A Fast Single-Key Two-Level Universal Hash Function

Debrup Chakraborty, Sebati Ghosh and Palash Sarkar
Indian Statistical Institute
203, B.T.Road, Kolkata, India - 700108.
{debrup,sebati_r,palash}@isical.ac.in

November 23, 2016

Abstract

Universal hash functions based on univariate polynomials are well known, e.g. Poly1305 and GHASH. Using Horner's rule to evaluate such hash functions require $\ell - 1$ field multiplications for hashing a message consisting of ℓ blocks where each block is one field element. A faster method is based on the class of Bernstein-Rabin-Winograd (BRW) polynomials which require $\lfloor \ell/2 \rfloor$ multiplications and $\lfloor \lg \ell \rfloor$ squarings for $\ell \geq 3$ blocks. Though this is significantly smaller than Horner's rule based hashing, implementation of BRW polynomials for variable length messages present significant difficulties. In this work, we propose a two-level hash function where BRW polynomial based hashing is done at the lower level and Horner's rule based hashing is done at the higher level. The BRW polynomial based hashing is applied to a fixed number of blocks and hence the difficulties in handling variable length messages is avoided. Even though the hash function has two levels, we show that it is sufficient to use a single field element as the hash key. The basic idea is instantiated to propose two new hash functions, one which hashes a single binary string and the other can hash a vector of binary strings. We describe two actual implementations, one over $\mathbb{F}_{2^{128}}$ and the other over $\mathbb{F}_{2^{256}}$ both using the `pclmulqdq` instruction available in modern Intel processors. On both the Haswell and Skylake processors, the implementation over $\mathbb{F}_{2^{128}}$ is faster than the highly optimised implementation of GHASH by Gueron. We further show that the Fast Fourier Transform based field multiplication over $\mathbb{F}_{2^{256}}$ proposed by Bernstein and Chou can be used to evaluate the new hash function at a cost of about at most 46 bit operations per bit of digest, but, unlike the Bernstein-Chou analysis, there is no hidden cost of generating the hash key. More generally, the new idea of building a two-level hash function having a single field element as the hash key can be applied to other finite fields to build new hash functions.

Keywords: universal hash function, Horner's rule, BRW polynomials, two-level hash function, MAC schemes.

1 Introduction

An important primitive in cryptography is a hash function family with provably low collision and differential probabilities. Hash functions with provably low collision probability are called almost universal (AU) and those with provably low differential probability are called almost XOR universal (AXU). Starting from the work of Carter and Wegman [6], such hash functions have been used to construct message authentication code (MAC) schemes. They have also been suggested for the construction of disk encryption and authenticated encryption.

A well known approach to the construction of an AU hash function is the multilinear map [8]. This requires ℓ field multiplications to obtain the digest when the message consists of ℓ field elements.

The computation can be reduced to about $\ell/2$ field multiplication by using the pseudo-dot product construction [25]. One problematic issue for both the multi-linear hash and the pseudo-dot product is that the key for the hash function is as long as the message.

The problem of long hash keys can be avoided by using another well known approach. In this approach, the digest is obtained by evaluating a univariate polynomial over a finite field. The coefficients of the polynomial are the message blocks and the point at which the polynomial is evaluated is the hash key. As a result, the hash key consists of a single field element. Using Horner's rule, a univariate polynomial of degree ℓ can be evaluated using $\ell - 1$ field multiplications. This cost is about the same as that required for multilinear map based hash function.

Bernstein [3] built on a previous work by Rabin and Winograd [18] to propose a hash function using a class of univariate polynomials called the BRW polynomials [19]. The hash key is still a single element of the field. The main advantage of BRW polynomial based hashing is that the number of multiplications required for hashing a message consisting of $\ell \geq 3$ blocks is $\lfloor \ell/2 \rfloor$ with an additional $\lfloor \lg \ell \rfloor$ squarings. In fact, what the pseudo-dot product is to the multilinear hash, the BRW polynomials is to the Horner based hash.

There is, however, an obstacle in efficient implementation of BRW polynomials. The definition of BRW polynomial is inherently recursive and the computation for an ℓ-block message requires two recursive calls on messages consisting of smaller number of blocks. In principle, the recursion can be simulated in a bottom-up fashion. The major problematic issue is that even the first recursive call cannot be made unless the length of the whole message is available. The whole message has to be buffered before even the first message block can be processed. A second problem is that at each point of the computation, it is quite complicated to figure out the operands that are to be multiplied. Again, in principle this can be done, but, actually determining the operands requires additional time. Possibly due to these issues, till date there has been no software efficient implementation of BRW based hash function. On the other hand, hardware implementations for fixed length inputs are known [7].

Our Contributions

We investigate the possibility of harnessing the speed of BRW polynomial based hashing without the associated difficulties in implementation. To this end, our first observation is that if the number of blocks in a message is a small fixed number, then the above mentioned difficulties disappear. Making effective use of this observation leads us to consider a two-level hash design. Suppose BRW is to be applied when the number of blocks is η. Let us call an η-block message to be a super-block. The input message blocks are divided into super-blocks and BRW is applied to each super-block. The outputs of these BRW calls are then combined using a Horner based hashing.

The number of multiplications required for a message consisting of ℓ super-blocks is about $\ell\eta/2 + \ell - 1$ (the precise count is provided later). Applying Horner to such a message will require $\ell\eta - 1$ multiplications while BRW will require $\ell\eta/2$ multiplications. By choosing a suitable value of η, the number of multiplications required by the new hash function can be made quite close to that of BRW. Such a two-level strategy has the advantage that it avoids the difficulties associated with implementing BRW on variable length messages.

The idea of two-level (or, multi-level) hashing is not new and has been proposed in the literature [22, 17, 20]. Two-level hashing in general requires independent keys for each level. So, applied directly, the hash key will consist of two field elements. For many applications, it is desirable to have only a single field element as the overall hash key.

An important aspect of our construction is the fact that the hash key consists of a single field element. Suppose the hash key for the BRW layer is τ. We show that it is possible to use a suitable power of τ as the key to the Horner layer. Moreover, if η is one less than some power of two, then the required power of τ can be computed using only one extra squaring over and above the computations required by BRW.

The underlying field for the new hash function can be any field. In particular, this field can be a suitable binary field or, it can be a field of large characteristic such as $\mathbb{F}_{2^{130}-5}$, the field which has been used in Poly1305.

To make the ideas concrete, we instantiate the two-level hash construction for the binary fields $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$. For implementing the new hash functions, the basic requirement is efficient implementation of multiplication over $\mathbb{F}_{2^n}$ for n equal to either 128 or 256. Being a binary field, it is possible to utilise the instruction `pclmulqdq` available in modern Intel processors for field multiplication. The instruction `pclmulqdq` multiplies two 64-bit polynomials and returns the 128-bit polynomial as the product. Our implementations for both $\mathbb{F}_{2^{128}}$ and $\mathbb{F}_{2^{256}}$ are based on the `pclmulqdq` instruction.

A field multiplication in $\mathbb{F}_{2^n}$ consists of a polynomial multiplication followed by a reduction modulo the irreducible polynomial representing the field. Late, or, delayed reduction is a well known technique for speeding up a group of field multiplications. Essentially, the idea is to perform several polynomial multiplications, add the results and then perform a single reduction for the entire group of multiplication. This technique cannot always be applied. We carefully analyse the structure of BRW and identify the groups of multiplications for which a single reduction suffices. Our implementations of the hash function for $n = 128$ and $n = 256$ make use of delayed reduction to achieve efficiency improvement.

Several other concrete efficiency issues for BRW have been identified and implemented. One of these is to perform independent multiplications together so that all the `pclmulqdq` instructions for these multiplications can be placed together. This permits possible utilisation of instruction level pipelining. For $n = 128$, the implementation of the new hash function is faster than the highly optimised implementation of GHASH by Gueron [9]; on the Haswell processor of Intel, we obtain speed improvements of about 15% to 19%, while on the Skylake processor, the speed improvements are about 10% to 15%.

The work by Bernstein and Chou [4] reports the implementation of a pseudo-dot product based hash function over $\mathbb{F}_{2^{256}}$. This implementation does not use the `pclmulqdq` instruction and is instead based on the Fast Fourier Transform (FFT) algorithm. The work shows that the hash function can be computed at the cost of 29 bit operations per bit of the digest. There is, however, a considerable hidden cost of generating the hash key which is as long as the message. This cost is not accounted for in the 29 bit operations per bit measure given in [4].

The FFT based multiplication algorithm can also be used with the new hash construction that we propose. The code for the multiplication algorithm described in [4] is not publicly available and so we could not carry out a concrete implementation. Instead, we used the operation counts for direct and inverse FFT, pointwise multiplication and the reduction algorithm reported in [4], to derive an expression for the number of bit operations per bit for the new hash function. For $\eta = 31$, this cost is at most about 46, while for $\eta = 63$ or 127, the cost is lower. The cost of 46 bit operations per bit is higher than the cost of 29 bit operations per bit reported in [4]. On the other hand, unlike [4], in our case there is no hidden cost of generating the hash key. Securely generating a long hash key will have a significant cost and if this cost is taken into account, then we expect the total cost in [4] to be significantly more than the 46 bit operations per bit that we obtain.

Previous Works

Universal hash functions were introduced by Carter and Wegman [6]. The multilinear hash function was proposed by Gilbert, MacWilliams and Sloane [8] while the pseudo-dot product appears in the work of Winograd [25]. Examples of Horner's rule based polynomial hashing are Poly1305 [1], PolyR [12] and GHASH [15]. Gueron and Kounavis [10] described an efficient method for reduction over binary fields. The most recent implementation of GHASH by Gueron [9] uses delayed reduction through the use of a pre-computed table. BRW polynomials were introduced in [3] and hardware implementation was reported in [7]. Well known constructions of hash functions based on the pseudo-dot product are UMAC and VMAC. As mentioned earlier, the construction in [4] is a more recent such construction over $\mathbb{F}_{2^{256}}$. Nandi [16] has shown a lower bound on the number of multiplications required for secure hashing which shows that the pseudo-dot and BRW based hashing essentially require an optimal number of field multiplications. Brief surveys on various constructions of universal hash functions can be found in [3, 21].

2 Preliminaries

Let $\mathcal{D}$ and $\mathcal{G}$ be finite non-empty sets. Let $\{H_\tau\}_{\tau\in\mathsf{T}}$ be an indexed family of functions such that for each τ, $H_\tau : \mathcal{D} \to \mathcal{G}$. The index set T is considered to be the set of all keys and a particular τ from T is considered to be the key for H_τ. We define two kinds of probabilities associated with such a function family.

Collision probability: For distinct $x, x' \in \mathcal{D}$, the collision probability of $\{H_\tau\}_{\tau\in\mathsf{T}}$ for the pair (x, x') is defined to be $\Pr_\tau[H_\tau(x) = H_\tau(x')]$.

Differential probability: Suppose $\mathcal{G}$ is an additively written group. For distinct $x, x' \in \mathcal{D}$ and any $y \in \mathcal{G}$, the differential probability of $\{H_\tau\}_{\tau\in\mathsf{T}}$ for the triplet (x, x', y) is defined to be $\Pr_\tau[H_\tau(x) - H_\tau(x') = y]$.

In the above, the probabilities are taken over uniform random choices of τ from T.

The family $\{H_\tau\}$ is said to be ϵ-almost universal (ϵ-AU) if for all distinct x, x' in $\mathcal{D}$, the collision probability for the pair (x, x') is at most ϵ. The family $\{H_\tau\}$ is said to be ϵ-almost XOR universal (ϵ-AXU) if for all distinct x, x' in $\mathcal{D}$ and any $y \in \mathcal{G}$, the differential probability for the triplet (x, x', y) is at most ϵ.

In the following, $\mathbb{F}$ will denote a finite field. The group $\mathcal{G}$ will be instantiated as the additive group of $\mathbb{F}$. The two standard operations over $\mathbb{F}$ are multiplication and addition. For $x, y \in \mathbb{F}$, the product (resp. sum) of x and y will be denoted as xy (resp. $x + y$) as is conventional. If $\mathbb{F}$ is a field of characteristic two, then the sum of x and y will be denoted as $x \oplus y$.

2.1 Polynomial Hashing

For $\ell \geq 0$, the polynomial $\mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell)$ in the variable τ with $m_1, \ldots, m_\ell \in \mathbb{F}$ is defined as follows:

If $\ell = 0$, then $\mathsf{Horner}_\tau() = 0$; and for $\ell > 0$,

$$\left.\begin{array}{ll} \multicolumn{2}{l}{\mathsf{Horner}_\tau(m_1, m_2, \cdots, m_\ell)} \\ = & m_1\tau^{\ell-1} + m_2\tau^{\ell-2} + \cdots + m_{\ell-1}\tau + m_\ell \\ = & (((m_1\tau + m_2)\tau + m_3)\tau + \cdots + m_{\ell-1})\tau + m_\ell. \end{array}\right\} \quad (1)$$

Note that computing Horner on ℓ field elements requires $\ell - 1$ additions and $\ell - 1$ multiplications.

It is well known that $\{\mathsf{Horner}_\tau\}_{\tau\in\mathbb{F}}$, is $((\ell - 1)/\#\mathbb{F})$-AU. Further, the hash function $\{\tau\mathsf{Horner}_\tau\}_{\tau\in\mathbb{F}}$ is $(\ell/\#\mathbb{F})$-AXU.

2.2 BRW Hashing

In [3], Bernstein defined a family of polynomials based on previous work by Rabin and Winograd [18], later called the BRW polynomials in [19]. For $\mathfrak{l} \geq 0$, $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\mathfrak{l})$ with $m_1, \ldots, m_\mathfrak{l} \in \mathbb{F}$ is a polynomial in the variable τ and is defined as follows:

- $\mathsf{BRW}_\tau() = 0$;
- $\mathsf{BRW}_\tau(m_1) = m_1$;
- $\mathsf{BRW}_\tau(m_1, m_2) = m_1\tau + m_2$;
- $\mathsf{BRW}_\tau(m_1, m_2, m_3) = (\tau + m_1)(\tau^2 + m_2) + m_3$;
- $\mathsf{BRW}_\tau(m_1, m_2, \cdots, m_\mathfrak{l})$
  $= \mathsf{BRW}_\tau(m_1, \cdots, m_{t-1})(\tau^t + m_t) + \mathsf{BRW}_\tau(m_{t+1}, \cdots, m_\mathfrak{l})$;
  if $t \in \{4, 8, 16, 32, \cdots\}$ and $t \leq \mathfrak{l} < 2t$.

Suppose $\mathfrak{l} \geq 3$. Following [3], it can be shown that $\mathsf{BRW}_\tau(m_1, \ldots, m_\mathfrak{l})$ can be computed using $\lfloor \mathfrak{l}/2 \rfloor$ multiplications and $\lfloor \lg \mathfrak{l} \rfloor$ additional squarings to compute $\tau^2, \tau^4, \ldots$.

Let $d(\mathfrak{l})$ denote the degree of $\mathsf{BRW}_\tau(m_1, \ldots, m_\mathfrak{l})$. Then $d(\mathfrak{l}) = 2^{\lfloor \lg \mathfrak{l} \rfloor + 1} - 1$ [3] and so $d(\mathfrak{l}) \leq 2\mathfrak{l} - 1$ where the bound is achieved if and only if $\mathfrak{l} = 2^r$ for some $r \geq 2$ and $d(\mathfrak{l}) = \mathfrak{l}$ if and only if $\mathfrak{l} = 2^{r+1} - 1$ for some $r \geq 1$.

It has been proved in [3] that the map from $\mathbb{F}^\mathfrak{l}$ to $\mathbb{F}[\tau]$ given by

$$(m_1, \ldots, m_\mathfrak{l}) \longmapsto \mathsf{BRW}_\tau(m_1, \ldots, m_\mathfrak{l})$$

is injective. As a consequence, the hash function $\{\mathsf{BRW}_\tau\}_{\tau \in \mathbb{F}}$, $\mathsf{BRW}_\tau : (m_1, \ldots, m_\mathfrak{l}) \mapsto \mathsf{BRW}_\tau(m_1, \ldots, m_\mathfrak{l})$ is $(d(\mathfrak{l})/\#\mathbb{F})$-AU.

3 Combining BRW with Horner

Both $\{\mathsf{Horner}_\tau\}$ and $\{\mathsf{BRW}_\tau\}$ use a single key $\tau \in \mathbb{F}$. The number of multiplications in $\mathbb{F}$ required to evaluate the two functions, though, are different. For a message consisting of ℓ field elements, Horner can be evaluated using $\ell - 1$ multiplications, while for $\ell \geq 3$, BRW requires $\lfloor \ell/2 \rfloor$ multiplications plus $\lfloor \log_2 \ell \rfloor$ squarings. In theory, this difference makes BRW much faster than Horner.

The problem, however, is that the definition of BRW is recursive. It is possible to have a recursive implementation of BRW. The overhead of such an implementation will nullify the benefit of lesser number of multiplications. On the other hand, if ℓ is a fixed integer, then it is possible to have a very fast non-recursive implementation of BRW.

Horner on the other hand can handle arbitrary values of ℓ quite easily. So, it makes sense to try and combine BRW and Horner so that the benefits of both the approaches can be obtained. One top-level strategy for doing this is the following. Suppose the message is a bit string which is formatted into a sequence of blocks where each block is an element of the field $\mathbb{F}$. Divide the sequence of field elements into groups of η blocks (assuming that η divides the number of blocks in the message). Each such group will be called a super-block.

We fix the value of η. The function BRW is used to process each super-block. Each invocation of BRW on a superblock produces a field element. These field elements are processed using Horner. So, there are two levels of the hash function. At the lower level, the message is formatted into super-blocks and BRW is used to process the super-blocks, while at the upper level, Horner is used to process the outputs of the BRW invocations. Since the number of blocks in a super-block is fixed, a fast non-recursive implementation of BRW can be used to process the super-blocks. A fast implementation of

Horner can be used to combine the outputs of BRW calls. The number of multiplications required by this approach is a little greater than that of BRW and is significantly smaller than that of Horner.

An important issue that needs to be properly tackled is the size of the key for the hash function. Generic approaches to multi-level hash [22, 17, 20] require the key to have independent components for each level of the hash. For a two-level hash, this would normally require two independent field elements as the key. It is, however, desirable to use a single field element as the key. We show how this can be done.

Proposition 1. *Let η, ℓ be positive integers. For $M \in \mathbb{F}^{\eta\ell}$ write $M = (M_1, \ldots, M_\ell)$ where each $M_i \in \mathbb{F}^{\eta}$. We define $G_\tau(M_1, \ldots, M_\ell)$ to be a polynomial in τ in the following manner.*

$$\begin{aligned} & G_\tau(M_1, \ldots, M_\ell) \\ = \ & \mathsf{Horner}_{\tau^{d(\eta)+1}}\left(\mathsf{BRW}_\tau(M_1), \ldots, \mathsf{BRW}_\tau(M_\ell)\right) \\ = \ & \tau^{(d(\eta)+1)(\ell-1)}\mathsf{BRW}_\tau(M_1) + \tau^{(d(\eta)+1)(\ell-2)}\mathsf{BRW}_\tau(M_2) + \\ & \quad \cdots + \tau^{(d(\eta)+1)}\mathsf{BRW}_\tau(M_{\ell-1}) + \mathsf{BRW}_\tau(M_\ell). \end{aligned} \tag{2}$$

The following hold for the function G given by (2).

1. *The degree of G in τ is $\ell d(\eta) + \ell - 1$.*
2. *G injectively maps $\mathbb{F}^{\eta\ell}$ to $\mathbb{F}[\tau]$.*

Consequently, the hash family $\{G_\tau\}_{\tau\in\mathbb{F}}$ is $((\ell d(\eta) + \ell - 1)/\#\mathbb{F})$-AU and the hash family $\{\tau\, G_\tau\}_{\tau\in\mathbb{F}}$ is $((\ell d(\eta) + \ell)/\#\mathbb{F})$-AXU.

Proof. Since each $M_i \in \mathbb{F}^{\eta}$, the degree of $\mathsf{BRW}_\tau(M_i)$ is $d(\eta)$ and so the degree of the polynomial G is $(d(\eta) + 1)(\ell - 1) + d(\eta) = \ell d(\eta) + \ell - 1$. This proves the first point.

Each $M_i \in \mathbb{F}^{\eta}$ and so for all i, $\mathsf{BRW}_\tau(M_i)$ has degree $d(\eta)$. Let

$$\mathsf{BRW}_\tau(M_i) = \tau^{d(\eta)}c_{i,d(\eta)} + \tau^{d(\eta)-1}c_{i,d(\eta)-1} + \cdots + \tau c_{i,1} + c_{i,0}$$

for some $c_{i,d(\eta)}, \ldots, c_{i,1}, c_{i,0} \in \mathbb{F}$ which depend on M_i. Using this, we write

$$\begin{aligned} & G_\tau(M_1, \ldots, M_\ell) \\ = \ & \tau^{(d(\eta)+1)(\ell-1)+d(\eta)}c_{1,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-1)+1}c_{1,1} + \tau^{(d(\eta)+1)(\ell-1)}c_{1,0} \\ & +\tau^{(d(\eta)+1)(\ell-2)+d(\eta)}c_{2,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-2)+1}c_{2,1} + \tau^{(d(\eta)+1)(\ell-2)}c_{2,0} \\ & +\cdots+ \\ & +\tau^{(d(\eta)+1)(\ell-i)+d(\eta)}c_{i,d(\eta)} + \cdots + \tau^{(d(\eta)+1)(\ell-i)+1}c_{i,1} + \tau^{(d(\eta)+1)(\ell-i)}c_{i,0} \\ & +\cdots+ \\ & +\tau^{2d(\eta)+1}c_{\ell-1,d(\eta)} + \cdots + \tau^{d(\eta)+2}c_{\ell-1,1} + \tau^{d(\eta)+1}c_{\ell-1,0} \\ & +\tau^{d(\eta)}c_{\ell,d(\eta)} + \cdots + \tau c_{\ell,1} + c_{\ell,0}. \end{aligned}$$

Considered as a polynomial in τ, the coefficients of $G_\tau(M_1, \ldots, M_\ell)$ are $c_{i,j}$ with $1 \le i \le \ell$ and $0 \le j \le d(\eta)$. Due to the choice of the key for Horner to be $\tau^{d(\eta)+1}$, each $c_{i,j}$ is associated with a unique power of τ.

Let $M, M' \in \mathbb{F}^{\eta\ell}$ with $M \neq M'$. Write $M' = (M'_1, \ldots, M'_\ell)$ with each $M'_i \in \mathbb{F}^{\eta}$. Let $c'_{i,j}$ be the coefficients of the polynomial $G_\tau(M'_1, \ldots, M'_\ell)$. Since $M \neq M'$, there is an i such that $M_i \neq M'_i$.

Table 1: Summary of the features of the basic scheme, Horner and BRW for hashing $\eta\ell$ blocks with $\eta = 2^{r+1} - 1$ for some $r \geq 1$.

scheme	# sqr	# mult	AU bound
Horner	–	$\eta\ell - 1$	$(\eta\ell - 1)/\#\mathbb{F}$
BRW	$\lfloor \lg \eta\ell \rfloor$	$\lfloor \eta\ell/2 \rfloor$	$d(\eta\ell)/\#\mathbb{F}$
G_τ	$r+1$	$\ell(\eta+1)/2 - 1$	$(\eta\ell + \ell - 1)/\#\mathbb{F}$

From the injectivity property of BRW, it follows that $\mathsf{BRW}_\tau(M_i) \neq \mathsf{BRW}_\tau(M'_i)$ and so there is a $j \in \{0, 1, \ldots, d(\eta)\}$ such that $c_{i,j} \neq c'_{i,j}$. From this it follows that $G_\tau(M_1, \ldots, M_\ell) \neq G_\tau(M'_1, \ldots, M'_\ell)$. This shows the second point.

Since for distinct M and M', $G_\tau(M_1, \ldots, M_\ell)$ and $G_\tau(M'_1, \ldots, M'_\ell)$ are distinct and have the same degree, it follows that $G_\tau(M_1, \ldots, M_\ell) - G_\tau(M'_1, \ldots, M'_\ell)$ is a non-zero polynomial of degree at most $\ell d(\eta) + \ell - 1$. Consequently, the probability that $G_\tau(M_1, \ldots, M_\ell)$ is equal to $G_\tau(M'_1, \ldots, M'_\ell)$ is the probability that τ is a root of the non-zero polynomial $G_\tau(M_1, \ldots, M_\ell) - G_\tau(M'_1, \ldots, M'_\ell)$. The number of distinct roots of a non-zero polynomial over a field is at most its degree from which it follows that the required probability is at most $(\ell d(\eta) + \ell - 1)/\#\mathbb{F}$. This shows the AU property.

For the AXU property, we note that the degree of $\tau\, G_\tau(M_1, \ldots, M_\ell)$ is $\ell d(\eta) + \ell$. The rest of the argument is similar to that of the AU propery. □ □

A crucial point in the above construction and the proof is the choice of the appropriate power of τ as the key for Horner so that the injectivity of G_τ follows directly from the injectivity of BRW_τ. The key for BRW_τ is τ and the degree of BRW_τ in τ is $d(\eta)$. Based on this, the key for Horner is chosen to be $\tau^{d(\eta)+1}$. This ensures that during the computation of Horner, the BRW polynomials arising from distinct super-blocks do not "overlap".

For $\eta \geq 3$ suppose $r \geq 1$ is such that $2^r \leq \eta < 2^{r+1}$. Then $d(\eta) = 2^{r+1} - 1$. The number of multiplications required in evaluating (2) is given by the number of multiplications required to evaluate all the BRW invocations and the number of multiplications required to evaluate the single Horner invocation. Each BRW requires $\lfloor \eta/2 \rfloor$ multiplications and Horner requires $\ell - 1$ multiplications for a total of $\ell\lfloor \eta/2 \rfloor + \ell - 1$ multiplications. Additionally, $\lfloor \lg \eta \rfloor = r$ squarings are required to compute the powers $\tau^2, \ldots, \tau^{2^r}$ which are used for evaluating BRW; an additional squaring is required to compute the power $\tau^{d(\eta)+1} = \tau^{2^{r+1}}$ which is used as a key to Horner. So a total of $\lfloor \lg \eta \rfloor + 1$ squarings are required to compute all the required powers of τ.

For $\eta = 2^{r+1} - 1$ with $r \geq 1$, Table 1 compares the efficiency and security of G_τ with that of Horner and BRW. The ratio of the number of multiplications required by G_τ to that required by Horner is $(\ell(\eta+1) - 2)/(2(\ell\eta - 1))$ and the ratio of the number of multiplications required by BRW to that required by G_τ is $2\lfloor \eta\ell/2 \rfloor/(\ell(\eta+1) - 2)$. Suppose $\eta = 31$: the first ratio is $(16\ell - 1)/(31\ell - 1)$ which equals $1/2$ for $\ell = 1$ and has the limiting upper bound of $16/31 \approx 0.52$; the second ratio is $\lfloor 31\ell/2 \rfloor/(16\ell - 1)$ which equals 1 for $\ell = 1$ and decreases to about 0.97 as ℓ increases. So, for $\eta = 31$, the number of multiplications required by G_τ is about 50% to 52% of that required by Horner while the number of multiplications required by BRW is about 97% to 100% of that required by G_τ.

For $\eta = 31$, the AU bound for Horner is $(31\ell - 1)/\#\mathbb{F}$; the AU bound for G_τ is $(32\ell - 1)/\#\mathbb{F}$; and the AU bound for BRW is $d(31\ell)/\#\mathbb{F} = (2^{\lfloor \lg(31\ell) \rfloor + 1} - 1)/\#\mathbb{F}$. The AU bound for BRW is in general higher than the AU bound for G_τ. The two bounds can be equal, e.g. for $\ell = 1, 2, 4, 8, \ldots$. On the other hand, the AU bound for BRW can be about twice as large as the AU bound for G_τ, e.g. for $\ell = 9$, the bound for G_τ is $287/\#\mathbb{F}$ and the bound for BRW is $511/\#\mathbb{F}$.

Overall, G_τ allows a range of efficiency/security trade-offs between BRW and Horner. By choosing an appropriate value for η, it is possible to attain speed nearly equal to that of BRW with AU bound not too larger than Horner.

Multi-level hashing: The idea of using BRW at the lower level and Horner at the upper level can be extended to more than one level. The critical issue is to choose an appropriate power of τ as the key for each level. While this can be done, extending to more than two levels results in a rather complicated construction which would mainly be of theoretical, rather than any practical, interest. So, we did not pursue the idea of multi-level hashing.

4 Two-Level Hash Function

For practical applications, it is required to handle variable length strings. We show how to modify the construction in Proposition 1 to be able to do this. For concreteness, *in the rest of the paper we will fix the finite field $\mathbb{F}$ to be $\mathbb{F}_{2^n}$ for some positive integer n.* The ideas, on the other hand, are quite general and can be adapted to work with other finite fields.

The following notation will be used.

- Given a binary string S, let $\mathsf{len}(S)$ denote the length of S, i.e., $\mathsf{len}(S)$ is the number of bits in S.
- Given an integer i with $0 \le i \le 2^k - 1$, let $\mathsf{bin}_k(i)$ denote the k-bit binary representation of i.
- Given a positive integer n and a binary string S, let $\mathsf{pad}_n(S)$ denote $S||0^i$ where i is the minimum non-negative integer such that $\mathsf{len}(S) + i$ is divisible by n.

Let

$$\mathcal{D} \;=\; \bigcup_{i=0}^{2^n-1} \{0,1\}^i. \qquad (3)$$

The reason for the bound $2^n - 1$ on the length of the strings in $\mathcal{D}$ is that we require the binary representation of the length of any string in $\mathcal{D}$ to fit into an n-bit string. For $M \in \mathcal{D}$, we define a function $\mathsf{superBlks}_{n,\eta}(M)$ as follows. Consider $\mathsf{pad}_n(M)$ to be formatted into a sequence of n-bit blocks. Let ℓ be such that

$$\frac{\mathsf{len}(\mathsf{pad}_n(M))}{n} = \eta(\ell - 1) + \lambda \qquad (4)$$

for some $\lambda \in \{1, \ldots, \eta\}$. Then $\mathsf{pad}_n(M)$ consists of $\ell - 1$ full super-blocks and one possibly partial super-block. Let $\mathsf{superBlks}_{n,\eta}(M)$ denote these super-blocks and we write

$$\mathsf{superBlks}_{n,\eta}(M) = (M_1, \ldots, M_\ell)$$

where $M_1, \ldots, M_{\ell-1}$ are full super-blocks (consisting of exactly η n-bit blocks each) and M_ℓ is a possibly partial superblock (consisting of at most η n-bit blocks).

Theorem 1. *Let $\mathcal{D}$ be as given in (3). Define a hash family $\{\mathsf{Hash2L}_\tau\}_{\tau\in\mathbb{F}_{2^n}}$ where $\mathsf{Hash2L}_\tau : \mathcal{D} \to \{0,1\}^n$ such that for $M \in \mathcal{D}$,*

$$\begin{aligned} &\mathsf{Hash2L}_\tau(M) \\ &= \tau^2 \mathsf{Horner}_{\tau^{d(\eta)+1}}(\mathsf{BRW}_\tau(M_1), \ldots, \mathsf{BRW}_\tau(M_\ell)) \oplus \tau \mathsf{bin}_n(\mathsf{len}(M)) \end{aligned} \qquad (5)$$

where $(M_1, \ldots, M_\ell) = \mathsf{superBlks}_{n,\eta}(M)$.

Let M *and* M' *be distinct elements of* $\mathcal{D}$ *with* $\mathsf{len}(M) \geq \mathsf{len}(M')$. *For a uniform random* $\tau \in \mathbb{F}_{2^n}$ *and any* $\beta \in \mathbb{F}_{2^n}$

$$\Pr_\tau \left[\mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') = \beta\right] \quad \leq \quad \frac{\ell(d(\eta)+1)+1}{2^n} \tag{6}$$

where ℓ *is the number of super-blocks in* M.

Proof. The proof follows if we can show that $\mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') \oplus \beta$ is a non-zero polynomial in τ of degree at most $\ell(d(\eta)+1)+1$. The maximum degree of $\mathsf{Hash2L}_\tau(M)$ as a polynomial in τ is $\ell d(\eta) + \ell - 1 + 2 = \ell(d(\eta)+1)+1$. So, we only need to argue that $P = \mathsf{Hash2L}_\tau(M) \oplus \mathsf{Hash2L}_\tau(M') \oplus \beta$ is a non-zero polynomial.

First suppose that $\mathsf{len}(M) \neq \mathsf{len}(M')$. The coefficient of τ in $\mathsf{Hash2L}_\tau(M)$ is $\mathsf{bin}_n(\mathsf{len}(M))$ and the coefficient of τ in $\mathsf{Hash2L}_\tau(M')$ is $\mathsf{bin}_n(\mathsf{len}(M')) \neq \mathsf{bin}_n(\mathsf{len}(M))$. So, P is a non-zero polynomial in τ.

So, suppose that $\mathsf{len}(M) = \mathsf{len}(M')$. Then $\ell = \ell'$ and an argument similar to that provided for Proposition 1 shows that P is a non-zero polynomial in τ. The only difference with the argument in Proposition 1 is that the last super-blocks M_ℓ and M'_ℓ may be partial. This, however, does not affect the argument, since the property that $M_\ell \neq M'_\ell$ implies $\mathsf{BRW}_\tau(M_\ell) \neq \mathsf{BRW}_\tau(M'_\ell)$ is preserved. □ □

Remark: The manner in which $\mathsf{Hash2L}_\tau(M)$ has been defined ensures the AXU property. If only the AU property is desired, then one can define $\mathsf{Hash2L}_\tau(M)$ to be

$$\tau\mathsf{Horner}_{\tau^{d(\eta)+1}}(\mathsf{BRW}_\tau(M_1), \ldots, \mathsf{BRW}_\tau(M_\ell)) \oplus \mathsf{bin}_n(\mathsf{len}(M)).$$

This requires one less multiplication.

4.1 Hashing a Vector of Strings

The hash family $\mathsf{Hash2L}$ handles a single binary string. We show how to extend it to handle a vector where each component is a binary string.

The parameters n and η are defined as in the case of $\mathsf{Hash2L}$. We define the hash family

$$\{\mathsf{vecHash2L}_\tau\}_{\tau \in \mathbb{F}_{2^n}} \text{ such that } \mathsf{vecHash2L}_\tau : \mathcal{VD} \to \mathbb{F}_{2^n}. \tag{7}$$

The domain $\mathcal{VD}$ consists of variable length vectors of binary strings. Formally,

$$\mathcal{VD} \quad = \quad \bigcup_{k=0}^{255} \left\{(M_1, \ldots, M_k) : 0 \leq \mathsf{len}(M_i) \leq 2^{n-16} - 1\right\}. \tag{8}$$

The reason for the bound $k \leq 255$ is that we require the binary representation of k to fit into a byte. Similarly, the reason for the bound $\mathsf{len}(M_i) \leq 2^{n-16} - 1$ is that we require the binary representation of the length of any M_i to fit into $n - 16$ bits. If $k = 0$, then the input is the empty list. Note that this input is different from the input where $k = 1$ and M_1 is the empty string.

The computation of the output of $\mathsf{vecHash2L}_\tau$ is shown in Table 2.

Theorem 2. *Let* $k \geq k' \geq 0$; $\mathbf{M} = (M_1, \ldots, M_k)$ *and* $\mathbf{M}' = (M'_1, \ldots, M'_{k'})$ *be two distinct vectors in* $\mathcal{VD}$. *For a uniform random* $\tau \in \mathbb{F}_{2^n}$ *and for any* $\beta \in \mathbb{F}_{2^n}$,

$$\Pr_\tau \left[\mathsf{vecHash2L}_\tau(\mathbf{M}) \oplus \mathsf{vecHash2L}_\tau(\mathbf{M}') = \beta\right] \leq \frac{\max\left(k + (d(\eta)+1)\Lambda, k' + (d(\eta)+1)\Lambda'\right)}{2^n} \tag{9}$$

where $\Lambda = \sum_{i=1}^{k} \ell_i$ *and* $\Lambda' = \sum_{i=1}^{k'} \ell'_i$.

Table 2: Computation of vecHash2L.

```
$\mathsf{vecHash2L}_\tau(M_1, \ldots, M_k)$
    if $k == 0$ return $1^n\tau$;
    $\mathsf{digest} = 0^n$;
    for $i = 1, \ldots, k-1$ do
        $(M_{i,1}, \ldots, M_{i,\ell_i}) = \mathsf{superBlks}_{n,\eta}(M_i)$;
        $L_i = \mathsf{bin}_n(\mathsf{len}(M_i))$;
        for $j = 1, \ldots, \ell_i$ do
            $\mathsf{digest} = \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{i,j})$;
        end for;
        $\mathsf{digest} = \tau\mathsf{digest} \oplus L_i$;
    end for;
    $(M_{k,1}, \ldots, M_{k,\ell_k}) = \mathsf{superBlks}_{n,\eta}(M_k)$;
    $L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k))$;
    for $j = 1, \ldots, \ell_k$ do
        $\mathsf{digest} = \tau^{d(\eta)+1}\mathsf{digest} \oplus \mathsf{BRW}_\tau(M_{k,j})$;
    end for;
    $\mathsf{digest} = \tau\mathsf{digest} \oplus L_k$;
    $\mathsf{digest} = \tau\mathsf{digest}$;
    return $\mathsf{digest}$.
```

Proof. Quantities corresponding to M' will have the superscript $'$.

For $i = 1, \ldots, k$ and $1 \leq j \leq \ell_i$, the degree of $\mathsf{BRW}_\tau(M_{i,j})$ is $d(\eta)$ for $1 \leq j < \ell_i$ and it is at most $d(\eta)$ for $j = \ell_i$. Write

$$\mathsf{BRW}_\tau(M_{i,j}) = c_{i,j,0} \oplus c_{i,j,1}\tau \oplus \cdots \oplus c_{i,j,d(\eta)}\tau^{d(\eta)}$$

where the c's depend on the M_i's. So, each M_i contributes at most $\ell_i(d(\eta)+1)+1$ coefficients

$$c_{i,1,0}, \ldots, c_{i,1,d(\eta)}, \ldots, c_{i,\ell_i,0}, \ldots, c_{i,\ell_i,d(\eta)}, L_i$$

to $\mathsf{vecHash2L}_\tau(\mathbf{M})$. The total number of coefficients is at most $k + (d(\eta)+1)\Lambda$. The last step in the digest computation increases the degree by one and so the maximum degree of $\mathsf{vecHash2L}_\tau(\mathbf{M})$ is equal to the maximum number of coefficients in $\mathsf{vecHash2L}_\tau(\mathbf{M})$. The degree of

$$P = \mathsf{vecHash2L}_\tau(\mathbf{M}) \oplus \mathsf{vecHash2L}_\tau(\mathbf{M}') \oplus \beta$$

is $\max(k + (d(\eta)+1)\Lambda, k' + (d(\eta)+1)\Lambda')$. The result follows if we can show that P is a non-zero polynomial in τ. The detailed proof is divided into several cases.

Case $k' = 0$: Since $\mathbf{M} \neq \mathbf{M}'$, it follows that $k > 0$. $\mathsf{vecHash2L}_\tau(M')$ equals $1^n\tau$. Since $k > 0$, $\mathsf{vecHash2L}_\tau(M)$ is of the form $L_k\tau \oplus \tau^2(\cdots)$ where

$$L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k)) \neq 1^n.$$

So, P is a non-zero polynomial.

Case $k > k' > 0$: In this case, $\mathsf{vecHash2L}_\tau(M)$ is of the form $L_k\tau \oplus \tau^2(\cdots)$ and $\mathsf{vecHash2L}_\tau(M')$ is of the form $L'_{k'}\tau \oplus \tau^2(\cdots)$ where

$$L_k = \mathsf{bin}_8(k)||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M_k)) \neq \mathsf{bin}_8(k')||0^8||\mathsf{bin}_{n-16}(\mathsf{len}(M'_{k'})) = L'_{k'}.$$

So, again P is a non-zero polynomial.

Case $k = k' > 0$: There are two subcases to consider.

Sub-case (a): There is some i such that $\mathsf{len}(M_i) \neq \mathsf{len}(M'_i)$. Let i be the maximum such value and so, $\mathsf{len}(M_j) = \mathsf{len}(M'_j)$ for $j = i+1,\ldots,k$. Since $\mathsf{len}(M_i) \neq \mathsf{len}(M'_i)$, it follows that $L_i \neq L'_i$. Let $s = 1 + (k-i) + \sum_{j=i+1}^{k} \ell_j(d(\eta)+1) = 1 + (k'-i) + \sum_{j=i+1}^{k'} \ell'_j(d(\eta)+1)$. Then the coefficient of τ^s in P is $L_i \oplus L'_i \neq 0$. So, again P is a non-zero polynomial.

Sub-case (b): In this case, $\mathsf{len}(M_i) = \mathsf{len}(M'_i)$ for $i = 1,\ldots,k$. As a result, in this case the number of components and the length of all the components in $\mathbf{M}$ and $\mathbf{M}'$ are equal. Since $\mathbf{M} \neq \mathbf{M}'$, it follows that there must be some s such that $M_s \neq M'_s$.

Since $\mathsf{len}(M_s) = \mathsf{len}(M'_s)$, it follows that the number of superblocks of M_s and M'_s are equal, i.e., $\ell_s = \ell'_s$. The super-blocks corresponding to M_s are $M_{s,1},\ldots,M_{s,\ell_s}$ while the super-blocks corresponding to M'_s are $M'_{s,1},\ldots,M'_{s,\ell_s}$. Since $M_s \neq M'_s$, at least one of the superblocks must be unequal. Let $t \in \{1,\ldots,\ell_s\}$ be such that $M_{s,t} \neq M'_{s,t}$. By the injectivity of BRW, it follows that $\mathsf{BRW}_\tau(M_{s,t}) \neq \mathsf{BRW}_\tau(M'_{s,t})$ and so there is a $k \in \{0,\ldots,d(\eta)\}$ such that $c_{s,t,k} \neq c'_{s,t,k}$. As a result, P is a non-zero polynomial.

This completes all the cases and the proof. $\square$ $\square$

5 Implementations Based on `pclmulqdq`

Our target platform were the Intel processors which support the `pclmulqdq` instruction. This instruction takes as input two degree 64 polynomials over $\mathbb{F}_2$ (represented as two 64-bit words) and returns as output the degree 128 polynomial which is the product of the two input polynomials. The implementation was done using Intel intrinsics.

We report implementations for $n = 128$ and $n = 256$. For $n = 128$, $\mathbb{F}_{2^{128}}$ was represented using the irreducible polynomial $\sigma(x) = x^{128} \oplus x^7 \oplus x^2 \oplus x \oplus 1$ and for $n = 256$, $\mathbb{F}_{2^{256}}$ was represented using the irreducible polynomial $\sigma(x) = x^{256} \oplus x^{10} \oplus x^5 \oplus x^2 \oplus 1$. In both cases, $\sigma(x)$ is of the form $x^n \oplus g_0(x)$ where $g_0(x)$ is a polynomial of degree less than $n/2$ having exactly 4 non-zero coefficients.

We report timings on two different machines. For the timing measurements, we followed the strategy of [13]. The first timing measurements were taken on a single core of a machine with Intel Core i7-4790 Haswell @ 3.60GHz. The second timing measurements were taken on a single core of a machine with Intel Core i7-6500U Skylake @ 2.5GHz. In both cases, the operating system was 64-bit Ubuntu-14.04-LTS and the C code was complied using GCC version 4.8.4. The code is publicly available[1].

5.1 Field Multiplication

The multiplication of two 128-bit polynomials using the schoolbook method requires 4 `pclmulqdq` calls and using Karatsuba's algorithm requires 3 `pclmulqdq` calls. The multiplication of two 256-bit

[1] `https://github.com/sebatighosh/HASH2L.git`

polynomials using the schoolbook method requires 16 `pclmulqdq` calls and using Karatsuba's algorithm requires 9 `pclmulqdq` calls. The reduction step can also be computed using `pclmulqdq` calls. From the work of Gueron and Kounavis [10] one obtains that for $n = 128$, 2 `pclmulqdq` calls are sufficient for the reduction while for $n = 256$, 4 `pclmulqdq` calls are sufficient. Details are provided in Section 5.2 below.

Batch multiplications: Suppose m independent multiplications are to be computed. The code can be arranged such that the `pclmulqdq` instructions for these multiplications can be grouped together. This may help the instruction scheduler to utilise instruction pipelining to speed up the computation. We have experimented with values of $m \leq 4$ and have found some speed improvements. In theory, the speed improvement should continue as m increases. In practice, however, this does not always happen.

5.2 Efficient Reduction

Let n be a positive even integer and $\mathbb{F}_{2^n}$ be represented by an irreducible polynomial $\sigma(x)$ of degree n over $\mathbb{F}_2$. Elements of $\mathbb{F}_{2^n}$ are represented using polynomials over $\mathbb{F}_2$ of degrees less than n. Let $a = a(x)$ and $b = b(x)$ be two elements of $\mathbb{F}_{2^n}$. The computation of $ab = a(x)b(x) \bmod \sigma(x)$ consists of two steps. In the first step, $a(x)$ and $b(x)$ are multiplied together to obtain a result $e(x)$ of degree less than $2n-1$ and then $e(x)$ is reduced modulo $\sigma(x)$ to obtain the desired result. Let $e(x) = a(x)b(x)$ and write $e(x) = d(x) \oplus c(x)x^n$ where $c(x)$ and $d(x)$ have degrees less than n. The essential task is to compute $c(x)x^n \bmod \sigma(x)$. A method for doing this was described by Gueron and Kounavis [10]. We review their method and determine the number of `pclmulqdq` instructions required for the reduction for both $n = 128$ and $n = 256$.

Let $q(x)$ and $p(x)$ be such that $c(x)x^n = q(x)\sigma(x) \oplus p(x)$ with $\deg(q), \deg(p) \leq n-1$. The goal is to find $p(x)$. Write $\sigma(x) = x^n \oplus \sigma^*(x)$, where $\deg(\sigma^*) \leq n-1$. The equation $c(x)x^n = q(x)\sigma(x) \oplus p(x)$ becomes $c(x)x^n = q(x)x^n \oplus q(x)\sigma^*(x) \oplus p(x)$ and so $p(x) = q(x)\sigma^*(x) \bmod x^n$. So, finding $q(x)$ is sufficient for obtaining $p(x)$.

Let $q^+(x)$ and $p^+(x)$ be such that $x^{2n} = q^+(x)\sigma(x) \oplus p^+(x)$ with $\deg(p^+) \leq n-1$ and $\deg(q^+) = n$. So,

$$\begin{array}{rrcl}
 & c(x)x^{2n} & = & q(x)\sigma(x)x^n \oplus p(x)x^n \\
\Rightarrow & c(x)(q^+(x)\sigma(x) \oplus p^+(x)) & = & q(x)\sigma(x)x^n \oplus p(x)x^n \\
\Rightarrow & c(x)q^+(x)\sigma(x) \oplus c(x)p^+(x) & = & q(x)\sigma(x)x^n \oplus p(x)x^n \\
\Rightarrow & \left\lfloor \frac{c(x)q^+(x)\sigma(x) \oplus c(x)p^+(x)}{x^{2n}} \right\rfloor & = & \left\lfloor \frac{q(x)\sigma(x)x^n \oplus p(x)x^n}{x^{2n}} \right\rfloor \\
\Rightarrow & \left\lfloor \frac{c(x)q^+(x)\sigma(x)}{x^{2n}} \right\rfloor & = & \left\lfloor \frac{q(x)\sigma(x)}{x^n} \right\rfloor .
\end{array}$$

In the above the following two facts have been used: $\deg(cp^+) \leq 2n-2$ and $\deg(p) \leq n-1$. Let $u(x)$ be of degree at most $n-1$, $v_1(x)$ of degree at most $2n-1$ and $v_2(x)$ of degree at most $n-1$ such that $c(x)q^+(x)\sigma(x) = u(x)x^{2n} \oplus v_1(x)$ and $q(x)\sigma(x) = u(x)x^n \oplus v_2(x)$. From this we obtain $c(x)q^+(x)\sigma(x)/x^n = u(x)x^n \oplus v_1(x)/x^n = q(x)\sigma(x) \oplus v_2(x) \oplus v_1(x)/x^n$. This is re-written as $c(x)q^+(x)/x^n = q(x) \oplus v_2(x)/\sigma(x) \oplus v_1(x)/\sigma(x)x^n$. Since $\deg(v_2) \leq n-1$, $\lfloor v_2(x)/\sigma(x) \rfloor = 0$ and since $\deg(v_1) \leq 2n-1$, $\lfloor v_1(x)/(\sigma(x)x^n) \rfloor = 0$. So, we obtain $q(x) = \lfloor c(x)q^+(x)/x^n \rfloor$.

Further simplifications: Suppose n is even, and $\sigma^*(x) = g_0(x)$ with $\deg(g_0) < n/2$. So, $\sigma(x) = x^n \oplus g_0(x)$. We have $x^n = \sigma(x) \oplus g_0(x)$ and so $x^{2n} = \sigma^2(x) \oplus g_0(x^2)$. Since $\deg(g_0) \leq n/2 - 1$, $\deg(g_0(x^2)) \leq n-2$. So, $q^+(x) = \sigma(x)$ and $p^+(x) = g_0(x^2)$. Write $c(x) = c_1(x)x^{n/2} \oplus c_0(x)$ where

$\deg(c_1), \deg(c_0) < n/2$. Consider the product

$$\begin{aligned} c(x)q^+(x) &= c(x)\sigma(x) = c(x)(x^n \oplus g_0(x)) = c(x)x^n \oplus c(x)g_0(x) \\ &= c(x)x^n \oplus (c_1(x)x^{n/2} \oplus c_0(x))g_0(x) = c(x)x^n \oplus x^{n/2}c_1(x)g_0(x) \oplus c_0(x)g_0(x). \end{aligned}$$

Since $\deg(c_0g_0) \leq n-2$, $\lfloor (c_0g_0)/x^n \rfloor = 0$ and we have

$$q(x) = \left\lfloor \frac{c(x)q^+(x)}{x^n} \right\rfloor = c(x) \oplus \left\lfloor \frac{x^{n/2}c_1(x)g_0(x)}{x^n} \right\rfloor = c(x) \oplus \left\lfloor \frac{c_1(x)g_0(x)}{x^{n/2}} \right\rfloor.$$

Write $q(x) = q_1(x)x^{n/2} \oplus q_0(x)$ with $\deg(q_1), \deg(q_0) < n/2$. Since $\deg(c_1), \deg(g_0) < n/2$, it follows that $\deg(c_1g_0) < n-1$ and so $\lfloor (c_1(x)g_0(x))/x^{n/2} \rfloor$ is a polynomial of degree less than $n/2$. So, we have $q(x) = c_1(x)x^{n/2} \oplus c_0(x) \oplus \lfloor (c_1(x)g_0(x))/x^{n/2} \rfloor$. In effect, $q_1(x) = c_1(x)$ and $q_0(x) = c_0(x) \oplus \lfloor (c_1(x)g_0(x))/x^{n/2} \rfloor$. Computing $q(x)$ requires computing $c_1(x)g_0(x)$ which accounts for one $n/2$-bit polynomial multiplication.

Given $q(x)$, $p(x)$ is obtained as $p(x) = q(x)\sigma^*(x) \bmod x^n = q(x)g_0(x) \bmod x^n = q_1(x)g_0(x)x^{n/2} \oplus q_0(x)g_0(x) \bmod x^n = c_1(x)g_0(x)x^{n/2} \oplus q_0(x)g_0(x) \bmod x^n$. The product $c_1(x)g_0(x)$ has already been computed. So, computing $p(x)$ requires another additional $n/2$-bit polynomial multiplication, namely $q_0(x)g_0(x)$. So, the entire reduction can be carried out using 2 $n/2$-bit polynomial multiplications. For $n = 128$, $n/2 = 64$ and the two $n/2$-bit polynomial multiplications can be computed using 2 `pclmulqdq` calls. The entire reduction $e(x) \bmod \sigma(x)$ requires a total of 7 instructions. For $n = 256$, $n/2 = 128$ and an $n/2$-bit polynomial multiplication is a 128-bit polynomial multiplication. We choose $g_0(x)$ to have degree less than 64. Since $c_1(x)$ is a polynomial of degree less than 128, the product $c_1(x)g_0(x)$ can be computed using 2 `pclmulqdq` instructions. Similarly, the product $q_0(x)g_0(x)$ can also be computed using 2 `pclmulqdq` instructions. So, 4 `pclmulqdq` instructions are sufficient for the reduction and the code for computing $e(x) \bmod \sigma(x)$ requires a total of 14 instructions.

5.3 Arithmetic Operations for Computing BRW

Let $\eta = 2^{r+1}-1 \geq 3$. Suppose A_{r+1} is the number of field additions required to evaluate $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$. Then $A_{r+1} = 2 + 2A_r$, $r \geq 2$ and using $A_2 = 3$, we have $A_{r+1} = 5 \cdot 2^{r-1} - 2$ for $r \geq 1$.

The number of multiplications for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ is $\lfloor \eta/2 \rfloor = 2^r - 1$. Two field elements β and γ are represented using polynomials over $\mathbb{F}_2$ of degrees less than n. Let us denote these polynomials as $\beta(x)$ and $\gamma(x)$. As described above, the computation of $\beta\gamma$ is done in two steps, namely a polynomial multiplication followed by a reduction.

For computing BRW, it is possible to reduce the number of reductions. We describe this with respect to $\mathbb{F}_{2^n}$, but, the general idea also applies to other fields. While computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $\eta = 2^{r+1}-1 \geq 3$, the product of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^r-1})$ and $(\tau^{2^r} + m_{2^r})$ is added to $\mathsf{BRW}_\tau(m_{2^r+1}, \ldots, m_{2^{r+1}-1})$. This involves a reduction step in the computation of the product $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^r-1})(\tau^{2^r} + m_{2^r})$ and a reduction step in the computation of the output of $\mathsf{BRW}_\tau(m_{2^r+1}, \ldots, m_{2^{r+1}-1})$. These two reductions can be combined into a single reduction in the following manner. Perform the polynomial multiplication of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^r-1})$ and $(\tau^{2^r} + m_{2^r})$; compute $\mathsf{BRW}_\tau(m_{2^r+1}, \ldots, m_{2^{r+1}-1})$ without the final reduction; add the two polynomials; then perform a reduction on the resulting polynomial.

For $\eta = 2^{r+1} - 1 \geq 3$, let R_{r+1} be the number of reductions required to compute $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $R_2 = 1$. The computation of $\mathsf{BRW}_\tau(m_1, \ldots, m_{2^r-1})$ requires R_r reductions; the computation of $\mathsf{BRW}_\tau(m_{2^r+1}, \ldots, m_{2^{r+1}-1})$ without the final reduction requires $R_r - 1$ reductions; and there is a final reduction. So, $R_{r+1} = R_r + (R_r - 1) + 1 = 2R_r$ for $r \geq 2$, $R_2 = 1$ and we obtain $R_{r+1} = 2^{r-1}$.

# sqr	# n-bit XORs	# poly mult	# red	AU bnd
r	$14 \cdot 2^{r-2} - 4$	$2^r - 1$	2^{r-1}	$\eta/2^n$

Table 3: Efficiency and AU bound for $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ over $\mathbb{F}_{2^n}$ with $\eta = 2^{r+1} - 1 \geq 3$.

A field addition in $\mathbb{F}_{2^n}$ is XOR of two n-bit strings. In trying to reduce the number of reductions, the number of n-bit XORs go up. Unreduced quantities are $2n$-bit polynomials and adding together two such polynomials require 2 n-bit XORs. Further, the cross product terms of the different multiplications are first added together and then shifted. This requires an extra n-bit XOR per delayed reduction. Let N_{r+1} be the number of n-bit XORs required to evaluate $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ with $\eta = 2^{r+1} - 1 \geq 3$. Then $N_{r+1} = 2N_r + 4$ for $r \geq 2$ with $N_2 = 3$ so that $N_{r+1} = 14 \cdot 2^{r-2} - 4$.

The relevant parameters for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ along with the AU bound are summarised in Table 3.

5.4 Computing BRW Polynomials

For the actual implementation, for both $n = 128$ and $n = 256$, we set $\eta = 31$, i.e., the number of n-bit blocks in a super-block is 31. For the two-level hash function, the last super-block can be partial. So, we did separate implementations of BRW for handling number of blocks from 1 to 31. Below we provide the details for the BRW implementation for 31-block inputs.

On a 31-block input, BRW requires a total of 15 n-bit multiplications. There is some amount of parallelism in these multiplications. A convenient way to bring out this parallelism is to represent the BRW computation using a tree as has been done in [7]. Such a tree depicts the dependencies among the multiplications required for BRW computation. We omit the details of how the tree is constructed as these details are not directly relevant to the present work.

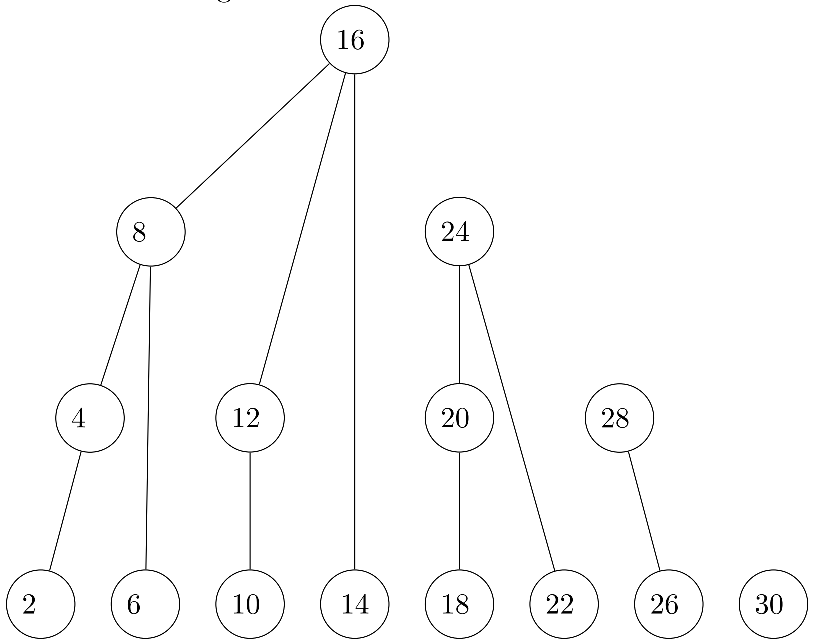
The relevant part of a 31-block BRW tree is shown in Figure 1. Each node is marked by an even number between 2 and 30 corresponding to the 15 multiplications that are required. (For the reason why the node labels are 2 to 30 instead of 1 to 15, we refer to [7].) If there is an edge from a lower marked node to a higher marked node, then the multiplication corresponding to the lower marked node has to be computed before the multiplication corresponding to the higher marked node. So, for example, the multiplication corresponding to node 2 has to be computed before the multiplication corresponding to node 4 can be computed and the multiplications corresponding to nodes 8, 12 and 14 have to be computed before the multiplication corresponding to node 16 can be computed.

Nodes which are not connected by an edge are independent and can be computed in parallel. For example, the eight multiplications at the lowest level are independent; the four multiplications at the next level are independent; and so on. There are, however, other ways to group the independent multiplications. Such groupings allow using batch multiplications to speed up the computations. Using batch size 3 as given below is particularly nice since the 15 multiplications can be cleanly grouped into 5 batches of 3 multiplications each.

Batch size 3: $\{2, 6, 10\}, \{14, 18, 22\}, \{26, 30, 4\}, \{20, 12, 8\}, \{28, 24, 16\}$.

In conjunction with the above, we also implemented the delayed reduction strategy described in Section 5.3. From Table 3, for $\eta = 31$, 15 multiplications of n-bit polynomials, 8 reductions and 52 n-bit XORs are required.

Figure 1: The 31-block BRW tree.



5.5 Decimated Horner

Given a sequence of n-bit blocks $m_1, \ldots, m_\ell$ and a positive integer $d \geq 1$, $\mathsf{Horner}_\tau(m_1, \ldots, m_\ell)$ can be computed as

$$\begin{aligned}
\mathsf{Horner}_\tau(m_1, \ldots, m_\ell) &= \tau^{\rho-1}\mathsf{Horner}_{\tau^d}(m_1, m_{d+1}, m_{2d+1}, \ldots) \\
&\quad \oplus \cdots \\
&\quad \oplus \tau^{\rho-\rho}\mathsf{Horner}_{\tau^d}(m_\rho, m_{d+\rho}, m_{2d+\rho}, \ldots) \\
&\quad \oplus \tau^{d-1}\mathsf{Horner}_{\tau^d}(m_{\rho+1}, m_{d+\rho+1}, m_{2d+\rho+1}, \ldots) \\
&\quad \oplus \cdots \\
&\quad \oplus \tau^{d-(d-\rho)}\mathsf{Horner}_{\tau^d}(m_d, m_{2d}, m_{3d}, \ldots) \qquad (10)
\end{aligned}$$

where $\rho = \ell \bmod d$. We call this d-decimated Horner computation. In (10), the d calls to Horner are independent leading to d independent multiplications at each step with the boundary conditions appropriately handled. These d independent multiplications can be computed as a batch multiplication. After the individual Horner calls are completed, the outputs are multiplied by $\tau^{\rho-1}, \ldots, 1, \tau^{d-1}, \ldots, \tau^{\rho}$ which can be done as a batch multiplication with batch size $d-1$ (since one multiplication is by 1).

5.6 Implementation of Hash2L

During implementation, there is a choice of batch size for BRW. For $n = 128$, we have found that choosing the batch size to be 3 provides slightly better speed compared to choosing the batch size to be 1. So, for $n = 128$, we implemented BRW using batch size 3 and 3-decimated Horner. For $n = 256$, however, there does not seem to be any noticeable improvement in speed by choosing the batch size to be greater than 1. So, in this case, we implemented both BRW and Horner using batch size 1.

	128-bit				256-bit			
	length of message in bytes				length of message in bytes			
	512	1024	4096	8192	512	1024	4096	8192
GHASH [9]	1.09	0.81	0.602	0.567	–	–	–	–
Hash2L	0.88 (19.27%)	0.687 (15.19%)	0.498 (17.28 %)	0.463 (18.3%)	1.4 –	0.95 –	0.718 –	0.67 –

Table 4: Cycles per byte for computing Hash2L and GHASH on Haswell. For both $n = 128$ and $n = 256$, Karatsuba gave better performance compared to the schoolbook method.

	128-bit				256-bit			
	length of message in bytes				length of message in bytes			
	512	1024	4096	8192	512	1024	4096	8192
GHASH [9]	0.79	0.55	0.369	0.339	–	–	–	–
Hash2L	0.667 (15.57%)	0.468 (14.9%)	0.33 (10.57%)	0.301 (11.2%)	1.11 –	0.758 –	0.562 –	0.525 –

Table 5: Cycles per byte for computing Hash2L and GHASH on Skylake. For $n = 128$, schoolbook was faster than Karatsuba, while for $n = 256$, Karatsuba was faster.

Timing results on the Haswell and the Skylake processors are presented in Tables 4 and 5 respectively. The percentage figures indicate the percentage of speed improvement obtained by our implementation of Hash2L over GHASH. The implementation of GHASH is by Gueron and has been taken from [9]. The implementation uses a delayed reduction strategy whereby a single reduction is done per eight polynomial multiplications. This strategy requires pre-computing a table consisting of 8 consecutive powers of the hash key. The code in [9] has been very carefully optimised. Both intrinsics and assembly codes are provided and it is mentioned that the performance of both the codes are similar. Since, we have implemented in intrinsics, we chose to compare to the intrinsics implementation in [9]. We measured the time required by the intrisics implementation of GHASH in [9] on the same machines where we measured the time required by Hash2L.

For timing both Hash2L and GHASH, the hash key was updated in each iteration. This ensured that the timing measurements included the time for pre-computing the powers of τ in case of Hash2L and the pre-computed table in Gueron's implementation of GHASH.

From the results in Tables 4 and 5 we find that Hash2L is about 15% to 19% faster than GHASH on the Haswell processor and is about 10% to 15% on the Skylake processor. In theory, the number of multiplications required by Hash2L is slightly more than half the number of multiplications required by GHASH. This does not directly turn into a roughly two times speed improvement for the following reasons.

First, the strategy of delayed reduction for GHASH used by Gueron to some extent mitigates the effect of requiring about two times as many multiplications. Second, the code for GHASH is simpler and smaller than that for Hash2L and this could have an effect on the overall speed. Third and perhaps the most important reason is that the code using the delayed reduction strategy in [9] is very carefully optimised. The ordering of instructions seem to have been carefully chosen to obtain the best possible speed. In this context, we note that [9] also provides an intrinsics implementation of GHASH without using delayed reduction. This code runs at over 2 cycles per byte for both Haswell and Skylake. The speed improvement obtained using the pre-computed table cannot be just explained by the algorithmic improvement of delayed reduction. It shows a very deep understanding of how the Intel processor executes the instructions.

To summarise, the speed improvements that we are able to achieve are indicative of the algorithmic superiority of Hash2L over GHASH. We do not claim that our code provides the fastest possible timing for Hash2L. Experts on intrinsics and assembly programming should be able to tune the code to achieve even higher speeds. Further, we have considered only $\eta = 31$ for implementation. It would be interesting to explore the speed achievable using other values of η. We leave these as interesting work for the future.

6 Implementation Strategy Without Using `pclmulqdq`

For $n = 256$, Bernstein and Chou [4] have provided a description of how to implement binary field arithmetic using the Fast Fourier Transform (FFT) algorithm. The method does not require the `pclmulqdq` instruction. The following counts of number of bit operations are provided in [4]. Forward Fourier transform: $4068 - 656 = 3412$ bit operations without radix conversions; pointwise multiplications: $64 \cdot 110$ bit operations; inverse Fourier transform: 5996 bit operations; reduction: 992 bit operations.

In the FFT based polynomial multiplication, the inverse Fourier transform is applied to the pointwise product. As pointed out in [4], to compute an expression of the type $a_1a_2 + b_1b_2$, it is equivalent to compute the pointwise multiplications for a_1, a_2 and b_1, b_2; add the vectors; and then perform a single inverse Fourier transform. So, whenever a sum of products of polynomials is to be computed, a single inverse Fourier transform suffices. In the present context, this means that the number of inverse Fourier transforms to be computed is equal to the number of reductions.

We consider the use of this strategy for computing Hash2L. The polynomial multiplication and reduction procedures used in [4] can be directly considered in the present context.

Suppose $\eta = 2^{r+1} - 1 \geq 3$. From Table 3, computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ requires $14 \cdot 2^{r-2} - 4$ 256-bit XORs; $2(2^r - 1)$ forward Fourier transforms (each polynomial multiplication requires two forward Fourier transforms); $2^r - 1$ pointwise multiplications; 2^{r-1} inverse Fourier transforms; and 2^{r-1} reductions. The total number of bit operations for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ comes to

$$\begin{aligned} 256(14 \cdot 2^{r-2} - 4) + 2 \cdot 3412 \cdot (2^r - 1) + (64 \cdot 110)(2^r - 1) + 5996 \cdot 2^{r-1} + 992 \cdot 2^{r-1} \\ = 18254 \cdot 2^r - 14888. \end{aligned} \tag{11}$$

The number of bits in $(m_1, \ldots, m_\eta)$ is $256\eta = 256 \cdot (2^{r+1} - 1)$ and so the number of bit operations per bit for computing $\mathsf{BRW}_\tau(m_1, \ldots, m_\eta)$ is

$$\mathfrak{B}_{r+1} = \frac{18254 \cdot 2^r - 14888}{256 \cdot (2^{r+1} - 1)}.$$

We have $\mathfrak{B}_2 \approx 28.2$, $\mathfrak{B}_3 \approx 32.4$, $\mathfrak{B}_4 \approx 34.2$, $\mathfrak{B}_5 \approx 34.9$, $\mathfrak{B}_6 \approx 35.3$, $\mathfrak{B}_7 \approx 35.5$.

For Hash2L having $\eta\ell$ 256-bit blocks, there are ℓ super-blocks consisting of η 256-bit blocks each. Processing these super-blocks require $\mathfrak{B}_{r+1}$ bit operations per block. Additionally, the ℓ blocks which are produced as the output of the ℓ BRW invocations are processed using Horner. For achieving AXU, this requires ℓ field multiplications. In the multiplications of Horner computation, one of the operands is always $\tau^{2^{r+1}}$ and so the number of forward Fourier transforms is $\ell + 1$ (one transform for each of the ℓ blocks, plus a transform for $\tau^{d(\eta)+1}$) instead of 2ℓ. In addition to these, there are ℓ pointwise multiplications; ℓ inverse Fourier transforms; ℓ reductions; and ℓ 256-bit XORs. Plugging in the number of bit operations for each of the aforementioned operations shows that a total of $17696\ell + 3412$ bit operations are required for evaluating Horner. Since there are a total of $\eta\ell$ 256-bit blocks, the number of bit operations per bit for evaluating Horner is $(17696\ell + 3412)/(256 \cdot \eta\ell) = 69.125/(2^{r+1} - 1) + 13.22/(\ell(2^{r+1} - 1))$.

There is an additional cost for computing the powers $\tau^2, \tau^4, \ldots, \tau^{2^{r+1}}$. Each of these is a squaring and requires 17440 bit operations for a total of $17440 \cdot r$ bit operations to compute all the powers. Amortised over the entire computation, the cost per bit for computing the powers is $(17440 \cdot r)/(256 \cdot \eta\ell)$.

So, the total number of bit operations per bit for computing Hash2L on $\eta\ell$ 256-bit blocks with $\eta = 2^{r+1} - 1$ is

$$\mathfrak{C}_{r+1} = \mathfrak{B}_{r+1} + 69.125/(2^{r+1} - 1) + (68.125 \cdot r + 13.22)/(\ell(2^{r+1} - 1)).$$

We have $\mathfrak{C}_2 \approx 51.2 + 27.1/\ell$, $\mathfrak{C}_3 \approx 42.3 + 21.4/\ell$, $\mathfrak{C}_4 \approx 38.8 + 14.5/\ell$, $\mathfrak{C}_5 \approx 37.2 + 9.2/\ell$, $\mathfrak{C}_6 \approx 36.4 + 5.6/\ell$, $\mathfrak{C}_7 \approx 36.0 + 3.3/\ell$.

Choosing $\eta = 31 = 2^5 - 1$ shows that the number of bit operations per bit for computing Hash2L is $37.2 + 9.2/\ell \leq 46.4$ By choosing $\eta = 63$ or 127, it is possible to lower the number of bit operations per bit though this is still greater than the 29 bit operations per bit required for the pseudo-dot product [4]. The main reason behind the cost of Hash2L being higher than that of the pseudo-dot product is that in the later case, there is a single inverse Fourier transform and a single reduction for the entire computation. The problem with the pseudo-dot product, however, is that the hash key is as long as the message. The cost of securely generating this key will be significant and has not been considered in [4].

Remark: The complete Hash2L requires an additional multiplication to process the block containing the message length. The above cost measure does not include this multiplication. The reason is that a complete hash function based on the pseudo-dot product will also require such a multiplication and this is not covered by the figure of 29 bit operations per bit reported in [4].

7 Message Authentication Code

A well known method for constructing a nonce-based MAC scheme from a hash function is the following [24]. Let $F : \mathcal{K} \times \mathcal{N} \rightarrow \{0,1\}^n$ be a mapping and $\{H_\tau\}_{\tau \in \mathsf{T}}$ with $H_\tau : \mathcal{M} \rightarrow \{0,1\}^n$ be a hash family. The key space for the MAC scheme is $\mathcal{K} \times \mathsf{T}$, the nonce space is $\mathcal{N}$ and the message space is $\mathcal{M}$. Given a nonce N and a message M, the output of the MAC scheme under a key (K, τ) is

$$(N, M) \xrightarrow{(K,\tau)} F_K(N) \oplus H_\tau(M). \tag{12}$$

It is possible to instantiate the hash function H using Hash2L. In this case, the message M is a binary string. More generally, it is possible to instantiate H using vecHash2L in which case the message M is a vector where each component is a binary string. The function F_K can be either a block cipher or a stream cipher.

Analysis of this scheme under the assumption that F is either a pseudo-random function (PRF) or a pseudo-random permutation (PRP) has a long history starting from [24] with the best known bounds appearing in [2]. If F is instantiated using a stream cipher, then security is based on the assumption that F is a PRF while if F is instantiated using a block cipher, then security is based on the assumption that F is a PRP. The overall security bound for the MAC scheme is obtained from the security assumption on F and the AXU bound on H_τ. These bounds are derived in [2] and so we do not repeat them here.

Instantiation at the 128-bit security level: It is possible to use a 128-bit block cipher such as AES to instantiate F_K. The size of K could be any of the options allowed for AES and the size of N will be 128 bits. It is also possible to instantiate F using a stream cipher whose key size is at least 128 bits.

Instantiation at the 256-bit security level: A 128-bit block cipher such as AES cannot be directly used to instantiate F at the 256-bit security level. Instead, a stream cipher supporting a 256-bit key can be directly used to instantiate F.

8 Comparison to Some Previous Works

We consider some of the important universal hash functions and corresponding MAC schemes that have been proposed. The discussion is divided into two parts. In the first part, we consider schemes for which the keys to the hash function are long and in the second part, we consider schemes for which the keys to the hash functions are short.

8.1 Comparison to Schemes Using Long Hash Keys

UMAC [5] and VMAC [11]: The core of the MAC scheme UMAC is the hash function NH^{T} which is based on integer arithmetic. This hash function processes an ℓ-block message with each block being w-bit long to produce a digest of size $2tw$ for some parameter $t \geq 1$. The construction is essentially the pseudo-dot product. The collision probability is 2^{-tw}. The hash key consists of $\ell + 2(t-1)$ w-bit blocks. So, the length of the hash key is longer than the length of the message to be hashed. The core of VMAC is the hash function VHASH which is also based on integer arithmetic and requires a key which is longer than the message.
Auth256 [4]: The core of Auth256 is the hash function Hash256. This hash function uses arithmetic over $GF(2^{256})$ to compute a pseudo-dot product. The key is as long as the message which results in collision probability being at most 2^{-256} and differential probability being at most 2^{-255}. The work [4] reports an implementation of Hash256 using a tower field representation and a new FFT-based algorithm for field multiplication. It does not use the `pclmulqdq` instruction on Intel processors.

A more recent proposal of a hash function which uses a long key is [14]. The idea of this construction is based on that of VHASH, except that the computation is over $\mathbb{F}_{2^{64}}$. The hash function produces 64-bit outputs. We note that more than 10 years ago, Bernstein had commented [23] that a 64-bit digest provides inadequate security.

For hash functions using long keys, in practice, the key has to be generated using either a stream cipher or a block cipher mode of operation. This leads to both efficiency and security issues as mentioned below.

Efficiency: Generation of the key can be either done on the fly, or, it could be pre-computed and stored. Both the approaches have problems. Generating the key on the fly requires significant additional time which should be included in the total time for hashing. However, the above mentioned schemes do not report this time. On the other hand, pre-computing and storing a large key has its own problems. To quote Bernstein [2], the large key "creates a huge speed penalty: cache misses become much more common and much more expensive."

Security: The analysis of the scheme given in (12) is well known when K and τ are chosen independently and uniformly at random with the best known bounds appearing in [2]. However, if τ is generated using a mode of operation of a block or a stream cipher, then there are two issues. If the key for the mode of operation used to generate τ is the same as that of F, then the independence condition is violated. Even if the key for the mode of operation is independent of the key for F, using a mode of operation to generate τ violates the uniform distribution property of τ. Consequently, if τ is generated using a mode of operation, the analysis and the bounds provided in [2] do not direcly apply and a fresh analysis and security bound need to be worked out. In fact, there has been a lengthy discussion on this issue [23] in the context of UMAC where Bernstein had strongly argued for the necessity of precise security statement and proof for UMAC. By the same reasoning, a precise security statement and proof is also required for Auth256 which is not available in [4].

While the above issues are relevant for hash functions which use long keys, we note below two issues which are particularly relevant to Hash256 and Auth256.

1. Hash256 avoids using `pclmulqdq` under the rationale that not all processors provide this instruction. Consider this issue in conjunction with the requirement of generating the hash key using AES in counter mode. Processors which do not provide an instruction similar to `pclmulqdq` are unlikely to provide support in the instruction set for AES. So, on such processors, the generation of the hash key will take significantly more time than the actual hashing. This time is neither reported nor considered in [4].

2. The digest size of Hash256 is 256 bits and so the goal of Auth256 is the 256-bit security level. It is suggested in [4] that the hash key can be generated using counter-mode AES. Since AES is a 128-bit block cipher, a direct use of counter-mode AES will not provide security at the 256-bit level. So, a combination of Hash256 with counter-mode AES is unlikely to provide 256-bit security. A further issue is that of instantiating F in (12) using AES. The output of F is required to be 256 bits long and since AES is a 128-bit cipher, it cannot be directly used to instantiate F. Since [4] does not provide a clear description of how the hash key for Hash256 is to be generated and how F is to be instantiated, the acutal security claim of Auth256 at the 256-bit level is unclear.

In terms of efficiency, [4] reports a cost of 29 bit operations per bit for computing Hash256 along with a hidded cost of generating the hash key. Any secure method for generating the long hash key will have a significant cost. In Section 6, we have shown that choosing $\eta = 31$ leads to a cost of at most 46.4 bit operations per bit. There is, however, no associated hidden cost of generating the hash key. The cost can be made lower by choosing a higher value of η. While the comparison in terms of bit operation counts is indicative, it would have been better to obtain the actual speed measurements. Since the code for Hash256 is not (yet) publicly available, we were unable to do this.

8.2 Comparison to Schemes Using Short Hash Keys

Poly1305 [1]: This is a usual univariate polynomial hashing using Horner. The arithmetic is over the prime field $\mathbb{F}_p$ with $p = 2^{130} - 5$. Clever use of floating point techniques are made to provide efficient implementation. For Haswell, the best reported speed we could find is 0.65 cycles/byte using 64-bit AVX2 instructions[2]. For Skylake, we were unable to locate a speed report.
GHASH [15]: This is also a usual univariate polynomial hashing using Horner. In this case, the arithmetic is over the field $\mathbb{F}_{2^{128}}$. Since this hash function forms a part of the NIST standard there has been much research in efficient implementation of this function. In fact, one of the reasons for Intel to include the

[2] `https://www.openssl.org/blog/blog/2016/02/15/poly1305-revised/`

pclmulqdq instruction is to be able to efficiently implement GHASH. The best known highly optimised implementation of GHASH using pclmulqdq is by Gueron [9].

Both Poly1305 and GHASH are usual Horner evaluations and hence, require $\ell - 1$ multiplications for evaluating an ℓ-block message. The design approach proposed here, on the other hand, requires a little more than $\ell/2$ multiplications. So, inherently this approach is faster than GHASH or Poly1305. We have instantiated this approach over binary fields to develop Hash2L. On the other hand, if one wishes to work over prime fields, it is equally possible to instantiate the approach over any appropriate field such as $\mathbb{F}_{2^{130}-5}$.

The hash key for Poly1305, GHASH and also Hash2L is a single field element. So, in terms of key agility, there is no difference between these three algorithms. The collision probabilities for Poly1305 and GHASH are those obtained from the usual Horner style hash and hence are only slightly lower than that of Hash2L. See Table 1 for more details.

Both Poly1305 and GHASH are designed for the 128-bit security level. The instantiation of Hash2L at the 128-bit security level turns out to be faster than both these functions on Haswell processor of Intel; it is faster than GHASH on Skylake; and we were unable to locate a speed report for Poly1305 on Skylake. We expect the 128-bit version of Hash2L to be faster than GHASH on any platform and to be faster than Poly1305 on any processor which supports the pclmulqdq instruction. The comparison of Hash2L to Poly1305 on processors which do not provide support for pclmulqdq cannot be determined without getting into the details of a particular processor.

9 Conclusion

In this work, we have shown how to combine the BRW family of polynomials with the Horner based polynomial evaluation to design a new hash function. The number of multiplications required for computing the digest is a little more than that for BRW polynomials. The advantage is that the implementation difficulties of BRW polynomials for variable length messages are eliminated. The combination is a two-level hash with BRW at the lower level and Horner at the higher level. The hash key is a single field element and has been appropriately used to work for both the levels. Concrete instantiations of the hash function over binary fields have been reported. The idea, on the other hand, is quite general and applies to other fields as well. A possible future work is to explore this idea to build concrete hash functions over other finite fields.

References

[1] 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.

[2] 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.

[3] Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. http://cr.yp.to/papers.html#pema.

[4] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field macs. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st*

International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers, volume 8781 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[5] 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.

[6] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[7] Debrup Chakraborty, Cuauhtemoc Mancillas-López, Francisco Rodríguez-Henríquez, and Palash Sarkar. Efficient hardware implementations of BRW polynomials and tweakable enciphering schemes. *IEEE Trans. Computers*, 62(2):279–294, 2013.

[8] Edgar N. Gilbert, F. Jessie MacWilliams, and Neil J. A. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53:405–424, 1974.

[9] Shay Gueron. AES-GCM-SIV implementations (128 and 256-bit). `https://github.com/Shay-Gueron/AES-GCM-SIV`, 2016.

[10] Shay Gueron and Michael E. Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Inf. Process. Lett.*, 110(14-15):549–553, 2010.

[11] Ted Krovetz and Wei Dai. VMAC: Message authentication code using universal hashing. `http://www.fastcrypto.org/vmac/draft-krovetz-vmac-01.txt`, 2007.

[12] 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.

[13] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[14] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptographic Engineering*, 6(3):171–185, 2016.

[15] 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.

[16] Mridul Nandi. On the minimum number of multiplications necessary for universal hash functions. In *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, pages 489–508, 2014.

[17] Wim Nevelsteen and Bart Preneel. Software performance of universal hash functions. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 24–41, 1999.

[18] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972.

[19] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4759, 2009.

[20] Palash Sarkar. A trade-off between collision probability and key size in universal hashing using polynomials. *Des. Codes Cryptography*, 58(3):271–278, 2011.

[21] Palash Sarkar. A new multi-linear universal hash family. *Des. Codes Cryptography*, 69(3):351–367, 2013.

[22] Douglas R. Stinson. Universal hashing and authentication codes. *Des. Codes Cryptography*, 4(4):369–380, 1994.

[23] UMAC. CFRG discussion on UMAC. `http://marc.info/?l=cfrg&m=143336318427068&w=2`, 2005.

[24] 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.

[25] Shmuel Winograd. A new algorithm for inner product. *IEEE Transactions on Computers*, 17:693–694, 1968.