x_0\alpha + x_1\alpha^2 + \cdots + x_n\alpha^n$. The result is now represented as powers of γ in each subfield (array). In the last part, adding $y_k[j]$ to $y_k[j-i]$ for all i such that $g_i = 1$ from $j = N-2$ down to m/L simply means reducing the powers of γ to the basis $\{\gamma^{m/L-1}, \ldots, \gamma, 1\}$. Hence the output of $\mathcal{G}_B[w,n,N]$ is $x_0\alpha + x_1\alpha^2 + \cdots + x_n\alpha^n \in GF(2^m)$, where $GF(2^m) = GF(2^{m/L}) \otimes \cdots \otimes GF(2^{m/L})$ and $GF(2^{m/L})$ is represented using the basis $\{\gamma^{m/L-1}, \ldots, \gamma, 1\}$. □

A similar result can be obtained for the family $\mathcal{G}_2$. For example, let the set of allowed lists be

$$\{([\alpha_1], [\alpha_2], \ldots, [\alpha_n]), \forall \alpha_i \in GF(2^m), 1 \leq i \leq n\},$$

i.e., the set of all possible lists. Then $\mathcal{G}_B[w,n,N]$ is equivalent to $\mathcal{G}_2^m$ with $D = 1$, i.e., the Cartesian product of w hash families $\mathcal{G}_2$ with $D = 1$.

5 Implementation and parameter choices

Clearly, the efficiency of the evaluation will depend on the choice of parameters in the above description. Let us consider some different ways to implement $\mathcal{G}_B[w,n,N]$. Note that the situation is very similar to Rogaway's bucket hashing. We can process word by word from input, or we can process bucket by bucket. Furthermore, we can use a self-modifying code (the actual hash function is implemented in the program code), or we can read the bucket/word locations from a table in memory. The fastest choice is a self-modifying code processing bucket by bucket. Then we can keep the current bucket in a register while processing, requiring only one `load` and one `xor` instruction for each input word and each row of buckets. Hence, for L rows this requires $2L$ simple instructions. For further details we refer to [22].

Furthermore, the compression of the arrays means LNc `load`, `add` and `store` operations, where c is the number of nonzero coefficients in the primitive polynomial defining γ (this can usually be chosen to be 2). For $n >> N$ the time to do the compression part is hence negligible compared to the first part. Initialization of the list h is done only once. Hence, when concatenating this hash function many times using Lemma 5, the time to execute this part is also negligible.

For tabulating some values, we regard $n = 8N$ as being sufficient for considering the compression to be negligible in time. Also, the generation of the list h is different depending on the actual choice of hash function. In all cases we are aware of the fact that we need to concatenate a few, say 10, hash families in order to make the time to process this part small. Alternatively, considering multiple use, we can assume that the list h is generated once and then kept fixed. We tabulate some values for different parameter values in Table 1. The input size, the output size and N are given in number of words; ϵ is the collision probability; the key size is given in bits; and the time column gives the minimal number of simple instructions per word for a self-modifying code processing bucket by bucket.

Hash function and parameters	Input size	Output size	ϵ	Key size	Time
Bucket hashing, $n = 4096$, $N = 40$	2^{12}	40	2^{-20}	94000	6
Bucket hashing, $n = 4096$, $N = 200$	2^{12}	200	2^{-34}	94000	6
$\mathcal{G}^1$, $N = 2^{10}$, $L = 3$	2^{13}	30	2^{-17}	30	6
$\mathcal{G}^1$, $N = 2^{10}$, $L = 4$	2^{13}	40	2^{-27}	40	8
$\mathcal{G}^1$, $N = 2^{10}$, $L = 5$	2^{13}	50	2^{-37}	50	10
$\mathcal{G}^1$, $N = 2^{10}$, $L = 7$	2^{13}	70	2^{-57}	70	14
$\mathcal{G}^1$, $N = 2^{20}$, $L = 4$	2^{23}	80	2^{-57}	80	8
$\mathcal{G}^2$, $N = 2^{10}$, $L = 4$, $D = 512$	2^{13}	40	2^{-31}	640	8

Table 1. A comparison for some different parameter choices.

In order to process each simple instruction in at most one clock cycle (we might execute several in parallel) on a usual processor, each reference to a memory location needs to be in the on-chip cache of the processor. Hence, for a self-modifying code processing bucket by bucket, the input to one hash function must fit the on-chip cache, giving restrictions on the input size and thus on N. Examining the sizes of the on-chip caches of todays processors, $N = 2^{10}$ is probably about the maximum size of the arrays under these circumstances.

Note the fact that some properties of Rogaway's bucket hashing and the proposed techniques are different and hence the techniques are not directly comparable. Especially, $\mathcal{G}_1$ gives a much higher compression, i.e., input size/output size is much smaller. This means that including $\mathcal{G}_1$, it is enough to concatenate two hash families using Lemma 4 to get the desired output size, whereas bucket hashing requires many concatenations to obtain the desired output size. This can be a problem for large messages since producing a large hash output that has to be written in memory and then further processed will produce cash-misses etc.

6 The universal hash approach in practice

Up to this point, we have only considered how to construct the ϵ-AU$_2$ hash family. This short section overviews how to use the ϵ-AU$_2$ hash family to produce an ϵ-ASU$_2$ hash family that gives an authentication tag (MAC) and also have the properties mentioned in Section 1.

The usage of ϵ-ASU$_2$ families of hash functions in the described way applies to the case of sending/storing one message with fixed length (variable length can easily be included [22]). Sometimes one is interested in multiple use, i.e., sending/storing many message where each message needs individual authentication. In the unconditionally secure approach, the solution is to add new random key bits for each additional messages to be hashed. If $h_{e_1}()$ is the ϵ-ASU$_2$ hash function, the MACs $(z_1, z_2, \ldots)$ for a sequence of messages $s_1, s_2, \ldots$ can be produced by

$$z_1 = h_{e_1}(s_1), \; z_2 = h_{e_1}(s_2) + e_2, \; z_3 = h_{e_1}(s_3) + e_3, \ldots,$$

where $e_2, e_3, \ldots$ are randomly chosen keys of same length as the MAC. It can be proved [28] that this procedure gives the same P_I and P_S as for the single message case.

In some cases, the number of messages is limited and then it is preferable to keep the unconditionally secure approach. In other cases, the set $\{e_2, e_3, \ldots\}$ of randomly chosen keys is too large to be kept secret in an unconditionally secure way. Instead, one uses a pseudo-random number generator to produce this set. In such a case, some of the motivation to consider ϵ-AU$_2$ hash families with a short key is lost, since the same pseudo-random number generator can be used to produce the hash function itself.

A complete ϵ-ASU$_2$ hash family obtained by Lemma 4 can be described as follows. Let x be the message that is to be hashed. Divide x into suitable sized substrings $x = x_1 x_2 \cdots x_n$. Apply a secretly chosen ϵ-AU$_2$ hash function h_1 and calculate $y_i = h_1(x_i)$, $1 \leq i \leq n$. For the obtained string $y = y_1 y_2 \cdots y_n$ (now of modest size) we have secretly selected another ϵ-ASU$_2$ hash function h_2 and calculates $w = h_2(y)$. In an unconditionally secure authentication code we would select a secret key e and form a MAC of the form MAC $= w + e$. For the next message, we use a new value of e, etc.

If we want to produce the sequence of keys using a pseudo-random number generator we can do as follows. We have a counter, call it *cnt*, which is initially zero. This counter is used together with a cryptographic primitive, e.g. RC5 [23], using a secret key e. The MAC for the message is given by

$$\mathrm{MAC} = w + \mathrm{RC5}_e(cnt),$$

together with the used value of the counter. Finally, *cnt* is incremented.

Example: As a particular example for $w = 32$, choose $\mathcal{G}_1$ with $N = 1024$ and $L = 7$ as the first hash family. The key e_1 to select the hash function is 70 bits. We have 8092 word input, producing a 70 word output and $\epsilon = 2^{-57}$. As the second hash family, choose the polynomial evaluation hash [6, 14] over $GF(2^{70})$. Our key e_2 for this hash family is 70 bits as well.

Divide the input x in 32Kbyte blocks $x = x_1 x_2 \cdots x_n$. Apply the methods described in Section 4 on each block x_i, receiving n 70-word blocks called y_i, by $y_i = g_{e_1}(x_i)$. Then form the string $y = y_1 y_2 \cdots y_n$ and interpret this as a polynomial over $GF(2^{70})$. This polynomial, call it $y(x)$, will then have degree $32n$. Evaluate the polynomial in e_2, obtaining $w = y(e_2)$. Then calculate the MAC as MAC $= w + e_3$, where e_3 is a third 70 bit key. Finally, we output (x, MAC). The value of ϵ will depend on n, but for input sizes smaller than 8Mbyte we have $\epsilon < 2^{-56}$.

Alternatively, using RC5 in multiple use we calculate the MAC as MAC $= w + \mathrm{RC5}_{e_3}(cnt)$, output (x, cnt, MAC), and increment the counter.

References

1. V. Afanassiev, C. Gehrmann, B. Smeets, Fast message authentication using efficient polynomial evaluation, *Proceedings of Fast Software Encryption Conference '97*, to appear.
2. M. Bellare, R. Canetti, H. Krawczyk, Keying hash functions for message authentication, *Lecture Notes in Computer Science* **1109** (1996), 1 15 (CRYPTO '96).
3. M. Bellare, R. Guérin, P. Rogaway, XOR MACs: New methods for message authentication, *Lecture Notes in Computer Science* **963** (1995), 15–28 (CRYPTO '95).
4. M. Bellare, J. Kilian, P. Rogaway, The security of cipher block chaining, *Lecture Notes in Computer Science* **839** (1994), 341–358 (CRYPTO '94).
5. J. Bierbrauer, T. Johansson, G. Kabatianskii, and B. Smeets, On families of hash functions via geometric codes and concatenation, *Lecture Notes in Computer Science*, **773** (1994), 331–342 (CRYPTO '93).
6. B. den Boer, A simple and key-economical unconditionally authentication scheme, *Journal of Computer Security*, **2** (1993), 65–71.
7. A. Bosselaers, R. Govaerts, J. Vandewalle, Fast hashing on the Pentium, *Lecture Notes in Computer Science* **1109** (1996), 298–313 (CRYPTO '96).
8. J.L. Carter, M.N. Wegman, Universal classes of hash functions, *J. Computer and System Sciences*, **18** (1979), 143–154.
9. H. Dobbertin, Cryptoanalysis of MD5 compress, presented at the rump session of EUROCRYPT'96.
10. E.N. Gilbert, F.J. MacWilliams, and N.J.A. Sloane, Codes which detect deception, *Bell Syst. Tech. J.*, **53** (1974), 405–424.
11. S. Halevi, H. Krawczyk, Software message authentication in the Gbit/second rates, *Proceedings of Fast Software Encryption Conference '97*, to appear.
12. T. Helleseth and T. Johansson, Universal hash functions from exponential sums over finite fields and Galois rings, *Lecture Notes in Computer Science* **1109** (1996), 31–44 (CRYPTO '96).
13. T. Johansson, A shift register construction of unconditionally secure authentication codes, *Designs, Codes and Cryptography*, **4** (1994), 69–81.
14. T. Johansson, G. Kabatianskii, B. Smeets, On the relation between A-codes and codes correcting independent errors, *Lecture Notes in Computer Science*, **765** (1994), 1–11 (EUROCRYPT'93).

15. G. Kabatianskii, B. Smeets, and T. Johansson, On the cardinality of systematic authentication codes via error correcting codes, *IEEE Trans. Inform. Theory*, **42** (1996), 566–578.
16. H. Krawczyk, LFSR-based hashing and authentication, *Lecture Notes in Computer Science*, **839** (1994), 129–139 (CRYPTO '94).
17. H. Krawczyk, New hash functions for message authentication, *Lecture Notes in Computer Science*, **921** (1995), 140-149 (EUROCRYPT '95).
18. B. Preneel, Cryptographic hash functions, *European Transactions on Telecommunications*, **5** (1994), 431–448.
19. B. Preneel, P. van Oorschot, MDx-MAC and building fast MACs from hash functions, *Lecture Notes in Computer Science*, **963** (1995), 1–14 (CRYPTO '95).
20. B. Preneel, P. van Oorschot, On the security of two MAC algorithms, *Lecture Notes in Computer Science*, **1070** (1996), 19–32 (EUROCRYPT '96).
21. R.L. Rivest, The MD5 message-digest algorithm, *Request for Comments 1321*, Internet Activities Board, Internet Privacy Task Force (1992).
22. P. Rogaway, Bucket hashing and its application to fast message authentication, *Lecture Notes in Computer Science*, **963** (1995), 29–42 (CRYPTO '95).
23. B. Schneier, *Applied Cryptography*, John Wiley & Sons (1996).
24. V. Shoup, On fast and provably secure message authentication based on universal hashing, *Lecture Notes in Computer Science*, **1109** (1996), 313–328 (CRYPTO '96).
25. G.J. Simmons, A game theory model of digital message authentication, *Congr. Numer.*, **34** (1992), 413–424.
26. G.J. Simmons, Authentication theory/coding theory, in *Lecture Notes in Computer Science*, **196** (1985), 411-431 (CRYPTO '84).
27. D.R. Stinson, Universal hashing and authentication codes, *Codes, Designs and Cryptography*, 4 (1994), 337–346.
28. M.N. Wegman and J.L. Carter, New hash functions and their use in authentication and set equality, *J. Computer and System Sciences*, **22** (1981), 265–279.