A New Multi-Linear Universal Hash Family

Palash Sarkar

Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108.
email: palash@isical.ac.in

Abstract. A new universal hash family is described. Messages are sequences over a finite field $\mathbb{F}_q$ while keys are sequences over an extension field $\mathbb{F}_{q^n}$. A linear map ψ from $\mathbb{F}_{q^n}$ to itself is used to compute the output digest. Of special interest is the case $q = 2$. For this case, we show that there is an efficient way to implement ψ using a tower field representation of $\mathbb{F}_{q^n}$. From a practical point of view, the focus of our constructions is small hardware and other resource constrained applications. For such platforms, our constructions compare favourably to previous work.

Keywords: universal hash function, tower field, message authentication code, resource constrained devices.

1 Introduction

Universal hash functions are useful in cryptography. These were introduced by Carter and Wegman [7] and have been extensively studied since then. Among their many applications, one of the most important is the construction of message authentication code (MAC) algorithms [26]. Previously, work on unconditionally secure authentication codes was done by Gilbert, MacWilliams and Sloane [9].

A well known construction[1] of a multi-linear universal hash function is the following. The message $\mathbf{M} = (M_1, \ldots, M_l)$ and key $\mathbf{K} = (K_1, \ldots, K_l)$ are sequences of elements over a finite field $\mathbb{F}$. The map

$$\mathsf{MLHash} : (M_1, \ldots, M_l) \xmapsto{\mathbf{K}} K_1M_1 + \cdots + K_lM_l \qquad (1)$$

is a multi-linear map and we will call this the multi-linear hash function. The probability (over random keys) that two distinct messages map to the same value is $1/|\mathbb{F}|$.

The basic idea of MLHash has been studied and extended later by Halevi and Krawczyk [10]. They describe an efficient software implementation of this construction over $\mathbb{F}_q$ with $q = 2^{32} + 15$. Modifications are made to the construction to align with 32-bit word boundaries and reduce the total number of modulo q operations.

The goal of obtaining universal hash functions have high-speed software implementations in general purpose computers has been pursued in several works leading to proposals such as UMAC [6], Poly1305 [2], PolyR [17], bucket hashing [21, 12], and [22]. Design of hash functions involving linear feedback shift registers (LFSRs) in some form has been investigated in [16, 11, 14]. In Section 7, we provide a short survey of some of these methods. The connection of universal hash functions to error-correcting codes was highlighted in [5]. Stinson [24] describes several methods of combining hash function constructions.

[1] This construction has been credited to Carter and Wegman [7] in [10]. Bernstein [4], mentions that this construction appears in an earlier work by Gilbert, MacWilliams and Sloane [9]. The description in [9] was in the language of finite geometries which according to Bernstein [4] seems to have deterred potential readers.

Our contributions. We generalize MLHash. The basic idea of the generalization is to work with two fields, a base field $\mathbb{F}_q$ and an extension $\mathbb{F}_{q^n}$, where n is a positive integer. The message is a sequence of elements from $\mathbb{F}_q$, whereas the key is a sequence of elements from $\mathbb{F}_{q^n}$. A linear operator ψ from $\mathbb{F}_{q^n}$ to itself is used. The contribution of $l \leq n$ message elements $M_1, \ldots, M_l$ under a key $K \in \mathbb{F}_{q^n}$ is the sum

$$M_1K + M_2\psi(K) + \cdots + M_l\psi^{l-1}(K).$$

The output is an element of $\mathbb{F}_{q^n}$.

This basic idea is developed into a definition of a hash function family LH. It is shown that MLHash can be seen as a special case of LH. For two equal length messages $\mathbf{M}$ and $\mathbf{M}'$ and any $\alpha \in \mathbb{F}_{q^n}$, we show the probability (over the uniform random choice of key $\mathbf{K}$) that $\mathsf{LH}_{\mathbf{K}}(\mathbf{M}') - \mathsf{LH}_{\mathbf{K}}(\mathbf{M})$ equals α is q^{-n}. Using multiple hashing with s independent keys brings this down to q^{-ns}. But, using a previously known technique, the so-called Toeplitz method, we show that the probability q^{-ns} can be achieved using only a few extra key elements.

The focus of this paper is to obtain designs which are suitable for implementation in resource constrained devices. For this goal, the choice of $q = 2$ is appropriate. In this case, we show that ψ can be instantiated very efficiently using a tower field representation of $\mathbb{F}_{q^n}$. Several different designs are discussed. The simplest design for 128-bit digests has the above probability to be 2^{-128}; can be implemented using only two 128-bit registers and requires only left shift and XOR operations. This design does not require the map ψ or any finite field arithmetic. As a result, this is ideal for environments which requires very small hardware solutions. For platforms which can support somewhat larger hardware, we propose solutions based on suitable instantiations of ψ. Issues of parallelism and bit slicing techniques are highlighted. The collection of designs presented here offer a wide variety of choices to a developer of message authentication code algorithms for resource constrained devices.

Section 7 is a rather long section which describes previous constructions of universal hash functions. In each case, we compare to the constructions in this work especially in the context of small-size hardware platforms. For such platforms, our conclusion is that the new constructions compare favourably to all previously known proposals. In particular, we show the impracticality of an LFSR based construction due to Johansson [11]. The LFSR in [11] can actually be done away with at the cost of increasing the key length by a few extra bits. It also turns out that the previous construction WH [15] is actually an inferior version of an earlier construction due to Winograd [27]. Winograd's construction is described in Appendix A. This is an issue of independent interest which is not directly related to the main focus of this work.

2 Preliminaries

2.1 Hash Function Definitions

Let $\mathcal{H} = \{H_k\}_{k \in \mathcal{K}}$ be a keyed family of functions, where for each $k \in \mathcal{K}$, $H_k : \mathcal{X} \to \mathcal{Y}$. Here $\mathcal{X}$ and $\mathcal{Y}$ are finite non-empty sets with $|\mathcal{X}| > |\mathcal{Y}|$. Let x and x' be distinct elements of $\mathcal{X}$. The collision probability (over uniform random choice of k from $\mathcal{K}$) of $\mathcal{H}$ associated with the elements x and x' is defined to be $\Pr_k[H_k(x) = H_k(x')]$. Further, if $\mathcal{Y}$ is a commutative (additively written) group, then for any fixed $\alpha \in \mathcal{Y}$, the differential probability (over uniform random choice of k from $\mathcal{K}$)

associated with (x, x', α) is defined to be $\Pr_k[H_k(x) - H_k(x') = \alpha]$. The terminology of collision probability and differential probability in the current context is from [3].

The family $\mathcal{H}$ is said to be ϵ-almost XOR universal (ϵ-AXU) if the differential probability for any (x, x', α) is bounded above by ϵ. The family $\mathcal{H}$ is said to be ϵ-almost universal (ϵ-AU) if the collision probability for any (x, x') is bounded above by ϵ. Clearly, if $\mathcal{H}$ is ϵ-AXU, then it is also ϵ-AU.

$\mathcal{H}$ is said to be universal if it is ϵ-almost universal with $\epsilon = 1/|\mathcal{Y}|$. All the function families that we define in this paper are universal. In fact, for each case, $\mathcal{Y}$ is a commutative group and we show that the differential probabilities are equal to $1/|\mathcal{Y}|$.

2.2 Some Basic Results

We need some elementary results on linear algebra over finite fields. The purpose of this section is to present the basic notation as well as the results that will be required.

Let q be a prime power and $\mathbb{F}_q$ be the finite field of q elements. For a positive integer n, the extension field $\mathbb{F}_{q^n}$ is a vector space over $\mathbb{F}_q$.

Let ψ be a linear transformation from $\mathbb{F}_{q^n}$ to itself. By ψ^i, we denote the usual iterate of ψ, i.e., $\psi^0 = \mathsf{id}$ and $\psi^i = \psi^{i-1} \circ \psi$, where id is the identity map from $\mathbb{F}_{q^n}$ to itself. Let $p(x) \in \mathbb{F}_q[x]$ be of the form $p(x) = a_m x^m + a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$, then by $p(\psi)$ we denote the linear operator $a_m\psi^m + a_{m-1}\psi^{m-1} + \cdots + a_1\psi + a_0\mathsf{id}$. We say that $p(x)$ annihilates ψ, if $p(\psi) = 0$, i.e., $p(\psi)$ maps all elements of $\mathbb{F}_{q^n}$ to zero. The minimum degree monic polynomial in $\mathbb{F}_q$ which annihilates ψ is said to be the minimal polynomial of ψ. The notion of minimal polynomial is relative to the base field. Suppose n_1 divides n. Then $\mathbb{F}_{q^{n_1}}$ is a subfield of $\mathbb{F}_{q^n}$ and the minimal polynomial of ψ over $\mathbb{F}_q$ is not the same as the minimal polynomial of ψ over $\mathbb{F}_{q^{n_1}}$.

If we fix a basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$, then a linear map ψ from $\mathbb{F}_{q^n}$ to itself is uniquely given by an $n \times n$ matrix A with entries from $\mathbb{F}_q$. The minimal polynomial $\tau(x)$ of ψ is also the minimal polynomial of A. Fixing a basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ allows the representation of the elements of $\mathbb{F}_{q^n}$ by n tuples over $\mathbb{F}_q$, so that, with respect to this basis, we can identify $\mathbb{F}_{q^n}$ with $\mathbb{F}_q^n$. For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_q^n$, $\langle \mathbf{a}, \mathbf{b} \rangle$ denotes the inner product, i.e., $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{n} a_i b_i$, where $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$.

We will require the following results about linear maps.

Lemma 1. *Let ψ be a linear map from $\mathbb{F}_{q^n}$ to itself such that its minimal polynomial $\tau(x)$ in $\mathbb{F}_q[x]$ is of degree n and is irreducible over $\mathbb{F}_q$. Let $p(x)$ be any polynomial in $\mathbb{F}_q[x]$. If $\tau(x)$ does not divide $p(x)$, then $p(\psi)$ is invertible. Equivalently, either $p(\psi) = 0$ or $p(\psi)$ is invertible.*

Proof: Fix a basis of $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$ and let ψ be given by a matrix A over $\mathbb{F}_q$. Then the minimal polynomial of A is $\tau(x)$ and since $\tau(x)$ is of degree n, it is also the characteristic polynomial of A. Let the characteristic roots of A (over $\mathbb{F}_{q^n}$) be $\zeta_1, \ldots, \zeta_n$. Since $\tau(x)$ is irreducible, none of the ζ_is are zero.

The matrix $p(A)$ represents the linear map $p(\psi)$ with respect to the previously fixed basis. The characteristic roots of $p(A)$ are $p(\zeta_1), \ldots, p(\zeta_n)$. If any of these values is 0, then from the irreducibility of $\tau(x)$, we get that $\tau(x)$ divides $p(x)$. Since $\tau(A) = 0$ (because $\tau(x)$ is the minimal polynomial of A), we have $p(A)$ also to be zero. On the other hand, if none of the $p(\zeta_1), \ldots, p(\zeta_n)$ are zero, the matrix $p(A)$ is non-singular and hence the transformation $p(\psi)$ is invertible. □

Lemma 2. *Let q be a prime power, $n = n_1 \times n_2$ and $q_1 = q^{n_1}$. Let $\psi : \mathbb{F}_{q_1^{n_2}} \to \mathbb{F}_{q_1^{n_2}}$ be a linear map whose minimal polynomial, $\mu(x)$, over $\mathbb{F}_{q_1}$ is irreducible and of degree n_2. Then the minimal polynomial of ψ over $\mathbb{F}_q$ is of degree n and is irreducible over $\mathbb{F}_q$.*

Proof: Fix a basis of $\mathbb{F}_{q_1^{n_2}}$ over $\mathbb{F}_{q_1}$. Then the linear map ψ is given by an $n_2 \times n_2$ matrix A with entries from $\mathbb{F}_{q_1}$ and whose characteristic polynomial $\mu(x)$ is irreducible over $\mathbb{F}_{q_1}$.

Following [18], let $I(q, n; x)$ be the product of all irreducible polynomials in x of degree n over $\mathbb{F}_q$. Using [18, Theorem 3.31], we have $I(q^{n_1}, n_2; x) = I(q, n_1n_2; x)$.

By definition, $\mu(x)$ divides $I(q^{n_1}, n_2; x)$. Also, by the Cayley-Hamilton theorem, $\mu(A) = 0$ and so $I(q, n_1n_2; A) = I(q^{n_1}, n_2; A) = 0$. Now $I(q, n_1n_2; x)$ is the product of all irreducible polynomials of degree $n = n_1n_2$ over $\mathbb{F}_q$. By Lemma 1, we have that for any irreducible polynomial $P(x)$ of degree n over $\mathbb{F}_q$, either $P(A) = 0$ or $P(A)$ is invertible. If for all irreducible factors $P(x)$ of $I(q, n; x)$, we have $P(A)$ to be invertible, then we clearly cannot have $I(q, n; A)$ to be zero. Therefore, there must be some irreducible factor $\tau(x)$ of $I(q, n; x)$ such that $\tau(A) = 0$. Since this $\tau(x)$ is a factor of $I(q, n; x)$, it is irreducible and of degree n. This $\tau(x)$ is then the minimal polynomial of A over $\mathbb{F}_q$, which completes our proof. $\square$

3 Basic Construction

In this section, we describe the basic idea.

Fix a field $\mathbb{F}_q$ and an integer $n \geq 1$. Consider the extension field $\mathbb{F}_{q^n}$. Let ψ be a linear map from $\mathbb{F}_{q^n}$ to itself such that the minimal polynomial $\tau(x)$ in $\mathbb{F}_q[x]$ of ψ is of degree n and is irreducible over $\mathbb{F}_q$. For each $K \in \mathbb{F}_{q^n}$, we define a function $G_K : \cup_{l=1}^{n} \mathbb{F}_q^l \to \mathbb{F}_{q^n}$ as follows. Let $\mathbf{a} = (a_1, \ldots, a_l)$, for some $l \in \{1, \ldots, n\}$. Then

$$\left.\begin{aligned} G_K(\mathbf{a}) &= \langle (a_1, \ldots, a_l), (K, \psi(K), \ldots, \psi^{l-1}(K)) \rangle \\ &= a_1K + a_2\psi(K) + \cdots + a_l\psi^{l-1}(K). \end{aligned}\right\} \tag{2}$$

In other words, $G_K(\mathbf{a})$ is the linear combination of $(a_1, \ldots, a_l)$ and $(K, \psi(K), \ldots, \psi^{l-1}(K))$. A term of $G_K()$ is of the form $a_i\psi^{i-1}(K)$, where a_i is an element of $\mathbb{F}_q$ and $\psi^{i-1}(K)$ is an element of $\mathbb{F}_{q^n}$. The efficiency of evaluating the term $a_i\psi^{i-1}(K)$ depends on the representation of $\mathbb{F}_{q^n}$ and the choice of the map ψ. We say more about this later.

Lemma 3. *Fix an l with $1 \leq l \leq n$. The following are true for the function defined in (2).*

1. *For a fixed K, the function G_K restricted to inputs with l components, is a multi-linear function, i.e., it is linear in every component of its input.*
2. *Fix a non-zero $\mathbf{a} \in \mathbb{F}_q^l$. If K is uniformly distributed over $\mathbb{F}_{q^n}$ then so is $G_K(\mathbf{a})$.*
3. *Consequently, for $\mathbf{a}, \mathbf{a}' \in \mathbb{F}_q^l$, $\mathbf{a} \neq \mathbf{a}'$ and for any $\alpha \in \mathbb{F}_{q^n}$, $\Pr_K[G_K(\mathbf{a}) - G_K(\mathbf{a}') = \alpha] = 1/q^n$.*

Proof: It is easy to see that G_K is linear for fixed K. We prove the second statement. Let $\mathbf{a} = (a_1, \ldots, a_l)$ where $a_i \in \mathbb{F}_q$ and define a polynomial $p(x) = a_1 + a_2x + \cdots + a_lx^{l-1}$. Since $\mathbf{a}$ is non-zero, $p(x)$ is also a non-zero polynomial over $\mathbb{F}_q$. Also, since $l \leq n$, the degree of $p(x)$ is less than n. Recall that the minimal polynomial $\tau(x)$ of ψ is irreducible over $\mathbb{F}_q$ and is of degree n. Thus, $p(x)$ is a non-zero polynomial which is coprime to $\tau(x)$. Using Lemma 1, we have $p(\psi)$ to be an invertible map from $\mathbb{F}_{q^n}$ to itself. Thus, if K is randomly distributed over $\mathbb{F}_{q^n}$, so is $p(\psi)(K)$. The second statement now follows on noting that $p(\psi)(K) = a_1 + a_2\psi(K) + \cdots + a_l\psi^{l-1}(K) = G_K(\mathbf{a})$.

By linearity of G_K, $G_K(\mathbf{a}) - G_K(\mathbf{a}') = G_K(\mathbf{a} - \mathbf{a}')$. So, the third statement follows directly from the second statement. $\square$

Key Length. For $G_K(\cdot)$, the key is K, which is an element of $\mathbb{F}_{q^n}$. One K can be used for messages consisting of upto n elements of $\mathbb{F}_q$. Thus, the key length is as long as the message. This is an inherent property of the multi-linear hash given by Equation (1) and is present in other extensions of the idea, such as [10, 6]. As mentioned in [6], in practice, the long key will be generated using a pseudorandom generator (PRG) from a short key.

3.1 Extending Message Length

In this and later sections, we use some terminology and notation from [6].

The function G can handle up to n elements of $\mathbb{F}_q$. It is the basic building block used for defining functions which can handle arbitrary length inputs. The idea is the following. Given m elements of $\mathbb{F}_q$, divide into (l_1+1) groups where each of the first l_1 groups has n elements and the last group has l_2 elements with $1 \le l_2 \le n$. To obtain the digest, apply G separately (with random and independent keys) to each of the groups and then add together all the individual digests.

More formally, we define a function family which is parametrized as $\mathsf{LH}[q,n,m]$. The domain is $\cup_{i=1}^{m}\mathbb{F}_q^i$, i.e., each element of the domain consists of upto m elements of $\mathbb{F}_q$; the range is $\mathbb{F}_{q^n}$. Let $t = \lceil m/n \rceil$. Each function in $\mathsf{LH}[q,n,m]$ is named by an element $\mathbf{K}$ of $\mathbb{F}_{q^n}^t$; a random function in $\mathsf{LH}[q,n,m]$ is given by a random element of $\mathbb{F}_{q^n}^t$. We write the function indicated by $\mathbf{K}$ as $\mathsf{LH}_{\mathbf{K}}(\cdot)$.

The message $\mathbf{M}$ consists of an l-tuple, $1 \le l \le m$, over $\mathbb{F}_q$. Let $l = l_1 n + l_2$, with $l_1 \ge 0$ and $1 \le l_2 \le n$. We consider $\mathbf{M}$ to be of the form $(\mathbf{M}_1, \ldots, \mathbf{M}_{l_1}, \mathbf{M}_{l_1+1})$, where $\mathbf{M}_1, \ldots, \mathbf{M}_{l_1}$ are in $\mathbb{F}_q^n$ and $\mathbf{M}_{l_1+1}$ is in $\mathbb{F}_q^{l_2}$. Further, let $\mathbf{K} = (K_1, \ldots, K_t)$ and note that $t \ge l_1 + 1$. Then, $\mathsf{LH}_{\mathbf{K}}(\mathbf{M})$ is defined as

$$\mathsf{LH}_{\mathbf{K}}(\mathbf{M}) = G_{K_1}(\mathbf{M}_1) + G_{K_2}(\mathbf{M}_2) + \cdots + G_{K_{l_1}}(\mathbf{M}_{l_1}) + G_{K_{l_1+1}}(\mathbf{M}_{l_1+1}). \tag{3}$$

Formally, we should write $\mathsf{LH}_{\mathbf{K}}[q,n,m](\mathbf{M})$ instead of $\mathsf{LH}_{\mathbf{K}}(\mathbf{M})$, but, we prefer the second notation as it is simpler. The parameters q, n and m will be clear from the context.

Theorem 1. *For any prime power q, any positive integers n and m, the differential probabilities of $\mathsf{LH}[q,n,m]$ for equal length strings are equal to q^{-n}.*

Proof: As in Lemma 3, it is sufficient to show that for any non-zero $\mathbf{M} \in \mathbb{F}_q^l$, $1 \le l \le m$; for any $\alpha \in \mathbb{F}_{q^n}$ and uniform random $\mathbf{K}$, $\Pr[\mathsf{LH}_{\mathbf{K}}(\mathbf{M}) = \alpha] = 1/q^n$.

Let $\mathbf{M}_{i_1}, \ldots, \mathbf{M}_{i_k}$ be the non-zero blocks of $\mathbf{M}$, where $i_1, \ldots, i_k$ are integers from the set $\{1, \ldots, l_1+1\}$. Since $\mathbf{M}$ is non-zero, $k \ge 1$.

$$\begin{aligned}
\mathsf{Pr}\left[\mathsf{LH}_{\mathbf{K}}(\mathbf{M}) = \alpha\right] &= \mathsf{Pr}\left[G_{K_{i_1}}(\mathbf{M}_{i_1}) + \cdots + G_{K_{i_k}}(\mathbf{M}_{i_k}) = \alpha\right] \\
&= \sum_{\gamma_2,\ldots,\gamma_k} \mathsf{Pr}\left[G_{K_{i_1}}(\mathbf{M}_{i_1}) + \cdots + G_{K_{i_k}}(\mathbf{M}_{i_k}) = \alpha | K_{i_2} = \gamma_2, \ldots, K_{i_k} = \gamma_k\right] \\
&\quad \times \mathsf{Pr}\left[K_{i_2} = \gamma_2, \ldots, K_{i_k} = \gamma_k\right] \\
&= \sum_{\gamma_2,\ldots,\gamma_k} \mathsf{Pr}\left[G_{K_{i_1}}(\mathbf{M}_{i_1}) + G_{\gamma_2}(\mathbf{M}_{i_2}) + \cdots + G_{\gamma_k}(\mathbf{M}_{i_k}) = \alpha\right] \\
&\quad \times \mathsf{Pr}\left[K_{i_2} = \gamma_2, \ldots, K_{i_k} = \gamma_k\right] \\
&= \sum_{\gamma_2,\ldots,\gamma_k} \mathsf{Pr}\left[G_{K_{i_1}}(\mathbf{M}_{i_1}) = \beta\right] \times \mathsf{Pr}\left[K_{i_2} = \gamma_2, \ldots, K_{i_k} = \gamma_k\right]
\end{aligned}$$

$$= \frac{1}{q^n} \sum_{\gamma_2,\ldots,\gamma_k} \mathsf{Pr}\left[K_{i_2} = \gamma_2, \ldots, K_{i_k} = \gamma_k\right]$$
$$= \frac{1}{q^n}.$$

The sum is over all possible choices of $(\gamma_2, \ldots, \gamma_k)$ and $\beta = \alpha - (G_{\gamma_2}(\mathbf{M}_{i_2}) + \cdots + G_{\gamma_k}(\mathbf{M}_{i_k}))$. The fact that $\mathsf{Pr}\left[G_{K_{i_1}}(\mathbf{M}_{i_1}) = \beta\right] = 1/q^n$ follows from Lemma 3. $\square$

The previous multi-linear construction. We show that LH is a generalization of MLHash given by (1). Let $n = 1$, then $\mathbb{F}_{q^n} = \mathbb{F}_q$. Consider the function G. The domain, range and the key space of G becomes $\mathbb{F}_q$ and the definition of G is now $G_K(M) = KM$. Now consider the function $\mathsf{LH}[q,1,m]$. The domain is $A = \cup_{i=1}^{m} \mathbb{F}_q$ and the range is $\mathbb{F}_{q^n} = \mathbb{F}_q$. Also $t = \lceil m/n \rceil = m$ and each key $\mathbf{K} = (K_1, \ldots, K_m) \in \mathbb{F}_q^m$. Then for $\mathbf{M} = (M_1, \ldots, M_l)$, $\mathsf{LH}_{\mathbf{K}}(\mathbf{M}) = K_1M_1 + \cdots + K_lM_l$. Thus, $\mathsf{LH}[q,1,m]$ is the multi-linear hash function MLHash in (1).

In the above, we set $n = 1$ and view the evaluation of $G_K(M)$ as the product KM over the field $\mathbb{F}_q$. An alternative way to see that MLHash is a special case of LH is to use a specific instantiation of the operator ψ. Let $n > 1$ so that $\mathbb{F}_{q^n}$ is a non-trivial extension of $\mathbb{F}_q$ which is represented using an irreducible polynomial $\rho(\alpha)$ of degree n over $\mathbb{F}_q$. Then elements of $\mathbb{F}_{q^n}$ can be seen as polynomials of degree at most $(n-1)$ over $\mathbb{F}_q$.

Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ be represented in $\mathbb{F}_{q^n}$ by the polynomial $a(\alpha) = a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}$ and let $K(\alpha) = k_0 + k_1\alpha + \cdots + k_{n-1}\alpha^{n-1}$ be another polynomial. Then

$$\begin{aligned}
&\left(a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}\right) \times \left(k_0 + k_1\alpha + \cdots + k_{n-1}\alpha^{n-1}\right)\\
&= a_0\left(k_0 + k_1\alpha + \cdots + k_{n-1}\alpha^{n-1}\right) + a_1\alpha\left(k_0 + k_1\alpha + \cdots + k_{n-1}\alpha^{n-1}\right)\\
&\cdot\ \cdots + a_{n-1}\alpha^{n-1}\left(k_0 + k_1\alpha + \cdots + k_{n-1}\alpha^{n-1}\right)\\
&= \left\langle (a_0, \ldots, a_{n-1}), (K(\alpha), \alpha K(\alpha), \ldots, \alpha^{n-1}K(\alpha)) \right\rangle.
\end{aligned}$$

Here the multiplications by α are done modulo $\rho(\alpha)$. Define $\psi : \mathbb{F}_{q^n} \to \mathbb{F}_{q^n}$ as

$$\psi : K(\alpha) \mapsto \alpha K(\alpha) \bmod \rho(\alpha).$$

Then it is easy to show that ψ is a linear map whose minimal polynomial is $\rho(\alpha)$. If ψ is instantiated as this "multiply by α" map, then for $K \in \mathbb{F}_{q^n}$

$$\begin{aligned}
G_K(\mathbf{a}) &= \langle \mathbf{a}, (K, \psi(K), \ldots, \psi^{n-1}(K)) \rangle\\
&= \langle \mathbf{a}, (K(\alpha), \alpha K(\alpha), \ldots, \alpha^{n-1}K(\alpha)) \rangle\\
&= a(\alpha)K(\alpha) \bmod \rho(\alpha).
\end{aligned}$$

In this case, the evaluation of $G_K(\mathbf{a})$ corresponds to a multiplication in the extension field $\mathbb{F}_{q^n}$. Again, the resulting LH is the same as MLHash given by (1), where the finite field is $\mathbb{F}_{q^n}$.

In the above, the "multiply by α" is a specific instantiation of ψ. We later discuss other possible instantiations where $\mathbb{F}_{q^n}$ is seen as a tower field over $\mathbb{F}_q$.

4 The Sliding Window (Toeplitz) Construction

The differential (and hence collision) probabilities of LH are equal to q^{-n}. By suitably choosing q and n, this can be made as low as one desires. On the other hand, it is also possible to introduce additional flexibility based on repeated hashing.

The idea is the following. Repeatedly hash the same message with independent keys and concatenate the output. If the message is hashed s times, then the collision probability becomes q^{-ns}. But, this approach requires s independent keys. The modification is to generate s keys using a sliding window technique. Suppose each hash call requires a key sequence of length t. We start with a sequence of length $t+s-1$ and slide (one element at a time) a window of length t over this sequence to obtain s different keys each of length t. This is called the Toeplitz procedure as the process can be visualized as a Toeplitz matrix and has been earlier used in [6, 15].

We define the hash function family $\mathsf{LH}^{\mathsf{T}}[q,n,m,s]$, where q, n and m are as in Section 3.1 and s is a positive integer. The value $t = \lceil m/n \rceil$, the domain A and the details of message parsing are the same as in Section 3.1. The range B is now an element of $\mathbb{F}_{q^n}^s$. A function in $\mathsf{LH}^{\mathsf{T}}[q,n,m,s]$ is named by a key $\mathbf{K} = (K_1, \ldots, K_{t+s-1}) \in \mathbb{F}_{q^n}^{t+s-1}$. For $1 \le i \le s$, let $\mathbf{K}_i = (K_i, K_{i+1}, \ldots, K_{t+i-1})$. For any $M \in A$, we define $\mathsf{LH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M})$ as

$$\mathsf{LH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M}) = (\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}), \ldots, \mathsf{LH}_{\mathbf{K}_s}(\mathbf{M})). \tag{4}$$

Theorem 2. *Let q be a prime power and n, m and s be positive integers. Then the differential probabilities of $\mathsf{LH}^{\mathsf{T}}[q,n,m,s]$ for equal length strings are equal to q^{-ns}.*

Proof: As before, since LH^{T} is linear, it is sufficient to show that for any non-zero $\mathbf{M} \in A$ and for any fixed $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_s)$ in $\mathbb{F}_{q^n}^s$, $\Pr_{\mathbf{K}}[\mathsf{LH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M}) = \boldsymbol{\alpha}] = q^{-ns}$.

Parse $\mathbf{M}$ as $\mathbf{M}_1, \ldots, \mathbf{M}_{l_1}, \mathbf{M}_{l_1+1}$ as in Section 3.1. Since $\mathbf{M}$ is non-zero, at least one of the $\mathbf{M}_i$'s will be non-zero. Let r be the maximum integer such that $\mathbf{M}_r$ is non-zero. The condition $\mathsf{LH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M}) = \boldsymbol{\alpha}$ is equivalent to

$$\left.\begin{array}{llll}
\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = & G_{K_1}(\mathbf{M}_1) + \cdots + G_{K_{r-1}}(\mathbf{M}_{r-1}) + G_{K_r}(\mathbf{M}_r) & = \alpha_1 \\
\mathsf{LH}_{\mathbf{K}_2}(\mathbf{M}) = & G_{K_2}(\mathbf{M}_1) + \cdots + G_{K_r}(\mathbf{M}_{r-1}) + G_{K_{r+1}}(\mathbf{M}_r) & = \alpha_2 \\
\vdots \quad \vdots & \qquad \vdots & \vdots \ \vdots \\
\mathsf{LH}_{\mathbf{K}_s}(\mathbf{M}) = & G_{K_s}(\mathbf{M}_1) + \cdots + G_{K_{r+s-2}}(\mathbf{M}_{r-1}) + G_{K_{r+s-1}}(\mathbf{M}_r) & = \alpha_s
\end{array}\right\} \tag{5}$$

We consider the probability

$$\begin{aligned}
&\Pr\left[\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = \alpha_1, \ldots, \mathsf{LH}_{\mathbf{K}_s}(\mathbf{M}) = \alpha_s\right] \\
&= \Pr\left[\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = \alpha_1\right] \times \Pr\left[\mathsf{LH}_{\mathbf{K}_2}(\mathbf{M}) = \alpha_2 | \mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = \alpha_1\right] \\
&\quad \cdots \\
&\quad \times \Pr\left[\mathsf{LH}_{\mathbf{K}_s}(\mathbf{M}) = \alpha_s | \mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = \alpha_1, \ldots, \mathsf{LH}_{\mathbf{K}_{s-1}}(\mathbf{M}) = \alpha_{s-1}\right].
\end{aligned}$$

Using Theorem 1, $\Pr\left[\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = \alpha_1\right] = q^{-n}$. By Lemma 3 and the fact that $\mathbf{M}_r$ is non-zero, each of $G_{K_r}(\mathbf{M}_r), G_{K_{r+1}}(\mathbf{M}_r), \ldots, G_{K_{r+s-1}}(\mathbf{M}_r)$ is uniformly distributed over $\mathbb{F}_{q^n}$. Since $K_r, K_{r+1}, \ldots, K_{r+s-1}$ are independent, the random variables $G_{K_r}(\mathbf{M}_r), G_{K_{r+1}}(\mathbf{M}_r), \ldots, G_{K_{r+s-1}}(\mathbf{M}_r)$ are also independent. From this it follows that each of the conditional probabilities in the above expression is also equal to q^{-n}. This proves the result. $\square$

4.1 Toeplitz version of MLHash

By setting $n = 1$, the description of the Toeplitz method turns out to be a Toeplitz version of the multi-linear hash function MLHash given by (1).

Messages are as before sequences of lengths at most m over $\mathbb{F}_q$. For $n = 1$, the key elements K_i's are from the field $\mathbb{F}_{q^n} = \mathbb{F}_q$, i.e., the key is a sequence of length $(m+s-1)$ over $\mathbb{F}_q$. For a message $\mathbf{M} = (M_1, \ldots, M_l)$ with $l \leq m$ a key $\mathbf{K} = (K_1, \ldots, K_{l+s-1})$ is used. Set $\mathbf{K}_i = (K_i, \ldots, K_{l+i-1})$. Then $\mathsf{LH}_{\mathbf{K}_i}(\mathbf{M})$ is computed as follows.

$$\left.\begin{array}{ll} \mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) = M_1K_1 + M_2K_2 + \cdots + M_lK_l \\ \mathsf{LH}_{\mathbf{K}_2}(\mathbf{M}) = M_1K_2 + M_2K_3 + \cdots + M_lK_{l+1} \\ \cdots \quad\quad\quad \cdot \cdots \\ \mathsf{LH}_{\mathbf{K}_s}(\mathbf{M}) = M_1K_s + M_2K_{s+1} + \cdots + M_lK_{s+l-1}. \end{array}\right\} \tag{6}$$

Note that this computation does not require the map ψ at all. From Theorem 2, the differential probabilities for this construction equals $q^{-ns} = q^{-s}$, since $n = 1$. One has to choose q and s suitably to obtain a desired probability.

Case $q = 2$. In this case, the product M_iK_j is simply an AND of two bits. The computation in (6) can be completed using the following simple algorithm which uses just two s-bit registers $\mathbf{T}$ and $\mathbf{R}$.

```
$\mathbf{T} = (K_1, \ldots, K_s)$;
if $M_1 = 0$, then $\mathbf{R} = (0, \ldots, 0)$; else $\mathbf{R} = \mathbf{T}$;
for $i = 2$ to $l$
    $\mathbf{T} = (\mathbf{T} \ll 1) \oplus (0, \ldots, 0, K_{s+i-1})$;
    if $(M_i = 1)$ then $\mathbf{R} = \mathbf{R} \oplus \mathbf{T}$;
end for;
return $\mathbf{R}$.
```

An implementation of the above algorithm can be done using very small hardware. Apart from the two s-bit registers, the only other operations are left shift and bitwise XOR of the two registers. A typical value of s is 128 giving rise to a differential probability of 2^{-128}. This is achieved using two 128-bit registers and 128 XOR gates. To the best of our knowledge such a small size design of universal hash function has not been done earlier.

5 Representations of $\mathbb{F}_{q^n}$ and the Linear Transformation ψ

Let $\mathbb{F}_{q^n}$ be represented using a polynomial $\tau(x)$ which is irreducible over $\mathbb{F}_q$. Then an element of $\mathbb{F}_{q^n}$ is given by an n-tuple of elements over $\mathbb{F}_q$. Let $\tau(x) = x^n - t_{n-1}x^{n-1} - \cdots - t_1x - t_0$ with $t_{n-1}, \ldots, t_0 \in \mathbb{F}_q$. Given an element $(x_0, \ldots, x_{n-1})$ of $\mathbb{F}_{q^n}$, we define $(y_0, \ldots, y_{n-1}) = \psi(x_0, \ldots, x_{n-1})$ in the following manner.

$$\begin{array}{ll} y_0 = t_{n-1}x_0 + t_{n-2}x_1 + \cdots + t_0x_{n-1}; & \text{and} \\ y_i = x_{i-1} & \text{for } 1 \leq i \leq n-1. \end{array} \tag{7}$$

This provides an easy way to implement the map ψ. This is essentially a linear feedback shift register (LFSR). See [18] for a general discussion on LFSRs. The "multiply-by-α" mentioned earlier can also be seen as an LFSR map.

Tower field representation of ψ for the case $q = 2$. For the rest of the paper we will fix $q = 2$. (The case of general q can be built along the lines of $q = 2$.) If $n = 128$, then one can use an LFSR of length n to obtain a ψ having minimal polynomial $x^{128} + x^{107} + x^{64} + x^{13} + 1$. But, it is possible to obtain other options for ψ by using a tower field representation of $\mathbb{F}_{2^n}$. We discuss this below.

Suppose $n = n_1 n_2$. Fix an irreducible polynomial $\rho(x)$ of degree n_1 over $\mathbb{F}_2$. This gives rise to a representation of $\mathbb{F}_{2^{n_1}}$ as $\mathbb{F}_2[\alpha]/(\rho(\alpha))$. Choose a monic irreducible polynomial $\mu(x)$ of degree n_2 over $\mathbb{F}_2[\alpha]/(\rho(\alpha))$. This gives a representation of $\mathbb{F}_{2^n}$ as a two-part extension ($\mathbb{F}_2 \to \mathbb{F}_{2^{n_1}} \to \mathbb{F}_{2^n}$). The pair of polynomials $(\rho(\alpha), \mu(x))$ defines the particular representation.

Let $\mu(x) = x^{n_2} - c_{n_2-1}x^{n_2-1} - \cdots - c_1 x - c_0$ where c_is are elements of $\mathbb{F}_2[\alpha]/(\rho(\alpha))$. Using $\mu(x)$, we define the linear map ψ in the following manner. Given $(x_0, \ldots, x_{n_2-1}) \in \mathbb{F}_{2^{n_1}}^{n_2}$, let $(y_0, \ldots, y_{n_2-1}) = \psi(x_0, \ldots, x_{n_2-1})$ where

$$\begin{array}{lll} y_0 &= c_{n_2-1}x_0 + c_{n_2-2}x_1 + \cdots + c_0 x_{n_2-1}; & \text{and} \\ y_i &= x_{i-1} & \text{for } 1 \leq i \leq n_2 - 1. \end{array} \tag{8}$$

Evaluating y_0 in general requires n_2 multiplications over $\mathbb{F}_{2^{n_1}}$ and is not a better option than working directly over $\mathbb{F}_2$. On the other hand, if most of the c_is are either 0 or 1, then evaluating y_0 becomes much more efficient. In the best case, we will have all but one of the c_is to be 0 or 1. Table 1 provides examples of tower field representations of $\mathbb{F}_{2^n}$. In all cases, only the constant term of $\mu(x)$ is equal to α and hence (8) can be evaluated by a single multiplication by α modulo the corresponding $\rho(\alpha)$.

Table 1. Examples of $\mathbb{F}_{2^n}$ represented as a tower field. In each case, $\mu(x)$ is a primitive polynomial. The differential (and collision) probabilities are 2^{-n}.

n_1	n_2	$n = n_1 \times n_2$	$\rho(\alpha)$	$\mu(x)$
32	2	64	$\alpha^{32} + \alpha^{31} + \alpha^{29} + \alpha^1 + 1$	$x^2 + x + \alpha$
16	5	80	$\alpha^{16} + \alpha^5 + \alpha^3 + \alpha^2 + 1$	$x^5 + x^3 + \alpha$
96	3	96	$\alpha^{32} + \alpha^{18} + \alpha^9 + \alpha^2 + 1$	$x^3 + x + \alpha$
32	4	128	$\alpha^{32} + \alpha^{18} + \alpha^6 + \alpha^5 + 1$	$x^4 + x^3 + x + \alpha$
16	8	128	$\alpha^{16} + \alpha^{10} + \alpha^9 + \alpha^6 + 1$	$x^8 + x^3 + x + \alpha$
8	16	128	$\alpha^8 + \alpha^7 + \alpha^3 + \alpha^2 + 1$	$x^{16} + x^7 + x + \alpha$

Now we have a linear map ψ from $\mathbb{F}_{2^n}$ to $\mathbb{F}_{2^n}$ (defined via the tower field representation of $\mathbb{F}_{2^n}$). The message $\mathbf{a}$ to be hashed is a bit string. Recall that the definition of $G_K(\mathbf{a})$ assumed that the minimal polynomial $\tau(x)$ of the linear map ψ is in $\mathbb{F}_2[x]$ and is irreducible over $\mathbb{F}_2$. Here, we have defined ψ to be a map from $\mathbb{F}_{(2^{n_1})^{n_2}}$ (where $\mathbb{F}_{2^{n_1}}$ is represented by $\rho(x)$) to itself, whose minimal polynomial $\mu(x)$ is in $\mathbb{F}_{2^{n_1}}[x]$ and is irreducible over $\mathbb{F}_{2^{n_1}}$. Lemma 2 assures us that the minimal polynomial of ψ *over* $\mathbb{F}_2$ is an irreducible polynomial of degree n as required.

5.1 Bit-Slice Computation of LH

A combination of message decimation and a bit slicing technique can be used to exploit the inherent parallelism in the construction. To understand this, consider the definition of

$$\mathsf{LH}_{\mathbf{K}}(\mathbf{M}) = G_{K_1}(\mathbf{M}_1) + \cdots + G_{K_{l_1}}(\mathbf{M}_{l_1}) + G_{K_{l_1+1}}(\mathbf{M}_{l_1+1})$$

(as given by (3)), where

$$
\begin{aligned}
G_{K_1}(\mathbf{M}_1) &= M_{1,1}K_1 + M_{1,2}\psi(K_1) + M_{1,3}\psi^2(K_1) + \cdots + M_{1,n}\psi^{n-1}(K_1)\\
G_{K_2}(\mathbf{M}_2) &= M_{2,1}K_2 + M_{2,2}\psi(K_2) + M_{2,3}\psi^2(K_2) + \cdots + M_{2,n}\psi^{n-1}(K_2)\\
G_{K_3}(\mathbf{M}_3) &= M_{3,1}K_3 + M_{3,2}\psi(K_3) + M_{3,3}\psi^2(K_3) + \cdots + M_{3,n}\psi^{n-1}(K_3)\\
\cdots & \cdot \cdots
\end{aligned}
$$

From this we clearly see that the rows can be computed independent of each other. This suggests that we can perform the row computations in parallel. The map ψ takes an element K of $\mathbb{F}_{2^n}$ to $\psi(K)$. Extend this to $(K_1, \ldots, K_w)$ by defining $\psi(K_1, \ldots, K_w)$ to be $(\psi(K_1), \ldots, \psi(K_w))$. The crucial observation is that w rows of the above computation can be processed in parallel by *simultaneous* application of ψ to w rows in the following manner.

$$
\begin{pmatrix} K_1 \\ K_2 \\ \vdots \\ K_w \end{pmatrix} \xrightarrow{\psi} \begin{pmatrix} \psi(K_1) \\ \psi(K_2) \\ \vdots \\ \psi(K_w) \end{pmatrix} \xrightarrow{\psi} \begin{pmatrix} \psi^2(K_1) \\ \psi^2(K_2) \\ \vdots \\ \psi^2(K_w) \end{pmatrix} \xrightarrow{\psi} \cdots \xrightarrow{\psi} \begin{pmatrix} \psi^{n-1}(K_1) \\ \psi^{n-1}(K_2) \\ \vdots \\ \psi^{n-1}(K_w) \end{pmatrix}
$$

Each application of ψ works in parallel on the w key blocks. In particular, the i-th bits of $K_1, \ldots, K_w$ are processed in exactly the same manner, which gives rise to a regular bit-slice architecture.

In this architecture, nw bits are processed using $(n-1)$ parallel applications of ψ and n inner product computations. As w increases, the speed of the computation increases, but, so does the size of the hardware. Based on the available resources, a suitable value of w is to be chosen.

Consider for example $n = 128$ and let ψ be instantiated using a tower field representation where $n = n_1 \times n_2$ with $n_1 = 32$ and $n_2 = 4$. A suitable choice of $\rho(\alpha)$ and $\mu(x)$ for these values of n_1 and n_2 is given in Table 1. For this choice, each key block consists of 4 32-bit words. The w key blocks $K_1, \ldots, K_w$ is given by $4w$ 32-bit words. The actual hardware implementation of ψ operating column-wise on the w key blocks require very simple shift and XOR operations.

Choosing $w = 4$ requires 16 32-bit registers which would be a good choice if the requirement is to implement on resource constrained devices. On the other hand, if sufficient hardware area is available, then one can even choose $w = 128$ which will give rise to a very fast implementation.

5.2 Instances of the Toeplitz Method and Parallel Implementation

Consider the definition of LH^{T} as given by (4).

$$
\mathsf{LH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M}) = (\mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}), \mathsf{LH}_{\mathbf{K}_2}(\mathbf{M}), \ldots, \mathsf{LH}_{\mathbf{K}_s}(\mathbf{M})).
$$

Here $\mathbf{K} = (K_1, \ldots, K_{t+s-1})$,

$$
\begin{aligned}
\mathbf{K}_1 &= (K_1, K_2, \ldots, K_{t-1})\\
\mathbf{K}_2 &= (K_2, K_3, \ldots, K_t)\\
\cdots & \cdot \cdots\\
\mathbf{K}_s &= (K_s, K_{s+1}, \ldots, K_{t+s-1}).
\end{aligned}
$$

The message consists of l elements of $\mathbb{F}_q$ and is written as $\mathbf{M} = (\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_{l_1}, \mathbf{M}_{l_1+1})$ where $\mathbf{M}_j = (M_{j,1}, \ldots, M_{j,n})$ for $1 \leq j \leq l_1$; $\mathbf{M}_{l_1+1} = (M_{l_1+1,1}, \ldots, M_{l_1+1,l_2})$ and $M_{j,k} \in \mathbb{F}_q$. The definition of the numbers, s, t, n, l_1, l_2 are as in Section 3.1.

Let $\mathbf{V}_i = (K_i, \psi(K_i), \ldots, \psi^{n-1}(K_i))$ for $i \geq 1$. We use the notation $\mathbf{V}_{i|l_2}$ to denote the first l_2 elements of $\mathbf{V}_i$. Then

$$\begin{aligned} G_{K_i}(\mathbf{M}_j) &= \left\langle (M_{j,1}, \ldots, M_{j,n}), (K_i, \psi(K_i), \ldots, \psi^{n-1}(K_i)) \right\rangle \\ &= \langle \mathbf{M}_j, \mathbf{V}_i \rangle . \end{aligned}$$

Using this notation, we can write

$$\left.\begin{aligned} \mathsf{LH}_{\mathbf{K}_1}(\mathbf{M}) &= \langle \mathbf{M}_1, \mathbf{V}_1 \rangle + \langle \mathbf{M}_2, \mathbf{V}_2 \rangle + \cdots + \langle \mathbf{M}_{l_1}, \mathbf{V}_{l_1} \rangle + \left\langle \mathbf{M}_{l_1+1}, \mathbf{V}_{l_1+1|l_2} \right\rangle \\ \mathsf{LH}_{\mathbf{K}_2}(\mathbf{M}) &= \langle \mathbf{M}_1, \mathbf{V}_2 \rangle + \langle \mathbf{M}_2, \mathbf{V}_3 \rangle + \cdots + \langle \mathbf{M}_{l_1}, \mathbf{V}_{l_1+1} \rangle + \left\langle \mathbf{M}_{l_1+1}, \mathbf{V}_{l_1+2|l_2} \right\rangle \\ \cdots \quad & \quad \cdot \cdots \\ \mathsf{LH}_{\mathbf{K}_s}(\mathbf{M}) &= \langle \mathbf{M}_1, \mathbf{V}_s \rangle + \langle \mathbf{M}_2, \mathbf{V}_{s+1} \rangle + \cdots + \langle \mathbf{M}_{l_1}, \mathbf{V}_{l_1+s-1} \rangle + \left\langle \mathbf{M}_{l_1+1}, \mathbf{V}_{l_1+s|l_2} \right\rangle . \end{aligned}\right\} \qquad (9)$$

There are several things to note regarding (9).

1. For each column, the message block remains the same.
2. The $\mathbf{V}$'s for the next column is obtained by shifting the previous column along with the computation of one extra $\mathbf{V}_k$.
3. The $\mathbf{V}$'s are independent of each other and can be performed in parallel.

A simple algorithm to perform the computation is as follows. The algorithm uses $R_1, \ldots, R_s$ to store the intermediate results, where the R_i's are elements of $\mathbb{F}_q$. Also, an array $\mathbf{W}$ of length s is used, where $\mathbf{W}[j]$ stores one of the $\mathbf{V}$'s which is an element of $\mathbb{F}_{q^n}^n$. For the simplicity of description of the algorithm, we assume that $l_2 = n$. It is easy to modify the algorithm to handle the case where $l_2 < n$.

```
$\mathbf{W} = (\mathbf{V}_1, \ldots, \mathbf{V}_s)$;
for $j = 1, \ldots, s$, $R_j = \langle \mathbf{M}_1, \mathbf{W}[j] \rangle$;
for $i = s + 1$ to $l_1 + s$ (here $l$ is the length of the message)
    compute $\mathbf{V}_i$;
    shift $\mathbf{W}$ left once;
    $\mathbf{W}[s] = \mathbf{V}_i$;
    for $j = 1, \ldots, s$, $R_j = R_j + \langle \mathbf{M}_i, \mathbf{W}[j] \rangle$;
end for;
return $(R_1, \ldots, R_s)$.
```

Computing $\mathbf{V}_i$ requires $(n-1)$ applications of ψ so that the total number of applications of ψ is equal to $(n-1)(l_1 + s) = l - l_1 - l_2 + ns$. (We have assumed that $l_2 = n$ but, this is true even if $l_2 < n$.) Each application of ψ is on an n-bit string.

The differential probability for $\mathsf{LH}^{\mathsf{T}}[q, n, m, s]$ is q^{-ns}. For $q = 2$, this is 2^{-ns}. By suitably choosing n and s, we can obtain configurations for different environments. For example, for obtaining the differential probability to be 2^{-128} we have to set $ns = 128$. So $(64, 2)$, $(32, 4)$, $(16, 8)$, $(8, 16)$ are some possible choices of (n, s). For a particular choice of n, the map ψ can be instantiated by choosing a suitable irreducible polynomial of degree n over $\mathbb{F}_2$. Alternatively, one can also use a tower field representation to instantiate ψ. If $n = 64$, then one can use the first row of Table 1.

The computation of the expressions in (9) can be divided into two parts: computation of the $\mathbf{V}$'s and computation of the inner products. At a broad level, this can be considered to be a two-stage pipeline. As mentioned earlier, the $\mathbf{V}$'s are independent of each other and can be computed in parallel. Suppose that w of the $\mathbf{V}$'s are computed in parallel. Then the number of parallel applications of ψ becomes $(l-l_1-l_2+ns)/w$, i.e., the computation becomes almost w times faster. The computed $\mathbf{V}$'s are forwarded to the second stage, where the inner products are computed. The inner products are also independent of each other and can be computed in parallel. This can be seen as follows.

$$\boxed{\text{Compute } \mathbf{V}\text{'s}} \longrightarrow \boxed{\text{Compute inner products}}$$

Suppose that ω of the inner products are computed in parallel. The parameters w and ω determine the cost of the hardware and the speed of the computation. To determine proper choices of these two values, the detailed design of the architecture is required.

6 Handling Variable Length Inputs

Theorem 1 and 2 ensure low differential probabilities only for equal length inputs. For designing a MAC algorithm, we need to handle variable length messages. We describe how this can be done for $q = 2$. For other values of q, it is possible to suitably modify the proposal.

For $q = 2$, the message is actually a bit string. So, there is no problem of handling partial blocks. The basic technique for handling variable length messages is to append the binary representation of the length to the message. In our case, a simple modification of this idea will ensure low differential probability for variable length messages. But, such a simple padding rule may result in messages which do not align properly to byte or word boundaries. So, we modify the construction so that the padded length is a multiple of 32. (This can easily be modified to obtain a construction where the padded length is a multiple of 8 or 16.)

We define $\mathsf{UH}[2, n]$ to be a family of functions. For each function in the family the domain is the set of all binary strings of lengths less than 2^{63} and the range is $\mathbb{F}_{2^n}$. The size restriction on the domain ensures that the length of any string in the domain can be represented using 63 bits.

Restricting the length of strings to be hashed to be less than 2^{63} is of no consequence for practical applications. The family $\mathsf{UH}[2, n]$ is indexed by the set of all binary strings of lengths equal to 2^{64} which form the key space for this family. In practice, the entire key will not be required. Only the initial segment of length approximately equal to the message will be required. We make this precise below.

Let $\mathbf{a} = (a_1, \ldots, a_l)$ be the message, where $l \geq 1$ and each a_i is a bit. Let $\mathsf{pad}(\mathbf{a})$ be the string obtained by padding a minimum number of zeros to $\mathbf{a}$ so as to ensure that the length of $\mathsf{pad}(\mathbf{a})$ is a multiple of 32. Then

$$\mathsf{UH}_{\mathbf{K}}(\mathbf{a}) = \mathsf{LH}_{\mathbf{K}_1}(\mathsf{pad}(\mathbf{a})||\lambda_{64}(l)). \tag{10}$$

$\mathbf{K}$ is a binary string of length 2^{64} while $\mathbf{K}_1$ is the initial segment of $\mathbf{K}$ of length $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)$ bits, where $\mathsf{blklen}_n(l) = n \times \lceil l/n \rceil$. By $\widehat{\mathbf{a}}$ we will denote the binary string $\mathsf{pad}(\mathbf{a})||\lambda_{64}(l)$. Here, $\lambda_{64}(l)$ is defined as follows:

1. Let y be the minimum length binary representation of l;
2. reverse y and pad on the left with 0^*1 to obtain a binary string of length 64;
3. this is our desired $\lambda_{64}(l)$.

For example, if $l = 12$, then the minimum length binary representation of 12 is 1100 and $\lambda_{64}(12) = 0^{59}10011$. If, on the other hand, $l = 3$, then the minimum length binary representation of 3 is 11 and $\lambda_{64}(3) = 0^{61}111$.

The function $\lambda(l)$ is actually a *prefix-free* encoding of the reverse of the minimum length binary representation of l.[2]

Proposition 1. *For λ_{64} defined above the following hold.*

1. *For $l \neq l'$, $\lambda_{64}(l) \neq \lambda_{64}(l')$.*
2. *The last bit of $\lambda_{64}(l)$ is always 1.*

Proof: The second point is directly seen from the construction. For the first point, suppose that str is the reverse of the minimum length binary representation of l. Let $s \leq 63$ be the length of str. Then $\lambda_{64}(l) = 0^{64-s-1}1\mathsf{str}$. Similarly, let $\lambda_{64}(l') = 0^{64-s'-1}1\mathsf{str}'$.

If $s = s'$, then the lengths of the binary representations of l and l' are equal. Since $l \neq l'$, it then necessarily follows that $\mathsf{str} \neq \mathsf{str}'$ and so $\lambda_{64}(l) \neq \lambda_{64}(l')$. On the other hand, if $s \neq s'$, then assume without loss of generality that $s > s'$. By definition, in $\lambda_{64}(l)$, the $(s+1)$st position from the right is 1, but, in $\lambda_{64}(l')$, the $(s+1)$st position from the right is 0. So, again it follows that $\lambda_{64}(l) \neq \lambda_{64}(l')$. $\square$

Theorem 3. *Let $\mathbf{a}$ and $\mathbf{a}'$ be two distinct binary strings having lengths l and l' respectively with $l \geq l'$. Let $\mathbf{K}$ be a key for UH and $\mathbf{K}_1$ be the initial segment of $\mathbf{K}$ of length equal to $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)$ bits. Let δ be any element of $\mathbb{F}_{2^n}$. Then*

$$\mathsf{Pr}[\mathsf{UH}_{\mathbf{K}}(\mathbf{a}) - \mathsf{UH}_{\mathbf{K}}(\mathbf{a}') = \delta] = \frac{1}{2^{ns}}.$$

Here the probability is over the random choice of $\mathbf{K}_1$ (and not the whole of $\mathbf{K}$).

Proof: We are given that $\mathbf{K}_1$ is the initial segment of $\mathbf{K}$ of length equal to $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)$ bits. Let $\mathbf{K}_1'$ be the initial segment of $\mathbf{K}$ whose length is equal to $\mathsf{blklen}_n(|\widehat{\mathbf{a}'}|)$ bits.

Let $k = |\widehat{\mathbf{a}}|$ and $k' = |\widehat{\mathbf{a}'}|$. By the condition of the theorem, we have $k \geq k'$.

The first point to note is that if $\mathbf{a} \neq \mathbf{a}'$, then $\mathbf{x} = \mathsf{pad}(\mathbf{a})||\lambda_{64}(l) \neq \mathsf{pad}(\mathbf{a}')||\lambda_{64}(l')||0^{k-k'} = \mathbf{x}'$. (Note that $\mathbf{x} = \widehat{\mathbf{a}}$ and $\mathbf{x}' = \widehat{\mathbf{a}'}||0^{k-k'}$.)

There are a few cases to see this. If $k > k'$, then the last bit of $\mathbf{x}'$ is 0, where as the last bit of $\mathbf{x}$ is 1 from the definition of $\lambda_{64}(l)$. Thus, $\mathbf{x} \neq \mathbf{x}'$.

If, on the other hand, $k = k'$, then the lengths of x and x' are equal. We need to consider two cases. If $l = l'$, then both $\mathbf{a}$ and $\mathbf{a}'$ are two distinct strings of the same length and then it follows that $\mathbf{x} \neq \mathbf{x}'$. If $l \neq l'$, then from Proposition 1, it follows that $\lambda_{64}(l) \neq \lambda_{64}(l)$. So, the last 64 bits of x and x' are different which again leads to $\mathbf{x} \neq \mathbf{x}'$.

The second point is that

$$\begin{aligned}
\mathsf{UH}_{\mathbf{K}}(\mathbf{a}) &= \mathsf{LH}_{\mathbf{K}_1}(\mathbf{x}) \text{ and} \\
\mathsf{UH}_{\mathbf{K}}(\mathbf{a}') &= \mathsf{LH}_{\mathbf{K}_1'}(\mathsf{pad}(\mathbf{a}')||\mathsf{bin}_{64}(l')) \\
&= \mathsf{LH}_{\mathbf{K}_1}(\mathbf{x}').
\end{aligned}$$

[2] In a previous version of the definition of λ, the padding on the left was only with zeros. This did not provide a prefix-free encoding and so the encodings of $l = 1$ and $l = 2$ became the same. This was kindly pointed out to us by Subhamoy Maitra and the ZUC design team.

Note that if $k > k'$, then the padding at the end by zeros does not affect the output of the computation of LH. Now the problem reduces to that of bounding the differential probability of LH for equal length strings. From Theorem 1, we obtain this probability to be $1/2^n$. $\square$

Note. Instead of using separate notation $\mathbf{K}$ and $\mathbf{K}_1$, we will use the convention that the key $\mathbf{K}$ in $\mathsf{UH}_{\mathbf{K}}(\mathbf{a})$ is of length equal to $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)$ (instead of being a binary string of length equal to 2^{64}). This is consistent with the definition of UH, since the later bits of $\mathbf{K}$ do not affect the computation of the hash value for $\mathbf{a}$.

The above description defines the family UH from the family LH. In a similar manner we define a family $\mathsf{UH}^{\mathsf{T}}[2,n,s]$ from the family LH^{T}, i.e.,

$$\mathsf{UH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}) = \mathsf{LH}^{\mathsf{T}}_{\mathbf{K}_1}(\mathsf{pad}(\mathbf{a})||\lambda_{64}(l)). \tag{11}$$

The only difference is that in this case, the family $\mathsf{UH}^{\mathsf{T}}[2,n,s]$ is indexed by the set of all binary strings of lengths equal to $2^{64}+ns$ bits. Also, $\mathbf{K}_1$ consists of the first $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)+ns$ bits of $\mathbf{K}$. The extra ns bits corresponds to the additional s n-bit blocks required for the s-stage slide over $\mathbf{K}$.

The proof of Theorem 3 essentially reduces to bounding the differential probability for LH on equal length strings. In a similar manner, the analysis of the differential probability for UH^{T} reduces to bounding the differential probability for LH^{T} on equal length strings. This gives the following result.

Theorem 4. *Let $\mathbf{a}$ and $\mathbf{a}'$ be two distinct binary strings having lengths l and l' respectively with $l \geq l'$. Let $\mathbf{K}$ be a key for UH and $\mathbf{K}_1$ be the initial segment of $\mathbf{K}$ of length equal to $\mathsf{blklen}_n(|\widehat{\mathbf{a}}|)+ns$ bits. Let δ be any element of $\mathbb{F}_{2^n}$. Then*

$$\mathsf{Pr}[\mathsf{UH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}) - \mathsf{UH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}') = \delta] = 1/2^n.$$

Here the probability is over the random choice of $\mathbf{K}_1$ (and not the whole of $\mathbf{K}$).

6.1 A Simpler Padding Scheme

The padding scheme described above ensures that the padded message consists of 32-bit words. This will align well with byte boundaries and is necessary for many applications. It can be modified to ensure that the padded message consists of 8-bit or 16-bit words. Also, the assumption is that the length is at most $2^{63}-1$ and the construction can be easily modified so that the length is at most $2^{31}-1$.

On the other hand, if we do not need to bother about the alignment of the message with byte boundaries, then there is a much simpler padding scheme. Given a binary string x, let $\chi(x)$ denote the reverse of the minimum length binary representation of the length of x. For example, if $x = 1100101010001001$, then the length of x is 12 which is 1100 in binary and so $\chi(x) = \chi(1100101010001001) = 0011$.

Given a binary string x, consider the encoding

$$\mathsf{simpleEncode} : x \mapsto x||\chi(x).$$

Two properties hold for this map.

Proposition 2. *1. If $x \neq x'$, then* $\mathsf{simpleEncode}(x) \neq \mathsf{simpleEncode}(x')$.

2. The last bit of $\mathsf{simpleEncode}(x)$ *is* 1.

Proof: The second statement follows directly from the definition of the map. For the first statement, we consider two cases.

If the lengths of x and x' are equal, then $\chi(x)$ equals $\chi(x')$. Since $x \neq x'$, it follows that $\mathsf{simpleEncode}(x) \neq \mathsf{simpleEncode}(x')$. On the other hand, if the lengths of x and x' are not equal, then the lengths of $\mathsf{simpleEncode}(x)$ and $\mathsf{simpleEncode}(x')$ are unequal implying that they are unequal. □

Note that for the above encoding we do need any bound on the length of the binary string x. For the hash function, we will however, work with finite length binary strings of some maximum length. The key for the hash function will as before be denoted by $\mathbf{K}$. It is now possible to define two hash functions.

$$\mathsf{SUH}_{\mathbf{K}}(\mathbf{a}) = \mathsf{LH}_{\mathbf{K}_1}(\mathsf{simpleEncode}(\mathbf{a})) \tag{12}$$

$$\mathsf{SUH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}) = \mathsf{LH}^{\mathsf{T}}_{\mathbf{K}_1}(\mathsf{simpleEncode}(\mathbf{a})). \tag{13}$$

Here $\mathbf{K}_1$ is the prefix of the key $\mathbf{K}$ whose length is equal to the length of $\mathsf{simpleEncode}(\mathbf{a})$. The following result can now be proved in a manner similar to that of Theorem 3.

Theorem 5. *Let* $\mathbf{a}$ *and* $\mathbf{a}'$ *be two distinct binary strings. Let* δ *be any element of* $\mathbb{F}_{2^n}$. *Then*

$$\Pr[\mathsf{SUH}_{\mathbf{K}}(\mathbf{a}) - \mathsf{SUH}_{\mathbf{K}}(\mathbf{a}') = \delta] = 1/2^n;$$
$$\Pr[\mathsf{SUH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}) - \mathsf{SUH}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{a}') = \delta] = 1/2^n.$$

This padding rule, though simple, may not be suitable for applications. The padded lengths do not respect byte boundaries which would cause problems for real-life applications.

7 Comparison to Previous Constructions

The focus of our construction is implementation in resource constrained devices. In this section, we survey the previously known constructions and compare to our construction in the above context.

Polynomial evaluation based hash functions. Let $\mathbf{M} = (M_1, \ldots, M_l)$ be a message with each M_i in $\mathbb{F}_q$. Define a polynomial $p_{\mathbf{M}}(x) = M_1 + xM_2 + \cdots + x^{l-1}M_l$. Given a key $K \in \mathbb{F}_q$, define the hash of $\mathbf{M}$ under the key K as follows.

$$\mathbf{M} \overset{K}{\longmapsto} p_{\mathbf{M}}(K).$$

This defines a map from l elements of $\mathbb{F}_q$ to a single element of $\mathbb{F}_q$. The key is a single element of $\mathbb{F}_q$. Collision probabilities for this hash function can be shown to be bounded above by $(l-1)/q$ using a simple argument on the number of roots of a polynomial of degree less than l. Further, the differential probabilities for the map $\mathbf{M} \overset{K}{\mapsto} K \times p_{\mathbf{M}}(K)$ can be shown to be bounded above by l/q. Evaluating the polynomial requires $(l-1)$ multiplications over $\mathbb{F}_q$.

In the context of authentication codes, the polynomial evaluation based hash function was proposed by [8, 13, 25]. Subsequent works have reported fast implementations by suitably choosing the underlying field $\mathbb{F}_q$. Poly1305 [2] and PolyR [17] are two constructions which work over general characteristic fields.

Shoup [22] described three methods for defining a universal hash function using binary extension field arithmetic. (A specific instance of polynomial hashing over $\mathbb{F}_{2^{128}}$ is GCM [19].) These are polynomial evaluation, division hash (earlier called "LFSR hashing" or cryptographic CRC by Krawzcyk [16]) and generalized division hash (GDH). The third method has the first two as special cases. In GDH, the message is a polynomial $m(x)$ over $\mathbb{F}_{2^k}$ of degree less than nl/k. The key is a random monic irreducible polynomial $\tau(x)$ of degree l/k over $\mathbb{F}_{2^k}$. The hash value is $m(x)x^{l/k}$ mod $\tau(x)$. The differential probabilities are bounded above by $nl/(k2^l)$.

In GDH, the message is a polynomial over $\mathbb{F}_{2^k}$, i.e., the message is broken into k-bit words. Also, $\tau(x)$ is a polynomial of degree l/k over $\mathbb{F}_{2^k}$. The digest computation $m(x)x^{l/k}$ mod $\tau(x)$ is done by successively computing the powers $\delta_i = x^i \bmod \tau(x)$, $i = l/k, l/k+1, \ldots$ and multiplying with the corresponding coefficient of $m(x)$. Each δ_i is a polynomial of degree less than l/k having coefficients from $\mathbb{F}_{2^k}$. Thus, the multiplication of a coefficient of $m(x)$ by some δ_i, in general, requires l/k multiplications over $\mathbb{F}_{2^k}$.

If $l = k$, then $\tau(x) = x + \alpha$ and the hash function evaluation reduces to computing $\alpha \cdot m(\alpha)$ over $\mathbb{F}_{2^k}$. This is the polynomial evaluation hash over $\mathbb{F}_{2^k}$. If $k = 1$, then the hash function computation reduces to evaluating $m(x)x^l$ mod $\tau(x)$, where $m(x)$ is a polynomial over $\mathbb{F}_2$ and $\tau(x)$ is an irreducible polynomial of degree l over $\mathbb{F}_2$. The case $k = 1$ corresponds to Krawzcyk's "LFSR hashing".

Note that for $l > k$ (and this includes Krawzcyk's LFSR hashing), the key is a random irreducible polynomial over $\mathbb{F}_{2^k}$. Choosing and changing a key requires running a program to find such a polynomial. So, changing keys is a much less efficient procedure compared to the constructions proposed in this paper.

Bernstein [4] introduced a new hash function based on polynomial evaluation. This function modifies a previous work due to Winograd and Rabin [20]. The advantage of the new function is that compared to the usual polynomial based hash function, it reduces the number of multiplications by almost a factor of two. See [4] for details.

In polynomial based hashing (including all of the above mentioned constructions), the collision probability degrades with the increase in the length of the message. In our constructions, an n-bit digest provides a differential probability of 2^{-n}. On the other hand, the key for polynomial hashing is small, while in all approaches based on Equation (1) approach, the key is as long as the message and has to be taken as the output of a PRG.

MMH **by Halevi and Krawczyk [10].** This is an implementation of the map MLHash in (1). The prime is chosen to be $q = 2^{32} + 15$. Certain modifications are made to improve the software performance of the map. The modified map is the following.

$$(M_1, \ldots, M_l) \overset{(K_1,\ldots,K_l)}{\longmapsto} \left(\left(\left(\sum_{i=1}^{l} M_i K_i \right) \bmod 2^{64} \right) \bmod \left(2^{32} + 15\right) \right) \bmod 2^{32}.$$

The digest size is 32 bits and the differential probabilities are bounded above by $1.5/2^{30}$.

UMAC **by Black et al [6].** The core of UMAC is a non-linear hash function NH^T. Here the superscript T stands for the Toeplitz construction and NH^T is built from a non-linear hash function NH. Let $\mathbf{M} = (M_1, \ldots, M_l)$, l even, be the message and $\mathbf{K} = (K_1, \ldots, K_n)$, with $n \geq l$ be the key.

Each M_i and K_j are w-bit integers. Then $\mathsf{NH}_{\mathbf{K}}(\mathbf{M})$ is computed as

$$\left(\sum_{i=1}^{l/2}((M_{2i-1}+K_{2i-1}) \bmod 2^w)\cdot((M_{2i}+K_{2i}) \bmod 2^w)\right) \bmod 2^{2w}. \tag{14}$$

The output is a $2w$-bit integer and the collision probability is shown to be 2^{-w}.

The cost of computing NH is $l/2$ multiplications modulo 2^{2w} and l additions modulo 2^w. The construction NH^T is an extension, which produces a digest of size $2tw$; has collision probability 2^{-tw} and the cost is $tl/2$ multiplications modulo 2^{2w} and l additions modulo 2^w. To provide collision probability of 2^{-128} with $w = 32$ will require a digest of 256 bits, i.e., $t = 4$. The cost will be $2l$ multiplications modulo 2^{64} and $4l$ additions modulo 2^{32}.

WH by Kaps et al [15]. Messages and keys are sequences of length l over $\mathbb{F}_{2^w}$, where l is assumed to be even. Let $\rho(x)$ be an irreducible polynomial of degree w over $\mathbb{F}_2$ and let α be a root of ρ. For a message $\mathbf{M} = (M_1, \ldots, M_l)$ and key $\mathbf{K} = (K_1, \ldots, K_l)$

$$\mathsf{WH}_{\mathbf{K}}(\mathbf{M}) = \sum_{i=1}^{l/2}(M_{2i-1}+K_{2i-1})(M_{2i}+K_{2i})\alpha^{(l/2-i)w}. \tag{15}$$

The authors describe the Toeplitz version of this map. This function and its Toeplitz version were introduced get a construction which is suitable for low power devices. A detailed hardware design along with different options is presented. In comparison with NH, it is shown in [15] that WH requires significantly less power. This is not very surprising, since NH is based on multiplication modulo 2^{2w} which requires more hardware area than implementing a finite field multiplier over $\mathbb{F}_{2^w}$.

A major criticism of the map WH is that the multiplication by $\alpha^{(l/2-i)w}$ is unnecessary. The authors have actually obtained a less efficient version of Winograd's pseudo-dot product based hash function. In Appendix A, we discuss this hash function and its Toeplitz variant. Using Winograd's construction over $\mathbb{F}_{2^{32}}$ will lead to a more efficient construction compared to WH. The motivation for WH was lower power consumption in comparison to UMAC. Winograd's construction will require even lesser power than WH. In view of Winograd's construction, we do not see any importance of WH.

Requirement of a multiplier. In all of the above mentioned constructions, i.e., polynomial hashing, MMH, UMAC (and WH), a multiplier is required: MMH and UMAC require integer multipliers while WH requires a finite field multiplier. Implementing a multiplier in hardware usually requires significant area. In contrast, we have described constructions which do not require any multipliers. For resource constrained devices, these constructions will be preferrable.

Bucket Hashing by Rogaway [21]. The message is a sequence of length l over $\mathbb{F}_{2^w}$ for some suitably chosen word size w. The key consists of an $L \times l$, $L < l$ binary matrix A where each column of A has exactly three ones. An additional requirement on the parameters is that $\binom{L}{3} \geq l$. For a message $\mathbf{M}$, the digest is defined to be $A\mathbf{M}$.

Since the matrix A is a binary matrix of a particularly simple form, the matrix-vector product $A\mathbf{M}$ can be computed using a simple method which can be envisioned as throwing words into L buckets. From a computation point of view, only XOR of w-bit words is required. For the key, it is not required to give the entire matrix A. Since there are exactly three ones in each column, a column is specified by three integers in the range $\{1,\ldots,L\}$ and so the entire matrix is specified by $3l$ integers in the range $\{1,\ldots,L\}$. Key size therefore is $3l\lceil\log_2 L\rceil$ bits. The size of the digest is $L \times w$ bits. Memory required for computing the hash value consists of L w-bit buckets. The construction is reportedly very fast in software.

Collision probability is given by a complicated expression and it is mentioned in [21], that this is about $3312 \times L^{-6}$. The collision probability does not depend on the word size w. Increasing w increases the size of the digest but does not lower the collision probability; it has the effect of making the computation of the digest faster.

For comparison to the constructions in this work, we note the following points.

1. Obtaining a collision probability of 2^{-128} requires L to be about 2^{28}. The requirement of L w-bit buckets to compute the hash value is prohibitively large.
2. Bucket hashing maps lw-bit strings to Lw-bit strings with $L(L-1)(L-2) \geq 6l$ and so, the amount of compression is not really good. As such this map has been suggested to be used at the first level of a multi-level hashing procedure.
3. For the constructions in this paper, the key is as long as the message (and slightly more longer for the Toeplitz construction). In comparison, the key for bucket hashing is about $3\lceil\log_2 L\rceil$ times longer than the message.

Bucket hashing with smaller key size by Johansson [12]. Let α be an element of $\mathbb{F}_{2^n}$ and $\mathbf{M} = (M_1,\ldots,M_l)$ be a message where M_i is a bit. Consider the map

$$\mathsf{JH}_\alpha(\mathbf{M}) : \mathbf{M} \overset{\alpha}{\longmapsto} M_1\alpha + M_2\alpha^2 + \cdots + M_l\alpha_l.$$

This is a hash function which maps an l-bit string to an n-bit string and the collision probability for this function is $l/2^n$. Extension to the case where M_i's are w-bit words is done as follows. Let $M_i^{(j)}$ be the j-th bit of M_i and $M^{(j)}$ be the sequence formed from the j-th bits of all the M_i's. Let $\alpha_l = (\alpha,\alpha^2,\ldots,\alpha^l)$. Then the map for w-bit strings is as follows.

$$\mathbf{M} \overset{\alpha}{\longmapsto} (\mathsf{JH}_\alpha(M^{(1)}),\ldots,\mathsf{JH}_\alpha(M^{(w)})).$$

The digest grows to wn bits but the collision probability remains the same as before, i.e., $l/2^n$. This seems like a bad thing to do, but, the advantage is that the computation now proceeds w times faster.

Another extension of JH is described in [12]. The key consists of k elements of $\mathbb{F}_{2^n}$, $\alpha_1,\ldots,\alpha_k$. Suppose $\mathbf{M}_1,\ldots,\mathbf{M}_k$ are k bit strings each of length l. Then the extension, EJH is the following.

$$\mathsf{EJH} : \mathbf{M}_1,\ldots,\mathbf{M}_k \overset{(\alpha_1,\ldots,\alpha_k)}{\longmapsto} \mathsf{JH}_{\alpha_1}(\mathbf{M}_1) + \cdots + \mathsf{JH}_{\alpha_k}(\mathbf{M}_k).$$

The digest is n bits and the collision probability is again $l/2^n$. Computation of the individual JH's can proceed in parallel. The trade-off is that the key size increases to nk bits. The case where the $\mathbf{M}_i$'s are w-bit strings can be handled in a manner similar to the one described for JH.

The core of the construction is the map JH. This is a special case of polynomial evaluation hash function where the M_i's are chosen to be bits instead of elements of $\mathbb{F}_{2^n}$. The advantage is that since the M_i's are bits, with some pre-computation, this does not require any multiplication during the actual hash computation. As explained in [12], the computation process for JH can also be seen as throwing words into buckets. The advantage over the method in [21] is that the key size is small.

Following the description in [12], the computation of JH requires intermediate memory to store the buckets. This requires L arrays of length $N = 2^{n/L}$ each where each entry of the array is a w-bit word. The quantity L is a parameter for the construction and determines *both* the size of the intermediate storage *and* the speed of the hash computation. The intermediate storage requirement is $L2^{n/L}w$ bits. For the bucket hashing procedure to be reasonably fast, the value of L must be quite small. Then for an 80-bit or 128-bit digest the intermediate storage requirement becomes very large. For example, [12] gives $l = 2^{13}$, $m = 80$ as an example and chooses $L = 4$ to obtain a speed of 8 operations per word. The memory requirement for buckets then becomes $2^{22}w$ bits. An implementation which wishes to obtain the speed promised by [12] must be prepared to use $2^{22}w$ bits of intermediate memory. This becomes prohibitive for low memory resource constrained devices. In general, for the construction in [12], there does not seem to be a case where high speed hashing can be done using a small amount of memory for the buckets.

Note. Bucket hashing techniques of both Rogaway [21] and Johansson [12] are actually meant for high-speed software implementation in general purpose computers. Apart from the different issues mentioned above, the basic technique of throwing words into a number of buckets does not suggest a simple systolic architecture.

LFSR based hashing by Johansson [11]. This is a suggestion for using LFSRs in the construction of authentication codes and can also be viewed as universal hash functions. The construction from [11] is the following. Let $\mathbb{F}_q$ be the finite field of q elements. Messages are *fixed* length sequences of over $\mathbb{F}_q$. Suppose that messages are of length m. Keys also consist of sequences of length m over $\mathbb{F}_q$ while digests are sequences of length s over $\mathbb{F}_q$ for a suitably chosen s.

Let $\mathcal{L}$ be an LFSR over $\mathbb{F}_q$ whose connection polynomial is of degree m and is irreducible over $\mathbb{F}_q$. So, the length of the LFSR is equal to that of the message. We consider $\mathcal{L}$ to be the state update function of the LFSR, i.e., $\mathcal{L}$ is a map from $\mathbb{F}_q^m$ to $\mathbb{F}_q^m$. For a message $\mathbf{M} \in \mathbb{F}_q^m$, the digest is defined to be

$$\left(\langle \mathbf{M}, \mathbf{K} \rangle, \langle \mathbf{M}, \mathcal{L}(\mathbf{K}) \rangle, \cdots, \langle \mathbf{M}, \mathcal{L}^{s-1}(\mathbf{K}) \rangle\right). \tag{16}$$

Each component is an element of $\mathbb{F}_q$ and hence the digest is an element of $\mathbb{F}_q^s$. In [11], the LFSR length is equal to the length of the message. In a related work [14], hash function construction based on Reed-Solomon codes have been proposed. This turns out to be usual polynomial based hashing. More specifically, the authors mention that processing a message consisting of at most 2^c elements over $\mathbb{F}_{2^{c+d}}$ requires evaluating a polynomial of degree at most 2^c over $\mathbb{F}_{2^{c+d}}$. They further mention that this can be done using several LFSRs each of length $c+d$. Note that in this case also, the length of the LFSR is determined by the length of the message and is in fact longer than the length of the message.

The main difference to our construction is in the manner the LFSR is used. For example, if $q = 2$, and the requirement is to hash messages of length 2^{20} bits, then [11] requires an LFSR of length 2^{20}. This in turn requires choosing an irreducible polynomial of length 2^{20}. A single clocking of such a long LFSR will be quite slow in both hardware and software. (Similar criticism apply to the suggestion in [14].) Further, [11] does not discuss how to handle variable length messages. A difficulty in doing this is that to hash messages of different lengths one requires LFSRs of different lengths. Again for $q = 2$, if the requirement is to hash messages of lengths between 2^{10} and 2^{20} bits, then the hash function specification has to include irreducible polynomials of all sizes between 2^{10} and 2^{20} (apart from other possible mechanisms to handle variable lengths). This makes the construction completely impractical for variable length messages.

A more conceptual issue is that the LFSR map $\mathcal{L}$ in (16) can actually be done away with. Consider the Toeplitz version of the multi-linear hash function as given in (6). This maps a message $\mathbf{M}$ of length l, to $(\langle \mathbf{M}, \mathbf{K}_1 \rangle, \ldots, \langle \mathbf{M}, \mathbf{K}_s \rangle)$ where $\mathbf{K}_i = (K_i, \ldots, K_{l+i-1})$. So, by including extra s elements of $\mathbb{F}_q$ as part of the key, it is possible to completely do away with the LFSR map $\mathcal{L}$ in (16). For $q = 2$ to obtain a 128-bit digest, we need $s = 128$, which means that compared to the map in (16) the key is longer by an extra 128 bits. This is a very small trade-off for eliminating the requirement of using LFSRs of length 2^{20} bits. We cannot think of any scenario where the hash function from [11] would be preferable to the constructions given in this paper.

Summary. We summarise the above discussion.

1. Polynomial based hashing, Shoup's GDH (including Krawczyk's LFSR hashing), MMH, UMAC and WH require multipliers. We provide multiplier-less designs. (In view of Appendix A, WH is actually of no interest.)
2. Bucket hashing is good for fast software implementation on general purpose computers. Obtaining high speed requires large intermediate storage which make them unsuitable for low-memory devices.
3. LFSR based hashing by Johansson (and a similar suggestion in [14]) is impractical for long and variable length messages. Further, by including a few extra bits in the key, it is possible to obtain a significantly simpler map.

Based on the above, we can conclude that compared to known work, the constructions given in this paper provide the most attractive options for implementing universal hash functions in resource constrained devices. The smallest such construction is described in Section 4.1. Somewhat larger and faster options are described in Sections 5.1 and 5.2. Since, in our constructions, the key is as long as the message, practical implementation requires a PRG to generate the key. A good choice for PRG to be used with the hash constructions in this paper would be one of the proposals in the estream [1] portfolio focussed towards hardware implementation. Thus, our work provides the best known designs to a developer of message authentication codes for resource constrained devices.

Acknowledgement.

We thank Kishan Chand Gupta for carefully reading the manuscript and providing useful comments. We also thank Subhamoy Maitra and the ZUC design team for pointing out a gap in the previous construction for handling variable length messages.

References

1. eSTREAM, the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`.
2. Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.
3. Daniel J. Bernstein. Stronger security bounds for Wegman-Carter-Shoup authenticators. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 2005.

4. Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. `http://cr.yp.to/papers.html#pema`.
5. Jürgen Bierbrauer, Thomas Johansson, Gregory Kabatianskii, and Ben J. M. Smeets. On families of hash functions via geometric codes and concatenation. In Stinson [23], pages 331–342.
6. John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1999.
7. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.
8. Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, 2:65–72, 1993.
9. Edgar N. Gilbert, F. Jessie MacWilliams, and Neil J. A. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53:405–424, 1974.
10. Shai Halevi and Hugo Krawczyk. MMH: Software message authentication in the gbit/second rates. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 172–189. Springer, 1997.
11. Thomas Johansson. A shift register construction of unconditionally secure authentication codes. *Des. Codes Cryptography*, 4(1):69–81, 1994.
12. Thomas Johansson. Bucket hashing with a small key size. In *EUROCRYPT*, pages 149–162, 1997.
13. Thomas Johansson, Gregory Kabatianskii, and Ben J. M. Smeets. On the relation between a-codes and codes correcting independent errors. In *EUROCRYPT*, pages 1–11, 1993.
14. Gregory Kabatianskii, Ben J. M. Smeets, and Thomas Johansson. On the cardinality of systematic authentication codes via error-correcting codes. *IEEE Transactions on Information Theory*, 42(2):566–578, 1996.
15. Jens-Peter Kaps, Kaan Yüksel, and Berk Sunar. Energy scalable universal hashing. *IEEE Trans. Computers*, 54(12):1484–1495, 2005.
16. Hugo Krawczyk. LFSR-based hashing and authentication. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 129–139. Springer, 1994.
17. Ted Krovetz and Phillip Rogaway. Fast universal hashing with small keys and no preprocessing: The polyr construction. In Dongho Won, editor, *ICISC*, volume 2015 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2000.
18. R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications, revised edition.* Cambridge University Press, 1994.
19. David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.
20. Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.
21. Phillip Rogaway. Bucket hashing and its application to fast message authentication. *J. Cryptology*, 12(2):91–115, 1999.
22. Victor Shoup. On fast and provably secure message authentication based on universal hashing. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 1996.
23. Douglas R. Stinson, editor. *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*. Springer, 1994.
24. Douglas R. Stinson. Universal hashing and authentication codes. *Des. Codes Cryptography*, 4(4):369–380, 1994.
25. Richard Taylor. An integrity check value algorithm for stream ciphers. In Stinson [23], pages 40–48.
26. Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.
27. Shmuel Winograd. A new algorithm for inner product. *IEEE Transactions on Computers*, 17:693–694, 1968.

A Winograd's Pseudo-Dot Product Based Hash Computation

Let $\mathbf{M} = (M_1, \ldots, M_l)$ be the message and $\mathbf{K} = (K_1, \ldots, K_l)$ be the key where each M_i and K_j are elements of $\mathbb{F}_q$ and assume that l is even. Winograd [27] defined

$$\left.\begin{array}{r}\mathsf{PD}_{\mathbf{K}}(\mathbf{M}) : \mathbf{M} \xmapsto{\mathbf{K}} (M_1 + K_1)(M_2 + K_2) + (M_3 + K_3)(M_4 + K_4) \\ + \cdots + (M_{l-1} + K_{l-1})(M_l + K_l)\end{array}\right\} \tag{17}$$

to be the pseudo-dot product of $\mathbf{M}$ and $\mathbf{K}$. Evaluating this expression requires $l/2$ multiplications over $\mathbb{F}_q$ and $3l/2$ additions over $\mathbb{F}_q$. In comparison the GMS hash function requires l multiplications and l additions. So, the pseudo-dot product computation trades off about $l/2$ multiplications for $l/2$ additions. In general, this is an advantage. But, over $\mathbb{F}_2$, addition and multiplication are actually the XOR and AND operation on bits and take the same time. So over $\mathbb{F}_2$, there is no particular reason to prefer PD over the function MLHash given by (1). Further, for PD there does not seem to be any analogue of the generalisation of MLHash that we have introduced.

Winograd had introduced the pseudo-dot product to speed up matrix multiplication. As pointed out by Bernstein [4], this idea in various forms was later used in the context of construction of hash functions by several later authors including [10, 6]. Black et al [6] consider a modification of this expression where the terms $(K_i + M_i)$ are computed modulo 2^w while the products and the outer sum are computed modulo 2^{2w} where w is a suitable word size. They also consider the Toeplitz version of their basic construction.

It is fairly easy to obtain the direct Toeplitz version of (17) over a finite field $\mathbb{F}_q$. To describe such a construction, let $\mathbf{K} = (K_1, \ldots, K_{l+2s-2})$ and $\mathbf{K}_i = (K_i, \ldots, K_{l+i-1})$, for $1 \leq i \leq s$. Define a map PD^{T} as follows.

$$\mathsf{PD}^{\mathsf{T}}_{\mathbf{K}}(\mathbf{M}) : \mathbf{M} \xmapsto{\mathbf{K}} (\mathsf{PD}_{\mathbf{K}_1}(\mathbf{M}), \mathsf{PD}_{\mathbf{K}_3}(\mathbf{M}), \ldots, \mathsf{PD}_{\mathbf{K}_{2s-1}}(\mathbf{M}))\,. \tag{18}$$

Note that in the above, each shift of the key is by two elements. This is to be contrasted with the construction in Section 4 where the shift is by one element. (For the pseudo-dot product based computation, shifts of even length will work, but shifts of odd length will not work.) As a result, the total key length in this case is s elements longer than the key length required in Section 4.

It is not too difficult to show that for equal length messages the differential probabilities for the map in (18) are q^{-s}. The actual computation of the hash value can be done using the following simple algorithm. The variables $R_1, \ldots, R_s$ each can store an element of $\mathbb{F}_q$. In the algorithm, the number of arithmetic operations per message element M_i of $\mathbb{F}_q$ is $s/2$ multiplications and $3s/2$ additions over $\mathbb{F}_q$.

```
$R_1 = \cdots = R_s = 0$;
for $i = 1$ to $l/2$
    for $j = 1$ to $s$
        $R_j = R_j + (M_{2i-1} + K_{2i+2j-3})(M_{2i} + K_{2i+2j-2})$;
    end for;
end for;
return $(R_1, \ldots, R_s)$.
```

For the case $q = 2$, the above algorithm can be stated in a form which is similar to the algorithm given in Section 4.1. The storage requirement and efficiency will be the same, but, one small disadvantage will be that the key will be longer by an extra s bits. This is due to the fact that in the present case the key shifts are of length two while in Section 4.1 the key shifts are of length one.