Polynomial Hashing over Prime Order Fields

Sreyosi Bhattacharyya[1], Kaushik Nath[1], and Palash Sarkar[1]

[1]Indian Statistical Institute, 203, B.T. Road, Kolkata, India 700108
Emails: bhattacharyya.sreyosi@gmail.com, kaushik.nath@yahoo.in, palash@isical.ac.in

October 29, 2023

Abstract

This paper makes a comprehensive study of two important strategies for polynomial hashing over a prime order field $\mathbb{F}_p$, namely usual polynomial based hashing and hashing based on Bernstein-Rabin-Winograd (BRW) polynomials, and the various ways to combine them. Several hash functions are proposed and upper bounds on their differential probabilities are derived. Concrete instantiations are provided for the primes $p = 2^{127} - 1$ and $p = 2^{130} - 5$. A major contribution of the paper is an extensive 64-bit implementation of all the proposed hash functions in assembly targeted at modern Intel processors. The timing results suggest that using the prime $2^{127} - 1$ is significantly faster than using the prime $2^{130} - 5$. Further, a judicious mix of the usual polynomial based hashing and BRW-polynomial based hashing can provide a significantly faster alternative to only usual polynomial based hashing. In particular, the timing results of our implementations show that our final hash function proposal for the prime $2^{127} - 1$ is much faster than the well known Poly1305 hash function defined over the prime $2^{130} - 5$, achieving speed improvements of up to 40%.

Keywords: almost XOR universal hash function, polynomial hash, BRW hash.

Mathematics Subject Classification: 94A60

Contents

1 Introduction

Let X be a binary string. A polynomial hashing strategy constructs a univariate polynomial over a finite field from X and evaluates the polynomial at a secret point of the field, called the key, to obtain the digest. The security requirement on such a hash function is that all differential probabilities are provably small, where a differential probability is the probability, over a random choice of the key, that the difference of the digests corresponding to two different strings is equal to an arbitrary element of the field. Such hash functions have important applications to cryptography including the construction of authentication schemes, methods for authenticated encryption and disk encryption. The well known Poly1305 construction by Bernstein [3] is a polynomial hash function which is widely used[1].

One way to construct the univariate polynomial corresponding to X was independently put forward in three papers in 1993, namely den Boer [11], Taylor [24], and Bierbrauer, Johansson, Kabatianskii and Smeets [6]. The idea is to partition X into ℓ distinct blocks which are considered to be elements of the field and let these ℓ blocks be the coefficients of the polynomial. Poly1305 adopts this approach. Another method of constructing the univariate polynomial was put forward by Bernstein [4] in 2007 and is based on an earlier work by Rabin and Winograd [19]. The polynomials obtained using the method in [4] were called the BRW polynomials in [20]. To distinguish between the two methods of constructing the polynomials, we will denote the polynomials obtained by the method of [11, 24, 6] by Poly (and call these the usual polynomials) and denote the polynomials obtained by the method of [4] by BRW. The major theoretical advantage of BRW over Poly is that for the same string X, evaluation of the BRW polynomial constructed from X requires about half the number of field multiplications compared to the evaluation of the usual polynomial Poly constructed from X.

In this paper we make a comprehensive study of polynomial hashing based on both Poly and BRW and their various combinations over a prime order field $\mathbb{F}_p$. We define four hash functions over $\mathbb{F}_p$, one based only on Poly, a second one based only on BRW and two others which use a combination of BRW and Poly. Theoretical properties of these hash functions including explicit upper bounds on their differential probabilities are derived. From timing results, we observe that for short messages, the Poly based hash function has the best performance, while for longer messages it becomes slower than the other hash functions. To obtain the best performance for messages of all lengths, we put forward a combination hash function which applies the Poly based hash function for short messages and switches to another hash function for longer messages. We show that the combination hash function is secure by obtaining an upper bound on the corresponding differential probability.

[1] `https://en.wikipedia.org/wiki/Poly1305`, accessed on October 17, 2023

Concrete instantiations of the hash functions are proposed for the primes $2^{127}-1$ and $2^{130}-5$. The Poly based hash function for the prime $2^{130}-5$ coincides with Poly1305 for messages whose lengths are multiples of eight. The Poly based hash for the prime $2^{127}-1$ is new (see below for a discussion on previous proposals of usual polynomial based hashing for this prime). As far as we know, none of the other hash functions proposed in this work have been reported in the literature.

A major aspect of our work is implementation. Horner's rule is the usual method of evaluating Poly. A previous suggestion by Gueron [14] in the context of binary extension fields had put forward a method, which we call grouped Horner, to eliminate some operations from Horner's rule by pre-computing certain key powers and performing the evaluation of Poly in groups of coefficients. We implemented both basic and grouped Horner's rule for both the primes. The definition of BRW is recursive which makes it difficult to implement efficiently. A non-recursive algorithm to evaluate BRW was proposed in [12] and an implementation over binary extension fields was reported. We simplify the algorithm from [12]. The simplification makes substantial changes to the manner in which intermediate quantities are stored and accessed. As a result, the proof of correctness provided in [12] no longer applies to the modified algorithm. So we provide a detailed proof of correctness of the new algorithm. We have implemented this new algorithm for both the primes.

We have made 64-bit assembly implementation of all the hash functions for both the primes for all practical values of design and implementation parameters. The code is targeted towards the Intel Broadwell and later generation processors. In particular, along with the usual integer arithmetic instructions, we use the instructions `mulx`, `adcx` and `adox` which are available from the Broadwell processor onwards. To the best of our knowledge, there is no previous implementation of Poly1305 using these instructions.

The extensive implementation exercise provides some important insights.

1. The hash functions defined using $2^{127}-1$ are faster than the corresponding hash functions defined using $2^{130}-5$. In particular, the Poly based hash function over $2^{127}-1$ is faster than Poly1305 and the speed improvements range from 10% to 30%.

2. While in theory, BRW evaluation requires about half the number of field multiplications compared to Poly evaluation, this is not true in practice. Compared to Horner's rule based evaluation of Poly, BRW produces a speed improvement of 30% to 40% for messages which are not too short. If, however, grouped Horner is used for evaluating Poly, then the speed improvement becomes around 10%. While this is much less than what is predicted by theory, a speed improvement of 10% is indeed significant in practice.

Our final hash function proposal is a combination of Poly and BRW over the prime $2^{127}-1$. If the number of (appropriate sized) blocks in the message is less than 16, then Poly is used. If the number of blocks is at least 16, then a 2-level hash function is used, where BRW is used at the first level and Poly is used at the second level. Timing data from our implementations show that the new hash function is significantly faster than Poly1305, achieving speed improvements of about 8.5% (for 10-byte messages) to about 40% (for 5000-byte messages).

1.1 Related works

Hash families with provably low differential probabilities are called almost XOR universal (AXU) and are a generalisation of universal hash functions [7]. There is a large literature on universal hash functions and their generalisations. We refer to the relevant discussions in [2, 3, 4, 21, 22] for an

overview of this literature. Here we focus on the previous works which are related to the present paper.

The prime $2^{127}-1$ was suggested by Taylor [24] for instantiating Poly based hashing. Concrete instantiations were proposed by Bernstein [2] and Kohno, Viega and Whiting [15]. The construction in [2] used 32-bit coefficients and floating point arithmetic for implementation, while the construction in [15] used 96-bit coefficients and integer arithmetic for implementation. In contrast, we use 126-bit coefficients (obtained by padding 120-bit message blocks) and integer arithmetic for implementation. To speed-up evaluation, [2] used large precomputed tables ("few kilobytes of data for each key", as mentioned in [3]). In contrast, in our implementation, for speeding up the computation of Poly using grouped Horner with group size 8, only 128 bytes of pre-computed data are required for each key. Apart from [24, 2, 15], we know of no other previous work which considered the prime $2^{127}-1$ for polynomial hashing. In the Poly1305 paper [3], under 'Design decisions', Bernstein writes "I considered various primes above 2^{128}". No discussion is provided on the reason for discarding the prime $2^{127}-1$ used by Bernstein in his previous work [2]. Our proposal of Poly based hash function using $2^{127}-1$ and its implementation suggests that $2^{127}-1$ is a much faster option compared to $2^{130}-5$.

Concrete BRW based hash functions and their implementations in hardware and software have been proposed over binary extension fields [20, 9, 10, 8, 12]. To the best of our knowledge, till date there has been no concrete proposal of BRW based hash functions over prime order fields.

2 Preliminaries

Let $\mathcal{D}$ be a non-empty set, $(\mathcal{R},+)$ be a finite group and $\mathcal{K}$ be a finite non-empty set. Let $\{\mathsf{Hash}_\tau\}_{\tau\in\mathcal{K}}$ be a family of functions, such that for each $\tau\in\mathcal{K}$, $\mathsf{Hash}_\tau:\mathcal{D}\to\mathcal{R}$. The sets $\mathcal{D}$, $\mathcal{K}$ and $\mathcal{R}$ are called the message, key and tag (or digest) spaces respectively. For distinct $a,a'\in\mathcal{D}$ and $b\in\mathcal{R}$, the differential probability corresponding to (a,a',b) is defined to be $\mathrm{Pr}_\tau[\mathsf{Hash}_\tau(a)-\mathsf{Hash}_\tau(a')=b]$, where the probability is taken over a uniform random choice of τ from $\mathcal{K}$. If for every choice of distinct a,a' in $\mathcal{D}$ and $b\in\mathcal{R}$, the differential probability corresponding to (a,a',b) is at most ϵ, then we say that the family $\{\mathsf{Hash}_\tau\}_{\tau\in\mathcal{K}}$ is ϵ-almost XOR universal (ϵ-AXU).

Let $\mathbb{F}$ be a finite field. For a non-zero polynomial $P(x)\in\mathbb{F}[x]$, by $\mathsf{deg}(P(x))$ we will denote the degree of $P(x)$.

For $i\geq 0$ and $M_1,\ldots,M_i\in\mathbb{F}$, we define polynomials $\mathsf{Poly}(x;M_1,\ldots,M_i)$ and $\mathsf{BRW}(x;M_1,M_2,\ldots,M_i)$ in $\mathbb{F}[x]$ with indeterminate x and parameters $M_1,\ldots,M_i$ as follows.

$$\mathsf{Poly}(x;M_1,\ldots,M_i) \;=\; \begin{cases} 0, & \text{if } i=0;\\ M_1x^{i-1}+M_2x^{i-2}+\cdots+M_{i-1}x+M_i, & \text{if } i>0, \end{cases} \tag{1}$$

and

- $\mathsf{BRW}(x;)=0$;
- $\mathsf{BRW}(x;M_1)=M_1$;
- $\mathsf{BRW}(x;M_1,M_2)=M_1x+M_2$;
- $\mathsf{BRW}(x;M_1,M_2,M_3)=(x+M_1)(x^2+M_2)+M_3$;
- $\mathsf{BRW}(x;M_1,M_2,\ldots,M_i)=\mathsf{BRW}(x;M_1,\ldots,M_{k-1})(x^k+M_k)+\mathsf{BRW}(x;M_{k+1},\ldots,M_i)$; if $k\in\{4,8,16,32,\ldots\}$ and $k\leq i<2k$, i.e. k is the largest power of 2 such that $i\geq k$.

For $\tau\in\mathbb{F}$, using Horner's rule $\mathsf{Poly}(\tau;M_1,\ldots,M_i)$ can be evaluated using $i-1$ multiplications and same number of additions.

p	m	k	n
$2^{127}-1$	127	126	120
$2^{130}-5$	130	128	128

Table 1: The parameters m, k and n for the two values of p considered in this work.

For $i \geq 3$, $\mathsf{BRW}(x; M_1, M_2, \ldots, M_i)$ is a monic polynomial. The following has been proved in [4].

Theorem 1. *[4]*

1. *For every $i \geq 0$, the map from $\mathbb{F}^i$ to $\mathbb{F}[x]$ given by $(M_1, \ldots, M_i) \mapsto \mathsf{BRW}(x; M_1, \ldots, M_i)$ is injective.*
2. *For $i \geq 1$, let $\mathfrak{d}(i)$ denote $\mathsf{deg}(\mathsf{BRW}(x; M_1, \ldots, M_i))$. For $i \geq 3$, $\mathfrak{d}(i) = 2^{\lfloor \lg i \rfloor + 1} - 1$ and so $\mathfrak{d}(i) \leq 2i-1$; the bound is achieved if and only if $i = 2^a$; and $\mathfrak{d}(i) = i$ if and only if $i = 2^a - 1$ for some integer $a \geq 2$.*
3. *For $\tau \in \mathbb{F}$ and $i \geq 3$, $\mathsf{BRW}(\tau; M_1, \ldots, M_i)$ can be computed using $\lfloor i/2 \rfloor$ field multiplications and $\lfloor \lg i \rfloor$ additional field squarings to compute $\tau^2, \tau^4, \ldots$.*

3 Constructions

Let p be a prime and $\mathbb{F}_p$ be the finite field of order p. We will work with the primes $2^{127}-1$ and $2^{130}-5$. Given the prime p, we define the integers m, k and n as shown in Table 1. Elements of $\mathbb{F}_p$ can be represented as m-bit strings. Since k and n are less than m, we will consider k-bit and n-bit strings to represent elements of $\mathbb{F}_p$, where the most significant $m-k$ and $m-n$ bits respectively are set to 0. We will also adopt the usual convention that the binary representation of a non-negative integer is written with the least significant bit on the right. For a positive integer i and $0 \leq j < 2^i$, by $\mathsf{bin}_i(j)$ we will denote the i-bit binary representation of j. For example, if $i = 4$ and $j = 13$, then $\mathsf{bin}_i(j)$ is the string 1101. We will denote logarithm to base two by lg.

Formatting and padding: A binary string X of length $L \geq 0$ is formatted (or partitioned) into ℓ blocks $X_1, \ldots, X_\ell$, where the length of X_i is n for $1 \leq i \leq \ell - 1$, the length of X_ℓ is s with $1 \leq s \leq n$, and $X = X_1 || X_2 || \cdots || X_\ell$. Note that if X is the empty string, i.e. if $L = 0$, then $\ell = 0$. If the length of a block is n, then we call it a full block, otherwise we call it a partial block. By $\mathsf{format}(X)$ we will denote the list $(X_1, \ldots, X_\ell)$ obtained from X using the above described procedure. We describe two padding schemes.

- $\mathsf{pad1}(X_1, \ldots, X_\ell)$ returns $(M_1, \ldots, M_\ell)$, where $M_i = 0^{m-n-2} || 1 || X_i$, for $i = 1, \ldots, \ell - 1$, and $M_\ell = 0^{m-s-2} || 1 || X_\ell$.
- $\mathsf{pad2}(X_1, \ldots, X_\ell)$ returns $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, where $M_i = 0^{m-n-1} || X_i$, for $i = 1, \ldots, \ell - 1$, $M_\ell = 0^{m-s-1} || X_\ell$.

Remark 1. *For both the padding schemes, the length of each M_i, $i = 1, \ldots, \ell$, is $m-1$ and we consider M_i to be an element of $\mathbb{F}_p$. Also, $\mathsf{bin}_{m-1}(L)$ is considered to be an element of $\mathbb{F}_p$. If $\ell = 0$, then the input list is empty; $\mathsf{pad1}()$ returns the empty list, while $\mathsf{pad2}()$ returns a singleton list containing 0^{m-1}.*

In Section 4.1, we provide explanations for the choice of the padding schemes.

We describe four constructions of hash function families, namely polyHash, BRWHash, t-BRWHash, and d-2LHash. The key space and digest space for all the four families are the following. The key space is $\{0,1\}^k$; τ denotes the k-bit key which is considered to be an element of $\mathbb{F}_p$. The digest space is the group $(\mathbb{Z}_{2^k}, +)$. The message space for polyHash is the set of all binary strings, while the message space for the other three hash function families is the set of all binary strings of lengths less than 2^{m-1}. See, however, Remark 5 in Section 4 for further discussion on the message length.

In the descriptions below, X denotes a message which is a binary string of length $L \geq 0$.

Construction polyHash**:** Given a binary string X, let $(M_1, \ldots, M_\ell)$ be the output of $\mathsf{pad1}(\mathsf{format}(X))$. We define

$$\mathsf{polyHash}_\tau(X) \quad = \quad (P_1(\tau; M_1, \ldots, M_\ell) \bmod p) \bmod 2^k, \tag{2}$$

where $P_1(x; M_1, \ldots, M_\ell)$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$P_1(x; M_1, \ldots, M_\ell) \quad = \quad x \cdot \mathsf{Poly}(x; M_1, \ldots, M_\ell). \tag{3}$$

The family polyHash is motivated by the Poly1305 [3] construction and for the prime $2^{130} - 5$, if X is a sequence of bytes, then polyHash is exactly the same as Poly1305.

Construction BRWHash**:** Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$\mathsf{BRWHash}_\tau(X) \quad = \quad (P_2(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \tag{4}$$

where $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \quad = \quad x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)). \tag{5}$$

Construction t-BRWHash**:** This construction is parameterised by an integer $t \geq 2$. Let $\mathfrak{m} = \ell - (\ell \bmod 2^t)$. Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$t\text{-}\mathsf{BRWHash}_\tau(X) \quad = \quad (P_3(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \tag{6}$$

where $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a polynomial in $\mathbb{F}_p[x]$ defined as follows.

$$\begin{aligned} &P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &\quad = \quad x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_{\mathfrak{m}}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned} \tag{7}$$

Note that $\mathfrak{m}$ is a multiple of 2^t. The idea in t-BRWHash is to process the first $\mathfrak{m}$ blocks using BRW to obtain a single output and then combine this output with the leftover $\ell - \mathfrak{m}$ blocks and the length block using Poly.

Construction d-2LHash**:** This construction is parameterised by an integer $d \geq 2$. Let $\delta = 2^d - 1$ and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$. Given a binary string X, let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. We define

$$d\text{-}\mathsf{2LHash}_\tau(X) \quad = \quad (P_4(\tau; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^k, \tag{8}$$

where the polynomial $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ in $\mathbb{F}_p[x]$ is defined in the following manner. For $i = 1, \ldots, \mathfrak{n}$, let $U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta})$ and $V(x) = \mathsf{Poly}(x^{2^d}; U_1(x), U_2(x), \ldots, U_{\mathfrak{n}}(x))$. Then

$$P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \quad = \quad x \cdot \mathsf{Poly}(x; V(x), M_{\delta\mathfrak{n}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)). \tag{9}$$

The idea in d-2LHash is to divide the input sequence of blocks into groups of δ blocks, process each such block using BRW with key τ to obtain $\mathfrak{n}$ outputs and then combine these $\mathfrak{n}$ outputs using Poly with x substituted by $\gamma = \tau^{2^d}$. Finally the output of the Poly call, the left over blocks, and the length block are combined using another Poly call with x substituted by τ. This can be seen as two-level hashing. For binary extension fields, a similar two-level hash function was proposed in [8].

The choice of $\delta = 2^d - 1$ is motivated by the fact that $\mathfrak{d}(\delta) = \delta$, i.e. the degree of BRW on δ blocks is δ (see the second point of Theorem 1). A consequence of this is that the coefficients of the $U_i(x)$ are also the coefficients of $V(x)$. In more details, if we write $U_i(x) = x^\delta + u_{i,\delta-1}x^{\delta-1} + \cdots + u_{i,1}x + u_{i,0}$, then noting that $x^{2^d} = x^{\delta+1}$, we have

$$\begin{aligned} V(x) \quad = \quad & u_{\mathfrak{n},0} + u_{\mathfrak{n},1}x + \cdots + u_{\mathfrak{n},\delta-1}x^{\delta-1} + x^\delta \\ & + x^{\delta+1}\left(u_{\mathfrak{n}-1,0} + u_{\mathfrak{n}-1,1}x + \cdots + u_{\mathfrak{n}-1,\delta-1}x^{\delta-1} + x^\delta\right) \\ & + x^{2(\delta+1)}\left(u_{\mathfrak{n}-2,0} + u_{\mathfrak{n}-2,1}x + \cdots + u_{\mathfrak{n}-2,\delta-1}x^{\delta-1} + x^\delta\right) \\ & + \cdots \\ & + x^{(\mathfrak{n}-1)(\delta+1)}\left(u_{1,0} + u_{1,1}x + \cdots + u_{1,\delta-1}x^{\delta-1} + x^\delta\right). \end{aligned} \tag{10}$$

Remark 2.

1. *When $M_1, \ldots, M_\ell$ are clear from the context, we will write $P_1(x)$ instead of $P_1(x; M_1, \ldots, M_\ell)$. Similarly, when $M_1, \ldots, M_\ell$ and $\mathsf{bin}_{m-1}(L)$ are clear from the context, we will write $P_2(x)$, $P_3(x)$ and $P_4(x)$ instead of $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ respectively.*

2. *The motivation for considering t-BRWHash and d-2LHash is to avoid an implementation difficulty with BRWHash. See Section 5.3 for a discussion on this issue.*

3.1 Combining hash functions

Timing results show that for messages with a small number of blocks polyHash is faster than the other three hash functions, while for larger number of blocks, either t-BRWHash or d-2LHash is faster. If we use one of the hash functions, then either the performance for short messages is sub-optimal, or the performance for long messages is sub-optimal. In this section, we describe a construction which allows obtaining the best of both the cases.

Let $d \geq 2$ be a positive integer. Let X be a binary string of length $L \geq 0$. Let $(X_1, \ldots, X_\ell)$ be the output of $\mathsf{format}(X)$. We define

$$d\text{-}\mathsf{Hash}_\tau(X) \quad = \quad \begin{cases} \mathsf{polyHash}_\tau(X) & \text{if } \ell < 2^d; \\ d\text{-}\mathsf{2LHash}_\tau(X) & \text{if } \ell \geq 2^d. \end{cases} \tag{11}$$

Note that polyHash uses pad1 while d-2LHash uses pad2. So if $\ell < 2^d$, then pad1 is used on $(X_1, \ldots, X_\ell)$ and if $\ell \geq 2^d$, then pad2 is used on $(X_1, \ldots, X_\ell)$. We later show that this combination

of polyHash and d-2LHash produces a secure hash function. In (11), it is possible to replace d-2LHash with either BRWHash or d-BRWHash (i.e. t-BRWHash with the parameter t equal to d) and still obtain a secure hash function.

Remark 3. *The hash functions have the design parameters t and d. See Section 5.4 for a discussion on these parameters as well as other implementation parameters.*

3.2 Naming convention

We adopt the following naming convention. For each of the hash functions, there are two possible sets of parameters in Table 1. The choice of the prime p determines the values of m, k and n. So for each of the hash functions, by specifying the value of p, we obtain two different instantiations. If p is chosen to be $2^{127}-1$, we append 1271 to the name of the hash function, and if p is chosen to be $2^{130}-5$, we append 1305 to the name of the hash function. For example, polyHash1271 denotes polyHash computed modulo $2^{127}-1$ and BRWHash1305 denotes BRWHash computed modulo $2^{130}-5$. The naming convention will become important in Section 8 where we provide explicit timings for the various hash functions.

Our naming convention distinguishes between Poly which is used to denote the polynomial in (1) and the hash function polyHash given by (2) built from Poly after formatting and padding the message. We require this distinction since we use Poly as a component in the other hash functions. As a result of this distinction, the naming of polyHash1305 is different from Poly1305, even though the two hash functions are identical for messages whose lengths in bits are multiples of eight.

4 AXU bounds

The following result will be useful for obtaining AXU bounds for the four hash functions. This result is a generalisation of an observation used in the proof of Theorem 3.3 in [3].

Lemma 1. *Let $p = 2^m - \delta$ be a prime and k be a positive integer such that $k < m$ and $\delta < 2^k - 1$. (The values of p, m and k given in Table 1 satisfy these conditions.) Let $\alpha \in \mathbb{Z}_{2^k}$, and $P(x)$ and $P'(x)$ be distinct polynomials in $\mathbb{F}_p[x]$ satisfying $P(0) = P'(0) = 0$. The number of distinct $\tau \in \mathbb{F}_p$ such that*

$$((P(\tau) \bmod p) \bmod 2^k) - ((P'(\tau) \bmod p) \bmod 2^k) \equiv \alpha \pmod{2^k} \tag{12}$$

is at most 2^{m-k+1} times the degree of the polynomial $P(x) - P'(x)$.

Consequently, for τ chosen uniformly at random from $\{0,1\}^k$ (which is considered to be a subset of $\mathbb{F}_p$), the probability that (12) holds is at most $2^{m-2k+1} \cdot \mathsf{deg}(P(x) - P'(x))$.

Proof. Let U be the set of integers in the interval $[-2^m, 2^m - 1]$ which are congruent to α modulo 2^k. Then $i \cdot 2^k + \alpha$ is in U if and only if $-2^m \le i \cdot 2^k + \alpha \le 2^m - 1$, or equivalently $-2^{m-k} - \alpha/2^k \le i \le 2^{m-k} - (\alpha+1)/2^k$. Since $\alpha \in \mathbb{Z}_{2^k}$, the values that i can take are $-2^{m-k}, \ldots, 2^{m-k} - 1$ and hence $\#U = 2^{m-k+1}$.

Suppose (12) holds for some $\tau \in \mathbb{F}_p$. Note that $P(\tau) \bmod p$ and $P'(\tau) \bmod p$ are integers in the interval $[0, p-1]$. Write $(P(\tau) \bmod p) = a_1 2^k + a_0$ where $0 \le a_0 < 2^k$ and

$$0 \le a_1 = \lfloor (P(\tau) \bmod p)/2^k \rfloor \le \lfloor (p-1)/2^k \rfloor = \lfloor (2^m - (\delta+1))/2^k \rfloor = 2^{m-k} - 1.$$

The last equality holds since $\delta < 2^k - 1$. Similarly, write $(P'(\tau) \bmod p) \bmod 2^k = b_0$, where $0 \le b_0 < 2^k$ and $0 \le b_1 \le 2^{m-k} - 1$. From the bounds on a_1 and b_1, it follows that

$$-2^{m-k} \le a_1 - b_1, a_1 - b_1 - 1 \le 2^{m-k} - 1.$$

Since τ satisfies (12), we have $a_0 - b_0 \equiv \alpha \bmod 2^k$. So over the integers either $a_0 - b_0 = \alpha$ or $a_0 - b_0 = -2^k + \alpha$ according as $a_0 \ge b_0$ or $a_0 < b_0$ respectively. Let $\beta = 2^k(a_1 - b_1) + \alpha$ if $a_0 \ge b_0$, and $\beta = 2^k(a_1 - b_1 - 1) + \alpha$ if $a_0 < b_0$. Then $\beta \equiv \alpha \bmod 2^k$ and from the above bounds on $a_1 - b_1$ and $a_1 - b_1 - 1$, we have $\beta \in U$. So it follows that if τ satisfies (12), then $(P(\tau) \bmod p) - (P'(\tau) \bmod p) = 2^k(a_1 - b_1) + a_0 - b_0$ which is equal to $2^k(a_1 - b_1) + \alpha$ if $a_0 \ge b_0$ and is equal to $2^k(a_1 - b_1 - 1) + \alpha$ if $a_0 < b_0$. So $(P(\tau) \bmod p) - (P'(\tau) \bmod p) = \beta$. Then τ is a root of the polynomial $P(x) - P'(x) - \beta \pmod p$.

Let $R_\beta(x) = P(x) - P'(x) - \beta \in \mathbb{F}_p[x]$. Since $P(x)$ and $P'(x)$ are distinct polynomials and $P(0) = P'(0) = 0$, we have $R_\beta(x) \in \mathbb{F}_p[x]$ to be a non-zero polynomial whose degree is at most the degree of $P(x) - P'(x)$. Since $\beta \in U$, $\#U = 2^{m-k+1}$ and the number of roots of $R_\beta(x)$ over $\mathbb{F}_p$ is at most the degree of $R_\beta(x)$, it follows that the number of distinct τ such that (12) holds is at most 2^{m-k+1} times the degree of $R_\beta(x)$. □

Lemma 1 reduces the problem of determining the probability that a uniform random k-bit string τ satisfies (12) to the simpler problem of determining the degree of the polynomial $P(x) - P'(x) \in \mathbb{F}_p[x]$.

Lemma 2. *Let X be a binary string of length $L > 0$. Let $\ell = \lceil L/n \rceil$ and suppose $(M_1, \ldots, M_\ell)$ is the output of* $\mathsf{pad1}(\mathsf{format}(X))$. *Then $M_i \ne 0^{m-1}$ for $i = 1, \ldots, \ell$.*

Lemma 3. *Let X be a binary string of length $L \ge 0$. Then the maps $X \mapsto \mathsf{pad1}(\mathsf{format}(X))$ and $X \mapsto \mathsf{pad2}(\mathsf{format}(X))$ are injections.*

Proof. Let X and X' be two distinct binary strings of lengths L and L' respectively and we assume without loss of generality that $L \ge L' \ge 0$. Let $\ell = \lceil L/n \rceil$ and $\ell' = \lceil L'/n \rceil$. Let $(X_1, \ldots, X_\ell)$ be the output of $\mathsf{format}(X)$ and $(X'_1, \ldots, X'_{\ell'})$ be the output of $\mathsf{format}(X')$. Let s and s' be the lengths of X_ℓ and $X'_{\ell'}$ respectively.

We first consider $\mathsf{pad1}(\mathsf{format}(X))$. Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ is the output of $\mathsf{pad1}(\mathsf{format}(X'))$. If $\ell \ne \ell'$, then clearly $(M_1, \ldots, M_\ell) \ne (M'_1, \ldots, M'_{\ell'})$.

So suppose that $\ell = \ell'$. Since $X \ne X'$, there must be an i in $\{1, \ldots, \ell\}$ such that $X_i \ne X'_i$. If $1 \le i \le \ell - 1$, then both X_i and X'_i are full blocks and then $M_i = 0^{m-n-2}||1||X_i \ne 0^{m-n-2}||1||X'_i = M'_i$. So suppose that $i = \ell$. Then $M_\ell = 0^{m-s-2}||1||X_\ell$ and $M'_\ell = 0^{m-s'-2}||1||X'_\ell$. If $s \ne s'$, then the positions of the leading 1 in M_ℓ and M'_ℓ are different and so $M_\ell \ne M'_\ell$; on the other hand, if $s = s'$, then X_ℓ and X'_ℓ have the same length and since they are distinct, there must be a bit position where they differ, in which case we again have $M_\ell \ne M'_\ell$.

Next consider $\mathsf{pad2}(\mathsf{format}(X))$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ be the output of $\mathsf{pad2}(\mathsf{format}(X'))$. If $\ell \ne \ell'$, then the number of components in the two outputs are different and so the outputs are different. If $\ell = \ell'$ but $L \ne L'$, then $\mathsf{bin}_{m-1}(L) \ne \mathsf{bin}_{m-1}(L')$ and again the two outputs are different. So suppose $L = L'$ which implies $s = s'$. Since $X \ne X'$, there must be an i in $\{1, \ldots, \ell\}$ such that $X_i \ne X'_i$. If $1 \le i \le \ell - 1$, then $M_i = 0^{m-n-1}||X_i \ne 0^{m-n-1}||X'_i = M'_i$ and if $i = \ell$, then $M_\ell = 0^{m-s-1}||X_\ell \ne 0^{m-s-1}||X'_\ell = M'_\ell$. □

Lemma 4. *Let X be a binary string of length $L \geq 0$. Let $\ell = \lceil L/n \rceil$.*

1. *Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Then*

$$X \;\mapsto\; P_1(x; M_1, \ldots, M_\ell)$$

is an injection.

2. *Suppose $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the output of $\mathsf{pad2}(\mathsf{format}(X))$. Then the maps*

$$\begin{aligned} X &\mapsto P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)), \\ X &\mapsto P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)), \\ X &\mapsto P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \end{aligned}$$

are injections.

Proof. Let X and X' be two distinct binary strings of lengths L and L' respectively and we assume without loss of generality that $L \geq L' \geq 0$. Let $\ell = \lceil L/n \rceil$ and $\ell' = \lceil L'/n \rceil$.

Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ is the output of $\mathsf{pad1}(\mathsf{format}(X'))$. From Lemma 3, we have $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_{\ell'})$ and from Lemma 2, we have $M_i \neq 0^{m-1}$ and $M'_j \neq 0^{m-1}$ for $i = 1, \ldots, \ell$ and $j = 1, \ldots, \ell'$. If $L' = 0$, i.e. X' is the empty string, then X is not the empty string and we have $L > 0$ and so $\ell > 0$. Since $M_1 \neq 0^{m-1}$ it follows that $P_1(x; M_1, \ldots, M_\ell)$ is a non-zero polynomial whereas $P_1(x; M'_1, \ldots, M'_{\ell'}) = P_1(x;) = 0$. Now suppose that $L' > 0$ and so $\ell \geq \ell' \geq 1$. We have $P_1(x; M_1, \ldots, M_\ell) = M_1 x^\ell + \cdots + M_{\ell-1} x^2 + M_\ell x$ and $P_1(x; M'_1, \ldots, M'_{\ell'}) = M'_1 x^\ell + \cdots + M'_{\ell'-1} x^2 + M'_{\ell'} x$. If $\ell > \ell'$, then since $M_1 \neq 0^{m-1}$, it follows that $P_1(x; M_1, \ldots, M_\ell) \neq P_1(x; M'_1, \ldots, M'_{\ell'})$. If $\ell = \ell'$, since $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_{\ell'})$, it again follows that $P_1(x; M_1, \ldots, M_\ell) \neq P_1(x; M'_1, \ldots, M'_{\ell'})$.

Now we turn to the other three maps. Suppose $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ is the output of $\mathsf{pad2}(\mathsf{format}(X'))$. From Lemma 3, we have $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \neq (M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$.

Note that $\mathsf{bin}_{m-1}(L)$ is the coefficient of x in $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, while $\mathsf{bin}_{m-1}(L')$ is the coefficient of x in $P_2(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$, $P_3(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$ and $P_4(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L'))$. If $\mathsf{bin}_{m-1}(L) \neq \mathsf{bin}_{m-1}(L')$, then clearly,

$$\begin{aligned} P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_2(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')), \\ P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_3(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')), \\ P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &\neq P_4(x; M'_1, \ldots, M'_{\ell'}, \mathsf{bin}_{m-1}(L')). \end{aligned}$$

So henceforth suppose that $\mathsf{bin}_{m-1}(L) = \mathsf{bin}_{m-1}(L')$, i.e. $L = L'$ and so $\ell = \ell'$. Since $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is not equal to $(M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L))$, it follows that $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$. From this point, the arguments for $P_2(x)$, $P_3(x)$ and $P_4(x)$ are different.

Consider $P_2(x)$. Using the injectivity of BRW (see the first point of Theorem 1), we have

$$\begin{aligned} P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)) \\ &\neq x(x \cdot \mathsf{BRW}(x; M'_1, \ldots, M'_\ell) + \mathsf{bin}_{m-1}(L)) \\ &= P_2(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned}$$

Next consider $P_3(x)$. Since $\ell = \ell'$, it follows that $\mathfrak{m} = \mathfrak{m}'$. Using the injectivity of BRW and Poly, we have

$$\begin{aligned} & P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ = \;& x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_{\mathfrak{m}}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ \neq \;& x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M'_1, M'_2, \ldots, M'_{\mathfrak{m}}), M'_{\mathfrak{m}+1}, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)) \\ = \;& P_3(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned}$$

Finally consider $P_4(x)$. Since $\ell = \ell'$, it follows that $\mathfrak{n} = \mathfrak{n}'$. If $(M_{1+\delta(i-1)}, \ldots, M_{i\delta}) \neq (M'_{1+\delta(i-1)}, \ldots, M'_{i\delta})$ for some i in $\{1, \ldots, \mathfrak{n}\}$, then by the injectivity of BRW,

$$U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta}) \neq \mathsf{BRW}(x; M'_{1+\delta(i-1)}, \ldots, M'_{i\delta}) = U'_i(x).$$

By construction, the coefficients of $V(x)$ are the coefficients of the U_i's (see (10)) and so $U_i(x) \neq U'_i(x)$ implies that $V(x) \neq V'(x)$. On the other hand, if $(M_{1+\delta(i-1)}, \ldots, M_{i\delta}) = (M'_{1+\delta(i-1)}, \ldots, M'_{i\delta})$ for all i in $\{1, \ldots, \mathfrak{n}\}$, then since $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$, it follows that $(M_{\delta\mathfrak{n}+1}, \ldots, M_\ell) \neq (M'_{\delta\mathfrak{n}+1}, \ldots, M'_\ell)$. In either case, we have

$$\begin{aligned} & P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ = \;& \mathsf{Poly}(x; V(x), M_{\delta\mathfrak{n}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ \neq \;& \mathsf{Poly}(x; V'(x), M'_{\delta\mathfrak{n}+1}, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)) \\ = \;& P_4(x; M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L)). \end{aligned}$$

□

Lemma 5. *Let X be a binary string of length $L \geq 1$ and n be a positive integer. Let $\ell = \lceil L/n \rceil$.*

1. *Let $(M_1, \ldots, M_\ell)$ be the output of $\mathsf{pad1}(\mathsf{format}(X))$. Then $\mathsf{deg}(P_1(x; M_1, \ldots, M_\ell)) = \ell$.*
2. *Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$.*
   (a) *Then $\mathsf{deg}(P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + 2\ell$.*
   (b) *Let $t \geq 2$ be the parameter of t-$\mathsf{BRWHash}$. Then*

$$\mathsf{deg}(P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + 2\ell - (\ell \bmod 2^t) \leq 1 + 2\ell.$$

   (c) *Let $d \geq 2$ be the parameter of d-$\mathsf{2LHash}$. Then*

$$\mathsf{deg}(P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 1 + (2^d/(2^d - 1))\ell \leq 1 + 2\ell.$$

Proof. Since $L \geq 1$, we have $\ell \geq 1$.

Since $\ell \geq 1$, using the definition of Poly, we have $P_1(x; M_1, \ldots, M_\ell) = M_1 x^\ell + M_2 x^{\ell-1} + \cdots + M_\ell x$. By Lemma 2, M_1 is a non-zero element of $\mathbb{F}_p$. So the degree of $P_1(x)$ is ℓ.

Let $\rho = \mathfrak{d}(\ell)$ and $a_0, \ldots, a_\rho$ be such that $\mathsf{BRW}(x; M_1, \ldots, M_\ell) = a_\rho x^\rho + \cdots + a_1 x + a_0$. (Note that if $\ell \geq 3$, then since $\mathsf{BRW}(x; M_1, \ldots, M_\ell)$ is monic, it follows that $a_\rho = 1$.) So from (5),

$$\begin{aligned} P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x(x \cdot \mathsf{BRW}(x; M_1, \ldots, M_\ell) + \mathsf{bin}_{m-1}(L)) \\ &= a_\rho x^{\rho+2} + \cdots + a_1 x^3 + a_0 x^2 + \mathsf{bin}_{m-1}(L) x. \end{aligned}$$

From the second point of Theorem 1, we have $\rho \leq 2\ell - 1$ and so the degree of $P_2(x)$ is at most $2\ell - 1 + 2 = 2\ell + 1$.

Let $\mathfrak{m} = \ell - (\ell \bmod 2^t)$. Let $\rho = \mathfrak{d}(\mathfrak{m})$ and $a_0, \ldots, a_\rho$ be such that $\mathsf{BRW}(x; M_1, \ldots, M_{\mathfrak{m}}) = a_\rho x^\rho + \cdots + a_1 x + a_0$. (As above, if $\mathfrak{m} \geq 3$, then $a_\rho = 1$.) So

$$\begin{aligned} P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) &= x \cdot \mathsf{Poly}(x; \mathsf{BRW}(x; M_1, M_2, \ldots, M_{\mathfrak{m}}), M_{\mathfrak{m}+1}, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \\ &= a_\rho x^{\ell-\mathfrak{m}+2+\rho} + \cdots + a_1 x^{\ell-\mathfrak{m}+3} + a_0 x^{\ell-\mathfrak{m}+2} \\ &\quad + M_{\mathfrak{m}+1} x^{\ell-\mathfrak{m}+1} + \cdots + M_\ell x^2 + \mathsf{bin}_{m-1}(L) x. \end{aligned}$$

The degree of $P_3(x)$ is at most $\ell - \mathfrak{m} + 2 + \rho$. From the second point of Theorem 1, $\rho \leq 2\mathfrak{m} - 1$. Using this in the expression for the degree and noting that $\mathfrak{m} \leq \ell$ gives us the desired bound.

Let $\delta = 2^d - 1 \geq 3$ and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$. Recall that $U_i(x) = \mathsf{BRW}(x; M_{1+\delta(i-1)}, \ldots, M_{i\delta})$. Note that $\delta \geq 3$ implies that each $U_i(x)$ is a monic polynomial. Further, since $\delta = 2^d - 1$, from the second point of Theorem 1, we have $\mathsf{deg}(U_i) = \delta$. Since $V(x) = \mathsf{Poly}(x^{2^d}; U_1(x), \ldots, U_{\mathfrak{n}}(x))$, the coefficients of $V(x)$ are the coefficients of the $U_i(x)$'s and $\mathsf{deg}(V) = (\delta+1)(\mathfrak{n}-1) + \delta = (\delta+1)\mathfrak{n} - 1$ (see (10)). From (9) the degree of $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is equal to $\mathsf{deg}(V) + \ell - \delta\mathfrak{n} + 2 = \mathfrak{n} + \ell + 1$. Since $\mathfrak{n} \leq \ell/\delta$, we obtain the desired bound on the degree of $P_4(x)$. □

We obtain the following simple lower bound on the degrees of P_2, P_3 and P_4 which will be required in arguing about the correctness of d-Hash given by (11).

Corollary 1. *Let $d \geq 2$ and n be positive integers. Let X be a binary string of length $L \geq 1$ such that $\ell = \lceil L/n \rceil \geq 2^d$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. Let the parameter t of t-$\mathsf{BRWHash}$ be equal to d. Then $P_2(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, $P_3(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, and $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ are monic polynomials having degrees at least $2^d + 2$.*

Proof. Since $d \geq 2$ we have $\ell \geq 2^d > 3$. So the BRW components of all the three polynomials P_2, P_3 and P_4 are for more than three blocks and from the definition of BRW polynomials, it follows that these BRW components are all monic. Using the definitions of P_2, P_3 and P_4, it follows that all the three polynomials are also monic.

Using $\ell \geq 2^d > 3$, from the second point of Theorem 1 and the proof of Lemma 5 we have the following.

1. $\mathsf{deg}(P_2) = \mathfrak{d}(\ell) + 2 = (2^{\lfloor \lg \ell \rfloor + 1} - 1) + 2 \geq 2^d + 2$.
2. $\mathfrak{m} = \ell - (\ell \bmod 2^d) \geq 2^d$ and so $\mathsf{deg}(P_3) = \ell - \mathfrak{m} + 2 + \mathfrak{d}(\mathfrak{m}) = (\ell \bmod 2^t) + \mathfrak{d}(m) + 2 = (\ell \bmod 2^t) + (2^{\lfloor \lg \mathfrak{m} \rfloor + 1} - 1) + 2 \geq 2^d + 2$.
3. $\mathfrak{n} = \lfloor \ell/(2^d - 1) \rfloor \geq 1$ and so $\mathsf{deg}(P_4) = \mathfrak{n} + \ell + 1 \geq 2^d + 2$.

□

Theorem 2. *Suppose the prime p and the parameters m, k and n are as defined in Table 1. Let X and X' be two distinct binary strings of lengths L and L' respectively with $L \geq L' \geq 0$, and α be an element of $\mathbb{Z}_{2^k}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Let $t \geq 2$ be the parameter of t-$\mathsf{BRWHash}$ and $d \geq 2$ be the parameter of d-$\mathsf{2LHash}$. Then*

$$\begin{aligned} \Pr[\mathsf{polyHash}_\tau(X) - \mathsf{polyHash}_\tau(X') = \alpha] &\leq \ell \cdot 2^{m-2k+1}, \\ \Pr[\mathsf{BRWHash}_\tau(X) - \mathsf{BRWHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}, \\ \Pr[t\text{-}\mathsf{BRWHash}_\tau(X) - t\text{-}\mathsf{BRWHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}, \\ \Pr[d\text{-}\mathsf{2LHash}_\tau(X) - d\text{-}\mathsf{2LHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-2k+1}. \end{aligned}$$

Proof. Lemma 1 reduces the problem of upper bounding the differential probabilities of the hash functions to the analysis of the polynomials over $\mathbb{F}_p$ which define the corresponding hash functions. Specifically, we need to show that the constant terms of these polynomials are 0 and distinct X and X' map to distinct polynomials. From the definitions of $P_1(x)$, $P_2(x)$, $P_3(x)$ and $P_4(x)$ it follows that the constant terms of these polynomials are 0. The distinctness of the polynomials corresponding to distinct X and X' is given by Lemma 4. So Lemma 1 can be applied. Using the expressions for the degrees of the relevant polynomials from Lemma 5, we obtain the desired bounds on the probabilities. □

We next show the AXU bound for d-Hash.

Theorem 3. *Suppose the prime p and the parameters m, k and n are as defined in Table 1. Let X and X' be two distinct binary strings of lengths L and L' respectively with $L \geq L' \geq 0$, and α be an element of $\mathbb{Z}_{2^k}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Then*

$$\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] \quad \leq \quad (1+2\ell) \cdot 2^{m-2k+1}. \tag{13}$$

Proof. By construction, d-Hash applies polyHash if the number of blocks is less than 2^d and applies d-2LHash if the number of blocks is at least 2^d. Let $\ell' = \lceil L'/n \rceil$. There are three cases to consider, namely $\ell, \ell' < 2^d$, $\ell, \ell' \geq 2^d$, and $\ell' < 2^d \leq \ell$.

First suppose $\ell, \ell' < 2^d$. In this case, for both X and X', polyHash is applied and using Theorem 2 we have,

$$\begin{aligned} \Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] &= \Pr[\mathsf{polyHash}_\tau(X) - \mathsf{polyHash}_\tau(X') = \alpha] \\ &\leq \ell \cdot 2^{m-2k+1} < (1+2\ell) \cdot 2^{m-2k+1}. \end{aligned}$$

Next suppose $\ell, \ell' \geq 2^d$. In this case, for both X and X', d-2LHash is applied and using Theorem 2 we have

$$\begin{aligned} \Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha] &= \Pr[d\text{-}\mathsf{2LHash}_\tau(X) - d\text{-}\mathsf{2LHash}_\tau(X') = \alpha] \\ &\leq (1+2\ell) \cdot 2^{m-2k+1}. \end{aligned}$$

Now suppose $\ell' < 2^d \leq \ell$. In this case, X is hashed with d-2LHash and X' is hashed with polyHash. Recall that d-2LHash processes X using pad2 while polyHash processes X' using pad1. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_{\ell'})$ be the output of $\mathsf{pad1}(\mathsf{format}(X'))$. The constant terms of both $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_1(x; M'_1, \ldots, M'_{\ell'})$ are zero. From (2) and Lemma 2, the degree of $P_1(x; M'_1, \ldots, M'_{\ell'})$ is equal to $\ell' < 2^d$. Since $\ell \geq 2^d$, by Corollary 1, $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is a monic polynomial of degree at least $2^d + 2$. So $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ and $P_1(x; M'_1, \ldots, M'_{\ell'})$ are distinct polynomials whose constant terms are zero. Applying Lemma 1 to these two polynomials, we have $\Pr[d\text{-}\mathsf{Hash}_\tau(X) - d\text{-}\mathsf{Hash}_\tau(X') = \alpha]$ to be at most 2^{m-2k+1} times the degree of $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) - P_1(x; M'_1, \ldots, M'_{\ell'})$ which is the degree of $P_4(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$, and by Lemma 5 this degree is at most $1 + 2\ell$. □

Remark 4. *Suppose that in d-Hash, the hash function d-2LHash is replaced by either* BRWHash *or t-*BRWHash*. Using the bounds on the degrees of these polynomials given by Lemma 5 and Corollary 1, the proof of Theorem 3 shows that the bound* (13) *also holds for such modified variations of d-Hash.*

	polyHash	BRWHash, t-BRWHash, d-2LHash, d-Hash
$2^{127}-1$	$\ell \cdot 2^{-125}$	$(2\ell+1)2^{-125}$
$2^{130}-5$	$\ell \cdot 2^{-124}$	$(2\ell+1)2^{-124}$

Table 2: For the two primes in Table 1, the values of ϵ such that the hash families are ϵ-AXU. Here ℓ is the number of message blocks.

Remark 5. *The probability bounds in Theorems 2 and 3 are in terms of the number ℓ of n-bit blocks. So setting an upper bound on ℓ provides the values of ϵ for the hash families to be ϵ-AXU. Suppose we set $\ell = 2^{48}$, i.e. we restrict the hash families to process messages having at most 2^{48} blocks. The corresponding values of ϵ for the four hash families and the two primes can be obtained from the expressions given in Table 2 and are at most 2^{-75}. Choosing a lower value of ℓ, will further decrease the value of ϵ.*

With $\ell \leq 2^{48}$, we have $L \leq 2^{55}$ for the values of n in Table 1. Since it is unlikely that there will be an application which will require to hash messages longer than 2^{55} bits, for practical purposes, the length of any message will fit in a 64-bit word. So in the constructions BRWHash, t-BRWHash *and* d-2LHash, $\mathsf{bin}_{m-1}(L)$ *is essentially* $\mathsf{bin}_{64}(L)$. *This provides a speed-up in the implementation of the field multiplication with the binary representation of the length of the message. The effect, however, is minor since there is only one such field multiplication.*

4.1 Explanations for the padding schemes

Suppose $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Lemma 2 states that each of the M_i's are non-zero. The proof of Lemma 4 uses this observation to show that the map $X \mapsto \mathsf{Poly}(x; M_1, \ldots, M_\ell)$ and hence the map $X \mapsto P_1(x; M_1, \ldots, M_\ell)$ are injections.

The fact that the M_i's are non-zero is not sufficient to argue that the map $X \mapsto \mathsf{BRW}(x; M_1, \ldots, M_\ell)$ is an injection. For example, suppose X and X' are distinct binary strings such that $(M_1, \ldots, M_5) = \mathsf{pad1}(\mathsf{format}(X))$ and $(M'_1, \ldots, M'_6) = \mathsf{pad1}(\mathsf{format}(X'))$. Then

$$\begin{aligned} \mathsf{BRW}(x; M_1, \ldots, M_5) &= ((x+M_1)(x^2+M_2)+M_3)(x^4+M_4)+M_5, \\ \mathsf{BRW}(x; M'_1, \ldots, M'_6) &= ((x+M'_1)(x^2+M'_2)+M'_3)(x^4+M'_4)+M'_5x+M'_6. \end{aligned}$$

Note that both $\mathsf{BRW}(x; M_1, \ldots, M_5)$ and $\mathsf{BRW}(x; M'_1, \ldots, M'_6)$ are monic polynomials of degrees equal to 7. Since the coefficients of $\mathsf{BRW}(x; M_1, \ldots, M_5)$ and $\mathsf{BRW}(x; M'_1, \ldots, M'_6)$ are nonlinear functions of $M_1, \ldots, M_5$ and $M'_1, \ldots, M'_6$ respectively, the fact that the M_i's and the M'_j's are non-zero is not sufficient to argue that $\mathsf{BRW}(x; M_1, \ldots, M_5) \neq \mathsf{BRW}(x; M'_1, \ldots, M'_6)$.

Theorem 5.6 of [4] shows that if S is a subset of $\mathbb{F}_p$ such that $S \cap (S+1) = \emptyset$, then BRW injectively maps $\cup_{i \geq 0} S^i$ to $\mathbb{F}_p[x]$. We considered using this result to avoid the above issue. For S we considered taking all elements of $\mathbb{F}_p$ whose least significant bit is 0 (i.e. S is the subset of the even numbers of $\mathbb{F}_p$). Such an S satisfies the requirement $S \cap (S+1) = \emptyset$. The problem arises in defining an appropriate padding scheme. For concreteness, consider the prime $2^{130}-5$. Suppose $(X_1, \ldots, X_\ell)$ is the output of $\mathsf{format}(X)$. Each X_i is an 128-bit string. We may define a padding scheme which left shifts X_i by one place to obtain M_i. The M_i's are ensured to be even and hence are in S. There are two problems to such a padding scheme. For one thing, each X_i is a 16-byte string which would be stored as 2 64-bit or 4 32-bit words. A left shift by one place would require multiple shifts of the words representing an X_i. This would result in some inefficiency. (In comparison, note that appending a 1 on the left, as in pad1, does not require any shift.) There is

another more basic problem with such a padding scheme. We need to ensure that a padded partial block is distinct from a padded full block and also that after padding, two partial blocks of different lengths map to distinct strings. We see no simple way of ensuring that a padded partial block is distinct from a padded full block.

Further, the hash functions t-BRWHash and d-2LHash are built using a combination of BRW and Poly. Using a padding scheme to ensure injectivity of only BRW is not sufficient to ensure the injectivity of the defining polynomials for the hash functions t-BRWHash and d-2LHash.

To handle the above problems, we introduced pad2 which includes the binary representation of the length of X in its output. By appropriately using the length block, we obtain injective maps as shown in Lemma 4. The use of pad2 simplifies the description of the three hash functions and also makes the injectivity argument quite easy.

5 Algorithms

Algorithms to evaluate the hash functions essentially boil down to evaluating the polynomials $P_1(x)$, $P_2(x)$, $P_3(x)$ and $P_4(x)$ at the point τ. These four polynomials are in turn defined from the two polynomials Poly and BRW. So we first consider algorithms to compute the values of the polynomials Poly and BRW at a point.

In Sections 5.1 and 5.2, the description of the algorithms to compute the values of Poly and BRW at a point are over an arbitrary finite field $\mathbb{F}$. For $l \geq 0$, let $M_1, \ldots, M_l$ be elements of $\mathbb{F}$ and τ be an element of $\mathbb{F}$. (Recall that ℓ is the number of blocks obtained by formatting a binary string X; l is not necessarily equal to ℓ.) We consider the evaluation of $\mathsf{Poly}(x; M_1, \ldots, M_l)$ and $\mathsf{BRW}(x; M_1, \ldots, M_l)$ at the point τ. Such evaluation involves multiplications and additions over $\mathbb{F}$. A field multiplication has two steps. In the first step, a multiplication is done over an appropriate structure (such as the ring of integers or polynomials) and in the second step, the product is reduced. By unreducedMult we will denote only the first step, i.e. the reduction step is not performed, while by reduce we will denote the reduction step. Similarly a field addition also has two steps, an addition over an appropriate structure followed by a reduction. In the algorithms for evaluating Poly and BRW, the reduction after addition is never performed. So in the algorithms, '+' denotes an unreduced addition.

5.1 Evaluation of Poly

For $l \geq 2$, Using Horner's rule, $\mathsf{Poly}(\tau; M_1, \ldots, M_l)$ can be evaluated as follows.

$$\mathsf{Poly}(\tau; M_1, \ldots, M_l) \quad = \quad \tau(\cdots \tau(\tau(\tau \cdot M_1 + M_2) + M_3) + \cdots + M_{l-1}) + M_l. \tag{14}$$

This requires $l - 1$ field multiplications. A delayed (or lazy) reduction strategy was used in [14] to implement the hash function GHASH which is defined over binary fields. It was also used in [13] in the context of evaluation of Poly1305 using vector instructions. The same strategy can also be employed for an arbitrary finite field $\mathbb{F}$. We describe the strategy below.

Let $g \geq 1$ be a parameter. The blocks $M_1, \ldots, M_l$ are divided into $\lceil l/g \rceil$ groups and one reduction will be applied for each group. Let $r \in \{1, \ldots, g\}$ be such that $r \equiv l \bmod g$. Let $k = (l - r)/g$ (so that $k + 1 = \lceil l/g \rceil$) and define

$$A_1 \quad = \quad M_1\tau^{r-1} + M_2\tau^{r-2} + \cdots + M_{r-1}\tau + M_r,$$

and for $i = 1, \ldots, k$,

$$A_{i+1} \quad = \quad M_{r+(i-1)g+1}\tau^{g-1} + M_{r+(i-1)g+2}\tau^{g-2} + \cdots + M_{r+ig-1}\tau + M_{r+ig}.$$

Then

$$\mathsf{Poly}(\tau; M_1, \ldots, M_l) \quad = \quad \tau^g(\cdots\tau^g(\tau^g(\tau^g \cdot A_1 + A_2) + A_3) + \cdots + A_k) + A_{k+1}. \tag{15}$$

Note that A_1 is defined using a group of $r \leq g$ blocks, while each of $A_2, \ldots, A_{k+1}$ is defined using exactly g blocks.

Suppose the elements $\tau^2, \tau^3, \ldots, \tau^{g-1}, \tau^g$ are computed over $\mathbb{F}$ and stored. Then the A_i's can be evaluated by multiplying the relevant power of g with the appropriate block and summing the products. The A_i's are not individually reduced during the computation as we explain next. Let $\mathsf{unreduced}(A_i)$ denote the computation of A_i without applying the reduction step, i.e. the outputs of all the multiplications and additions in the expression for A_i are kept unreduced.

Let res be a variable which stores the result of the partial computations. Initially the value of res is set to $\mathsf{reduce}(\mathsf{unreduced}(A_1))$. Next, for $i = 1, \ldots, k$, update res to

$$\mathsf{reduce}(\mathsf{unreducedMult}(\mathsf{res}, \tau^g) + \mathsf{unreduced}(A_{i+1})).$$

The final value of res provides the required result.

Computing $\mathsf{unreduced}(A_1)$ requires $r - 1$ unreduced multiplications. For $i = 2, \ldots, k + 1$, computing $\mathsf{unreduced}(A_i)$ requires $g - 1$ unreduced multiplications. Finally, the k updations of res require k unreduced multiplications. So the total number of unreduced multiplications required is $r - 1 + k(g - 1) + k = l - 1$. This number is the same as that for evaluating (14). The advantage is that for $g > 1$, the number of reductions decreases. From the above description, the number of reductions required to compute (15) is $k + 1 = \lceil l/g \rceil$. In comparison, computing (14) requires $l - 1$ reductions.

The trade-off is that the powers $\tau^2, \ldots, \tau^g$ are required to be computed and this requires $g - 1$ field multiplications. For short messages, the time to compute the powers of τ will make the strategy inefficient. To avoid this inefficiency one may pre-compute and store the required powers of τ. There are advantages and disadvantages to both the approaches, i.e. to compute the powers on-the-fly and to precompute and store these powers. In our timing results given later, we provide timings for both the approaches.

The general idea of the delayed reduction strategy suggests that as g increases, the efficiency should improve. This, however, is not true in practice. For efficient execution, it is important that the powers of τ be available in the cache. If the value of g is high, then all the powers of τ cannot be stored in the cache and the efficiency of the method decreases. We have experimented with $g = 4, 8, 16, 32$. For long messages, the choice of $g = 16$ provides the best performance, while for shorter messages, lower values of g provide better performance.

Remark 6. *The explicit advantage of the delayed reduction strategy is the decrease in the number of reductions. Since the number of unreduced multiplications does not decrease, one does not expect a very sharp increase in efficiency due to the use of delayed reduction. Our experiments, on the other hand, show that using $g = 4$ or $g = 8$ provide a very high jump in speed. (Details of the timing results are given later.) Such a jump cannot be solely explained by the decrease in the number of reductions. We investigated the issue deeply. It turns out that there is a hidden advantage of the delayed reduction strategy. The evaluation of the A_k's require multiplications by the powers of τ.*

These powers are fixed for the entire duration of the computation. So in effect, one of the operands of the multiplications is a constant. During execution, the powers of τ are kept in the data buffer of the multiplication units of the CPU. It is due to this effect that there is a very significant speed improvement. If we modify the expression where instead of a fixed power of τ, multiplication is done with a variable quantity (which no longer evaluates Poly*), then the speed drops to the level which would be explained by the decrease in the number of reductions.*

5.2 Evaluation of BRW

The definition of BRW is recursive. It is possible to write a recursive program to evaluate BRW. Such a program, however, will be quite inefficient. If the number of blocks is fixed, then it is possible to implement BRW using a straight line code. See Appendix A for some examples. A general non-recursive algorithm to evaluate BRW was developed in [12] and it was shown that $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ can be evaluated using $\lfloor l/2 \rfloor$ unreduced multiplications and $1 + \lfloor l/4 \rfloor$ reductions (plus an additional $\lfloor \lg l \rfloor$ field squarings to compute the required power of τ). This is an improvement over the requirement of $\lfloor l/2 \rfloor$ field multiplications stated in Theorem 1. Following Lemma 1 of [12], the evaluation of $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ can be written as

$$\mathsf{BRW}(\tau; M_1, \ldots, M_l) \quad = \quad \mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l)),$$

where

- $\mathsf{unreducedBRW}(\tau;) = 0$;
- $\mathsf{unreducedBRW}(\tau; M_1) = M_1$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2) = \mathsf{unreducedMult}(M_1, \tau) + M_2$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, M_3) = \mathsf{unreducedMult}((\tau + M_1), (\tau^2 + M_2)) + M_3$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, \ldots, M_k)$
  $= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{k-1})), (\tau^k + M_k))$,
  if $k \in \{4, 8, 16, 32, \ldots\}$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, \ldots, M_l)$
  $= \mathsf{unreducedBRW}(\tau; M_1, \ldots, M_k) + \mathsf{unreducedBRW}(\tau; M_{k+1}, \ldots, M_l)$,
  if $k \in \{4, 8, 16, 32, \ldots\}$ and $k < l < 2k$.

The idea of the algorithm in [12] is to process groups of 2^t blocks at a time for some integer $t \geq 2$. For each such group, unreducedBRW corresponding to the first $2^t - 1$ blocks is evaluated using a straight line code. Partial results are stored in an array. Depending on the number of blocks that have been processed, some of the partial results are taken from the array and combined with the output of the present iteration and the resulting new partial result is again added to the array. Here we present a variant of the algorithm given in [12]. The variant simplifies the algorithm in [12] by using a different method to store partial results. The number of unreduced multiplications and reductions remain unchanged.

The modified algorithm ComputeBRW is shown in Algorithm 1. The partial results are stored in stack and top points to the top of stack. The stack is implemented as an array $\mathsf{stack}[0, ..., \lfloor \lg l \rfloor - t]$. The operation $\mathsf{ntz}(i)$ in Step 11 returns the number of trailing zeros in i; it can be implemented in assembly using the instruction `tzcnt`. The operation $\mathsf{wt}(\lfloor l/2^t \rfloor)$ in Step 20 returns the Hamming weight of the binary representation of $\lfloor l/2^t \rfloor$; it can be implemented in assembly using the instruction `popcnt`.

Algorithm 1 differs from the algorithm in [12] in the manner in which the partial results are stored. Since the manner of storage determines the overall correctness of the algorithm, the proof of correctness provided in [12] needs substantial modifications to apply to Algorithm 1. The following result states the correctness and complexity of the new algorithm. The proof is given in Appendix B.

Theorem 4. *For any $l \geq 0$ and any $t \geq 2$, Algorithm 1 correctly computes* $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$. *The number of* unreducedMult *required is $\lfloor l/2 \rfloor$ and the number of reductions required is $1 + \lfloor l/4 \rfloor$. Additionally $\lfloor \lg l \rfloor$ field squarings are required to compute the powers of τ. The maximum size of* stack *is at most $\lfloor \lg l \rfloor - t + 1$.*

Algorithm 1 Evaluation of $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$, $l \geq 0$. In the algorithm $t \geq 2$ is a parameter.

1: **function** ComputeBRW$(\tau, M_1, \ldots, M_l)$
2: keyPow$[0] \leftarrow \tau$
3: **if** $l > 2$ **then**
4: **for** $j \leftarrow 1$ to $\lfloor \lg l \rfloor$ **do**
5: keyPow$[j] \leftarrow$ keyPow$[j-1]^2$
6: **end for**
7: **end if**
8: top $\leftarrow -1$
9: **for** $i \leftarrow 1$ to $\lfloor l/2^t \rfloor$ **do**
10: tmp $\leftarrow$ unreducedBRW$(\tau; M_{2^t(i-1)+1}, \ldots, M_{2^t \cdot i-1})$;
11: $k \leftarrow$ ntz(i)
12: **for** $j \leftarrow 0$ to $k-1$ **do**
13: tmp $\leftarrow$ tmp + stack[top]; top $\leftarrow$ top $- 1$
14: **end for**
15: tmp $\leftarrow$ unreducedMult(reduce(tmp), $M_{2^t \cdot i}$ + keyPow$[t+k]$)
16: top $\leftarrow$ top $+ 1$; stack[top] $\leftarrow$ tmp
17: **end for**;
18: $r \leftarrow l \bmod 2^t$;
19: tmp $\leftarrow$ unreducedBRW$(\tau; M_{l-r+1}, \ldots, M_l)$;
20: $i \leftarrow$ wt$(\lfloor l/2^t \rfloor)$
21: **for** $j \leftarrow 0$ to $i-1$ **do**
22: tmp $\leftarrow$ tmp + stack[top]; top $\leftarrow$ top $- 1$
23: **end for**
24: return reduce(tmp);
25: **end function**.

The parameter t in Algorithm 1 does not have any effect on the correctness of the algorithm or on the number of operations that are required. Step 10 uses a straight line code to compute unreducedBRW on $2^t - 1$ blocks. So the parameter t determines the extent of loop unrolling. This has an effect on the practical efficiency of implementation as our timing results given later show.

5.3 Rationale for t-BRWHash and d-2LHash

Step 19 of Algorithm 1 performs an unreduced BRW computation on the last r blocks of the message, where $r = l \bmod 2^t$. Let us consider the implementation of Step 19. Since the value of r

depends on the number l of blocks in the input, the implementation of Step 19 needs to account for all the $2^t - 1$ possible positive values of r. For each such value of r, a straight line code is required to implement the unreduced BRW computation. So the implementation of Step 19 requires a total of $2^t - 1$ separate fragments of straight line codes to implement unreduced BRW computation for all the $2^t - 1$ possible positive values of r. If $t = 2$ or 3, then this accounts for 3 or 7 fragments of straight line codes respectively, which is reasonable. On the other hand, for $t = 4$ or 5, the number of straight line code fragments is 15 or 31 respectively. Having so many fragments of straight line codes make the overall program messy, difficult to optimise and increases the code size.

The design of t-BRWHash solves the above issue. Recall that in this design, BRW is used to process a number of blocks which is a multiple of 2^t. The output of this BRW computation, the leftover blocks and the length block are then processed using Poly. So when ComputeBRW is used to compute the BRW part of t-BRWHash, the computation of Step 19 is not required (i.e. since $r = 0$, it becomes trivial). As a result, the entire issue of using $2^t - 1$ separate fragments of straight line codes become irrelevant. This leads to a much shorter and more compact code.

We have implemented both BRWHash and t-BRWHash. For BRWHash, ComputeBRW was implemented using $t = 2$ and $t = 3$, while t-BRWHash was implemented for $t = 2, 3, 4$ and 5.

We next consider the design rationale for d-2LHash. In d-2LHash, each of the individual BRW computations is performed on δ blocks, where $\delta = 2^d - 1$ is a fixed number. For small d, the δ-block BRW computation can be performed using a straight line code. As a result, it is not required to implement Algorithm 1 at all. So the design motivation for d-2LHash is to completely avoid the implementation of Algorithm 1. This reduces the implementation complexity of the BRW part. In our implementations, we have considered $d = 2, 3, 4$ and 5. Examples of straight line code for the corresponding δ-block BRW computations are given in Appendix A.

One may also consider values of $d \geq 6$. A problem with such values of d is that the straight line code for δ-block BRW computation becomes large. For example, if $d = 6$, then a straight line code for a 63-block BRW computation is required. With larger values of d, the storage requirement for pre-computed keys will increase and efficiency benefits will be observed for longer messages. Also, having too high a value of d may have the effect that intermediate results no longer fit in the cache, which would lead to a slowdown. Due to these reasons, we did not investigate values of $d \geq 6$.

5.4 Explanation of parameters

We distinguish between two types of parameters, namely design parameter and implementation parameter. The output of a hash function does not depend on an implementation parameter, i.e. if the value of an implementation parameter is changed, then for the same input, the output does not change. On the other hand, the output of a hash function does depend on a design parameter. For the same input, if the value of a design parameter is changed, then the corresponding output will be different.

The parameter g which determines the group size in Horner evaluation is an implementation parameter. Similarly, t in Algorithm 1 is also an implementation parameter. On the other hand, the parameter t in t-BRWHash and the parameter d in d-2LHash are design parameters. Also, d is a design parameter in d-Hash.

In d-2LHash, the parameter d indicates that the hash function uses straight line BRW computation on $\delta = 2^d - 1$ blocks. In d-Hash, the parameter d indicates that for messages with less than 2^d blocks polyHash is used, and for messages with at least 2^d blocks d-2LHash is used. So in d-Hash, apart from its role in d-2LHash, the parameter d also indicates the switchover point from polyHash

to d-2LHash.

5.5 Operation counts

The four hash families are built out of various combinations of Poly and BRW. Having determined the number of integer multiplications and reductions required by Poly and BRW, we can now specify these numbers for the hash families. For Poly, we consider both the options $g = 1$ and $g > 1$. Table 3 provides the operation counts for the four hash functions for $g = 1$, while Table 4 provides the operation counts for $g > 1$. In these tables, the columns labeled 'mult' and 'red' provide the numbers of integer multiplications and reductions required for the evaluation. The column labeled 'storage' provides the number of powers of the key τ that need to be stored, while the column labeled 'pre-comp' provides the numbers of operations that are required to compute the key powers.

Elements are stored as m-bit quantities. An integer multiplication of two m-bit quantities results in a $2m$-bit quantity, while the integer addition of two m-bit quantities results in an $(m + 1)$-bit quantity. As explained earlier, in the algorithms to compute Poly and BRW, we do not immediately reduce the result of an integer multiplication. Any subsequent additions are also performed without reduction. A reduction is performed only when the intermediate quantity is to be multiplied with another m-bit element. The reductions counted in Tables 3 and 4 are such reductions.

Remark 7. *The storage requirements for key powers in Table 4 are overestimates. The values are simply the sums of the number of key powers required for* BRW *and the number of key powers required for* Poly *with $g > 1$. There will typically be an overlap between these key powers, which will reduce the storage requirement. For example, suppose in d-*2LHash*, we choose $d = 5$ and $g = 8$. According to Table 4, the number of key powers that need to be stored is $d + 2g = 21$. Let us consider the required key powers in more details. The* BRW *computation will require the key powers $\tau, \tau^2, \tau^4, \tau^8, \tau^{16}$. The computation of $V(x)$ with $g = 8$ will require the key powers $\gamma, \gamma^2, \gamma^3, \ldots, \gamma^8$, where $\gamma = \tau^{32}$. The final computation of* Poly *will require the key powers $\tau, \tau^2, \tau^3, \ldots, \tau^8$. Out of these, τ, τ^2, τ^4 and τ^8 are also required for the* BRW *computation. So the total number of key powers that will be required to be stored is $5 + 8 + 4 = 17$. Similarly, the numbers of operations required to compute the key powers are also overestimates. Since it is messy to obtain a general formula for the exact number of key powers that are required, we have chosen to provide the simpler overestimates. Later when we consider specific values of the parameters, we provide the corresponding accurate number of required key powers.*

The computation of the key powers $\tau^2, \tau^4, \tau^8, \ldots$ required for BRW computation can be done using squarings rather than multiplications. Squarings are faster than multiplications. The operation counts in Tables 3 and 4 do not make the distinction between multiplications and squarings. In our implementations, however, we have used squarings to compute the above mentioned key powers. The computations of these key powers are the only part of the entire computation which require squarings.

From Tables 3 and 4, it may be noted that the number of multiplications required by Poly is ℓ which is the maximum among the four hash functions. BRWHash, t-BRWHash and d-2LHash require about $2 + \ell/2$, $2 + \ell/2 + (\ell \bmod 2^t)/2$ and $1 + (2^{d-1}/(2^d - 1))\ell$ multiplications respectively. The number of reductions depends on the value of g and the comparison between the hash functions on this feature is more complicated. Nonetheless, from the operation counts one would expect BRWHash to be significantly faster than Poly. Our timing results reported later show that this is

				pre-comp	
	mult	red	storage	mult	red
polyHash	ℓ	ℓ	1	-	-
BRWHash	$2 + \lfloor \ell/2 \rfloor$	$2 + \lfloor \ell/4 \rfloor$	$\lfloor \lg \ell \rfloor$	$\lfloor \lg \ell \rfloor$	$\lfloor \lg \ell \rfloor$
t-BRWHash	$\ell - \lceil \mathfrak{m}/2 \rceil + 2$	$\lfloor \mathfrak{m}/4 \rfloor + \ell - \mathfrak{m} + 3$	$\lfloor \lg \mathfrak{m} \rfloor$	$\lfloor \lg \mathfrak{m} \rfloor$	$\lfloor \lg \mathfrak{m} \rfloor$
d-2LHash	$\ell + 1 - \mathfrak{n}(2^{d-1} - 1)$	$\ell + 1 - \mathfrak{n}(3 \cdot 2^{d-2} - 2)$	$d + 1$	d	d

Table 3: Operation counts for the hash functions for ℓ blocks with $g = 1$. In the table $\mathfrak{m} = \ell - (\ell \bmod 2^t)$, $\delta = 2^d - 1$, and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$.

				pre-comp	
	mult	red	storage	mult	red
polyHash	ℓ	$\lceil \ell/g \rceil$	g	$g - 1$	$g - 1$
t-BRWHash	$\ell - \lceil \mathfrak{m}/2 \rceil + 2$	$\lfloor \mathfrak{m}/4 \rfloor + \lceil (\ell - \mathfrak{m} + 2)/g \rceil + 1$	$\lfloor \lg \mathfrak{m} \rfloor + g$	$\lfloor \lg \mathfrak{m} \rfloor + g - 1$	$\lfloor \lg \mathfrak{m} \rfloor + g - 1$
d-2LHash	$\ell + 1 - \mathfrak{n}(2^{d-1} - 1)$	$\mathfrak{n} \cdot 2^{d-2} + \lceil \mathfrak{n}/g \rceil$ $+ \lceil (\ell - \delta\mathfrak{n} + 2)/g \rceil + 1$	$d + 2g$	$d + 2g - 1$	$d + 2g - 1$

Table 4: Operation counts for the hash functions for ℓ blocks with $g > 1$. In the table $\mathfrak{m} = \ell - (\ell \bmod 2^t)$, $\delta = 2^d - 1$, and $\mathfrak{n} = \lfloor \ell/\delta \rfloor$.

indeed true for $g = 1$. On the other hand, for $g > 1$ the speed improvement is much more modest. See Remark 6 for an explanation of this observation.

6 Implementation details

The implementations of the hash functions require the implementation of three arithmetic operations, namely integer addition, integer multiplication and modular reduction. Of these, the latter two operations are more complicated and require substantially more time than integer addition. So we focus on the description of integer multiplication and modular reduction. As explained in Section 5, for speed improvement we adopted the lazy reduction strategy. This introduces certain complications, which we explain below.

We have made 64-bit assembly implementations of the hash functions for the Intel Skylake and later generation processors. A 64-bit word will be called a limb. Depending on the choice of the prime p, elements of $\mathbb{F}_p$ have representations of different sizes.

Case $p = 2^{127} - 1$: In this case, a general element of $\mathbb{F}_p$ can be represented using 127 bits. So padded message blocks, key powers, and intermediate results have 2-limb representations for all the hash functions. For Poly evaluation, the multiplicand is a padded message block and the multiplier is a key power, both of which are 2-limb quantities. For BRW evaluation, the multiplicand is a sum of a padded message block and a key power, and the multiplier is a reduced partial result. Since padded message blocks are 126-bit quantities and key powers are 127-bit quantities, the result of the sum is a 128-bit quantity. A reduced partial result is also a 2-limb quantity. So for all the hash functions, both the multiplier and the multiplicand are 2-limb quantities. Consequently, the product can be stored in a 4-limb quantity. Since we adopt the lazy reduction strategy, the result is not immediately reduced. As explained in Section 5, the results of several multiplications are added together and a reduction is performed on the sum. The sum of the results of several multiplications may not fit in a 4-limb quantity, and we store such a sum in a 5-limb quantity. So the reduction algorithm is applied to a 5-limb quantity to reduce it to a 127-bit quantity.

	$2^{127}-1$	$2^{130}-5$
polyHash	2-limb x 2-limb	(2-limb x 3-limb)+
BRWHash	2-limb x 2-limb	3-limb x 3-limb
t-BRWHash	2-limb x 2-limb	3-limb x 3-limb
d-2LHash	2-limb x 2-limb	3-limb x 3-limb

Table 5: Types of multiplications for the various hash functions.

Case $p = 2^{130} - 5$: In this case, a general element of $\mathbb{F}_p$ can be represented using 3 limbs, where the two least significant bits of the third limb (i.e. the most significant limb) are information bits. In certain cases, instead of full reduction we apply partial reduction due to which the last three significant bits of the third limb are information bits. Compared to full reduction, partial reduction requires fewer assembly instructions and leads to overall efficiency improvement. The key τ is a 128-bit string while the key powers τ^i, $i \geq 2$, are general elements of $\mathbb{F}_p$. The representation of the message blocks depends on the hash function.

1. For polyHash, after padding, full message blocks are 129-bit strings where the 129-th bit is 1, while after padding, partial message blocks are 129-bit strings where the 129-th bit is 0. Since we know whether a message block is full or partial (only the last block can be partial), padded message blocks are stored as 2-limb quantities. For multiplications involving padded full blocks, we perform the required number of additions corresponding to the 129-th bit. Multiplications involve a padded message block and a key power. The padded message block is a 2-limb quantity (with the 129-th bit 1 if the block is full), and a key power is a 3-limb quantity, where the third limb has two information bits. The integer multiplication is of the type 2-limb x 3-limb, plus an additional number of 64-bit additions in case of full message blocks. The output is stored as a 5-limb quantity.

2. For the other hash functions, padded message blocks are defined to be 129-bit strings, where the 129-th bit is 0. So for these hash functions, padded message blocks are stored as 2-limb quantities. For BRWHash evaluation, the multiplier is a sum of a padded message block and a key power, and so in general is a 131-bit quantity. The multiplicand is an intermediate result which is kept partially reduced and stored as a 131-bit quantity. So a general multiplication for BRWHash evaluation is of the type 3-limb x 3-limb and the output is stored as a 5-limb quantity. For a 2-block message, $\mathsf{BRW}(\tau; M_1, M_2) = \tau \cdot M_1 + M_2$, and in this case the multiplication $\tau \cdot M_1$ is of the type 2-limb x 2-limb.

For both $p = 2^{127} - 1$ and $p = 2^{130} - 5$, the length block, i.e. the binary representation of the length of the message, is stored as a 1-limb quantity (see Remark 5), and this is multiplied with the key τ. Correspondingly, the multiplication involving the length block is of the type 1-limb x 2-limb.

Table 5 provides a summary of the general types of multiplications required by the various hash functions for the two primes. In the table, (2-limb x 3-limb)+ denotes a 2-limb x 3-limb type multiplication plus a number of 64-bit additions which arises due to the 129-th bit of a padded full message block being 1. Multiplication by the length block and the 2-limb x 2-limb multiplication required by BRWHash for a 2-block message for the prime $2^{130} - 5$ are not shown in the table.

6.1 Size increase due to lazy reduction

The strategy of lazy reduction in BRW evaluation requires accumulating a number of outputs of unreducedBRW computations. Such accumulation takes place at two places in Algorithm 1, namely in the **for** loop of Steps 12 to 14 and in the **for** loop of Steps 21 to 23. In both these loops, a number of elements from the stack are popped and added to the value of tmp. The result is not reduced. We have mentioned that we use a 5-limb quantity to store the result of the accumulation. We need to argue that a 5-limb quantity is sufficient for the purpose and does not cause any overflow. Suppose that $\mathfrak{k}$ is the number of elements that are popped from the stack and added to tmp. Then $\mathfrak{k}$ is at most the size of the stack. Recall from Theorem 4 that the maximum size of the stack is $\lfloor \lg \ell \rfloor - t + 1$ and so $\mathfrak{k} \leq \lfloor \lg \ell \rfloor - t + 1$. Assuming that messages are of lengths less than 2^{64} bits, i.e. $L < 2^{64}$, we have $\ell = \lceil L/n \rceil$ and so $\ell < 2^{57}$ for $n = 128$ and $\ell < 2^{58}$ for $n = 120$. Taking the smaller of these bounds (i.e. restricting messages to have less than 2^{57} blocks), we obtain $\mathfrak{k} < 58 - t$. Since $2 \leq t \leq 5$, we have $\mathfrak{k} < 56$. We need to argue that storing tmp as a 5-limb quantity is sufficient to store the result of $\mathfrak{k}$ additions, where $\mathfrak{k} < 56$. This argument is provided separately for the two primes.

1. For $p = 2^{127} - 1$, the multiplications are of the type 2-limb x 2-limb and the result is a 4-limb quantity. Consequently, the value of tmp computed in Step 10 of Algorithm 1 is a 4-limb quantity. So storing tmp as a 5-limb quantity allows the value of $\mathfrak{k}$ to be up to 64.

2. For $p = 2^{130} - 5$, the multiplications are of the type 3-limb x 3-limb, where the 3-limb quantities are at most 131-bit strings. Consequently, the value of tmp computed in Step 10 is a 262-bit string which is stored as a 5-limb quantity. So there are a total of $64 - 6 = 58$ bits in the fifth limb (i.e. the most significant limb) which are unused. So for $\mathfrak{k} < 56$, accumulating $\mathfrak{k}$ quantities does not lead to any overflow.

In view of the above, for both the primes $p = 2^{127} - 1$ and $p = 2^{130} - 5$, storing tmp as a 5-limb quantity is sufficient for all practical sized messages.

The strategy of lazy reduction for grouped evaluation of Poly with group size g requires accumulating the results of g multiplications. Each addition increases the size of the result by one bit and so accumulating the results of g multiplications, leads to an increase in size by g bits. In a manner similar to the above, it can be argued that for $g \leq 56$, using a 5-limb quantity to store the result of the accumulation does not lead to an overflow. The bound of 56 for g is well past the point where grouping leads to efficiency improvement for practical sized messages (see Section 5.1).

6.2 Integer multiplication

The Intel Skylake and later processors provide three instructions, namely `mulx`, `adox` and `adcx`, which permit the implementation of the so-called double carry chain strategy for multi-limb integer multiplication and squaring using the schoolbook method. The approach is outlined in two Intel white papers [18, 17] using specific examples. General algorithms for multi-limb double carry chain multiplication and squaring are described in [16]. We have used the algorithms in [16] to perform the multiplications in Table 5. Since the number of limbs is small (2 or 3), Karatsuba multplication will not be competitive with the schoolbook method.

6.3 Reduction

The prime $p = 2^{127} - 1$ is a Mersenne prime. Algorithm 4 of [16] provides a general method for reduction modulo a Mersenne prime. This algorithm reduces the output of the integer multiplication or squaring algorithm. Applied to $p = 2^{127} - 1$, Algorithm 4 of [16] will reduce a 4-limb quantity to a 2-limb (more precisely, a 127-bit) quantity. However as described above, for our present application, it is required to reduce a 5-limb quantity to a 2-limb one. This requires some modifications to Algorithm 4 of [16] which somewhat increases the number of instructions required. Our implementation of reduction modulo $p = 2^{127} - 1$ makes the required modifications. The modifications are straightforward and so we skip the details.

The prime $p = 2^{130} - 5$ is a so-called pseudo-Mersenne prime. Algorithm 5 of [16] provides a method of reduction modulo such a prime. Theorem 6.3 of [16] states the condition under which Algorithm 5 applies. Let $\delta = 5$ and $\alpha = 3$ so that $2^{\alpha-1} \leq \delta < 2^{\alpha}$. Consider the 3-limb representation of elements of $\mathbb{F}_p$, where the first two limbs are $\eta = 64$ bits long and the third limb is $\nu = 2$ bits long. According to Theorem 6.3 of [16], a condition for Algorithm 5 of [16] to apply is that $\alpha < \nu + 1$. This condition fails for the prime $p = 2^{130} - 5$. So Algorithm 5 of [16] cannot be applied to perform reduction modulo $p = 2^{130} - 5$. Below we describe a new method for performing this reduction. Computation modulo the prime $p = 2^{130} - 5$ underlies the computation of the well known hash function Poly1305. We have, however, not been able to locate the reduction method that is described below in the literature.

Write the 5-limb quantity A to be reduced as $A = a_0 + 2^{130}a_1$, where a_0 is a 130-bit non-negative integer and a_1 is a non-negative integer. Then $A \equiv a_0 + 5a_1 \pmod{2^{130} - 5}$. Write $5a_1 = a_1 + 4a_1$. A key observation is that $4a_1$ can be obtained easily from the 5-limb representation of A. If we set the two least significant bits of the third limb of A to 0, then the last three limbs (i.e. the three most significant limbs) of A provide $4a_1$. From $4a_1$, the value of a_1 can be obtained by a right shift of two places. So our reduction strategy is the following. Given A, obtain a_0, next obtain $4a_1$ as described and add to a_0, then obtain a_1 and add to the sum of a_0 and $4a_1$. This gives $a_0 + 5a_1$. From the above description of the size of representation of elements required for the lazy implementation strategy, at least the 8 most signficant bits of the fifth limb of A are 0 for the prime $2^{130} - 5$. So $4a_1$ is a 3-limb quantity, where at least the 8 most significant bits of the third limb of $4a_1$ are 0. The sum $B = a_0 + 5a_1$ computed as $a_0 + 4a_1 + a_1$ results in a 3-limb quantity. Now write $B = b_0 + 2^{130}b_1$, where b_0 is a 130-bit non-negative integer and b_1 is a 1-limb non-negative integer whose at least 8 most significant bits are 0. So $5b_1$ fits in a single limb. We have $B \equiv b_0 + 5b_1 \pmod{2^{130} - 5}$. The next step is to compute $5b_1$ and add to b_0. This provides a 131-bit quantity. Our partial reduction strategy is not to reduce this any further. It is only at the end of the entire hash function computation, that a final reduction to a 130-bit quantity is performed.

6.4 Storage of key powers

For $2^{127} - 1$, the key τ is a 126-bit string which is stored as a 2-limb quantity. The higher powers of τ are 127-bit quantities and are also stored as 2-limb quantities. Since a 2-limb quantity requires 16 bytes to be stored, so for $2^{127} - 1$ each of the key powers is stored as a 16-byte quantity.

For $2^{130} - 5$, τ is a 128-bit string while the higher powers of τ are 130-bit quantities. To avoid converting from byte representation to limb representation, the key powers are stored as multi-limb quantities rather than multi-byte quantities. In the case of polyHash with $g = 1$, only τ is required and it is stored as a 2-limb quantity requiring 16 bytes. For all other cases, along with τ other higher key powers are required. In principle, τ can be stored as a 2-limb quantity and the higher

	# fld elts	# bytes	
		$2^{127}-1$	$2^{130}-5$
$g=1$	1	16	16
$g=4$	4	64	96
$g=8$	8	128	192
$g=16$	16	256	384
$g=32$	32	512	768

(a) Storage requirement of key powers for the evaluation of polyHash.

	# fld elts	# bytes	
		$2^{127}-1$	$2^{130}-5$
#blks $=\ell$	$\lceil \lg \ell \rceil$	$16\lceil \lg \ell \rceil$	$24\lceil \lg \ell \rceil$
#blks $=\ell \geq 4$, $g=4$	$1+\lceil \lg \ell \rceil$	$16+16\lceil \lg \ell \rceil$	$24+24\lceil \lg \ell \rceil$

(b) Storage requirement of key powers for the evaluation of BRWHash and t-BRWHash. The first row is for BRWHash and t-BRWHash with $g=1$, while the second row is only for t-BRWHash with $g=4$.

	# fld elts	# bytes	
		$2^{127}-1$	$2^{130}-5$
$d=2,\ g=1$	3	48	72
$d=2,\ g=4$	7	112	168
$d=2,\ g=8$	14	224	336
$d=3,\ g=1$	4	64	96
$d=3,\ g=4$	8	128	192
$d=3,\ g=8$	15	240	360
$d=4,\ g=1$	5	80	120
$d=4,\ g=4$	9	144	216
$d=4,\ g=8$	16	256	384
$d=5,\ g=1$	6	96	144
$d=5,\ g=4$	10	160	240
$d=5,\ g=8$	17	272	408

(c) Storage requirement of key powers for the evaluation of d-2LHash.

Table 6: Storage requirements of the different hash functions.

powers of τ as 3-limb quantities. The non-uniformity, however, makes access to the key powers less efficient. So τ as well as the higher key powers are stored as 3-limb quantities. So each key power requires 24 bytes to be stored.

For polyHash, the evaluation can be done by using groups of size $g \geq 1$. For d-2LHash, the value of d determines the number of key powers required for the BRW part of the computation. There are two Poly computations in d-2LHash, and we have considered the same value of g for both of these computations. Depending upon the values of d and g, d-2LHash will require to store a number of key powers. BRWHash will require a number of key powers which depends upon the number of blocks in the message. Similarly, the number of key powers required by t-BRWHash depends on the number of blocks in the message and the value of g used to perform the Poly part of the computation.

In Tables 6a, 6b and 6c, we provide the storage requirements of the various hash functions for specific values of the parameters. This is provided in two ways, first as the number of field elements that are required to be stored, and second as the number of bytes that are required to be stored. For $2^{127}-1$, each key power is stored as a 2-limb quantity, and so the number of bytes required to store all the key powers is 16 times the number of field elements. For $2^{130}-5$, other than the case of polyHash with $g=1$, in all other cases, each key power is stored as a 3-limb quantity, and so the number of bytes required to store all the key powers is 24 times the number of field elements.

6.5 Code

We have developed assembly code for the four hash functions modulo the two primes. The various options are as follows.

1. polyHash: Implementations with group size $g = 1, 4, 8, 16$ and 32.

2. BRWHash: Implementations with the parameter t in evalBRW taking the values 2 and 3.

3. t-BRWHash: Implementations with the parameter $t = 2, 3, 4$ and 5. The Poly part of the hash function has been implemented with group size $g = 1$ and 4.

4. d-2LHash: Implementations with the parameter $d = 2, 3, 4$ and 5. There are two Poly parts of the hash function. The same value of g has been used for both the parts. We have implemented the Poly parts with $g = 4$ and 8.

The hash function polyHash1305 coincides with Poly1305 for messages whose lengths are multiples of 8 (i.e. messages which are a byte stream). There are several public implementations of Poly1305. To the best of our knowledge, none of the implementations consider group size g to be greater than 1. Also, none of the implementations uses the double carry chain method for integer multiplication or the reduction algorithm that we have used. So on Intel Broadwell and later generation processors, our implementations provide new code for Poly1305.

The codes for our implementations of the hash functions are available from the following links.

```
https://github.com/kn-cs/polyHash
https://github.com/kn-cs/d2LHash
https://github.com/Sreyosi/EvalBRW-p1271
https://github.com/Sreyosi/EvalBRW-p1305
https://github.com/Sreyosi/t-BRWHash-p1271
https://github.com/Sreyosi/t-BRWHash-p1305
```

7 Trade-off between $2^{127} - 1$ and $2^{130} - 5$

Let $p_1 = 2^{127} - 1$ and $p_2 = 2^{130} - 5$.

There are several aspects of the trade-off. From the security point of view, it is required to compare the AXU bounds. Suppose messages of length at most L bits are to be hashed. The number of blocks is $\ell = \lceil L/n \rceil$, where $n = 120$ for p_1 and $n = 128$ for p_2. Using ℓ in Table 2, the AXU bound for polyHash1271 is about $2^{-132} \cdot 16L/15$, while the AXU bound for polyHash1305 is about $2^{-131} \cdot L$. Similarly, the AXU bounds for BRWHash1271, t-BRWHash1271, d-2LHash1271 and d-Hash1271 are all about $2^{-131} \cdot 16L/15 + 2^{-125}$, while the AXU bounds for BRWHash1305, t-BRWHash1305, d-2LHash1305 and d-Hash1305 are all about $2^{-130} \cdot L + 2^{-124}$. So in all cases, the AXU bounds for p_1 is about half the AXU bounds for p_2. This difference in the AXU bounds is negligible.

From an implementation point of view, there are two aspects to be considered, namely storage required for the key powers and the efficiency of computation. From Table 6, we see that other than polyHash with $g = 1$, the number of bytes required to store the key powers in the case of p_1 is two-thirds of the number of bytes required in the case of p_2.

Next we consider the efficiency of computation. From Table 1, message blocks in the case of p_1 are 120-bit strings and in the case of p_2 are 128-bit strings. So given a message, the number of message blocks in the case of p_1 is about 16/15 times the number of message blocks in the case of p_2. Consequently, the number of multiplications required by any of the four hash functions to process a message in the case of p_1 is about 16/15 times the number of multiplications in the case

of p_2. The fact that hashing in the case of p_1 requires more multiplications than hashing in the case of p_2 is one dimension of the efficiency trade-off.

The other dimension of the efficiency trade-off is the time required for a single field multiplication. From Table 5, one may note that the multiplications in the case of p_1 are all of 2-limb x 2-limb type, while in the case of p_2 these are either (2-limb x 3-limb)+ or 3-limb x 3-limb type. So the integer multiplication part of a field multiplication is faster in the case of p_1 than in the case of p_2. Further, since p_1 is a Mersenne prime, reduction modulo p_1 requires substantially fewer operations compared to reduction modulo p_2. The combined effect of faster integer multiplication and reduction is that a field multiplication modulo p_1 is substantially faster than a field multiplication modulo p_2.

So the overall efficiency trade-off is that hashing in the case of p_1 requires more field multiplications than hashing in the case of p_2, while the cost of an individual field multiplication modulo p_1 is less than that modulo p_2. The question then is whether for p_1 the faster speed of multiplication compensates the requirement of more multiplications? Our timing results reported in Table 9 of Section 8 shows that this is indeed the case by a very healthy margin.

To summarise, using p_1 instead of p_2 decreases the AXU bound by a negligible factor, and provides significant implementation advantages in terms of requiring lower storage and faster computation.

8 Timing results

There are two approaches to the implementation.

1. Pre-computed key powers. In this approach, the required key powers are pre-computed and stored. The computation and storage may be for a particular session, or on a more long term basis. With the pre-computed approach, the time for computing the key powers can be ignored while considering the time for hashing a message.

2. On-the-fly: In this approach, the required key powers are computed on a per message basis. Consequently, the time for generating the key powers is considered to be part of the entire time for hashing a message.

Depending on the application at hand, one of the above two approaches will be desirable. To understand the trade-off between the two approaches, in our timing experiments, we have obtained time measurements for both the approaches.

The timing measurements were taken on a single core of an Intel Core i5-8250U Kaby Lake processor running at 1.60GHz. During the experiments, turbo boost and hyperthreading options were turned off. The OS was Ubuntu 20.05.5 LTS and the code was compiled using gcc version 10.3.0. The following flags were used during compilation.

-march=native -mtune=native -m64 -O3 -funroll-loops -fomit-frame-pointer

The various options for the implementation of the hash functions have been described in Section 6.5. We have taken measurements for all the options of the four hash functions for both the primes by varying the number of message blocks from 1 to 512. This resulted in a large number of measurements. Providing all of these measurements in table form will require too much space. On the other hand, plotting so many points on a graph will make it very difficult to understand the

		# msg blks									
		50	100	150	200	250	300	350	400	450	500
polyHash1271	$g = 1$	1.30	1.28	1.28	1.28	1.27	1.27	1.27	1.27	1.27	1.27
	$g = 8$	0.66	0.64	0.65	0.63	0.63	0.63	0.63	0.62	0.62	0.62
		0.80	0.71	0.69	0.66	0.66	0.65	0.65	0.64	0.64	0.63
d-2LHash1271	$d = 4$	0.65	0.59	0.56	0.55	0.55	0.54	0.54	0.54	0.54	0.54
		0.94	0.74	0.66	0.62	0.61	0.59	0.59	0.57	0.57	0.57
d-2LHash1271	$d = 5$	0.66	0.58	0.58	0.55	0.54	0.55	0.53	0.54	0.53	0.53
		0.98	0.74	0.68	0.63	0.60	0.60	0.58	0.58	0.57	0.56
t-BRWHash1271	$t = 4$	0.73	0.65	0.62	0.61	0.61	0.62	0.60	0.59	0.59	0.61
		0.84	0.72	0.70	0.66	0.64	0.65	0.63	0.61	0.61	0.61
t-BRWHash1271	$t = 5$	0.77	0.66	0.65	0.61	0.63	0.61	0.62	0.59	0.58	0.59
		0.92	0.71	0.70	0.65	0.71	0.63	0.64	0.62	0.62	0.61

Table 7: Cycles/byte measurements for 50 to 500 blocks for the various hash functions based on the prime $2^{127} - 1$.

comparative performances of the different hash functions. So we have instead adopted the following approach.

In Appendix C, we provide the measurements for all the options where the number of message blocks varies from 1 to 32. This provides comprehensive comparison among the hash functions for short messages. From the large set of experimental results that we have recorded, we observed the following for messages with 32 or more blocks.

1. t-BRWHash with $t = 2$ or $t = 3$ does not perform better than either $t = 4$ or $t = 5$.
2. d-2LHash with $d = 2$ or $d = 3$ does not perform better than either $d = 4$ or $d = 5$.
3. BRWHash does not perform better than either d-2LHash or t-BRWHash, for $d = 4, 5$ and $t = 4, 5$.

In view of these observations, for 32 or more blocks, we do not provide the timings for BRWHash, t-BRWHash with $t = 2$ or $t = 3$ and d-2LHash with $d = 2$ or $d = 3$. For the other hash functions, the timing results are shown in Tables 7, 8 and 9. We note the following points regarding these tables.

1. Timing results are provided as the number of cycles per byte.
2. Each cell of all the tables has two entries. The upper entry denotes the time required when the key powers are pre-computed, while the lower entry denotes the time required when the key powers are computed on-the-fly. The only exception is the case of polyHash with $g = 1$, since in this case, other than the key itself, no other key powers are required, and so the issue of pre-computed versus on-the-fly computation of key powers does not arise.
3. Other than Table 9, the time measurements in all the other tables are for a particular prime and the message size is given as number of blocks. The comparison between the hash functions based on the two primes is given in Table 9 and in this table, the message size is given as number of bytes.

In Tables 7 and 8, we compare polyHash, d-2LHash with $d = 4, 5$, and t-BRWHash with $t = 4, 5$. For polyHash we considered $g = 1$ and $g = 8$.

There are several dimensions to the comparison between the various options.

		# msg blks									
		50	100	150	200	250	300	350	400	450	500
polyHash1305	$g = 1$	1.81	1.79	1.78	1.78	1.78	1.77	1.77	1.77	1.76	1.75
	$g = 8$	0.98	0.93	0.93	0.92	0.92	0.91	0.91	0.91	0.91	0.91
		1.18	1.03	1.00	0.97	0.96	0.95	0.94	0.93	0.93	0.93
d-2LHash1305	$d = 4$	0.88	0.83	0.81	0.80	0.79	0.79	0.79	0.79	0.78	0.78
		1.33	1.05	0.96	0.91	0.89	0.87	0.85	0.84	0.83	0.83
d-2LHash1305	$d = 5$	0.87	0.82	0.80	0.78	0.78	0.77	0.77	0.77	0.77	0.76
		1.36	1.06	0.96	0.90	0.87	0.85	0.84	0.83	0.82	0.81
t-BRWHash1305	$t = 4$	0.95	0.87	0.84	0.84	0.83	0.83	0.82	0.82	0.82	0.81
		1.16	0.98	0.95	0.96	0.89	0.87	0.87	0.85	0.85	0.84
t-BRWHash1305	$t = 5$	0.95	0.88	0.85	0.85	0.83	0.82	0.82	0.89	0.82	0.81
		1.16	0.97	0.94	0.90	0.89	0.87	0.87	0.85	0.85	0.84

Table 8: Cycles/byte measurements for 50 to 500 blocks for the various hash functions based on the prime $2^{130} - 5$.

	$\mathbb{F}_{2^{130}-5}$						$\mathbb{F}_{2^{127}-1}$					
	polyHash1305		d-2LHash1305		t-BRWHash1305		polyHash1271		d-2LHash1271		t-BRWHash1271	
bytes	$g = 1$	$g = 8$	$d = 4$	$d = 5$	$t = 4$	$t = 5$	$g = 1$	$g = 8$	$d = 4$	$d = 5$	$t = 4$	$t = 5$
10	5.94	5.98	12.66	12.89	9.11	9.11	5.43	5.57	10.17	10.19	7.38	7.39
		20.22	48.48	51.51	12.40	11.92		14.11	32.55	34.38	9.99	9.85
50	3.11	2.54	3.07	3.09	2.45	2.45	2.11	1.61	2.73	2.71	2.11	2.12
		5.51	10.32	10.92	4.22	3.50		3.68	7.22	7.54	2.99	2.93
100	2.42	1.61	2.03	2.03	1.51	1.51	1.62	1.03	1.70	1.72	1.34	1.34
		3.14	5.67	5.95	2.61	2.25		2.07	3.94	4.11	1.87	1.82
500	1.90	1.04	1.00	0.97	1.04	1.03	1.35	0.72	0.71	0.68	0.81	0.81
		1.35	1.72	1.75	1.45	1.30		0.92	1.16	1.18	0.99	1.03
1000	1.77	0.96	0.87	0.86	0.92	0.89	1.30	0.66	0.62	0.61	0.68	0.68
		1.12	1.23	1.25	1.15	1.11		0.76	0.84	0.85	0.79	0.78
2000	1.79	0.94	0.81	0.80	0.86	0.84	1.29	0.66	0.58	0.57	0.63	0.65
		1.02	0.99	0.99	0.99	1.02		0.71	0.69	0.69	0.70	0.69
3000	1.73	0.93	0.81	0.78	0.84	0.83	1.27	0.63	0.55	0.55	0.61	0.60
		0.98	0.93	0.91	0.94	0.91		0.66	0.62	0.63	0.67	0.65
5000	1.72	0.91	0.79	0.77	0.82	0.81	1.27	0.63	0.54	0.55	0.61	0.60
		0.95	0.87	0.85	0.89	0.87		0.65	0.59	0.60	0.63	0.65

Table 9: Cycles/byte measurements for 10 to 5000 bytes for the various hash functions based on the primes $2^{127} - 1$ and $2^{130} - 5$.

8.1 Comparison between $2^{127}-1$ and $2^{130}-5$

The results show that for each of the hash functions, instantiation using $2^{127}-1$ is faster than $2^{130}-5$ in every possible scenario that we considered. Consider Table 9 which compares the instantiations of the hash functions for the two primes for different length messages. For polyHash with $g=1$, the speed gain for a 10-byte message is about 10% and it increases to about 25% for a 5000-byte message. For polyHash with $g=8$ and pre-computed key powers, the speed gain for a 10-byte message is about 7% and it increases to about 30% for a 5000-byte message. For polyHash with $g=8$ with keys computed on-the-fly, the speed gain is about 30% for message lengths from 10 bytes to 5000 bytes. Similar substantial speed improvements are observed for all the other hash functions.

8.2 Comparison between polyHash for $g=1$ and $g=8$

Table 7 provides the comparison for the prime $2^{127}-1$, while Table 8 provides the comparison for the prime $2^{130}-5$. For messages having 50 or more blocks, there is a significant speed improvement by considering $g=8$ in comparison to $g=1$. This holds irrespective of whether the key powers are pre-computed or computed on-the-fly. For example, for the prime $2^{127}-1$, the speed improvement for a 50-block message is about 50% if the key powers are pre-computed and is about 38% if the key powers are computed on-the-fly. For a 500-block message, the speed improvement for pre-computed key powers remains about the same, while the speed improvement for on-the-fly computation of key powers increases to about 50%. Similar significant speed improvement is observed for the prime $2^{130}-5$.

For messages having up to 32 blocks, the tables in Appendix C show that for very short messages, if key powers are computed on-the-fly then there is a substantial loss of speed in using $g>1$. This, however, is expected since the time to compute the key powers is taken into the measurement, but due to the small length of the message, the benefit of using the key powers cannot be obtained. On the other hand, if the key powers are pre-computed, then substantial speed improvement is observed for $g=4$ and $g=8$ over $g=1$ for messages as small as having only 3 blocks.

8.3 Comparison between polyHash and the various BRW-based hash functions

From a theoretical standpoint, the number of integer multiplications required by polyHash is about two times that in BRWHash and the number of reductions required is about four times that in BRWHash. So one may expect BRWHash to perform about two times faster than polyHash. From the tables in Appendix C, one may observe that compared to polyHash with $g=1$, BRWHash with pre-computed key powers achieve about 30% to 40% speed improvement for messages having 25 or more blocks. The picture, however, changes when polyHash is computed with $g>1$. For short messages, BRWHash no longer provides speed improvement over polyHash. See Remark 6 for an explanation of the substantial speed gain for polyHash achieved by considering $g>1$. If we switch to d-2LHash with pre-computed key powers, then from Tables 7 and 8, we observe speed improvements of around 10% over polyHash with $g=8$ for messages having 50 or more blocks; if the key powers are computed on-the-fly, then the speed improvements are observed for messages having 150 or more blocks. A similar, though lower, speed improvement behaviour is observed for t-BRWHash over polyHash.

The rationale for t-BRWHash has been explained in Section 5.3. The timings measurements show that d-2LHash is in general faster than t-BRWHash. On the other hand, from Table 6c it

follows that for practical sized messages, the storage requirement for d-2LHash is more than that required for t-BRWHash.

8.4 The hash function d-Hash

In Section 3.1, it was mentioned that polyHash is the fastest when the number of blocks is small, and its performance lags behind the others when the number of blocks grows. This motivated the design of the hash function d-Hash in (11) which applies polyHash when the number of blocks is less than 2^d and applies d-2LHash when the number of blocks is at least 2^d. The AXU bound on d-Hash is given by Theorem 3.

We put forward 4-Hash as a secure hash function which provides the speed benefit of polyHash for short messages and the speed benefit of 4-2LHash for long messages. In particular, 4-Hash applies polyHash when the number of blocks is less than 16 and applies 4-2LHash when the number of blocks is at least 16. Instantiating 4-Hash with the prime $2^{127} - 1$ gives us the hash function 4-Hash1271 which provides the fastest hashing (among all the options) for messages of all lengths.

8.5 Comparison between 4-Hash1271 and Poly1305

Presently, the fastest polynomial hash function over prime order fields is Poly1305. This is a widely used hash function and is part of TLS. So it is important to determine what improvement is achieved by the best construction in the present paper over Poly1305. As per our naming convention (see Section 3.2), the hash function polyHash1305 is identical to Poly1305 for messages whose lengths are multiples of eight.

Based on the discussion in Section 8.4, the most efficient hash function for messages of all sizes is 4-Hash1271. From Table 9, we observe that 4-Hash1271 is faster than polyHash1305 for messages of all sizes, with the speed improvement ranging from about 8.5% (for 10-byte messages) to about 40% (for 5-kilobyte messages). This makes 4-Hash1271 an attractive target for further implementation studies to confirm whether the speed improvement over Poly1305 is indeed preserved for other implementations and on other platforms.

9 Conclusion

A major finding of the present work is that for polynomial hashing over prime order fields, using the prime $2^{127} - 1$ results in significantly faster hashing compared to the prime $2^{130} - 5$. While this finding is based on our specific implementations for Intel processors, we do not envisage any platform where using $2^{127} - 1$ will result in slower hashing than using $2^{130} - 5$. Confirming (or not) this prediction is a possible direction of future implementation work.

The other major finding is that a judicious mix of Poly and BRW polynomials can lead to significant speed improvement over using only Poly. A direction of future work is to examine whether the speed improvement holds for SIMD implementations. For Poly1305, SIMD implementations have been reported in [13, 1, 5]. Possible improved SIMD implementation of Poly1305 and new SIMD implementation of polyHash1271 remains to be done. For BRW, parallel hardware implementation for binary extension fields have been reported in [9]. Whether these can be translated to SIMD implementation in software, or whether there is a different method of implementing BRW using SIMD in software is a topic for future research. The other direction of possible future implementation work is to obtain implementations of the hash functions proposed in this paper on ARM

processors.

Acknowledgement

We are grateful to the reviewers for their detailed comments which have helped in improving the paper.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: high-assurance and high-speed cryptographic implementations. *CoRR*, abs/1904.04606, 2019.

[2] Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2004. `https://cr.yp.to/papers.html#hash127`.

[3] 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.

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

[5] Sreyosi Bhattacharyya and Palash Sarkar. Improved SIMD implementation of Poly1305. *IET Inf. Secur.*, 14(5):521–530, 2020.

[6] 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.

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

[8] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. *IACR Trans. Symmetric Cryptol.*, 2017(1):106–128, 2017.

[9] 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.

[10] Debrup Chakraborty, Cuauhtemoc Mancillas-López, and Palash Sarkar. STES: A stream cipher based low cost scheme for securing stored data. *IEEE Trans. Computers*, 64(9):2691–2707, 2015.

[11] Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, 2:65–72, 1993.

[12] Sebati Ghosh and Palash Sarkar. Evaluating Bernstein-Rabin-Winograd polynomials. *Designs, Codes, and Cryptography*, 87(2-3):527–546, 2019.

[13] Martin Goll and Shay Gueron. Vectorization of Poly1305 message authentication code. In *2015 12th International Conference on Information Technology - New Generations*, pages 145–150. IEEE, April 2015. 10.1109/ITNG.2015.28.

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

[15] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authenticated encryption mode. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 408–426. Springer, 2004.

[16] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)Mersenne prime order fields. *Advances in Mathematics of Communications*, 16(2):303–348, 2022.

[17] E. Ozturk, J. Guilford, and V. Gopal. Large integer squaring on Intel architecture processors, intel white paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf, 2013.

[18] E. Ozturk, J. Guilford, V. Gopal, and W. Feghali. New instructions supporting large integer arithmetic on Intel architecture processors, intel white paper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf`, 2012.

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

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

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

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

[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] Richard Taylor. An integrity check value algorithm for stream ciphers. In Stinson [23], pages 40–48, `https://link.springer.com/content/pdf/10.1007/3-540-48329-2_4.pdf`.

A Straight line code for computing BRW

For d-2LHash, we provide examples of straight line codes to compute BRW for various values of d. While these are illustrative, our implementations do not exactly correspond to these straight line codes. Through experimentation, we developed variations which are more efficient in practice though the total number of integer multiplications and reductions remain unchanged.

Case $d = 2$ and $\delta = 3$: The output of $\mathsf{BRW}_\tau(M_1, M_2, M_3)$ is the following expression.

$$(M_1 + \tau)(M_2 + \tau^2) + M_3.$$

A straight line code to compute this expression using one integer multiplication and one reduction is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$
2. $T_2 \leftarrow T_1 T_2$
3. $T_2 \leftarrow T_2 + M_3$
4. Output $\mathsf{reduce}(T_2)$

Case $d = 3$ and $\delta = 7$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_7)$ is the following expression.

$$((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7.$$

A straight line code to compute this expression using three integer multiplication and two reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_3 \leftarrow M_5 + \tau$; $T_4 \leftarrow M_6 + \tau^2$
2. $T_2 \leftarrow T_1 T_2$; $T_4 \leftarrow T_3 T_4$
3. $T_2 \leftarrow T_2 + M_3$; $T_4 \leftarrow T_4 + M_7$
4. $T_1 \leftarrow M_4 + \tau^4$
5. $T_2 \leftarrow \mathsf{reduce}(T_2)$; $T_2 \leftarrow T_1 T_2$
6. $T_4 \leftarrow T_2 + T_4$
7. Output $\mathsf{reduce}(T_4)$

Case $d = 4$ and $\delta = 15$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_{15})$ is the following expression.

$$\begin{aligned}&(((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7)(M_8 + \tau^8)\\ &\quad +((M_9 + \tau)(M_{10} + \tau^2) + M_{11})(M_{12} + \tau^4) + (M_{13} + \tau)(M_{14} + \tau^2) + M_{15}.\end{aligned}$$

A straight line code to compute this expression using seven integer multiplication and four reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_3 \leftarrow M_5 + \tau$; $T_4 \leftarrow M_6 + \tau^2$;
   $T_5 \leftarrow M_9 + \tau$; $T_6 \leftarrow M_{10} + \tau^2$; $T_7 \leftarrow M_{13} + \tau$; $T_8 \leftarrow M_{14} + \tau^2$
2. $T_2 \leftarrow T_1 T_2$; $T_4 \leftarrow T_3 T_4$; $T_6 \leftarrow T_5 T_6$; $T_8 \leftarrow T_7 T_8$
3. $T_2 \leftarrow T_2 + M_3$; $T_4 \leftarrow T_4 + M_7$; $T_6 \leftarrow T_6 + M_{11}$; $T_8 \leftarrow T_8 + M_{15}$
4. $T_1 \leftarrow M_4 + \tau^4$; $T_5 \leftarrow M_{12} + \tau^4$;
5. $T_2 \leftarrow \mathsf{reduce}(T_2)$; $T_2 \leftarrow T_1 T_2$; $T_6 \leftarrow \mathsf{reduce}(T_6)$; $T_6 \leftarrow T_5 T_6$
6. $T_4 \leftarrow T_2 + T_4$; $T_8 \leftarrow T_6 + T_8$
7. $T_3 \leftarrow M_8 + \tau^8$
8. $T_4 \leftarrow \mathsf{reduce}(T_4)$; $T_4 \leftarrow T_3 T_4$
9. $T_8 \leftarrow T_4 + T_8$
10. Output $\mathsf{reduce}(T_8)$

Case $d = 5$ and $\delta = 31$: The output of $\mathsf{BRW}_\tau(M_1, \ldots, M_{31})$ is the following expression.

$$\begin{aligned}&((((M_1 + \tau)(M_2 + \tau^2) + M_3)(M_4 + \tau^4) + (M_5 + \tau)(M_6 + \tau^2) + M_7)(M_8 + \tau^8)\\ &\quad +((M_9 + \tau)(M_{10} + \tau^2) + M_{11})(M_{12} + \tau^4) + (M_{13} + \tau)(M_{14} + \tau^2) + M_{15})(M_{16} + \tau^{16})\\ &+(((M_{17} + \tau)(M_{18} + \tau^2) + M_{19})(M_{20} + \tau^4) + (M_{21} + \tau)(M_{22} + \tau^2) + M_{23})(M_{24} + \tau^8)\\ &\quad +((M_{25} + \tau)(M_{26} + \tau^2) + M_{27})(M_{28} + \tau^4) + (M_{29} + \tau)(M_{30} + \tau^2) + M_{31}.\end{aligned}$$

A straight line code to compute this expression using fifteen integer multiplication and eight reductions is the following.

1. $T_1 \leftarrow M_1 + \tau$; $T_2 \leftarrow M_2 + \tau^2$; $T_5 \leftarrow M_5 + \tau$; $T_6 \leftarrow M_6 + \tau^2$
   $T_9 \leftarrow M_9 + \tau$; $T_{10} \leftarrow M_{10} + \tau^2$
2. $T_2 \leftarrow T_1 T_2$; $T_6 \leftarrow T_5 T_6$; $T_{10} \leftarrow T_9 T_{10}$
3. $T_{13} \leftarrow M_{13} + \tau$; $T_{14} \leftarrow M_{14} + \tau^2$; $T_{17} \leftarrow M_{17} + \tau$; $T_{18} \leftarrow M_{18} + \tau^2$
   $T_{21} \leftarrow M_{21} + \tau$; $T_{22} \leftarrow M_{22} + \tau^2$
4. $T_{14} \leftarrow T_{13} T_{14}$; $T_{18} \leftarrow T_{17} T_{18}$; $T_{22} \leftarrow T_{21} T_{22}$
5. $T_{25} \leftarrow M_{25} + \tau$; $T_{26} \leftarrow M_{26} + \tau^2$; $T_{29} \leftarrow M_{29} + \tau$; $T_{30} \leftarrow M_{30} + \tau^2$
   $T_2 \leftarrow T_2 + M_3$; $\mathsf{reduce}(T_2)$; $T_4 \leftarrow M_4 + \tau^4$
6. $T_4 \leftarrow T_2 T_4$; $T_{26} \leftarrow T_{25} T_{26}$; $T_{30} \leftarrow T_{29} T_{30}$
7. $T_{10} \leftarrow T_{10} + M_{11}$; $\mathsf{reduce}(T_{10})$; $T_{12} \leftarrow M_{12} + \tau^4$; $T_{20} \leftarrow M_{20} + \tau^4$
   $T_{18} \leftarrow T_{18} + M_{19}$; $\mathsf{reduce}(T_{18})$; $T_8 \leftarrow M_8 + \tau^8$; $T_6 \leftarrow T_6 + M_7$; $T_6 \leftarrow T_4 + T_6$; $\mathsf{reduce}(T_6)$
8. $T_{12} \leftarrow T_{10} T_{12}$; $T_{20} \leftarrow T_{18} T_{20}$; $T_8 \leftarrow T_6 T_8$
9. $T_{28} \leftarrow M_{28} + \tau^4$; $T_{26} \leftarrow T_{26} + M_{27}$; $\mathsf{reduce}(T_{26})$; $T_{24} \leftarrow M_{24} + \tau^8$; $T_{22} \leftarrow T_{22} + M_{23}$; $\mathsf{reduce}(T_{22})$
   $T_{16} \leftarrow M_{16} + \tau^{16}$; $T_{14} \leftarrow T_{14} + M_{15}$; $T_{14} \leftarrow T_{12} + T_{14}$; $T_{14} \leftarrow T_8 + T_{14}$; $\mathsf{reduce}(T_{14})$
10. $T_{28} \leftarrow T_{26} T_{28}$; $T_{24} \leftarrow T_{22} T_{24}$; $T_{16} \leftarrow T_{14} T_{16}$
11. $T_{30} \leftarrow T_{30} + M_{31}$; $T_{30} \leftarrow T_{28} + T_{30}$; $T_{30} \leftarrow T_{24} + T_{30}$; $T_{30} \leftarrow T_{16} + T_{30}$
12. Output $\mathsf{reduce}(T_{30})$

B Correctness and complexity of Algorithm 1

We require the following result from [12].

Lemma 6 (Lemma 2 of [12])**.** *Let $t \geq 2$ be an integer. For any $l \geq 2^t$, write*

$$\left\lfloor \frac{l}{2^t} \right\rfloor \quad = \quad 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}, \tag{16}$$

where $k_1, \ldots, k_s$ are integers such that $k_1 > k_2 > \cdots > k_s \geq 0$. Let $K_0 = 0$ and for $\jmath = 0, \ldots, s-1$, let $K_{\jmath+1} = K_\jmath + 2^{t+k_{\jmath+1}}$. Then

$$\begin{aligned} &\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l) \\ &= \mathsf{unreducedBRW}(\tau; M_{K_0+1}, \ldots, M_{K_1}) + \mathsf{unreducedBRW}(\tau; M_{K_1+1}, \ldots, M_{K_2}) \\ &\quad + \cdots + \mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1}, \ldots, M_{K_s}) + \mathsf{unreducedBRW}(\tau; M_{K_s+1}, \ldots, M_l). \end{aligned} \tag{17}$$

It is easy to see that for $l > 2$, Steps 3 to 7 of Algorithm 1 ensure that $\mathsf{keyPow}[j] = \tau^{2^j}$, for $j = 1, \ldots, \lfloor \lg l \rfloor$. The main result required to argue the correctness of Algorithm 1 is the following which shows that the partial results are correctly computed and stored. This result is the counterpart of Lemma 5 of [12].

Lemma 7. *Let $t \geq 2$ and $l \geq 2^t$. Let the loop counter $i \in \{1, \ldots, i_{\max}\}$, with $i_{\max} = \lfloor l/2^t \rfloor$, in Step 9 of Algorithm 1 be written as*

$$i \quad = \quad 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \tag{18}$$

where $k_{i,1} > k_{i,2} > \cdots > k_{i,s_i} \geq 0$. Let $K_{i,0} = 0$ and for $\jmath = 0, \ldots, s_i - 1$, let $K_{i,\jmath+1} = K_{i,\jmath} + 2^{t+k_{i,\jmath+1}}$. After i iterations of the loop given by Steps 9 to 17, the following properties hold:

$\mathsf{top} = s_i - 1$, *and for* $\jmath \in \{0, \ldots, s_i - 1\}$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$.

Proof. First note that from the definition of $K_{i,\jmath}$, we have

$$K_{i,0} = 0,\ K_{i,1} = 2^{t+k_{i,1}}, K_{i,2} = 2^{t+k_{i,1}} + 2^{t+k_{i,2}}, \ldots, K_{i,s_i} = 2^{t+k_{i,1}} + \cdots + 2^{t+k_{i,s_i}} = 2^t \cdot i. \tag{19}$$

The proof is by induction on $i \geq 1$. It mirrors the proof of Lemma 5 of [12], with modifications to handle the different manner in which the partial results are stored and accessed by Algorithm 1 from the algorithm in [12].

The base case is $i = 1$. In this case, $s_1 = 1$, $k_{1,1} = 0$ and $K_{1,1} = 2^t$. The variable tmp is set to $\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{2^t-1})$; the variable k is set to $\mathsf{ntz}(1) = 0$ and so the loop in Steps 12 to 14 is not executed. In Step 15, tmp is updated to $\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{tmp}), M_{2^t} + \mathsf{keyPow}[t])$. In Step 16, the variable top is incremented to 0 (from -1) and $\mathsf{stack}[0]$ is assigned the value of tmp. At the end of the first iteration, the value of top is $0 = s_1 - 1$. The argument for the base case of $i = 1$ will be completed if we are able to show that

$$\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{2^t-1})), M_{2^t} + \mathsf{keyPow}[t])$$

equals $\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{2^t})$. This equality follows from the definition of $\mathsf{unreducedBRW}$ (see Section 5.2).

For the inductive step suppose the result holds for $i = 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} \geq 1$. We show that the result holds for $i + 1$. We have

$$i + 1 \quad = \quad 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} + 1 = 2^{k_{i+1,1}} + 2^{k_{i+1,2}} + \cdots + 2^{k_{i+1,s_{i+1}}}.$$

Note that s_{i+1} can be smaller than s_i. For example, if $i = 11 = 2^3 + 2 + 1$, then $s_{11} = 3$, and $i + 1 = 12 = 2^3 + 2^2$ with $s_{12} = 2$. Such a situation arises when i is odd. So the proof now divides into two cases of i even and i odd.

First suppose that i is even, since this is the simpler of the two cases. Since i is even, $k_{i,s_i} > 0$ and so $i + 1 = 2^{k_{i,1}} + 2^{k_{i,2}} + \cdots + 2^{k_{i,s_i}} + 1$ leading to $s_{i+1} = s_i + 1$, $k_{i+1,1} = k_{i,1}, \ldots, k_{i+1,s_i} = k_{i,s_i}$ and $k_{i+1,s_{i+1}} = 0$. So

$$K_{i+1,1} = K_{i,1}, \ldots, K_{i+1,s_i} = K_{i,s_i} \text{ and } K_{i+1,s_{i+1}} = K_{i,s_i} + 2^t = 2^t(i+1), \tag{20}$$

where we use (19) to note that $K_{i,s_i} = 2^t \cdot i$.

By the induction hypothesis, at the end of the i-th iteration, the value of top is $s_i - 1$ and $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$ for $\jmath \in \{0, \ldots, s_i - 1\}$. In the $(i+1)$-st iteration, Step 10 sets tmp to $\mathsf{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})$, i.e. to

$$\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}).$$

Since $i + 1$ is odd, $\mathsf{ntz}(i + 1) = 0$, so Step 11 sets k to 0 and as a result, the loop in Steps 12 to 14 is not executed. Step 15 updates tmp to

$$\begin{aligned} &\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1})), M_{2^t(i+1)} + \mathsf{keyPow}[t]) \\ = \quad &\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1})), M_{2^t(i+1)} + \tau^{2^t}). \end{aligned} \tag{21}$$

Step 16 increments top to $s_i = s_{i+1} - 1$ and sets $\mathsf{stack}[s_{i+1} - 1]$ to the value in (21). From (19), $K_{i,s_i} = 2^t \cdot i$ and from (20), $K_{i+1,s_{i+1}} = 2^t(i+1)$. So the expression given by (21) can be written as

$$\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})), M_{2^t(i+1)} + \tau^{2^t}). \tag{22}$$

The number of blocks involved in (22) is 2^t and so using the definition of $\mathsf{unreducedBRW}$, we have

$$\begin{aligned} &\mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})), M_{2^t(i+1)} + \tau^{2^t}) \\ &\qquad = \quad \mathsf{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)}) \\ &\qquad = \quad \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}}). \end{aligned} \tag{23}$$

To see (23), we note that $s_i = s_{i+1} - 1$ and so $K_{i+1,s_{i+1}-1} = K_{i+1,s_i} = K_{i,s_i} = 2^t \cdot i$ and $K_{i+1,s_{i+1}} = 2^t(i+1)$. Putting together (21), (22) and (23), we see that $\mathsf{stack}[s_{i+1} - 1]$ contains the value as stated in the result. So at the end of the $(i+1)$-st iteration, the value of top and the entries of stack are as stated in the result.

Now suppose that i is odd. This case is more complicated since in this case $i+1$ is even and in the $(i+1)$-st iteration, the value of k in Step 11 will be positive and the loop in Steps 12 to 14 will be executed. Since i is odd, we have $k_{i,s_i} = 0$. Let $\beta \geq 1$ be such that $k_{i,s_i} = 0, k_{i,s_i-1} = 1, \ldots, k_{i,s_i-\beta+1} = \beta - 1$ and $k_{i,s_i-\beta} > \beta$. Then we can write

$$i = 2^0 + 2^1 + \cdots + 2^{\beta-1} + 2^{k_{i,s_i-\beta}} + \cdots + 2^{k_{i,1}} \quad \text{and} \quad i+1 = 2^\beta + 2^{k_{i,s_i-\beta}} + \cdots + 2^{k_{i,1}},$$

Consequently, $s_{i+1} = s_i - \beta + 1$, $k_{i+1,1} = k_{i,1}, \ldots, k_{i+1,s_i-\beta} = k_{i,s_i-\beta}$ and $k_{i+1,s_{i+1}} = k_{i+1,s_i-\beta+1} = \beta$. So

$$K_{i+1,0} = K_{i,0} = 0, \ldots, K_{i+1,s_i-\beta} = K_{i,s_i-\beta} \text{ and } K_{i+1,s_{i+1}} = K_{i+1,s_i-\beta} + 2^{t+\beta}.$$

By the induction hypothesis, at the end of the i-th iteration, $\mathsf{top} = s_i - 1$ and for $\jmath \in \{0, \ldots, s_i - 1\}$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$. For $\jmath = s_i - \beta, \ldots, s_i - 1$, let

$$X_\jmath = \mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}}).$$

Note that for $\imath = 0, \ldots, \beta - 1$, $2^{t+\imath}$ blocks are used in the computation of $X_{s_i-\imath-1}$.

Let us now consider what happens in the $(i+1)$-st iteration. In Step 10, tmp is assigned the value $Y = \mathsf{unreducedBRW}(\tau; M_{2^t \cdot i+1}, \ldots, M_{2^t(i+1)-1})$. Note that the computation of Y involves $2^t - 1$ blocks. From the expression for $i+1$ given above and the condition that $k_{i,s_i-\beta} > \beta$, we have $\mathsf{ntz}(i+1) = \beta$. So Step 11 sets k to β. The loop in Steps 12 to 14 is executed β times which removes β elements $X_{s_i-\beta}, \ldots, X_{s_i-1}$ from the top of the stack and adds these to the value of tmp. At the end of this loop, the value of top is $s_i - 1 - \beta$ and this value is incremented in Step 16 so that at the end of the $(i+1)$-st iteration, the value of top is $s_i - \beta = s_{i+1} - 1$ as required. For $\jmath \in \{0, \ldots, s_i - \beta - 1\}$, the value of $\mathsf{stack}[\jmath]$ does not change, and in Step 16, the new value of $\mathsf{stack}[s_i - \beta]$ is set. Our proof will be complete if we can argue that the values in stack at the end of the $(i+1)$-st iteration are as stated in the result. For $\jmath = 0, \ldots, s_i - \beta - 1$, the value in $\mathsf{stack}[\jmath]$ at the end of the $(i+1)$-st round is the same as that at the end of the i-th round and this value is $\mathsf{unreducedBRW}(\tau; M_{K_{i,\jmath}+1}, \ldots, M_{K_{i,\jmath+1}})$. As argued above, $K_{i+1,\jmath} = K_{i,\jmath}$ for $\jmath = 0, \ldots, s_i - \beta$. So the values in $\mathsf{stack}[\jmath]$, $\jmath = 0, \ldots, s_i - \beta - 1$, at the end of the $(i+1)$-st round are as stated in the result. Noting that $s_{i+1} - 1 = s_i - \beta$, the value in $\mathsf{stack}[s_{i+1} - 1]$ at the end of the $(i+1)$-st round is

$$\begin{aligned} &\mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta} + \cdots + X_{s_i-1} + Y), (M_{2^t(i+1)} + \mathsf{keyPow}[t+\beta])) \\ &\quad = \quad \mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta} + \cdots + X_{s_i-1} + Y), (M_{2^t(i+1)} + \tau^{2^{t+\beta}})). \end{aligned} \tag{24}$$

The $X_\jmath$'s are the outputs of unreduced BRW computations on consecutives blocks. Further, as mentioned above, $2^{t+\beta-1}$ blocks are used in the computation of $X_{s_i-\beta}$; $2^{t+\beta-2}$ blocks are used in the computation of $X_{s_i-\beta+1}$; and so on, finally 2^t blocks are used in the computation of X_{s_i-1}. The quantity Y is the output of an unreduced BRW computation on $2^t - 1$ consecutive blocks

immediately following the blocks used in the computation of the X_i's. So we have

$$\begin{aligned}
&X_{s_i-\beta} + \cdots + X_{s_i-1} + Y \\
&\quad = \mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta}+1}, \ldots, M_{K_{i,s_i-\beta+1}}) \\
&\qquad +\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta+1}+1}, \ldots, M_{K_{i,s_i-\beta+2}}) \\
&\qquad +\cdots \\
&\qquad +\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-1}+1}, \ldots, M_{K_{i,s_i}}) \\
&\qquad +\mathsf{unreducedBRW}(\tau; M_{K_{i,s_i}+1}, \ldots, M_{2^t(i+1)-1}) \\
&\quad = \mathsf{unreducedBRW}(\tau; M_{K_{i,s_i-\beta}+1}, \ldots, M_{2^t(i+1)-1}) \\
&\quad = \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}-1}). \qquad (25)
\end{aligned}$$

The last but one equality follows from Lemma 6 and the last equality follows from $K_{i+1,s_{i+1}-1} = K_{i+1,s_i-\beta} = K_{i,s_i-\beta}$ and $K_{i+1,s_{i+1}} = 2^t(i+1)$. Using (25) in (24), we obtain

$$\begin{aligned}
&\mathsf{unreducedMult}(\mathsf{reduce}(X_{s_i-\beta} + \cdots + X_{s_i-1} + Y), (M_{2^t(i+1)} + \mathsf{keyPow}[t+\beta])) \\
&\qquad = \mathsf{unreducedBRW}(\tau; M_{K_{i+1,s_{i+1}-1}+1}, \ldots, M_{K_{i+1,s_{i+1}}}).
\end{aligned}$$

This completes the proof. □

Proof of Theorem 4. To show that Algorithm 1 correctly computes $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$, it is sufficient to show that the value of tmp in Step 24 is equal to $\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l)$. The argument is similar to the proof of Theorem 1 of [12] with modifications required to handle the different manner in which partial results are stored.

If $l < 2^t$, then the loop in Steps 9 to 23 is not executed and in Step 19, tmp is assigned the value $\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l)$ which shows the result for $l < 2^t$. So suppose $l \geq 2^t$ and let

$$\lfloor l/2^t \rfloor = 2^{k_1} + \cdots + 2^{k_s},$$

$K_0 = 0$, $K_1 = 2^{t+k_1}$, $K_2 = 2^{t+k_1} + 2^{t+k_2}, \ldots, K_s = 2^{t+k_1} + \cdots + 2^{t+k_s}$. Let $r = l \bmod 2^t$ and write $l = 2^t(2^{k_1} + \cdots + 2^{k_s}) + r$ so that $K_s = l - r$. From Lemma 6 we have

$$\begin{aligned}
&\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_l) \\
&\quad = \mathsf{unreducedBRW}(\tau; M_{K_0+1}, \ldots, M_{K_1}) + \cdots + \mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1}, \ldots, M_{K_s}) \\
&\qquad +\mathsf{unreducedBRW}(\tau; M_{K_s+1}, \ldots, M_l) \\
&\quad = \mathsf{unreducedBRW}(\tau; M_{K_0+1}, \ldots, M_{K_1}) + \cdots + \mathsf{unreducedBRW}(\tau; M_{K_{s-1}+1}, \ldots, M_{K_s}) \\
&\qquad +\mathsf{unreducedBRW}(\tau; M_{l-r+1}, \ldots, M_l). \qquad (26)
\end{aligned}$$

From Lemma 7, at the end of loop from Steps 9 to 18, $\mathsf{top} = s-1$ and for $\jmath = 0, \ldots, s-1$, $\mathsf{stack}[\jmath] = \mathsf{unreducedBRW}(\tau; M_{K_\jmath+1}, \ldots, M_{K_{\jmath+1}})$. Step 19 assigns $\mathsf{unreducedBRW}(\tau; M_{l-r+1}, \ldots, M_l)$ to tmp. The value of i in Step 20 is set to $\mathsf{wt}(\lfloor l/2^t \rfloor) = s$. The loop from Steps 21 to 23 adds the values of $\mathsf{stack}[\jmath]$, $\jmath = 0, \ldots, s-1$, to tmp. So in Step 24, the value of tmp is given by (26). This completes the proof of correctness.

The argument for the operation counts are exactly the same as in the proof of Theorem 2 of [12].

From Lemma 7, the maximum value of top is

$$\max_{1 \leq i \leq \lfloor l/2^t \rfloor} s_i - 1 \quad = \quad \max_{1 \leq i \leq \lfloor l/2^t \rfloor} \mathsf{wt}(i) - 1. \qquad (27)$$

Claim: For $l \geq 2^t$,

$$\max_{1 \leq i \leq \lfloor l/2^t \rfloor} \mathsf{wt}(i) \quad \leq \quad \lfloor \lg l \rfloor - t + 1. \tag{28}$$

Proof of claim: Note that the maximum on the left hand side of (28) is achieved for the value of i which is maximum in the given range and is of the form one less than some power of two. For $l = 2^t$, it can be easily verified that equality holds in (28). For $l = 2^{t_1}$ for some $t_1 > t$, the left hand side of (28) equals $\mathsf{wt}(2^{t_1-t} - 1) = t_1 - t$, while the right hand side of (28) equals $t_1 - t + 1$. For $t_1 \geq t$ and $2^{t_1} < l < 2^{t_1+1}$, the right hand side of (27) equals $t_1 - t + 1$. Also, we have $2^{t_1-t} \leq \lfloor l/2^t \rfloor \leq 2^{t_1-t+1} - 1$ and so the left hand side of (28) is at most $t_1 - t + 1$. □

Using (28) in (27), we obtain the maximum value of top to be

$$\max_{1 \leq i \leq \lfloor l/2^t \rfloor} \mathsf{wt}(i) - 1 \quad \leq \quad \lfloor \lg l \rfloor - t. \tag{29}$$

So the maximum size of stack at any point in the algorithm is at most $\lfloor \lg l \rfloor - t + 1$. (Note that (29) is tight, for example one may choose $l = 255$ and $t = 3$; On the other hand, for $l = 256$ and $t = 3$, the bound is loose, so one cannot replace the inequality by equality.) □

C Timing measurements for messages with few blocks

For each prime, the measurements for the different hash functions are divided into four tables, providing measurements for block sizes from 1 to 8, from 9 to 16, from 17 to 24, and from 25 to 32. The explanation of the entries in the tables are the same as that mentioned in Section 8.

		# msg blks							
		1	2	3	4	5	6	7	8
polyHash1271	$g = 1$	3.35	2.23	1.90	1.73	1.65	1.58	1.54	1.50
	$g = 4$	3.38	2.27	1.52	1.23	1.24	1.08	0.98	0.95
		5.65	3.03	2.13	1.77	1.65	1.44	1.30	1.23
	$g = 8$	3.38	2.28	1.55	1.24	1.07	0.99	0.90	0.89
		9.31	5.36	3.79	2.97	2.47	2.14	1.92	1.74
	$g = 16$	3.38	2.28	1.56	1.24	1.07	0.98	0.91	0.86
		19.24	10.04	6.85	5.24	4.33	3.71	3.25	2.91
	$g = 32$	3.38	2.28	1.55	1.23	1.07	0.98	0.91	0.88
		41.94	21.33	14.43	10.93	8.87	7.50	6.51	5.77
BRWHash1271	$t = 2$	4.49	2.74	1.87	2.22	1.93	1.61	1.41	1.40
		4.99	3.39	2.23	2.72	2.29	1.95	1.67	1.74
	$t = 3$	4.44	2.77	1.91	1.80	1.47	1.27	1.21	1.40
		7.60	3.07	2.10	2.19	1.81	1.55	1.37	1.70
d-2LHash1271	$d = 2$	5.48	3.15	1.90	1.62	1.45	1.27	1.22	1.14
		14.50	7.78	4.88	3.87	3.30	2.78	2.51	2.32
	$d = 3$	6.63	3.60	2.58	2.16	1.85	1.62	1.35	1.20
		20.47	10.45	7.163	5.62	4.59	3.92	3.28	2.88
	$d = 4$	6.58	3.59	2.56	2.15	1.85	1.65	1.59	1.39
		21.52	11.01	7.52	5.88	4.82	4.11	3.72	3.26
	$d = 5$	6.57	3.58	2.57	2.16	1.85	1.64	1.59	1.40
		22.75	11.62	7.94	6.20	5.07	4.32	3.88	3.40
t-BRWHash1271	$t = 2$	4.90	2.84	2.13	2.16	2.05	1.75	1.58	1.38
		9.93	5.29	3.60	3.62	3.23	2.72	2.47	2.18
	$t = 3$	4.99	2.87	2.10	1.71	1.54	1.41	1.30	1.40
		6.56	3.71	2.65	2.40	2.28	1.92	1.75	1.85
	$t = 4$	4.90	2.83	2.07	1.71	1.61	1.41	1.28	1.20
		6.50	3.71	2.69	2.37	2.29	1.91	1.73	1.75
	$t = 5$	4.99	2.88	2.11	1.77	1.59	1.51	1.33	1.24
		6.46	3.81	2.66	2.38	2.29	1.92	1.82	1.72

Table 10: Cycles/byte measurements for 1 to 8 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		9	10	11	12	13	14	15	16
polyHash1271	$g = 1$	1.48	1.45	1.44	1.42	1.41	1.40	1.39	1.38
	$g = 4$	0.95	0.91	0.86	0.84	0.86	0.84	0.81	0.80
		1.20	1.13	1.07	1.02	1.03	1.00	0.96	0.94
	$g = 8$	0.90	0.87	0.83	0.81	0.79	0.84	0.81	0.75
		1.67	1.55	1.46	1.38	1.32	1.27	1.22	1.19
	$g = 16$	0.83	0.81	0.82	0.76	0.81	0.73	0.72	0.82
		2.67	2.45	2.29	2.14	2.05	1.95	1.84	1.81
	$g = 32$	0.82	0.81	0.77	0.76	0.75	0.81	0.78	0.78
		5.18	4.73	4.35	4.03	3.78	3.57	3.37	3.21
BRWHash1271	$t = 2$	1.34	1.20	1.10	1.12	1.13	1.03	0.99	1.00
		1.61	1.49	1.34	1.77	1.30	1.32	1.15	1.25
	$t = 3$	1.33	1.20	1.09	1.14	1.08	1.01	0.98	1.02
		1.54	1.43	1.34	1.35	1.23	1.18	1.12	1.22
d-2LHash1271	$d = 2$	1.02	1.02	0.99	0.89	0.91	0.90	0.82	0.83
		2.04	1.92	1.82	1.66	1.59	1.55	1.43	1.39
	$d = 3$	1.11	1.04	1.07	1.05	0.98	0.91	0.90	0.88
		2.60	2.39	2.30	2.16	2.02	1.88	1.79	1.72
	$d = 4$	1.30	1.25	1.19	1.22	1.14	1.07	0.87	0.84
		2.95	2.72	2.51	2.41	2.27	2.13	1.85	1.75
	$d = 5$	1.29	1.21	1.17	1.15	1.10	1.06	1.09	1.02
		3.08	2.82	2.61	2.49	2.34	2.20	2.15	2.03
t-BRWHash1271	$t = 2$	1.45	1.37	1.25	1.11	1.15	1.09	1.05	1.03
		2.14	1.93	1.79	1.64	1.65	1.53	1.47	1.45
	$t = 3$	1.42	1.32	1.22	1.18	1.34	1.11	1.07	0.99
		1.81	1.65	1.54	1.46	1.48	1.39	1.34	1.28
	$t = 4$	1.26	1.19	1.12	1.08	1.07	1.03	0.99	0.98
		1.73	1.57	1.48	1.40	1.48	1.35	1.29	1.31
	$t = 5$	1.32	1.21	1.15	1.09	1.07	1.05	1.02	0.99
		1.77	1.62	1.51	1.40	1.45	1.37	1.29	1.31

Table 11: Cycles/byte measurements for 9 to 16 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		17	18	19	20	21	22	23	24
polyHash1271	$g = 1$	1.37	1.37	1.37	1.36	1.36	1.35	1.35	1.34
	$g = 4$	0.83	0.81	0.80	0.77	0.80	0.77	0.77	0.76
		0.96	0.93	0.91	0.88	0.90	0.87	0.87	0.85
	$g = 8$	0.77	0.76	0.74	0.74	0.72	0.73	0.72	0.71
		1.17	1.13	1.10	1.07	1.05	1.03	1.01	0.99
	$g = 16$	0.83	0.80	0.79	0.78	0.77	0.76	0.76	0.75
		1.77	1.69	1.63	1.59	1.53	1.50	1.46	1.42
	$g = 32$	0.82	0.69	0.77	0.67	0.66	0.67	0.66	0.66
		3.09	2.92	2.82	2.67	2.61	2.55	2.44	2.36
BRWHash1271	$t = 2$	0.99	0.98	0.89	0.92	0.90	0.94	0.87	0.89
		1.20	1.15	1.09	1.14	1.09	1.09	1.01	1.05
	$t = 3$	0.96	0.94	0.90	0.92	0.92	0.88	0.86	0.87
		1.17	1.14	1.10	1.11	1.07	1.05	0.99	1.00
d-2LHash1271	$d = 2$	0.82	0.80	0.80	0.79	0.77	0.77	0.76	0.74
		1.36	1.30	1.28	1.25	1.21	1.19	1.16	1.12
	$d = 3$	0.85	0.86	0.84	0.84	0.81	0.79	0.79	0.76
		1.64	1.60	1.55	1.51	1.45	1.40	1.38	1.32
	$d = 4$	0.82	0.80	0.81	0.80	0.79	0.81	0.79	0.78
		1.68	1.61	1.58	1.53	1.48	1.48	1.42	1.39
	$d = 5$	1.00	0.97	0.95	0.95	0.93	0.91	0.94	0.90
		1.94	1.86	1.79	1.76	1.70	1.64	1.63	1.56
t-BRWHash1271	$t = 2$	1.07	1.02	1.00	0.94	0.97	0.90	0.93	0.86
		1.44	1.39	1.33	1.26	1.28	1.26	1.22	1.16
	$t = 3$	1.03	0.98	0.96	0.97	0.94	0.92	0.91	0.86
		1.33	1.24	1.20	1.17	1.21	1.16	1.14	1.05
	$t = 4$	1.01	0.97	0.95	0.94	0.93	0.92	0.90	0.89
		1.28	1.22	1.20	1.16	1.19	1.14	1.13	1.10
	$t = 5$	1.04	1.01	0.98	0.96	0.96	0.95	0.92	0.92
		1.36	1.33	1.25	1.23	1.30	1.22	1.17	1.17

Table 12: Cycles/byte measurements for 17 to 24 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		25	26	27	28	29	30	31	32
polyHash1271	$g = 1$	1.34	1.34	1.34	1.33	1.33	1.33	1.33	1.32
	$g = 4$	0.78	0.76	0.75	0.75	0.76	0.75	0.75	0.73
		0.87	0.84	0.83	0.82	0.84	0.82	0.82	0.80
	$g = 8$	0.73	0.71	0.71	0.70	0.69	0.69	0.69	0.68
		1.00	0.97	0.97	0.94	0.93	0.92	0.90	0.89
	$g = 16$	0.74	0.74	0.76	0.72	0.78	0.72	0.77	0.78
		1.39	1.35	1.35	1.31	1.32	1.27	1.28	1.27
	$g = 32$	0.77	0.65	0.78	0.76	0.75	0.76	0.75	0.79
		2.32	2.24	2.22	2.16	2.10	2.06	2.02	1.99
BRWHash1271	$t = 2$	0.85	0.85	0.81	0.83	0.83	0.80	0.79	0.81
		1.01	1.00	0.96	0.97	0.98	0.93	0.90	0.97
	$t = 3$	0.85	0.82	0.85	0.83	0.81	0.82	0.78	0.82
		1.00	0.96	0.93	0.95	0.92	0.92	0.90	1.18
d-2LHash1271	$d = 2$	0.74	0.75	0.75	0.74	0.74	0.71	0.72	0.71
		1.11	1.10	1.08	1.07	1.05	1.02	1.01	1.00
	$d = 3$	0.78	0.76	0.75	0.72	0.73	0.71	0.71	0.72
		1.32	1.29	1.25	1.21	1.19	1.15	1.15	1.15
	$d = 4$	0.76	0.76	0.80	0.79	0.75	0.71	0.69	0.68
		1.35	1.33	1.34	1.31	1.26	1.20	1.16	1.14
	$d = 5$	0.89	0.87	0.86	0.87	0.85	0.84	0.68	0.69
		1.52	1.48	1.45	1.44	1.41	1.37	1.20	1.18
t-BRWHash1271	$t = 2$	0.88	0.85	0.89	0.82	0.88	0.82	0.82	0.80
		1.20	1.13	1.13	1.11	1.08	1.05	1.07	1.06
	$t = 3$	0.88	0.87	0.86	0.84	0.85	0.84	0.84	0.80
		1.08	1.041	1.01	1.00	1.06	1.00	1.00	0.99
	$t = 4$	0.93	0.89	0.89	0.96	0.97	0.87	0.86	0.79
		1.15	1.09	1.08	1.08	1.08	1.05	1.05	0.96
	$t = 5$	0.96	0.92	0.91	0.90	0.91	0.88	0.88	0.78
		1.19	1.15	1.14	1.12	1.13	1.10	1.13	0.96

Table 13: Cycles/byte measurements for 25 to 32 blocks for the various hash functions based on the prime $2^{127} - 1$.

		# msg blks							
		1	2	3	4	5	6	7	8
polyHash1305	$g = 1$	3.44	2.97	2.65	2.39	2.27	2.20	2.12	2.09
	$g = 4$	3.40	3.09	2.22	1.95	1.78	1.72	1.55	1.47
		5.78	4.41	3.44	2.75	2.45	2.29	2.04	1.88
	$g = 8$	3.40	3.09	2.23	1.93	1.69	1.54	1.40	1.37
		11.85	7.63	5.41	4.20	3.51	3.09	2.72	2.55
	$g = 16$	3.40	3.09	2.23	1.93	1.69	1.54	1.42	1.35
		25.38	14.43	9.94	7.61	6.23	5.35	4.69	4.20
	$g = 32$	3.40	3.09	2.23	1.94	1.71	1.54	1.42	1.35
		53.46	28.40	19.27	14.60	11.85	10.02	8.69	7.70
BRWHash1305	$t = 2$	5.34	3.26	2.38	2.63	2.27	1.90	1.75	1.77
		6.01	7.09	5.04	3.80	3.15	2.70	2.42	2.63
	$t = 3$	5.42	3.27	2.36	2.22	1.85	1.64	1.47	1.68
		6.00	6.69	4.46	3.86	2.99	2.49	2.13	2.84
d-2LHash1305	$d = 2$	6.18	3.53	2.30	1.95	1.73	1.68	1.57	1.48
		21.56	11.18	7.39	5.77	4.79	4.23	3.76	3.39
	$d = 3$	7.81	4.17	3.00	2.39	2.01	1.76	1.65	1.54
		28.93	14.72	10.05	7.66	6.26	5.31	4.63	4.15
	$d = 4$	7.87	4.16	2.99	2.39	2.04	1.78	1.81	1.64
		30.62	15.56	10.61	8.07	6.57	5.56	5.05	4.49
	$d = 5$	7.79	4.11	2.94	2.33	2.01	1.75	1.79	1.62
		32.28	16.35	11.12	8.47	6.87	5.83	5.27	4.69
t-BRWHash1305	$t = 2$	5.45	3.26	2.34	2.65	2.55	2.17	1.94	1.77
		7.83	4.98	3.61	4.11	3.71	3.15	2.77	2.77
	$t = 3$	4.99	2.87	2.09	1.71	1.54	1.40	1.30	1.40
		7.84	4.96	3.57	3.30	3.04	2.61	2.32	2.69
	$t = 4$	5.67	3.19	2.32	1.91	1.65	1.47	1.36	1.39
		7.79	4.96	3.63	3.28	3.03	2.61	2.32	2.37
	$t = 5$	5.79	3.30	2.41	1.99	1.69	1.52	1.40	1.46
		8.01	4.36	3.18	2.95	2.76	2.38	2.13	2.16

Table 14: Cycles/byte measurements for 1 to 8 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		9	10	11	12	13	14	15	16
polyHash1305	$g = 1$	2.04	2.02	2.00	1.98	1.96	1.95	1.93	1.92
	$g = 4$	1.43	1.43	1.35	1.31	1.29	1.30	1.25	1.23
		1.82	1.76	1.66	1.59	1.55	1.54	1.48	1.44
	$g = 8$	1.33	1.33	1.29	1.23	1.21	1.18	1.14	1.12
		2.39	2.28	2.16	2.02	1.95	1.86	1.78	1.73
	$g = 16$	1.27	1.24	1.19	1.18	1.21	1.13	1.08	1.19
		3.84	3.55	3.30	3.14	2.97	2.91	2.69	2.63
	$g = 32$	1.29	1.24	1.19	1.18	1.12	1.09	1.11	1.19
		6.93	6.35	5.87	5.43	5.11	4.80	4.58	4.37
BRWHash1305	$t = 2$	1.63	1.51	1.44	1.48	1.41	1.34	1.30	1.34
		1.83	1.74	1.68	1.69	1.64	1.58	1.54	1.55
	$t = 3$	1.85	1.44	1.48	1.46	1.32	1.26	1.24	1.27
		2.35	2.16	2.03	2.00	1.90	1.81	1.71	1.83
d-2LHash1305	$d = 2$	1.41	1.36	1.30	1.26	1.23	1.20	1.19	1.16
		3.11	2.89	2.68	2.54	2.40	2.29	2.21	2.11
	$d = 3$	1.43	1.36	1.28	1.23	1.18	1.26	1.22	1.18
		3.75	3.45	3.22	2.97	2.79	2.76	2.62	2.49
	$d = 4$	1.53	1.45	1.36	1.31	1.25	1.21	1.17	1.13
		4.06	3.72	3.43	3.20	3.00	2.84	2.67	2.55
	$d = 5$	1.51	1.44	1.36	1.31	1.24	1.20	1.26	1.21
		4.24	3.87	3.58	3.34	3.12	2.96	2.89	2.74
t-BRWHash1305	$t = 2$	1.80	1.66	1.55	1.49	1.53	1.45	1.39	1.34
		2.70	2.47	2.29	2.16	2.15	2.02	1.92	1.97
	$t = 3$	1.42	1.32	1.22	1.18	1.34	1.11	1.07	0.99
		2.62	2.40	2.24	2.10	2.09	1.98	1.89	1.88
	$t = 4$	1.35	1.25	1.20	1.15	1.11	1.07	1.03	1.24
		2.33	2.14	2.01	1.89	1.90	1.79	1.72	1.89
	$t = 5$	1.39	1.28	1.22	1.18	1.13	1.11	1.06	1.10
		2.12	1.95	1.84	1.73	1.77	1.65	1.59	1.66

Table 15: Cycles/byte measurements for 9 to 16 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		17	18	19	20	21	22	23	24
polyHash1305	$g = 1$	1.91	1.91	1.90	1.89	1.88	1.88	1.87	1.87
	$g = 4$	1.22	1.23	1.20	1.18	1.18	1.19	1.17	1.15
		1.42	1.42	1.39	1.35	1.34	1.34	1.32	1.29
	$g = 8$	1.12	1.13	1.11	1.10	1.08	1.07	1.05	1.05
		1.69	1.67	1.63	1.58	1.55	1.51	1.48	1.45
	$g = 16$	1.21	1.18	1.15	1.13	1.12	1.10	1.09	1.08
		2.56	2.46	2.37	2.29	2.22	2.15	2.10	2.04
	$g = 32$	1.07	1.15	1.05	1.05	1.76	1.11	1.06	1.14
		4.15	4.00	3.81	3.66	4.19	3.44	3.36	3.25
BRWHash1305	$t = 2$	1.31	1.24	1.22	1.25	1.22	1.20	1.18	1.18
		1.54	1.49	1.45	1.48	1.44	1.41	1.40	1.46
	$t = 3$	1.21	1.15	1.19	1.27	1.13	1.10	1.11	1.13
		2.00	1.70	1.63	1.66	1.57	1.56	1.49	1.54
d-2LHash1305	$d = 2$	1.14	1.12	1.11	1.09	1.08	1.08	1.07	1.06
		2.06	1.97	1.92	1.86	1.80	1.77	1.73	1.69
	$d = 3$	1.15	1.12	1.10	1.07	1.10	1.08	1.06	1.04
		2.38	2.28	2.20	2.11	2.10	2.03	1.97	1.92
	$d = 4$	1.10	1.08	1.05	1.03	1.01	1.05	1.03	1.01
		2.43	2.33	2.24	2.16	2.08	2.08	2.01	1.96
	$d = 5$	1.17	1.15	1.12	1.10	1.07	1.05	1.10	1.07
		2.62	2.51	2.41	2.32	2.24	2.17	2.17	2.09
t-BRWHash1305	$t = 2$	1.38	1.74	1.28	1.30	1.29	1.25	1.22	1.22
		1.98	1.89	1.82	1.77	1.78	1.71	1.66	1.62
	$t = 3$	1.03	0.98	0.96	0.97	0.94	0.92	0.91	0.86
		1.90	1.81	1.81	1.70	1.70	1.65	1.60	1.52
	$t = 4$	1.28	1.23	1.19	1.17	1.14	1.12	1.09	1.12
		1.90	1.84	1.75	1.69	1.70	1.65	1.62	1.56
	$t = 5$	1.10	1.06	1.03	1.01	0.99	0.98	0.96	0.99
		1.68	1.61	1.56	1.51	1.54	1.48	1.45	1.48

Table 16: Cycles/byte measurements for 17 to 24 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks							
		25	26	27	28	29	30	31	32
polyHash1305	$g = 1$	1.87	1.86	1.86	1.85	1.85	1.85	1.85	1.84
	$g = 4$	1.15	1.16	1.14	1.13	1.13	1.14	1.12	1.11
		1.28	1.29	1.27	1.25	1.24	1.25	1.23	1.22
	$g = 8$	1.05	1.06	1.05	1.04	1.03	1.03	1.01	1.01
		1.44	1.44	1.41	1.38	1.37	1.35	1.33	1.31
	$g = 16$	1.08	1.08	1.08	1.05	1.10	1.10	1.10	1.11
		2.01	1.98	1.95	1.89	1.90	1.88	1.85	1.83
	$g = 32$	1.13	1.02	1.08	1.01	1.02	1.03	1.04	1.12
		3.17	3.07	2.98	2.92	2.85	2.80	2.72	2.72
BRWHash1305	$t = 2$	1.16	1.13	1.14	1.15	1.13	1.10	1.12	1.12
		1.53	1.48	1.46	1.48	1.46	1.40	1.39	1.47
	$t = 3$	1.12	1.08	1.09	1.07	1.04	1.02	1.04	1.06
		1.64	1.58	1.55	1.52	1.54	1.49	1.46	1.50
d-2LHash1305	$d = 2$	1.06	1.05	1.07	1.05	1.04	1.04	1.03	1.02
		1.67	1.64	1.63	1.60	1.57	1.55	1.53	1.50
	$d = 3$	1.02	1.01	0.99	1.01	1.02	0.99	0.98	0.97
		1.86	1.81	1.77	1.76	1.74	1.69	1.66	1.63
	$d = 4$	1.00	0.98	0.97	0.96	0.95	0.98	0.97	0.96
		1.90	1.85	1.81	1.76	1.73	1.74	1.70	1.67
	$d = 5$	1.05	1.04	1.03	1.02	1.00	0.98	0.94	0.94
		2.03	1.98	1.93	1.89	1.84	1.81	1.73	1.70
t-BRWHash1305	$t = 2$	1.23	1.19	1.17	1.15	1.19	1.16	1.14	1.13
		1.64	1.58	1.55	1.52	1.54	1.49	1.46	1.50
	$t = 3$	0.88	0.87	0.86	0.84	0.85	0.84	0.84	0.80
		1.54	1.49	1.46	1.43	1.45	1.42	1.40	1.40
	$t = 4$	1.11	1.08	1.06	1.05	1.04	1.02	1.00	1.02
		1.63	1.54	1.53	1.47	1.49	1.45	1.43	1.41
	$t = 5$	0.99	0.97	0.96	0.95	0.94	0.92	0.91	1.03
		1.45	1.40	1.37	1.34	1.38	1.33	1.31	1.44

Table 17: Cycles/byte measurements for 25 to 32 blocks for the various hash functions based on the prime $2^{130} - 5$.