Vectorised Hashing Based on Bernstein-Rabin-Winograd Polynomials over Prime Order Fields

Kaushik Nath[1] and Palash Sarkar[1]

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

November 15, 2026

Abstract

We introduce the new AXU hash function c-decBRWHash, which is parameterised by the positive integer c and is based on Bernstein-Rabin-Winograd (BRW) polynomials. Choosing $c > 1$ gives a hash function which can be implemented using c-way single instruction multiple data (SIMD) instructions. We report a set of very comprehensive hand optimised assembly implementations of 4-decBRWHash using avx2 SIMD instructions available on modern Intel processors. For comparison, we also report similar carefully optimised avx2 assembly implementations of polyHash, an AXU hash function based on usual polynomials. Our implementations are over prime order fields, specifically the primes $2^{127} - 1$ and $2^{130} - 5$. For the prime $2^{130} - 5$, for avx2 implementations, compared to the famous Poly1305 hash function, 4-decBRWHash is faster for messages which are a few hundred bytes long and achieves a speed-up of about 16% for message lengths in a few kilobytes range and improves to a speed-up of about 22% for message lengths in a few megabytes range.
Keywords: almost XOR universal, BRW polynomials, SIMD, assembly implementation, avx2.
Mathematics Subject Classification: 94A60.

1 Introduction

Authentication and authenticated encryption are two of the major functionalities of modern symmetric key cryptography. Almost XOR universal (AXU) hash functions play an important role in both of these tasks. One of the most famous AXU hash functions is Poly1305 [3], and in combination with XChaCha20 [5] provides one of the most used authenticated encryption algorithm. Poly1305 is based on usual polynomials with arithmetic done modulo the prime $2^{130} - 5$.

Present generation processors provide support for single input multiple data (SIMD) instructions. These instructions permit implementation of vectorised algorithms. In a vectorised algorithm, at every step a single instruction is applied to a number of data items. Such vectorised algorithms have the potential to provide significant efficiency improvements over conventional sequential algorithms. However, to apply SIMD instructions it is required to rewrite the basic algorithm in vectorised form. For Poly1305, SIMD implementation was earlier reported in [20].

A class of polynomials was introduced in [4] to construct AXU hash functions, and later these polynomials were named the BRW polynomials [32]. An important theoretical advantage of hash functions based on BRW polynomials is that the number of field multiplications required by such

hash functions is about half the number of field multiplications required by hash functions based on usual polynomials [4] (see [19, 8] for further complexity improvements). This feature makes BRW polynomials an attractive option for constructing AXU hash function. An extensive study of both BRW polynomials based hash functions (named BRWHash) and usual polynomial based hash functions (named polyHash) for the primes $2^{127}-1$ and $2^{130}-5$ was carried out in [8] (see [25] for an update on [8]). The implementations reported in [8, 25] were sequential, i.e. not SIMD, and for such implementations it was observed that BRW based AXU hash functions indeed provides significant speed improvements (though less than what is theoretically predicted) for both of the primes that were considered. The natural question that arises is whether a similar speed improvement can be achieved for SIMD implementation.

Efficient algorithms for computing the value of a BRW polynomial at a particular point were reported in [19, 8]. Unfortunately, there is no good way to rewrite these algorithms in vectorised form. There is a certain amount of parallelism present in the computation of BRW polynomials, which has been exploited for hardware implementation [13]. This parallelism, however, does not permit vector computation.

The main theoretical contribution of the present work is to present a new AXU hash function based on BRW polynomials which permits an efficient vector implementation. We define the hash function c-decBRWHash which is parameterised by the positive integer c. Suppose $c = 4$, which is the case that we implement. The basic idea is to decimate the input stream into four parallel streams of the same length and perform independent BRW polynomial computation on each of the streams. Finally the four outputs of the four streams are combined using usual polynomial hashing. So usual polynomial hashing is applied only to a small and fixed number of blocks. Since the four BRW computations are independent and are on the sequences of the same length, it is possible to apply the algorithm from [8] in a vectorised manner to perform simultaneous computations of the four BRW polynomials. As a result, the entire algorithm becomes amenable to SIMD implementation.

The above strategy is the natural way to build an SIMD friendly hashing strategy based on BRW polynomials. While the value of c is a design parameter and is fixed, we show that for any c' dividing c, it is possible to implement c-decBRWHash using c'-way SIMD instructions. For example, 4-decBRWHash can be implemented both sequentially and also using 2-way SIMD along with 4-way SIMD instructions.

The hash function c-decBRWHash is a generalisation of BRWHash in the sense that for $c = 1$, c-decBRWHash becomes exactly BRWHash. While the idea behind c-decBRWHash is simple, there is a subtlety in the choice of key used for combining the outputs of the c streams. We prove that c-decBRWHash is indeed an AXU hash function, whose AXU bound is almost the same as the AXU bound of BRWHash for small c.

From a practical point of view, we report implementations of 4-decBRWHash for both the primes $2^{127}-1$ and $2^{130}-5$ using the `avx2` SIMD instructions available on modern Intel processors. For completeness and for the sake of comparison, we also report new implementations of polyHash using `avx2` instructions for both $2^{127}-1$ and $2^{130}-5$. Our implementations are comprehensive in the sense that we consider all feasible values of the various implementation parameters for both 4-decBRWHash and polyHash. The implementations that we report are in assembly language targeting the `avx2` and `mulx`/`adcx`/adox instructions of the Intel architecture and were meticulously hand optimised. Our hand optimised `avx2` implementation in assembly language of polyHash1305 (i.e. polyHash based on the prime $2^{130}-5$) is of independent interest, since if the message length is a multiple of eight and an appropriate key clamping is used, then polyHash1305 is exactly the well known hash function Poly1305. To the best of our knowledge, there is no previous hand optimised

avx2 assembly language implementation of Poly1305 which systematically considers all feasible values of the implementation parameters. The independent work [28] considers SIMD implementation of Poly1305 using avx2, avx512, and avx512_ifma instruction sets with feasible values of implementation parameters, but the code is in C with intrinsics, and not assembly. Further, our avx2 implementation outperforms the avx2 implementation reported in [28].

We obtained extensive timing results for all our implementations. These results show that for avx2 implementations, the prime $2^{127}-1$ is a slower option than the prime $2^{130}-5$ for both the hash functions 4-decBRWHash and polyHash. We provide a detailed explanation for this observation. In view of this observation, in this work we present only the timing results for the prime $2^{130}-5$. For the case of polyHash1305, the timing results show that for files which are longer than a few hundred bytes, the avx2 implementation is faster than the previously reported sequential implementation [8]. Of more interest in the present context is the comparison between the hash functions 4-decBRWHash and polyHash. The timings results for the prime $2^{130}-5$ show that for avx2 implementations, 4-decBRWHash is faster than polyHash for messages which are a few hundred bytes long, and achieves a speed-up of about 16% (for kilobyte size messages) to 22% (for megabyte size messages). Since Poly1305 and polyHash over the prime $2^{130}-5$ have the same speed, the previous statement for $2^{130}-5$ also applies to the speed-up of 4-decBRWHash over Poly1305. In a typical file system, text files are usually about a few kilobytes long, while media files such as high resolution pictures, audio and video files, are a few megabytes long (see [18]). Compared to Poly1305, the new hash function 4-decBRWHash1305 provides a faster option for authentication. For authenticated encryption, the speed gain will be smaller since the hash function will have to be coupled with encryption based on a block or a stream cipher, and so the speed improvement of only the hash function will be amortised over the time taken by the encryption module.

Previous and related works. AXU hash functions are a generalisation of the notion of universal hash functions [11]. Research over the last few decades have resulted in a sizeable literature on AXU hash functions. Overviews of the literature can be found in [2, 3, 4, 33, 34, 15]. We mention only the works which are relevant to the present paper.

Polynomial hash functions were proposed independently in three papers [17, 36, 10]. The prime $2^{127}-1$ for use in polynomial hashing was first proposed in [36], and was later used in [2, 21, 8]. BRW polynomials were proposed by Bernstein [4] based on earlier work by Rabin and Winograd [31]. Implementations of BRW polynomials in both software and hardware over binary extension fields were reported in [13, 14, 12, 19]. For prime order fields, sequential software implementations of BRW polynomials were reported in [8, 25].

Two recent works, namely [15] and [28], study implementation of polynomial hashing. Sequential implementation was considered in [15], while [28] considered SIMD implementations. With respect to code, both [15] and [28] focus on developing code generators, rather than direct hand written assembly implementation. We note that the present work was done independently of [28] – the ePrint version (available at https://eprint.iacr.org/archive/2025/1262/20250708:155337) of the present paper is dated 8 July 2025, whereas the received date of [28] is 15 July 2025. The main thrust of the present paper is to study SIMD friendly hash function based on BRW polynomials, which was not considered in [15, 28]. For Poly1305, we report hand written avx2 assembly code, whereas [28] provides generated C with intrinsics code for avx2, avx512, and avx512_ifma instruction sets, and also Jasmin code for avx2 instruction set. Our avx2 implementation of polyHash1305 is faster than the avx2 implementation of Poly1305 in [28].

Overview of the paper. Section 2 provides the preliminaries. The new construction of decimated BRW hash function is presented in Section 3. The various aspects of implementation are given in Sections 4 and 5. Descriptions of the implementations and the timing results are given in Section 6. Finally Section 7 concludes the paper. For the ease of reference, in Appendix A we provide the algorithm from [8] for computing BRW polynomials.

2 Preliminaries

The cardinality of a finite set S will be denoted as $\#S$. Logarithms to the base two will be denoted by lg. 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, $\mathsf{bin}_4(3) = 0011$ and $\mathsf{bin}_4(13) = 1101$.

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, where 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. Let $a, a' \in \mathcal{D}$ with $a \neq a'$ and $b \in \mathcal{R}$. The differential probability corresponding to the triple (a, a', b) is defined to be $\Pr_\tau[\mathsf{Hash}_\tau(a) - \mathsf{Hash}_\tau(a') = b]$, 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. Given a non-zero polynomial $P(x) \in \mathbb{F}[x]$, $\mathsf{deg}(P(x))$ denotes the degree of $P(x)$. Given $l \geq 0$ elements $M_1, \ldots, M_l$ in $\mathbb{F}$, we define two polynomials $\mathsf{Poly}(x; M_1, \ldots, M_l)$ and $\mathsf{BRW}(x; M_1, M_2, \ldots, M_l)$ in $\mathbb{F}[x]$ with indeterminate x and parameters $M_1, \ldots, M_l$ as follows.

$$\mathsf{Poly}(x; M_1, \ldots, M_l) \;=\; \begin{cases} 0, & \text{if } l = 0; \\ M_1 x^{l-1} + M_2 x^{l-2} + \cdots + M_{l-1}x + M_l, & \text{if } l > 0, \end{cases} \tag{1}$$

and

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

The BRW polynomials were introduced in [4] and named in [32]. Note that for $l \geq 3$, $\mathsf{BRW}(x; M_1, M_2, \ldots, M_l)$ is a monic polynomial.

For $\tau \in \mathbb{F}$, using Horner's rule $\mathsf{Poly}(\tau; M_1, \ldots, M_l)$ can be evaluated using $l - 1$ multiplications and same number of additions. For the BRW polynomials the following was proved in [4].

Theorem 1. *[4]*

1. *For every $l \geq 0$, the map from $\mathbb{F}^l$ to $\mathbb{F}[x]$ given by*

$$(M_1, \ldots, M_l) \mapsto \mathsf{BRW}(x; M_1, \ldots, M_l)$$

is injective.

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

Table 1: The parameters m, n, k and μ for the primes $2^{127}-1$ and $2^{130}-5$.

2. *For $l \geq 1$, let $\mathfrak{d}(l)$ denote $\mathsf{deg}(\mathsf{BRW}(x; M_1, \ldots, M_l))$. For $l \geq 3$, $\mathfrak{d}(l) = 2^{\lfloor \lg l \rfloor + 1} - 1$ and so $\mathfrak{d}(l) \leq 2l-1$; the bound is achieved if and only if $l = 2^a$; and $\mathfrak{d}(l) = l$ if and only if $l = 2^a - 1$ for some integer $a \geq 2$.*

3. *For $\tau \in \mathbb{F}$ and $l \geq 3$, $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ can be computed using $\lfloor l/2 \rfloor$ field multiplications (i.e. a multiplication over the field $\mathbb{F}$) and $\lfloor \lg l \rfloor$ additional field squarings to compute $\tau^2, \tau^4, \ldots, \tau^{2^{\lfloor \lg l \rfloor}}$.*

A field multiplication in $\mathbb{F}$ has two steps, namely a multiplication over the underlying ring (either the ring of integers, or the ring of polynomials), followed by a reduction step. So $\lfloor l/2 \rfloor$ field multiplications amounts to $\lfloor l/2 \rfloor$ ring multiplications and $\lfloor l/2 \rfloor$ reductions. By an unreduced multiplication we mean the ring multiplication with possibly a partial reduction. The following complexity improvement in computing $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ was proved in [8, 19].

Theorem 2. *[8, 19] For $\tau \in \mathbb{F}$ and $l \geq 3$, computing $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ requires $\lfloor l/2 \rfloor$ unreduced multiplications, $1 + \lfloor l/4 \rfloor$ reductions, and additionally requires $\lfloor \lg l \rfloor$ field squarings to compute $\tau^2, \tau^4, \ldots, \tau^{2^{\lfloor \lg l \rfloor}}$.*

The algorithm for evaluating $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ given in [8] is provided in Appendix A. The algorithm uses a parameter t which is a small integer. The values $t = 2, 3, 4$ and 5 were considered in [8] and the same values of t will also be considered in the present work.

Proposition 1 (From Theorem 5.2 of [8])**.** *Applying Algorithm 1 in Appendix A to compute $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$ requires the stack size to be at most $\lfloor \lg l \rfloor - t + 1$.*

2.1 Hash Functions polyHash and BRWHash

Let p be a prime and $\mathbb{F}_p$ be the finite field of order p. Our primary focus will be $2^{130}-5$ which is the prime underlying the hash function Poly1305. We will also consider the prime $2^{127}-1$ which has turned out to be quite important (see [8, 25]). Given the prime p, we define the integers m, n, k and μ as shown in Table 1. Elements of $\mathbb{F}_p$ can be represented as m-bit strings. Since n, k and μ are less than m, we will consider n-bit, k-bit and μ-bit strings to represent elements of $\mathbb{F}_p$, where the most significant $m-n$, $m-k$, and $m-\mu$ bits respectively are set to 0.

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 $\ell = 0$. We call each X_i to be a block. 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. The following two padding schemes were described in [8].

- $\mathsf{pad1}(X_1, \ldots, X_\ell)$ 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 $i = 1, \ldots, \ell - 1$, and $M_\ell = 0^{m-s-1}||X_\ell$.

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 $\mathbb{F}_p$. For the padding scheme pad1, there is no restriction on the value of L. On the other hand, for pad2, the value of L has to be less than 2^{m-1}. From Table 1, the values of $m - 1$ for the two primes are 127 and 130, and so the restriction on L is a non-issue in practice. In fact, in our implementations we consider L to be less than 2^{64}, so that the binary representation of L can be stored as a 64-bit quantity. This is sufficient for all conceivable applications.

The hash functions polyHash and BRWHash were introduced in [8]. In particular, the hash function polyHash is based on the idea behind the design of the hash function Poly1305. The key space and digest space for both the families polyHash and BRWHash are $\{0,1\}^k$; τ denotes the k-bit key which is considered to be an element of $\mathbb{F}_p$. The digest space is the group $(\mathbb{Z}_{2^\mu}, +)$, and so the digest can be represented using a μ-bit string. The message space for polyHash is the set of all binary strings. The message space for BRWHash is the set of all binary strings of lengths less than 2^{m-1}; as mentioned above in our implementations we considered messages of lengths less than 2^{64}.

Remark 1. *The descriptions of* polyHash *and* BRWHash *in [8] did not include the parameter μ. Instead both the key and tag spaces were defined to be $\{0,1\}^k$. In this paper, we make the formal distinction between the key and the tag spaces by introducing the additional parameter μ to denote the size of tags. This generalises the descriptions of the* polyHash *and* BRWHash*, and we restate the result on the AXU bounds proved in [8] in terms of μ and k.*

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

Construction 1. *Given a binary string X, let $(M_1, \ldots, M_\ell)$ be the output of* $\mathsf{pad1}(\mathsf{format}(X))$. *We define*

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

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

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

Note that if X is the empty string, then $L = \ell = 0$ and so $\mathsf{polyHash}_\tau(X) = 0$. The family polyHash is motivated by the design of Poly1305 [3] for the prime $2^{130} - 5$. The differences between Poly1305 and polyHash are as follows.

1. Poly1305 considers X to be a sequence of bytes, whereas polyHash considers X to be a sequence of bits.

2. In Poly1305, certain bits of the key τ are "clamped", i.e. they are set to 0. In [3] the clamping of key bits helped in efficient floating point implementation. On the other hand, however, clamping reduces security. Since we are not interested in floating point implementation, we do not include clamping of key bits in the specification of polyHash.

3. Poly1305 is defined only for the prime $2^{130} - 5$, whereas polyHash can be instantiated by any appropriate prime. In [8], instantiations of polyHash were proposed using both $2^{130} - 5$ and $2^{127} - 1$.

Suppose $\mathsf{pad1}(\mathsf{format}(X))$ returns $(M_1, \ldots, M_\ell)$. Computing $\mathsf{polyHash}_\tau(X)$ requires ℓ field multiplications. A delayed reduction strategy was proposed in [7] and developed further in [20] for computing $\mathsf{Poly}(\tau; M_1, \ldots, M_\ell)$. For a parameter $g \geq 1$, the idea is to perform a sequence of g unreduced multiplications and additions and then perform a single reduction. This strategy requires pre-computing the powers $\tau, \tau^2, \tau^3, \ldots, \tau^g$. Using this strategy, it is possible to compute $\mathsf{polyHash}(X)$ using ℓ unreduced multiplications, $\lceil \ell/g \rceil$ reductions, and additionally $g-1$ field multiplications [20, 8]. The key powers $\tau, \tau^2, \tau^3, \ldots, \tau^g$ are required to be pre-computed before the actual computation of $\mathsf{polyHash}$. See Table 2.

Construction 2. *Given a binary string X, let the output of $\mathsf{pad2}(\mathsf{format}(X))$ be $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$. We define*

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

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.

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

Note that if X is the empty string, then $L = \ell = 0$ and so $\mathsf{BRWHash}_\tau(X) = 0$. Suppose $\mathsf{pad2}(\mathsf{format}(X))$ returns $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$. Computing $\mathsf{BRWHash}(X)$ requires $2 + \lfloor \ell/2 \rfloor$ unreduced multiplications, $2 + \lfloor \ell/4 \rfloor$ reductions, and additionally $\lfloor \lg \ell \rfloor$ field squarings [8]. The key powers $\tau, \tau^2, \tau^{2^2}, \ldots, \tau^{2^{\lfloor \lg \ell \rfloor}}$ are required to be pre-computed before the actual computation of $\mathsf{BRWHash}$. See Table 2.

The following two results were proved in [8] for the case $k = \mu$. Below we state the results for the more general case of separate k and μ. The proofs are essentially the same as the proofs of the case $k = \mu$ given in [8].

Lemma 1 (Based on Lemma 4.1 of [8])**.** *Let $p = 2^m - \delta$ be a prime and μ be a positive integer such that $\mu < m$ and $\delta < 2^\mu - 1$. Let $\alpha \in \mathbb{Z}_{2^\mu}$, 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*

$$((P(\tau) \bmod p) \bmod 2^\mu) - ((P'(\tau) \bmod p) \bmod 2^\mu) \quad \equiv \quad \alpha \pmod{2^\mu} \tag{6}$$

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

Consequently, for τ chosen uniformly at random from a set $\emptyset \neq S \subseteq \mathbb{F}_p$, the probability that (6) holds is at most $(2^{m-\mu+1} \cdot \mathsf{deg}(P(x) - P'(x)))/\#S$.

In our applications of Lemma 1, the set S will be the key space of the hash functions. For the hash functions that we construct, the key space (and hence S) will be the set $\{0,1\}^k$ for some k such that $2^k < p$, and the k-bit keys are considered to be elements of $\mathbb{F}_p$. Lemma 1 reduces the problem of determining the probability that a uniform random k-bit string τ satisfies (6) to the simpler problem of determining the degree of the non-zero polynomial $P(x) - P'(x) \in \mathbb{F}_p[x]$. The values of p, m, k and μ given in Table 1 satisfy the conditions stated in Lemma 1.

Theorem 3 (Based on Theorem 4.7 of [8])**.** *Let $p = 2^m - \delta$ be a prime and μ be a positive integer such that $\mu < m$ and $\delta < 2^\mu - 1$. 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 $\mathbb{Z}_{2^\mu}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Then*

$$\begin{aligned} \Pr[\mathsf{polyHash}_\tau(X) - \mathsf{polyHash}_\tau(X') = \alpha] &\leq \ell \cdot 2^{m-k-\mu+1}, \\ \Pr[\mathsf{BRWHash}_\tau(X) - \mathsf{BRWHash}_\tau(X') = \alpha] &\leq (1 + 2\ell) \cdot 2^{m-k-\mu+1}, \end{aligned}$$

Apart from polyHash and BRWHash, two other hash functions, named t-BRWHash and d-2LHash, as well as the hash function d-Hash (which is a combination of polyHash and d-2LHash) were defined in [8]. Timing results from the sequential implementations reported in [8] indicated that among all the hash functions, for short messages polyHash is the fastest, while d-2LHash is the fastest for longer messages. However, fresh implementations of BRWHash (and also t-BRWHash) reported in [25] showed that among all the hash functions considered in [8], BRWHash is the fastest among all the four hash functions polyHash, BRWHash, t-BRWHash and d-2LHash for all message lengths and for both the primes $2^{127}-1$ and $2^{130}-5$. In view of the fact that BRWHash is faster than both t-BRWHash and d-2LHash, we do not consider the hash functions t-BRWHash and d-2LHash (and also d-Hash) in this work.

3 Decimated BRW Hash

We describe the hash function family c-decBRWHash. 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 $(\mathbb{Z}_{2^\mu},+)$. The message space is the set of all binary strings of lengths less than 2^{m-1}. For concreteness we refer to the primes and the parameters m, n, k and μ given in Table 1.

Construction 3. *The hash function is parameterised by a positive integer c. Given a binary string X of length $L \geq 0$, let $(M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L))$ be the output of* pad2(format(X)). *We define*

$$c\text{-decBRWHash}_\tau(X) \;=\; (Q(\tau;M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L)) \bmod p) \bmod 2^\mu, \tag{7}$$

where $Q(x;M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L))$ is the polynomial in $\mathbb{F}_p[x]$ defined in the following manner.

Let $\mathfrak{n} = \lceil \ell/c \rceil$ and $\mathfrak{m} = c\mathfrak{n}$. Define $M_{\ell+1} = \cdots = M_{\mathfrak{m}} = 0^{m-1}$. Let

$$\begin{aligned}
Q_1(x) &= \mathsf{BRW}(x;M_1,M_{c+1},M_{2c+1},\ldots,M_{\mathfrak{m}-c+1}),\\
Q_2(x) &= \mathsf{BRW}(x;M_2,M_{c+2},M_{2c+2}\ldots,M_{\mathfrak{m}-c+2}),\\
&\cdots\\
Q_c(x) &= \mathsf{BRW}(x;M_c,M_{2c},M_{3c}\ldots,M_{\mathfrak{m}}).
\end{aligned}$$

Note that each of the Q_i's is a BRW polynomial on $\mathfrak{n}$ blocks. If $L=0$, let $d=1$ and if $L>0$, let $d = 2^{\lfloor \lg \mathfrak{n} \rfloor + 1}$. Define

$$\begin{aligned}
Q_{c+1}(x) &= \mathsf{Poly}(x^d;Q_1(x),Q_2(x),\ldots,Q_c(x))\\
&= x^{(c-1)d}Q_1(x) + x^{(c-2)d}Q_2(x) + \cdots + x^dQ_{c-1}(x) + Q_c(x).
\end{aligned} \tag{8}$$

Finally,

$$Q(x;M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L)) \;=\; x(x\cdot Q_{c+1}(x) + \mathsf{bin}_{m-1}(L)). \tag{9}$$

When the quantities $M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L)$ are clear from the context, we will write $Q(x)$ instead of $Q(x;M_1,\ldots,M_\ell,\mathsf{bin}_{m-1}(L))$. Note that if X is the empty string, then $L=\ell=0$, and c-decBRWHash$_\tau(X)=0$.

The idea behind the construction of decBRWHash is to decimate the message blocks into c independent streams, process each stream using BRW and then combine the outputs of the streams using Horner with an appropriate power of the key τ. Choosing the proper power of τ for the Horner

	unred mult	red	storage	pre-comp (mult)
polyHash	ℓ	$\lceil \ell/g \rceil$	g	$g-1$
BRWHash	$2+\lfloor \ell/2 \rfloor$	$3+\lfloor \ell/4 \rfloor$	$1+\lfloor \lg \ell \rfloor$	$\lfloor \lg \ell \rfloor$
c-decBRWHash	$1+c(1+\lfloor \lceil \ell/c \rceil/2 \rfloor)$	$1+c(2+\lfloor \lceil \ell/c \rceil/4 \rfloor)$	$2+\lfloor \lg \lceil \ell/c \rceil \rfloor$	$1+\lfloor \lg \lceil \ell/c \rceil \rfloor$

Table 2: Operation counts and storage requirement for the hash functions for ℓ message blocks. For polyHash, the parameter g is a positive integer. For c-decBRWHash, $c > 1$.

evaluation is important to ensure security. We prove later that the choice of τ^d is appropriate. Further, the key powers, $\tau, \tau^2, \ldots, \tau^{d/2}$ are required for the BRW computations. So the key power τ^d for the Horner computation is obtained from the last key power $\tau^{d/2}$ required for the BRW computation by one squaring.

Remark 2. *Suppose $c = 1$. Then $Q_1(x) = \mathsf{BRW}(x; M_1, \ldots, M_\ell)$, $Q_2(x) = Q_1(x)$, and $Q(x) = x(x \cdot Q_1(x) + \mathsf{bin}_{m-1}(L))$. So with $c = 1$, the hash function c-decBRWHash becomes exactly the hash function BRWHash. Consequently, c-decBRWHash is a generalisation of BRWHash. Note that for $c = 1$, since $Q_2(x) = Q_1(x)$, x^d is not required.*

Remark 3. *The parameter c in the definition of c-decBRWHash is a design parameter, i.e. changing the value of c changes the hash function. For example, on the same input, the outputs of 2-decBRWHash and 4-decBRWHash are not necessarily the same. The AXU bound that we prove below is for a fixed value of c.*

The complexity of computing c-decBRWHash$_\tau(X)$ for $c > 1$ is stated in the following result.

Proposition 2. *Let $c > 1$ be an integer and X be a binary string. Suppose $\mathsf{pad2}(\mathsf{format}(X))$ returns $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$. Computing c-decBRWHash$_\tau(X)$ requires $c\lfloor \mathfrak{n}/2 \rfloor$ unreduced multiplications, $c(1 + \lfloor \mathfrak{n}/4 \rfloor)$ reductions, $c + 1$ field multiplications, and $\lfloor \lg \mathfrak{n} \rfloor$ field squarings.*

Proof. From Theorem 2, computing each Q_i, $i = 1, \ldots, c$, requires $\lfloor \mathfrak{n}/2 \rfloor$ unreduced multiplications and $1 + \lfloor \mathfrak{n}/4 \rfloor$ reductions. The key powers $\tau^2, \tau^4, \ldots, \tau^{2^{\lfloor \lg \mathfrak{n} \rfloor}}$ are required in the computation of all the Q_i's, and are computed only once using $\lfloor \lg \mathfrak{n} \rfloor$ field squarings. Computing Q_{c+1} from $Q_1, \ldots, Q_c$ requires $c - 1$ field multiplications, and computing Q from Q_{c+1} requires two additional field multiplications. □

For $c > 1$, the key powers $\tau, \tau^2, \tau^4, \ldots, \tau^{2^{\lfloor \lg \mathfrak{n} \rfloor + 1}}$ are required to be stored. See Table 2 which compares the operation counts and storage requirements for polyHash and c-BRWHash. Compared to BRWHash, for a small value of $c > 1$, the hash function c-BRWHash requires a few extra unreduced multiplications and reductions, and a little less storage. Computed sequentially, both BRWHash and c-BRWHash have similar efficiencies for message which are longer than a few blocks (for short messages BRWHash will be faster than c-BRWHash). The main advantage of c-BRWHash is that can be implemented using SIMD operations, as we describe later.

Naming convention. We adopt the following naming convention. For all the hash functions considered in this paper, there are two possible sets of parameters in Table 1. The choice of the prime p determines the values of m, k, n and μ. 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.

3.1 Computing c-decBRWHash with c'-way SIMD Instructions, where c' Divides c

The definition of c-decBRWHash shows a natural c-way SIMD implementation is possible. We show that in fact, it is possible to compute c-decBRWHash using c'-way SIMD for any c' which divides c.

Let $c'' = c/c'$. To start with consider a c-way SIMD implementation. This requires an overall data-structure D such that it is possible to apply c-way SIMD instructions on D. In particular, D will consist of several constituent sub-structures each of which has a c-way vector representation. On such c-way vectors, individual c-way SIMD instructions are applied.

Now consider each of these c-way vectors to be partitioned into c'' c'-way vectors. As a result, the overall data structure D is partitioned into c'' data structures $D_1, \ldots, D_{c''}$.

A loop of the c-way SIMD algorithm operating on D, updates D in each iteration using c-way SIMD instructions. Let Update_c denote the operation which applies one or more c-way SIMD instructions to its input. Then a loop which iteratively applies c-way instructions to D can be written in the following form.

```
for i = 1, 2, 3, ...
    D ← Update_c(D)
end for
```

The above loop of c-way SIMD instructions can be replaced by a loop of c'-way SIMD instructions in the following manner.

```
for i = 1, 2, 3, ...
    for j ← 1 to c''
        D_j ← Update_{c'}(D_j)
    end for
end for
```

The essential idea is that at every step, instead of applying c-way SIMD instructions to D, one applies c'-way SIMD instructions to each D_j, for $j = 1, \ldots, c''$. The correctness of the procedure is immediate.

So c-decBRWHash can be computed using c'-way SIMD instructions for any c' which divides c. For example, if we choose $c = 4$, then 4-decBRWHash can be computed sequentially (corresponding to $c' = 1$), using 2-way SIMD instructions (corresponding to $c' = 2$), and using 4-way SIMD instructions (corresponding to $c' = 4$). Similarly, 8-decBRWHash can be computed either sequentially, or using 2-way, 4-way, or 8-way SIMD instructions.

3.2 AXU Bounds

The following result from [8] states the basic property of pad2.

Lemma 2 (Lemma 4.3 of [8])**.** *Let X be a binary string of length $L \geq 0$. Then the map $X \mapsto \mathsf{pad2}(\mathsf{format}(X))$ is an injection.*

Lemma 3. *Let X be a binary string of length $L \geq 0$. Let $\ell = \lceil L/n \rceil$. Suppose $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the output of $\mathsf{pad2}(\mathsf{format}(X))$ and $Q(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is the polynomial constructed from X as in (9). Then $X \mapsto Q(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ is an injection.*

Proof. Let X and X' be two distinct binary strings of lengths L and L' respectively. We assume without loss of generality that $L \geq L' \geq 0$. Let $\ell = \lceil L/n \rceil$, $\mathfrak{n} = \lceil \ell/c \rceil$, $\mathfrak{m} = c\mathfrak{n}$, $d = 2^{\lfloor \lg \mathfrak{n} \rfloor + 1}$, and $\ell' = \lceil L'/n \rceil$, $\mathfrak{n}' = \lceil \ell'/c \rceil$, $\mathfrak{m}' = c\mathfrak{n}'$, $d' = 2^{\lfloor \lg \mathfrak{n}' \rfloor + 1}$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$, and let $(M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L'))$ be the output of $\mathsf{pad2}(\mathsf{format}(X'))$.

Let $Q_1(x), \ldots, Q_c(x), Q_{c+1}$ and $Q(x)$ be the polynomials arising from X, and $Q'_1(x), \ldots, Q'_c(x)$, $Q'_{c+1}(x)$ and $Q'(x)$ be the polynomials arising from X'. By construction, the coefficient of x in $Q(x)$ is $\mathsf{bin}_{m-1}(L)$ and the coefficient of x in $Q'(x)$ is $\mathsf{bin}_{m-1}(L')$. So if $L \neq L'$, then $Q(x) \neq Q'(x)$.

Now suppose that $L = L'$, which implies $\mathsf{bin}_{m-1}(L) = \mathsf{bin}_{m-1}(L')$, $\ell = \ell'$, $\mathfrak{n} = \mathfrak{n}'$, $\mathfrak{m} = \mathfrak{m}'$, and $d = d'$. Since there is exactly one string of length 0, $L = L'$ and $X \neq X'$ implies that both the strings X and X' are non-empty and so $\ell = \ell' > 0$. Since $X \neq X'$, by the injectivity of $\mathsf{pad2}$ (see Lemma 2), $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L)) \neq (M'_1, \ldots, M'_\ell, \mathsf{bin}_{m-1}(L'))$. Since $\mathsf{bin}_{m-1}(L) = \mathsf{bin}_{m-1}(L')$, it follows that $(M_1, \ldots, M_\ell) \neq (M'_1, \ldots, M'_\ell)$. Let $\imath$ be such that $M_\imath \neq M'_\imath$, and suppose that $\imath = \jmath + cj$, for some $\jmath \in \{1, \ldots, c\}$. By construction

$$\begin{aligned} Q_\jmath(x) &= \mathsf{BRW}(x; M_\jmath, M_{\jmath+c}, \ldots, M_{\jmath+c(j-1)}, M_{\jmath+cj}, M_{\jmath+c(j+1)}, \ldots, M_{\mathfrak{m}-c+\jmath}), \\ Q'_\jmath(x) &= \mathsf{BRW}(x; M'_\jmath, M'_{\jmath+c}, \ldots, M'_{\jmath+c(j-1)}, M'_{\jmath+cj}, M'_{\jmath+c(j+1)}, \ldots, M'_{\mathfrak{m}-c+\jmath}). \end{aligned}$$

Since $M_{\jmath+cj} = M_\imath \neq M'_\imath = M'_{\jmath+cj}$, by the injectivity of BRW polynomials (first point of Theorem 1), $Q_\jmath(x) \neq Q'_\jmath(x)$.

For each $i = 1, \ldots, c$, both $Q_i(x)$ and $Q'_i(x)$ are BRW polynomials built from $\mathfrak{n}$ blocks (where $\mathfrak{n} > 0$ since $\ell > 0$) and hence from the second point of Theorem 1, the degree of both $Q_i(x)$ and $Q'_i(x)$ is $2^{\lfloor \lg \mathfrak{n} \rfloor + 1} - 1 = d - 1$. From the definition of $Q_{c+1}(x)$ in (8), the coefficients of $Q_{c+1}(x)$ are exactly the coefficients of $Q_i(x)$, $i = 1, \ldots, c$, and similarly the coefficients of $Q'_{c+1}(x)$ are exactly the coefficients of $Q'_i(x)$, $i = 1, \ldots, c$. Since $Q_\jmath(x) \neq Q'_\jmath(x)$, it follows that $Q_{c+1}(x) \neq Q'_{c+1}(x)$ and hence $Q(x) \neq Q'(x)$. □

Lemma 4. *Let X be a binary string of length $L \geq 1$ and n be a positive integer. Let $\ell = \lceil L/n \rceil$. Let $(M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))$ be the output of $\mathsf{pad2}(\mathsf{format}(X))$. Then the following holds.*

1. *If $c \mid \ell$, then $\ell + 1 < \mathsf{deg}(Q(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) \leq 2\ell + 1$.*
2. *If $c \nmid \ell$, then $\ell + 1 < \mathsf{deg}(Q(x; M_1, \ldots, M_\ell, \mathsf{bin}_{m-1}(L))) < 2\ell + 2c + 1$.*

Proof. As argued in the proof of Lemma 3 the degree of $Q_i(x)$ is $d - 1$ for $i = 1, \ldots, c$. So from (8), the degree of $Q_{c+1}(x)$ is $cd - 1$ and hence the degree of $Q(x)$ is $cd + 1$, where $d = 2^{\lfloor \lg \mathfrak{n} \rfloor + 1}$, and $\mathfrak{n} = \lceil \ell/c \rceil$. Suppose $\lfloor \lg \mathfrak{n} \rfloor = \rho$, i.e. $2^\rho \leq \mathfrak{n} = \lceil \ell/c \rceil < 2^{\rho+1}$. So the degree of $Q(x)$ is $c2^{\rho+1} + 1$.

If $c \mid \ell$, then $c2^\rho \leq \ell < c2^{\rho+1}$ from which we obtain the first point. If $c \nmid \ell$, then write $\ell/c = a + f$, where a is an integer and $0 < f < 1$. So $\lceil \ell/c \rceil = a + 1$ and $2^\rho \leq a + 1 < 2^{\rho+1}$. Using $a = \ell/c - f$, we obtain $c2^\rho \leq \ell + c(1 - f) < c2^{\rho+1}$. This yields $\ell + 1 + c(1 - f) < c2^{\rho+1} + 1 \leq 2\ell + 2c(1 - f) + 1$. Since $0 < f < 1$, we obtain the second point. □

Theorem 4. *Let $p = 2^m - \delta$ be a prime and μ be a positive integer such that $\mu < m$ and $\delta < 2^\mu - 1$. 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 $\mathbb{Z}_{2^\mu}$. Let $\ell = \lceil L/n \rceil$. Suppose τ is chosen uniformly at random from $\{0,1\}^k$. Then the following holds.*

1. *If $c = 1$, then $\Pr[c\text{-}\mathsf{decBRWHash}_\tau(X) - c\text{-}\mathsf{decBRWHash}_\tau(X') = \alpha] \leq (2\ell + 1) \cdot 2^{m-k-\mu+1}$.*
2. *If $c > 1$, then $\Pr[c\text{-}\mathsf{decBRWHash}_\tau(X) - c\text{-}\mathsf{decBRWHash}_\tau(X') = \alpha] < (2\ell + 2c + 1) \cdot 2^{m-k-\mu+1}$.*

Proof. Since X and X' are distinct, by Lemma 3, the corresponding polynomials $Q(x)$ and $Q'(x)$ are also distinct. Further, by construction the constant terms of both $Q(x)$ and $Q'(x)$ are zero. Using Lemma 1, the required probability is at most $2^{m-k-\mu+1}$ times the degree of $Q(x)$. From Lemma 4, the degree of $Q(x)$ is at most $2\ell+1$ if $c=1$, and is less than $2\ell+2c+1$ if $c>1$. □

Note that for $c=1$, the AXU bound of c-decBRWHash is the same as that of BRWHash. This is a consequence of the fact that c-decBRWHash is a generalisation of BRWHash. Note that the AXU bound degrades with c. In practice, c will be a small constant, and hence the degradation will be insignificant. In our implementations, we take $c=4$, and we do not envisage a future situation where it will be useful to take c to be greater than 8.

4 Field Arithmetic

The focus of our implementation is SIMD operations. In particular, we focus on the `avx2` instructions of Intel processors. The presently available `avx2` instructions determine how the elements of the field $\mathbb{F}_p$ are represented and the field arithmetic is performed. First we describe the representation of individual field elements and arithmetic for a pair of field elements. Later we describe how the representation of a single field element can be lifted to a vector of 4 field elements, and how simultaneous arithmetic is performed on 4 pairs of field elements. We note that previous works [20, 8, 15] and also the independent work [28] provide descriptions of field arithmetic. The descriptions in [8, 15] are for 64-bit sequential implementations, while the descriptions in [20, 28] are for SIMD implementation. For `avx2` implementations, compared to previous works, we provide more details regarding overflow analysis based on limb sizes, partial reduction, delayed reduction, and reduction chain. In particular, in comparison to [28], we provide the complete details specific to the primes $2^{130}-5$ and $2^{127}-1$, while [28] sets up an optimisation problem for overflow analysis corresponding to pseudo-Mersenne primes to derive such details; the actual implementation details for $2^{130}-5$ and $2^{127}-1$ are not explicitly discussed.

The `avx2` instructions allow applying the same operation simultaneously on four different pairs of operands. The basic data type is a 256-bit quantity which is considered to be 4 64-bit words. Given two such 256-bit quantities, it is possible to simultaneously add or multiply the four pairs of 64-bit operands that arise from the same 64-bit positions of the two 256-bit quantities. In particular, the instruction `vpmuludq` performs 4 simultaneous multiplications and `vpaddq` performs 4 simultaneous additions; two other relevant instructions are `vpand` (which performs 4 simultaneous bitwise AND operations), `vpsllq` (which performs 4 simultaneous left shifts), and `vpsrlq` (which performs 4 simultaneous right shifts).

There is, however, no scope for handling overflow (i.e. the result of an arithmetic instruction is greater than or equal to 2^{64}) with `avx2` instructions. So to ensure correctness of the computation, the result of the addition and multiplication instructions must also fit within a 64-bit word. In particular, the add-with-carry operation is not available with `avx2` instructions. Since there is no scope for handling overflow, to ensure the correctness of the results of addition and multiplication, the whole 64 bits of the operands cannot be information bits. For addition, at most the 63 least significant bits of the operands can contain information, so that the result of the addition is at most a 64-bit quantity. For multiplication, at most the 32 least significant bits of the operands can contain information, so that the result of the multiplication is at most a 64-bit quantity. So in effect the `avx2` instructions support 32-bit multiplication.

Remark 4. *Intel processors also provide support for 64-bit integer multiplication. In particular, from the Haswell processor onwards three instructions, namely* `mulx`*,* `adcx`*, and* `adox`*, are provided which allow a double carry chain multiplication and squaring to be performed [27, 26, 24]. Implementations which utilise these instructions have been called* `maax` *implementations [24]. For both* polyHash *and* BRWHash*,* `maax` *implementations were reported in [8, 25] for both the primes* $2^{130}-5$ *and* $2^{127}-1$*. Later we compare the speeds of these* `maax` *implementations with the speeds of the new* `avx2` *implementations that are reported in this paper.*

Below we describe the representation and field arithmetic separately for the primes $2^{130}-5$ and $2^{127}-1$.

4.1 Case of $p = 2^{130} - 5$

Elements in $\mathbb{F}_p$ are represented using 130-bit quantities. An element f of the field $\mathbb{F}_p$ is represented as a 5-limb quantity, where each limb is a 26-bit quantity, i.e.

$$f \;=\; f_0 + f_1 2^{26} + f_2 2^{26\cdot 2} + f_3 2^{26\cdot 3} + f_4 2^{26\cdot 4},$$

where each f_i is a 26-bit quantity. We call the coefficients of the powers of 2^{26} to be the limbs of f. Sometimes we write f as $(f_0, f_1, f_2, f_3, f_4)$. The reason for choosing base 2^{26} representation is that `avx2` supports only 32-bit multiplications, so that multiplication of two 26-bit operands results in a 52-bit operand which fits within a 64-bit word. Using 32 bits to store a 26-bit limb is an example of unsaturated representation; see [24] for explanations of saturated and unsaturated representations which introduced these terminology (as far as we are aware).

Suppose e is another field element whose limbs are e_0, e_1, e_2, e_3, e_4. The product $ef \bmod p$ can be written as a 5-limb quantity $h = h_0 + h_1 2^{26} + h_2 2^{26\cdot 2} + h_3 2^{26\cdot 3} + h_4 2^{26\cdot 4}$ as follows (see [20]).

$$\begin{array}{rcl} h_0 &=& e_0f_0 + 5(e_1f_4 + e_2f_3 + e_3g_2 + e_4g_1) \\ h_1 &=& e_0f_1 + e_1f_0 + 5(e_2f_4 + e_3f_3 + e_4f_2) \\ h_2 &=& e_0f_2 + e_1f_1 + e_2f_0 + 5(e_3f_4 + e_4f_3) \\ h_3 &=& e_0f_3 + e_1f_2 + e_2f_1 + e_3f_0 + 5e_4f_4 \\ h_4 &=& e_0f_4 + e_1f_f + e_2f_2 + e_3f_1 + e_4f_0. \end{array} \tag{10}$$

In the above we have used $2^{130} \equiv 5 \bmod p$. Consider $h_0 = e_0f_0+5(e_1f_4+e_2f_3+e_3f_2+e_4f_1) = (e_0f_0+e_1f_4+e_2f_3+e_3f_2+e_4f_1)+4(e_1f_4+e_2f_3+e_3f_2+e_4f_1) = u+v$, where $u = e_0f_0+e_1f_4+e_2f_3+e_3f_2+e_4f_1$ and $v = 4(e_1f_4 + e_2f_3 + e_3f_2 + e_4f_1)$. Each of the cross product terms e_if_j is 52-bit long; the sum of four such quantities is at most 54-bit long; the multiplication by 4 increases the length by 2 bits, so v is at most 56-bit long; by a similar reasoning u is at most 55-bit long; so the sum $h_0 = u + v$ is at most 57-bit long. By a similar argument, the lengths of the other h_j's are also at most 57 bits. So the limbs of h are (at most) 57-bit quantities. *By an unreduced multiplication we mean obtaining* $(h_0, \ldots, h_4)$ *from* $(e_0, \ldots, e_4)$ *and* $(f_0, \ldots, f_4)$ *as given in* (10).

Further reduction of the limbs of h to 26-bit quantities are not immediately done. Recall that both grouped Horner and BRW evaluation support lazy or delayed reduction. The limbs of h are stored as 64-bit quantities. If we perform limb-wise addition of at most 64 5-limb quantities all of whose limbs are 57 bits long, then the limbs of the final sum are at most 63 bits long, and so there is no overflow. So delayed reduction strategy can be applied up to the sum of 64 quantities. Note that instead of being 26-bit quantities, if at most one of the e_i's and at most one of the f_j's were 27-bit quantities, then the limb sizes would be 58 bits (instead of 57 bits), and there would

be no overflow when delayed reduction is employed up to the addition of 32 quantities. We take advantage of this observation during the reduction step, where we allow one limb to be a 27-bit quantity. We found that for grouped Horner implementing delayed reduction beyond group size 4 did not lead to speed improvement. For BRW evaluation on $\mathfrak{n} = \lceil \ell/4 \rceil$ vector blocks, the maximum number of additions required by Algorithm 1 in Appendix A due to delayed reduction is the size of the stack which by Proposition 1 is at most $\lfloor \lg \mathfrak{n} \rfloor - t + 1$. So if the number of blocks ℓ is at most about 2^{31+t}, then there is no problem with delayed reduction.

Remark 5. *In computing h using* (10)*, suppose e is a fixed quantity, while f varies. In such a situation, given $e = (e_0, e_1, \ldots, e_4)$ it is advantageous to pre-compute $\tilde{e} = (5 \cdot e_1, 5 \cdot e_2, 5 \cdot e_3, 5 \cdot e_4)$, as then h_0 can be computed as $h_0 = e_0 f_0 + (5 \cdot e_1) f_4 + (5 \cdot e_2) f_3 + (5 \cdot e_3) f_2 + (5 \cdot e_4) f_1$, and similarly for h_1, h_2 and h_3. This saves the four multiplications by 5. This technique was used in [20].*

Remark 6. *The product form shown in* (10) *corresponds to schoolbook multiplication. We have also experimented with Karatsuba multiplication, but did not find it to provide better performance than schoolbook multiplication.*

Next we consider the reduction. Suppose $h = h_0 + h_1 2^{26} + h_2 2^{26 \cdot 2} + h_3 2^{26 \cdot 3} + h_4 2^{26 \cdot 4}$, where we assume (due to possible delayed reduction) that each h_i is a 63-bit quantity. The goal of the reduction is to reduce (modulo p) so that each limb is a 26-bit quantity. This is a complete reduction and is done once at the end of the computation. For intermediate reductions, we reduce all limbs other than h_1 to 26-bit quantitites and reduce h_1 to a 27-bit quantity. Such a partial reduction is faster than a complete reduction. As mentioned above, this does not cause an overflow when implementing delayed reduction.

The basic idea of the reduction for a limb is to retain the 26 least significant bits in the limb and add the other bits (which we call the carry out) to the next limb. In other words, for any $i \in \{0, \ldots, 3\}$, write $h_i = h_{i,0} + h_{i,1} 2^{26}$ with $h_{i,0} = h_i \bmod 2^{26}$, update h_i to $h_{i,0}$ and add $h_{i,1}$ to h_{i+1}; write $h_4 = h_{4,0} + h_{4,1} 2^{26}$ with $h_{4,0} = h_4 \bmod 2^{26}$, update h_4 to $h_{4,0}$ and add $5h_{4,1}$ (using once again $2^{130} \equiv 5 \bmod p$) to h_0. Since h_{i+1}, $i = 0, \ldots, 3$ is a 63-bit quantity, adding $h_{i,1}$ to h_{i+1} does not cause an overflow. It is important to note that the reduction procedure can start from any $i \in \{0, \ldots, 3\}$, and in particular the procedure does not have to start from h_0. For the reduction, we use the reduction chain $h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_0 \to h_1$, which is a chain having 6 steps. The first five steps reduce $h_0, \ldots, h_4$ to 26-bit quantities, and the carry out of h_4 is at most a 37-bit quantity. Multiplying this carry out by 5 creates at most a 40-bit quantity, and adding it to the 26-bit h_0 makes h_0 at most a 41-bit quantity. The last step $h_0 \to h_1$ reduces h_0 to 26 bits and adds the at most 15-bit carry out to the 26-bit h_1 to make the new h_1 a 27-bit quantity.

SIMD implementation. Suppose a_0, a_1, a_2 and a_3 are four field elements, and for $i = 0, \ldots, 3$, suppose the limbs of a_i are $a_{i,0}, a_{i,1}, \ldots, a_{i,4}$, where each $a_{i,j}$ is a 26-bit (or 27-bit) quantity. The total of 20 limbs of a_0, a_1, a_2 and a_3 are packed into 5 256-bit words $U_0, \ldots, U_4$ in the following manner. Consider U_j to be $U_{j,0} || U_{j,1} || U_{j,2} || U_{j,3}$, where each $U_{j,i}$ is a 64-bit word. For $i = 0, \ldots, 3$, $a_{i,j}$ is stored in the 26 least significant bits of $U_{j,i}$. See Figure 1 for an illustration. This idea of packing is from [20] and was also used in [28].

Similarly, suppose b_0, b_1, b_2 and b_3 are four field elements which are packed into 5 256-bits words $V_0, \ldots, V_4$. Let $c_i = a_i b_i \bmod p$, $i = 0, \ldots, 3$, where the 5-limb representation of c_i is obtained from the 5-limb representations of a_i and b_i in a manner similar to (10). Then the 5-limb representations of c_0, c_1, c_2, c_3 are obtained in 5 256-bit words $W_0, W_1, \ldots, W_4$. Using the `avx2` instructions

U_0	$a_{3,0}$	$a_{2,0}$	$a_{1,0}$	$a_{0,0}$
U_1	$a_{3,1}$	$a_{2,1}$	$a_{1,1}$	$a_{0,1}$
U_2	$a_{3,2}$	$a_{2,2}$	$a_{1,2}$	$a_{0,2}$
U_3	$a_{3,3}$	$a_{2,3}$	$a_{1,3}$	$a_{0,3}$
U_4	$a_{3,4}$	$a_{2,4}$	$a_{1,4}$	$a_{0,4}$

Figure 1: Packing of four field elements $a_0, \ldots, a_3$ into 5 256-bit words $U_0, \ldots, U_4$.

`vpmuludq` and `vpaddq`, it is possible to obtain $W_0, W_1, \ldots, W_4$ from $U_0, U_1, \ldots, U_4$ and $V_0, V_1, \ldots, V_4$. In particular, we note that 25 `vpmuludq` instructions are required to obtain all the cross product terms and additionally 4 `vpmuludq` instructions are required to perform the multiplications by 5. The multiplications by 5 are not required if one of the operands in each of the four multiplications is fixed (see Remark 5).

The computation of grouped Horner and BRW proceeds using the delayed reduction strategy. So the limbs of the W_j's grow to at most 63-bit quantities. Then the reduction strategy described above is applied using SIMD instructions to the 5 words $W_0, W_1, \ldots, W_4$.

A different SIMD representation. While the above method of packing was suggested in [20], another method of packing four field elements into 256-bit words was actually used in [20]. Each of the four field elements is represented using 5 26-bit quantities and hence can fit in 5 32-bit words. So the four field elements together require 20 32-bit words to be stored. Three 256-words provide a total of 24 32-bit words. So the 20 32-bit words representing the four field elements can be stored in three 256-bit words. This is the representation of 4 field elements that was used in [20]. Stored in this manner, it is not possible to directly perform the 4-way SIMD multiplication, and requires more instructions for unpacking and repacking. The rationale for adopting such a strategy is that using 3 instead of 5 256-bit words to store operands frees up some of the 256-bit registers for performing the actual arithmetic, and this compensates for the penalty incurred due to packing and unpacking. In our implementations, on the other hand, we have used 5 256-words to represent the operands so that the multiplication operations can be directly applied. By carefully managing register allocation, we did not encounter the problem of unavailable registers. This was possible since we implemented directly in assembly, whereas the implementation in [20] is in Intel intrinsics which is at a higher level. Since in our approach the problem of unavailable registers does not arise, using the packed representation of field elements used in [20] would incur an unnecessary penalty. So we chose not to use that strategy.

4.2 Case of $p = 2^{127} - 1$

Elements in $\mathbb{F}_p$ are represented using 128-bit quantities. Keeping in mind the fact that SIMD supports 32-bit multiplication, there are two possible representations of elements of $\mathbb{F}_p$, namely a 4-limb, or a 5-limb representation.

5-limb representation. The 5-limb representation is almost the same as that of the 5-limb representation for $2^{130} - 5$ described in Section 4.1, i.e. a base 2^{26} representation can be used. The only difference in the multiplication procedure shown in (10) is that the constant 5 is replaced by 8, since $2^{130} \equiv 8 \bmod (2^{127} - 1)$. For the reduction algorithm, we use the chain $h_3 \rightarrow h_4 \rightarrow h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow$

$h_3 \rightarrow h_4$, i.e. we start the chain at h_3 instead of starting at h_0. The chain consists of 6 steps as in the case for $2^{130}-5$. For the step $h_4 \rightarrow h_0$, we reduce h_4 to 23 bits (note that $4 \times 26 + 23 = 127$) and produce a carry out of at most 40 bits which is then added to h_0 (since $2^{127} \equiv 1 \bmod (2^{127}-1)$, there is no need to multiply the carry out by any constant). The chain finally stops at h_4 which results in h_4 being at most a 24-bit quantity. The number of operations required for multiplication and reduction using the 5-limb representation of $2^{127}-1$ is almost the same as the number of operations required for multiplication and reduction using the 5-limb representation of $2^{130}-5$.

The SIMD implementation of the 5-limb representation for $2^{127}-1$ is also very similar to the SIMD implementation of the 5-limb representation for $2^{130}-5$. Four field elements are stored in 5 256-bit words as shown in Figure 1 and multiplication is done using `avx2` instructions. In particular, with the 5-limb representation, multiplication requires requires 25 `vpmuludq` instructions to compute the cross product terms, plus 4 `vpsllq` instructions for the multiplications by 8.

4-limb representation. For the 4-limb representation, an element f of the field $\mathbb{F}_p$ is represented as a 4-limb quantity, where each limb is a 32-bit quantity, i.e.

$$f \;=\; f_0 + f_1 2^{32} + f_2 2^{32 \cdot 2} + f_3 2^{32 \cdot 3},$$

where each f_i is a 32-bit quantity. Suppose e is another field element whose limbs are e_0, e_1, e_2, e_3. The product $ef \bmod p$ can be written as a 4-limb quantity $h = h_0 + h_1 2^{32} + h_2 2^{32 \cdot 2} + h_3 2^{32 \cdot 3}$ as follows.

$$\begin{array}{rcl} h_0 &=& e_0 f_0 + 2(e_1 f_3 + e_2 f_2 + e_3 f_1) \\ h_1 &=& e_0 f_1 + e_1 f_0 + 2(e_2 f_3 + e_3 f_2) \\ h_2 &=& e_0 f_2 + e_1 f_1 + e_2 f_0 + 2 e_3 f_3 \\ h_3 &=& e_0 f_3 + e_1 f_2 + e_2 f_1 + e_3 f_0. \end{array} \tag{11}$$

In the above, we have used $2^{128} \equiv 2 \bmod p$. Each of the cross product terms $e_i f_j$ is 64-bit long. Adding together such terms increases the size of the sum beyond 64 bits. Since `avx2` instructions do not provide any mechanism to handle the carry arising out of additions, one cannot directly add the cross product terms. An alternative procedure needs to be used. Consider h_0. Write $e_i f_j$ as $u + v 2^{32}$, where both u and v are 32-bit quantities. The u's arising from the terms $e_1 f_3$, $e_2 f_2$ and $e_3 f_1$ are added, the sum multiplied by 2, and the result added to the u arising from the the term $e_0 f_0$. This results in h_0 being a 36-bit quantity. Similarly the v's arising from the terms $e_1 f_3$, $e_2 f_2$ and $e_3 f_1$ are added, the sum multiplied by 2, and the result added to the v arising from the term $e_0 f_0$, giving a value v'. The computation of h_1 starts with the initial value v', and is updated by adding the u's arising from the cross product terms in the expression for h_1, while the v's arising from the cross product terms in the expression for h_1 contribute to the value of h_2.

For the reduction algorithm, the chain is $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3$ which consists of 7 steps (which is one step more than the chain for the 5-limb representation). Note that the chain makes two full iterations over the limbs and reduces all the limbs to 32-bit quantities. This is required, since if any limb is greater than 32 bits, then subsequent multiplication with such limbs will cause an overflow.

SIMD implementation packs four field elements with 32-bit limbs into 4 256-bit words in much the same as the packing of four field elements with 26-bit limbs into 5 256-bit words. Using this packed representation, multiplication of four pairs of field elements is done using `avx2` instructions following the description given earlier. An advantage of using the 4-limb representation is that the

number of `vpmuludq` instructions required to compute the multiplication comes down to 16 from 25. However, there is a significant increase in the number additions and shifts. Multiplication using the 4-limb representation requires 16 `vpmuludq`, 40 `vpaddq`, 16 `vmovdqa`, 16 `vpand` and 16 `vpsrlq` operations (plus 3 `vpsllq` instructions for the multiplications by 2). In contrast, multiplication using the 5-limb representation requires 25 `vpmuludq` and 20 `vpaddq` instructions (plus 4 `vpsrlq` instructions for the multiplications by 8). The latencies of the various instructions on the Skylake processor are as follows: `vpmuludq` - 5, `vpaddq` -1, `vpand` - 1, `vpsrlq` - 1, `vmovdqa` - 7 for load and 5 for store. This shows that the penalty due to the additional instructions required for the 4-limb representation more than cancels the benefit of requiring a less number of `vpmuludq` instructions.

Non-availability of carry handling instructions. A major reason that 4-limb representation turns out to be slower is that SIMD instructions do not provide any mechanism to handle the carry out of an addition operation. Since the cross product terms in (11) are all 64-bit quantities, without the availability of an efficient carry handling mechanism, many more operations are required to prevent an overflow condition. If in the future SIMD instructions provide some mechanism for obtaining the carry out of an addition, and/or the add-with-carry operation, then it is likely that the 4-limb representation will provide a significantly faster multiplication algorithm than the 5-limb representation. We note that for the `maax` implementation using 64-bit arithmetic there is excellent support for carry operations (see Remark 4).

Non-availability of 64-bit multiplications. SIMD operations presently do not support full 64-bit multiplication. If in the future, full 64-bit SIMD multiplication is supported (with 2 such simultaneous multiplications using 256-bit words, or 4 such simultaneous multiplications using 512-bit words), then for the prime $2^{130} - 5$, a 3-limb representation can be used, while for $2^{127} - 1$, either a 2-limb, or a 3-limb representation can be used. The speed of multiplication using such a 2-limb representation for $2^{127} - 1$ has the potential to significantly outperform the speed of multiplication of a 3-limb representation of $2^{130} - 5$, especially if along with 64-bit multiplication, carry handling SIMD operations are used. This observation arises from the comparative speed performances of multiplication algorithms for $2^{127} - 1$ and $2^{130} - 5$ for non-SIMD implementation using `maax` instructions (see [8, 25]).

We note that both `avx512` and `avx512_ifma` work on 512-bit registers, but neither instruction set support 4 simultaneous full 64-bit multiplications.

5 Vectorised Algorithms

We describe algorithms for vectorised computation of polyHash and decBRWHash.

5.1 Vectorised Computation of polyHash

One way to exploit parallelism in the computation of polyHash is to divide the sequence of blocks $(M_1, \ldots, M_\ell)$ into $c \geq 2$ subsequences and apply Horner's rule to each of the subsequence. For $c = 4$, such a strategy was used in [20] for vectorised implementation of Poly1305. In [12], this strategy was called c-decimated Horner evaluation.

From (2) and (3), the computation of $\mathsf{polyHash}_\tau(X)$ requires the computation of $\tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell)$, where $(M_1, \ldots, M_\ell)$ is the output of $\mathsf{pad1}(\mathsf{format}(X))$. Let $\rho = \ell \bmod c$ and $\ell' = (\ell - \rho)/c =$

$\lfloor \ell/c \rfloor$. The computation of $\tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell)$ can be done in the following manner. First compute

$$P \;=\; \left\{ \begin{array}{lll} & \tau^c & \cdot \;\; \mathsf{Poly}(\tau^c; M_1, M_{c+1}, M_{2c+1}, \ldots, M_{c\ell'-c+1}) \\ + & \tau^{c-1} & \cdot \;\; \mathsf{Poly}(\tau^c; M_2, M_{c+2}, M_{2c+2}, \ldots, M_{c\ell'-c+2}) \\ + & \cdots & \\ + & \tau & \cdot \;\; \mathsf{Poly}(\tau^c; M_c, M_{2c}, M_{3c}, \ldots, M_{c\ell'}). \end{array} \right. \tag{12}$$

Then

$$\begin{array}{rcll} \tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell) & = & P & \text{if } \rho = 0, \\ \tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell) & = & \tau \cdot \mathsf{Poly}(\tau; P + M_{\ell-\rho+1}, M_{\ell-\rho+2}, \ldots, M_\ell) & \text{if } \rho > 0. \end{array}$$

We focus on the computation of P. For $i = 1, \ldots, c$, let

$$C_i \;=\; \tau^{c+1-i} \cdot \mathsf{Poly}(\tau^c; M_i, M_{c+i}, \ldots, M_{c\ell'-c+i}).$$

Then $P = C_1 + C_2 + \cdots + C_c$. So the computation of P consists of computing $C_1, \ldots, C_c$ and then adding these together. The computation of the vector

$$\mathbf{C} \;=\; (C_1, \ldots, C_c) \tag{13}$$

can be done using an SIMD strategy. The computation of P from $\mathbf{C}$, and the subsequent computation of $\tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell)$ from P (in the case $\rho > 0$) is done using a small number of `maax` operations.

For a non-negative integer j, let $\boldsymbol{\tau}^j = (\tau^j, \ldots, \tau^j)$ be a vector of length c. Further, let $\boldsymbol{\tau}_\theta = (\tau^c, \tau^{c-1}, \ldots, \tau)$ also be a vector of length c. For $i = 1, \ldots, \ell'$, let $\mathbf{M}_i = (M_{ci-c+1}, \ldots, M_{ci})$. Then

$$\mathbf{C} \;=\; \boldsymbol{\tau}_\theta \circ (\boldsymbol{\tau}^c \circ (\cdots (\boldsymbol{\tau}^c \circ (\boldsymbol{\tau}^c \circ \mathbf{M}_1 + \mathbf{M}_2) + \mathbf{M}_3) + \cdots + \mathbf{M}_{\ell'-1}) + \mathbf{M}_{\ell'}), \tag{14}$$

where $\circ$ and $+$ denote SIMD (i.e. component wise) field multiplication and field addition of two c-dimensional vectors respectively.

The computation in (14) requires a total of ℓ' SIMD field multiplications and $\ell' - 1$ SIMD additions. Of the ℓ' SIMD field multiplications, $\ell' - 1$ SIMD field multiplications have $\boldsymbol{\tau}^c$ as one of the operands, while one SIMD field multiplication has $\boldsymbol{\tau}_\theta$ as one of the operands.

As mentioned in Section 2, a delayed (or lazy) reduction strategy was used originally in [7] and developed in [20] to decrease the number of reductions. This strategy was used in a number of follow-up works [8, 15, 28]. The technique is important and provides significant speed improvement (see Remark 5.1 in [8] for a detailed explanation of the speed improvement). We provide a detailed mathematical description of the technique, which includes the exact partitioning technique ($g + 1$ in the first group, g in the second to last but one group, and the remaining in the last group), the recurrence, and the handling of the last group. Such a description is not present in other works – for example, [28] illustrates the technique using a 12-block example and group size 3. Also, we provide precise counts of the number of SIMD operations required by delayed reduction (see Table 3), and the number of bytes required to store the key material (see Table 4a); such details are not available in previous works.

Let $g \geq 1$ be a parameter, and ℓ'' and r be integers such that $(\ell' - g - 1) = g(\ell'' - 1) + r$, with $\ell'' \geq 1$ and $1 \leq r \leq g$. So $\ell' - 1 = g\ell'' + r$, and since $1 \leq r \leq g$, we have $\ell'' = \lceil (\ell' - 1)/g \rceil - 1$. Let $\boldsymbol{\gamma} = \boldsymbol{\tau}^c$.

$$\mathbf{A}_1 \;=\; \left\{ \begin{array}{ll} \mathbf{M}_1 \circ \boldsymbol{\gamma}^{\ell'-1} + \mathbf{M}_2 \circ \boldsymbol{\gamma}^{\ell'-2} + \cdots + \mathbf{M}_{\ell'-1} \circ \boldsymbol{\gamma} + \mathbf{M}_{\ell'} & \text{if } \ell' \leq g+1, \\ \mathbf{M}_1 \circ \boldsymbol{\gamma}^g + \mathbf{M}_2 \circ \boldsymbol{\gamma}^{g-1} + \cdots + \mathbf{M}_g \circ \boldsymbol{\gamma} + \mathbf{M}_{g+1} & \text{if } \ell' > g+1. \end{array} \right.$$

For $i = 1 \ldots, \ell'' - 1$, let

$$\mathbf{A}_{i+1} \;=\; \mathbf{A}_i \circ \boldsymbol{\gamma}^g + \mathbf{M}_{ig+2} \circ \boldsymbol{\gamma}^{g-1} + \cdots + \mathbf{M}_{(i+1)g} \circ \boldsymbol{\gamma} + \mathbf{M}_{(i+1)g+1},$$

and

$$\mathbf{A}_{\ell''+1} \;=\; \mathbf{A}_{\ell''} \circ \boldsymbol{\gamma}^r + \mathbf{M}_{\ell''g+2} \circ \boldsymbol{\gamma}^{r-1} + \mathbf{M}_{\ell''g+3} \circ \boldsymbol{\gamma}^{r-2} + \cdots + \mathbf{M}_{\ell''g+r} \circ \boldsymbol{\gamma} + \mathbf{M}_{\ell''g+r+1}.$$

Then it is not difficult to verify that

$$\mathbf{C} \;=\; \begin{cases} \boldsymbol{\tau}_\theta \circ \mathbf{A}_1 & \text{if } \ell' \leq g+1, \\ \boldsymbol{\tau}_\theta \circ \mathbf{A}_{\ell''+1} & \text{if } \ell' > g+1. \end{cases} \tag{15}$$

Suppose that the key powers $\boldsymbol{\gamma}, \boldsymbol{\gamma}^2, \boldsymbol{\gamma}^3, \ldots, \boldsymbol{\gamma}^g$ are pre-computed. If $\ell' = 1$, then $\mathbf{A}_1 = \mathbf{M}_1$ and no SIMD multiplication or SIMD reduction are required to compute $\mathbf{A}_1$. If $1 < \ell' \leq g+1$, then computing $\mathbf{A}_1$ requires $\ell' - 1$ unreduced SIMD multiplications and one SIMD reduction. If $\ell' > g+1$, then computing $\mathbf{A}_1$ requires g unreduced SIMD multiplications and one SIMD reduction; for $1 \leq i \leq \ell'' - 1$, computing $\mathbf{A}_{i+1}$ from $\mathbf{A}_i$, $1 \leq i \leq \ell'' - 1$, requires g unreduced SIMD multiplications and one SIMD reduction; and computing $\mathbf{A}_{\ell''+1}$ from $\mathbf{A}_{\ell''}$ requires r unreduced SIMD multiplications and one reduction. So if $\ell' > g+1$, then to compute $\mathbf{A}_{\ell''+1}$ the total number of unreduced SIMD multiplications required is equal to $\ell''g + r = \ell' - 1$, and the total number of SIMD reductions is equal to $\ell'' + 1 = \lceil (\ell' - 1)/g \rceil$. In fact, for all values of $\ell' \geq 1$, the total number of unreduced SIMD multiplications required is $\ell' - 1$, and the total number of SIMD reductions is $\lceil (\ell' - 1)/g \rceil$.

The computation of $\mathbf{C}$ in (15) from $\mathbf{A}_1$ or $\mathbf{A}_{\ell''+1}$ requires one unreduced SIMD multiplication and one SIMD reduction. However, the SIMD reduction can be avoided. The unreduced SIMD multiplication by $\boldsymbol{\tau}_\theta$ results in $\mathsf{unreduced}(\mathbf{C})$. The four components $\mathsf{unreduced}(C_i)$, $i = 1, \ldots, 4$, of $\mathsf{unreduced}(\mathbf{C})$ are added together and a single reduction applied to the sum using `maax` operations. So for the computation of P, the total number of unreduced SIMD multiplications is equal to ℓ', and the total number of SIMD reductions is equal to $\lceil (\ell' - 1)/g \rceil$.

In our implementations, we have taken $c = 4$, i.e. we have made 4-way SIMD implementation using `avx2` instructions. Given the delayed reduction parameter g, the key power vectors $\boldsymbol{\tau}^4, \boldsymbol{\tau}^8, \ldots, \boldsymbol{\tau}^{4g}$ are required. Additionally, the vector $\boldsymbol{\tau}_\theta$ is required. To compute $\boldsymbol{\tau}_\theta$ and the key power vectors $\boldsymbol{\tau}^{4i}$, $i = 1, \ldots, g$, the key powers $\tau, \tau^2, \tau^3, \tau^4, \tau^8, \tau^{12}, \ldots, \tau^{4g}$ are first computed using `maax` instructions and then appropriately organised into the required key power vectors. The actual SIMD computation of $\mathbf{C}$ starts after the required key power vectors have been computed. The number of unreduced SIMD multiplications and the number of SIMD reductions for the computation of C are shown in Table 3. Additionally, there are a small number of non-SIMD operations required at the end to compute P and to compute $\tau \cdot \mathsf{Poly}(\tau; M_1, \ldots, M_\ell)$ from P. These operations are implemented using `maax` instructions and their counts are not shown in Table 3.

As explained after (12) if the number of blocks is not a multiple of 4, then the last few blocks (between 1 and 3) need to be tackled sequentially. As a result, the entire computation cannot proceed uniformly as a 4-way SIMD computation. It is possible to pre-pend a number of zero blocks, so that the total number of blocks becomes a multiple of 4, and the 4-way SIMD can be employed for the entire message. Such pre-pending does not alter the hash function and is only an implementation issue. This technique was used in [9]. There is, however, an efficiency issue which does not combine well with the technique of pre-pending zero blocks. If the message is stored as 32-byte aligned data, then the instruction `vmovdqa` (i.e. aligned move) can be used to read the data.

	unred mult (SIMD)	red (SIMD)
polyHash	$\lfloor \ell/4 \rfloor$	$\lceil (\lfloor \ell/4 \rfloor - 1)/g \rceil$
4-decBRWHash	$\lfloor \lceil \ell/4 \rceil /2 \rfloor$	$1 + \lfloor \lceil \ell/4 \rceil /4 \rfloor$

Table 3: Counts of 4-way SIMD operations required for computing P (as part of the computation of polyHash) and decBRWHash for ℓ message blocks. Full computations of polyHash and decBRWHash require additionally a small number of `maax` operations which are not shown in the table. For polyHash, the parameter g is a positive integer.

Such aligned read is faster than the unaligned read `vmovdqu`. If the zero pre-pending technique is not used, then with successive aligned moves successive 32 bytes of data can be read. On the other hand, if zero-prepending is used, then this is no longer possible. As a result, the reading of the data becomes slower. We have implemented both the technique of zero pre-pending without support of aligned moves, and not using zero pre-pending which supports aligned moves. We find no significant difference in the timings.

5.2 Vectorised Computation of c-decBRWHash

The computation of c-decBRWHash$_\tau(X)$ requires the evaluation of $Q_1(\tau), Q_2(\tau), \ldots, Q_c(\tau)$. Evaluations of the $Q_i(\tau)$'s require evaluations of c BRW polynomials, where the inputs to the BRW polynomials are disjoint and the number of message blocks provided as input is the same for all the BRW polynomials. Consequently, the c BRW polynomials can be simultaneously evaluated at τ using an SIMD strategy.

For $i = 1, \ldots, \mathfrak{n} = \lceil \ell/c \rceil$, let $\mathbf{M}_i = (M_{(i-1)c+1}, M_{(i-1)c+2}, \ldots, M_{ic})$, where the message blocks $M_1, M_2, \ldots$ are as defined in the description of c-decBRWHash. Algorithm 1 in Appendix A can be used to compute c simultaneous BRW polynomials, i.e. Algorithm 1 can be used in a c-way SIMD manner, where the input consists of the sequence of c-way vectors $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_\mathfrak{n}$. The key vector is the vector $\boldsymbol{\tau} = (\tau, \ldots, \tau)$ of length c. The key power vectors required are $\boldsymbol{\tau}^{2^j}$, for $j = 0, \ldots, \lfloor \lg \mathfrak{n} \rfloor$. The key power τ^{2^j} are computed from which the key power vector $\boldsymbol{\tau}^{2^j}$ is computed. The algorithm uses keyPow, stack and tmp as internal arrays and variables. The SIMD version of the algorithm uses vector versions of these arrays and variables. In particular, keyPow$[j]$ stores the vector $\boldsymbol{\tau}^{2^j}$, stack$[j]$ stores a vector of length c, and tmp is a vector of length c, where the i-th components of stack$[j]$ and tmp correspond to the computation of the i-th BRW polynomial. All unreduced multiplications, reductions and unreduced additions are done component wise on the vectors. With this strategy Algorithm 1 becomes an SIMD algorithm for the simultaneous computation of the c BRW polynomials.

We have implemented the SIMD version of Algorithm 1 for $c = 4$. Table 3 provides the number of 4-way SIMD operations required for computing 4-decBRWHash using the SIMD version of Algorithm 1. In addition to these SIMD operations, there are a small number of other operations which are required to compute Q_5 from Q_1, Q_2, Q_3, Q_4 and to compute Q from Q_5 (see Section 3). These are implemented using `maax` operations, and the count of these operations are not shown in Table 3.

Remark 7. *Taking $c = 4$ gives* 4-decBRWHash*. Our implementation of* 4-decBRWHash *was made using the* `avx2` *instruction set of Intel. Taking $c = 8$, we obtain* 8-decBRWHash*. It is possible to implement* 8-decBRWHash *using either the* `avx512` *or the* `avx512_ifma` *instruction sets of Intel. Following the discussion in Section 3.1,* 4-decBRWHash *can be implemented using the 64-bit* `maax`

instructions, and 8-decBRWHash *can be implemented using both the* avx2 *instruction set, and the 64-bit* maax *instructions. We leave these as possible future works.*

5.3 Comparison between polyHash and 4-decBRWHash

We make a comparison between the 4-way SIMD computation of polyHash and 4-decBRWHash. There are two aspects to the comparison, the efficiency and the storage requirement.

Efficiency. The SIMD operation counts for polyHash and 4-decBRWHash are shown in Table 3. The number of unreduced SIMD multiplications required for 4-decBRWHash is about half of what is required for polyHash. The number of SIMD reductions required by 4-decBRWHash is at most that required by polyHash for $g \leq 4$, while for $g > 4$, polyHash requires less number of SIMD reductions. In our implementations, we found that taking $g > 4$ does not provide speed improvement. The halving of the number of unreduced SIMD multiplications indicates that 4-decBRWHash should be substantially faster than polyHash for all values of g. Experimental results for the primes $2^{130} - 5$ and $2^{127} - 1$, however, show that for avx2 implementations while there is a noticeable speed-up of 4-decBRWHash over polyHash, the actual speed-up obtained is less than what is theoretically predicted by operation counts. We explain the reasons for such an observation.

Consider the prime $2^{130} - 5$. (A similar reasoning applies for the 5-limb representation based on the prime $2^{127} - 1$.) For the 4-way SIMD computation of polyHash1305, the vectors of key powers $\boldsymbol{\tau}^4, \boldsymbol{\tau}^8, \ldots, \boldsymbol{\tau}^{4g}$ are fixed. All the 4-way multiplications involved in the 4-way computation of polyHash1305 have some key power vector $\boldsymbol{\tau}^{4i}$ as one of the arguments. This has two effects.

First, as noted in Remark 5 when the operand e to the multiplication is fixed, it is possible to pre-compute $\widetilde{e}$ so that the 4 multiplications by 5 are not required. Extending this to SIMD operations, for each key power $\boldsymbol{\tau}^{4i}$, we compute the corresponding $\widetilde{\boldsymbol{\tau}^{4i}}$, and can avoid the SIMD multiplications by 5. Also, for the vector $\boldsymbol{\tau}_\theta$ we compute $\widetilde{\boldsymbol{\tau}_\theta}$, so that the SIMD multiplications by 5 can be avoided while multiplying by $\boldsymbol{\tau}_\theta$. As a result the number of vpmuludq instructions required for an unreduced multiplication is 25 in the case of SIMD computation of polyHash. In contrast, in the computation of 4-decBRWHash, none of the multiplications are by a fixed element. So the number of vpmuludq instructions required for an unreduced multiplication is 29 in the case of SIMD computation of 4-decBRWHash.

The second effect of multiplying with a fixed element is more generic. When a fixed element is used for repeated multiplication, during actual execution this element is kept either on chip or in the cache memory. This significantly reduces the time for reading the element from the memory and leads to a significant increase in speed which is not explained by simply counting the number of arithmetic operations. For the maax implementation also, a similar speed-up was observed and explained in details in [8].

As a combined result of the above two effects, the speed improvement of SIMD computation of 4-decBRWHash over SIMD computation of polyHash is less than what is theoretically predicted, though a significant improvement in the speed is observed for messages which are a few kilo bytes or longer. We provide detailed timing results in Section 6.

Storage. Consider the prime $2^{130} - 5$ where elements are represented using 5 limbs. The storage requirement based on using the 5-limb representation of the prime $2^{127} - 1$ is the same as the storage requirement for the prime $2^{130} - 5$.

	$g=1$	$g=2$	$g=3$	$g=4$
# bytes	448	736	1024	1312

(a) For various values of g, the number of bytes of key material required to be stored for computing either polyHash1305 or polyHash1271 using 4-way SIMD.

	# of blocks ℓ						
	1-4	5-12	13-28	29-60	61-124	125-252	253-508
# bytes	0	160	320	480	640	800	960

(b) The number of bytes of key material required to be stored for computing either decBRWHash1305 or decBRWHash1271.

Table 4: Key storage requirements.

The 4-way SIMD computation of polyHash requires the pre-computation and storage of the key power vectors $\boldsymbol{\tau}^4, \boldsymbol{\tau}^8, \ldots, \boldsymbol{\tau}^{4g}$ as well as the vector $\boldsymbol{\tau}_\theta$. Additionally, to avoid the multiplications by 5, it is also required to store the associated vectors $\widetilde{\boldsymbol{\tau}^4}, \widetilde{\boldsymbol{\tau}^8}, \ldots, \widetilde{\boldsymbol{\tau}^{4g}}$. The associated vector $\widetilde{\boldsymbol{\tau}_\theta}$ is also required, but it is required only at the end, and so is not pre-computed and carried forward. Each $\boldsymbol{\tau}^{4i}$ and also $\boldsymbol{\tau}_\theta$ requires 5 256-bit words to be stored, i.e. a total of 160 bytes. The associated vector $\widetilde{\boldsymbol{\tau}^{4i}}$ requires 4 256-bit words to be stored, i.e. a total of 128 bytes. So the total number of bytes required for storing all the key power vectors is $g \cdot (160 + 128) + 160 = 160 + 288g$. Concrete values of the storage requirement for various values of g are shown in Table 4a.

For computing 4-decBRWHash, the key power vectors $\boldsymbol{\tau}^{2^j}$, $j = 0, \ldots, \lfloor \lg \mathfrak{n} \rfloor = \lfloor \lg \lceil \ell/4 \rceil \rfloor$ are required. Each of these key power vector requires $5 \cdot 32 = 160$ bytes to be stored, so for an ℓ-block message, the total number of bytes required to store all the key power vectors is $160 \cdot \lfloor \lg \lceil \ell/4 \rceil \rfloor$. The key power vector $\boldsymbol{\tau}^d$ is required at the end to combine the four independent BRW computations. Since this vector is required only at the end, it is not pre-computed. Concrete values of the storage requirement for various values of ℓ are shown in Table 4b.

5.4 Efficiency Trade-Off Between $2^{130} - 5$ and $2^{127} - 1$

There is a generic disadvantage of $2^{127} - 1$ in comparison to $2^{130} - 5$. From Table 1, the block size n is 120 for $2^{127} - 1$ and 128 for $2^{130} - 5$. So for any given message, the number of blocks for $2^{127} - 1$ will be about 16/15 times the number of blocks for $2^{130} - 5$. Being required to process more blocks, suggests that hash functions based on $2^{127} - 1$ will be slower than the corresponding hash functions based on $2^{130} - 5$. The other aspect to consider is the speed of an individual multiplication. If the individual multiplication for $2^{127} - 1$ is faster than the individual multiplication for $2^{130} - 5$, then this may compensate the requirement of processing more blocks. For `maax` implementations, it is indeed the case that the individual multiplication for $2^{127} - 1$ is substantially faster than the individual multiplication for $2^{130} - 5$, resulting in the hash functions based on $2^{127} - 1$ being faster than the hash functions based on $2^{130} - 5$. See [8, 25] for the timing results of `maax` implementation which supports this statement. However, for `avx2` implementation, an individual multiplication for $2^{127} - 1$ has efficiency similar to an individual multiplication for $2^{130} - 5$ (see Section 4). As a result, due to the requirement of processing more blocks, for `avx2` implementations, the hash functions based on $2^{127} - 1$ are slower than the corresponding hash functions based on $2^{130} - 5$.

Remark 8. *In the context of* `avx2` *implementation, there is another disadvantage for* $2^{127} - 1$. *For vectorised processing, it is advantageous to use 16-byte block sizes (or block sizes which are multiples of 16 bytes). With a 16-byte block size, it is possible to use two* `vmovdqa(u)` *instructions to read 512 bits of the input into two* `ymm` *registers. The first* `vmovdqa(u)` *instruction reads 32 bytes*

which brings two input blocks into an `ymm` *register, and so does the second. If the block size is not 16 bytes, then such a smooth read operation will not be possible. Reading the input and allocating it to two* `ymm` *registers will be more complicated and hence will require more operations. Since the block size for* $2^{127} - 1$ *is 15 bytes, while the block size for* $2^{130} - 5$ *is 16 bytes, reading the message bytes and allocating to* `ymm` *registers require more operations for* $2^{127} - 1$ *than for* $2^{130} - 5$*.*

Primes other than $2^{127} - 1$ and $2^{130} - 5$ were considered in [15, 28]; see Section 5.5. SIMD implementation based on any prime for which the block size is not a power of 2 will suffer from the disadvantage mentioned in Remark 8. For `avx2` implementations, the performance model proposed in [28] includes the cost of loading data, but does not consider the effect of Remark 8.

5.5 Other Primes

Several possible primes were considered in [15, 28] for different security levels. For about the 128-bit security level, as alternatives to $2^{130} - 5$, the primes $2^{116} - 3$, $2^{122} - 3$, and $2^{150} - 3$ were considered in [15]. On the other hand, in [28] all primes of the form $2^m - \delta$, with $100 \leq m \leq 400$ and the smallest admissible δ were considered as part of an automated search. For about the 128-bit security level, individual primes other than $2^{130} - 5$ that were mentioned in the performance analysis in [28, Table 2] are $2^{116} - 3$, and $2^{150} - 3$.

For about the 128-bit security level, apart from $2^{127} - 1$ and $2^{130} - 5$, we considered the following other primes: $2^{116} - 3$, $2^{122} - 3$, $2^{137} - 13$, $2^{140} - 27$, $2^{141} - 9$, and $2^{150} - 3$. Based on our analysis of implementation using 4-way SIMD we provide the reasons for not implementing hash functions for these primes.

1. For the primes $2^{137} - 13$, $2^{140} - 27$, and $2^{141} - 9$ the elements of the corresponding fields can be represented using 5 limbs. These primes would support 17-byte blocks instead of the 16-byte blocks supported by the prime $2^{130} - 5$, and also provide a few extra bits of security. However, with these three primes there will not be sufficiently many free bits left after multiplication to support delayed reduction. So the overall hash computation will be slower than that for $2^{130} - 5$. Further, while using 17-byte blocks instead of 16-byte blocks will reduce the number of blocks, this advantage will be more than offset due to the problem of reading the input (see Remark 8).
2. One may use an even greater prime such as $2^{150} - 3$; the problem in this case is that a 5-limb representation will cause overflow during the multiplication procedure, while a 6-limb representation will be slower than $2^{130} - 5$ (though it will support delayed reduction). The block size will be 18 bytes, which will reduce the number of blocks, but also run into the problem of inefficiency of reading the input (see Remark 8).
3. The prime $2^{122} - 3$ is smaller than $2^{127} - 1$. With $2^{122} - 3$, a 4-limb representation will lead to an overflow in the first limb of the product, and a 5-limb representation gains nothing over $2^{127} - 1$. The block size will be 15 bytes, and will share the reading inefficiency problem of $2^{127} - 1$ (see Remark 8).

The case of the prime $2^{116} - 3$**.** The hash function Poly1163 based on the prime $2^{116} - 3$ was put forward as an alternative to Poly1305 in [15]. The argument provided in [15] in favour of Poly1163 is that it is faster than Poly1305, and at the same time provides security comparable to Poly1305. In the terminology of [8], the hash function Poly1163 would correspond to polyHash1163. We compare polyHash1163 with polyHash1271 and polyHash1305 in details.

We first consider the security argument. The argument that Poly1163 provides comparable security to Poly1305 crucially depends on the fact that Poly1305 uses clamping to reduce a 128-bit key to a 106-bit key. This argument, however, does not provide the complete picture. The key clamping procedure of Poly1305 is not a security requirement. Key clamping was introduced in [3] only to support high speed floating point computation. One can build a hash function based on the prime $2^{130} - 5$ with 128-bit keys. The function polyHash1305 that we introduced is in fact based on the prime $2^{130} - 5$ and uses 128-bit keys. In particular, we do not propose any key clamping for polyHash1305. The key and tag for Poly1163 are both 14 bytes long, while the key and tag for polyHash1305 are 16 bytes long (see Table 1). Applying the AXU bound for polyHash (see Theorem 3), the loss in security of Poly1163 in comparison to polyHash1305 is 18 bits. Similarly, the loss in security of Poly1163 in comparison to polyHash1271 is 17 bits (using Table 1 and Theorem 3).

Next we consider the speed argument for different implementations. The issue which is common for all the implementations is that the block sizes for polyHash1163, polyHash1271, and polyHash1305 are 14-byte, 15-byte, and 16-byte respectively. Correspondingly, for a fixed message length, the number of blocks will be the smallest for polyHash1305, largest for polyHash1163, with polyHash1271 in the middle.

64-bit `maax` implementation. For polyHash1163, polyHash1271, and polyHash1305 the limb representations are 2-limb, 2-limb, and 3-limb respectively. So individual multiplications for polyHash1163 and polyHash1271 will have the same speed, which will be faster than an individual multiplication for polyHash1305. Overall efficiency depends on the relative trade-off between the decrease in the time for an individual multiplication versus the increase in the number of blocks. This trade-off favours polyHash1271 over polyHash1305 as reported in [8]. Whether this trade-off also favours polyHash1163 over polyHash1305 is not known, since [15] did not perform a comparison of a `maax` implementation of Poly1163 with the `maax` implementation of polyHash1305 in [8].

`avx2` implementation. The limb representations for polyHash1163, polyHash1271, and polyHash1305 are 4-limb, 5-limb, and 5-limb respectively. So individual multiplications for polyHash1271 and polyHash1305 will have the same speed which will be slower than an individual multiplication for polyHash1163. Overall efficiency again depends on the relative trade-off between the decrease in the time for an individual multiplication versus the increase in the number of blocks. Further, there is the issue of inefficiency in loading registers from memory when the block size is not a power of 2 (see Remark 8), which is a disadvantage for both polyHash1163, polyHash1271 in comparison to polyHash1305. Our `avx2` implementations show that the overall trade-off favours polyHash1305 over polyHash1271. Timing results reported in [29] for `avx2` implementations of Poly1163 and Poly1305 show that the trade-off is in favour of Poly1163. However, the speed reported for `avx2` implementation of Poly1163 in [29] on the Intel Sapphire Rapids processor is about 0.45 cycles/byte for 22456 bytes, while for a similar number of bytes, the speed of our `avx2` implementation of polyHash1305 on the Intel Tiger Lake processor (an earlier generation processor, see Section 6 for a comparison of Sapphire Rapids and Tiger Lake processors) is between 0.43–0.45 cycles/byte (see Table 10 in Appendix B). So at present there is no definitive evidence that Poly1163 will necessarily be faster than polyHash1305.

`avx512` implementation. The comparison between polyHash1163, polyHash1271, and polyHash1305 is the same as that for `avx2` implementations.

`avx512_ifma` implementation. With `avx512_ifma`, 4 simultaneous full 52-bit multiplications can be performed (using two instructions, rather than a single instruction). All three of

polyHash1163, polyHash1271, and polyHash1305 will have 3-limb representations. So the speed of individual multiplication will be the same in all the three cases. Since polyHash1305 has the least number of blocks, and has the block size to be a power of 2 (see Remark 8), it follows that polyHash1305 will be fastest of the three hash functions. Between polyHash1163, polyHash1271, since polyHash1271 will have less number of blocks, it will be faster than polyHash1163.

The overall summary of the efficiency comparison is the following. For `maax` implementation, polyHash1271 will be faster than both polyHash1163 and polyHash1305, while the comparative speed of polyHash1163 and polyHash1305 is not clear. For `avx2` and `avx512` implementations, polyHash1305 will be faster than polyHash1271, but the comparative speed of polyHash1163 and polyHash1305 is not clear. For `avx512_ifma` implementations, polyHash1305 will be the fastest, polyHash1163 will be the slowest, with polyHash1271 somewhere in the middle. Considering the efficiency comparison together with the 17-bit and 18-bit loss of security with respect to polyHash1271 and polyHash1305 respectively, in our opinion, the security/efficiency trade-off does not favour polyHash1163 over either polyHash1271 or polyHash1305.

6 Implementation and Timing Results

We have made `avx2` implementations of polyHash and 4-decBRWHash for both the primes $2^{130}-5$ and $2^{127}-1$ in hand optimised assembly language. For the polyHash implementations we considered the delayed reduction parameter g to take the values $1, 2, 3$ and 4. Using higher values of g leads to a loss of speed. For the 4-decBRWHash implementations we considered the parameter t in Algorithm 1 to take the values $t = 2, 3, 4$ and 5. Higher values of t would lead to a very large code size (since $2^t - 1$ fragments of straight line code are required to implement Step 19, as r can take $2^t - 1$ values).

Recall that if the key clamping mandated by Poly1305 is implemented, then polyHash1305 is the same as Poly1305 for messages whose lengths are multiples of 8. Since Poly1305 is an extensively used hash function, different implementations of the hash function targeted at various platforms are publicly available; see for example [7, 20, 16, 1, 9, 8, 15, 28, 23, 30, 22, 6]. As far as we are aware, our implementations of polyHash1305 provide the first hand optimised `avx2` assembly language implementations of Poly1305 with different values of the delayed reduction parameter g.

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

`https://github.com/kn-cs/dec-BRWHash`
`https://github.com/kn-cs/vec-polyHash`

All the implementations were verfied with test vectors of various sizes generated by separate SAGE implementations of the all hash functions.

We recorded an extensive set of timings for all the hash functions. The timing measurements were taken on a single core of 11th Gen Intel Core i7-1185G7 @ 3.00GHz × 4 Tiger Lake processor using 31.1 GiB memory. During the experiments, turbo boost and hyperthreading options were turned off. The OS was Ubuntu 20.04.3 LTS and the code was compiled using gcc version 9.4.0. The following flags were used during compilation.

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

We counted CPU cycles using the microlibrary "libcpucycles" (see `https://cpucycles.cr.yp.to/`) through the `amd64-pmc` counter (see `https://cpucycles.cr.yp.to/counters.html`) which

requires a 64-bit Intel/AMD platform and Linux perf_event interface. The `amd64-pmc` counter accesses a cycle counter through RDPMC and requires `/proc/sys/kernel/perf_event_paranoid` to be at most 2 for user-level RDPMC access. This counter runs at clock frequency of the CPU core.

The timing results show that the `avx2` implementation of polyHash1271 is slower than the `avx2` implementation of polyHash1305, and the `avx2` implementation of 4-decBRWHash1271 is slower than the `avx2` implementation of 4-decBRWHash1305. This confirms the theoretically predicted slowdown discussed in Section 5.4. In view of this slowdown, we do not present the timing results for $2^{127} - 1$.

For purpose of comparison, we provide timing results for polyHash1305 and BRWHash1305 from [8, 25], which are sequential `maax` implementations. For BRWHash1305, there are no other previous implementations. For `avx2` implementations of polyHash1305, we compare to the measurements reported in [29] of the implementation in the recent work [28], and also to the measurements of the performance of Poly1305 code in OpenSSL [30] and HACL* [37] reported in [29].

We provide the detailed timing results in Appendix B. In the case of polyHash1305, timing results are provided for different values of the group reduction parameter g – for `maax` implementation $g \in \{1, 4, 8, 16, 32\}$, and for `avx2` implementation $g \in \{1, 2, 3, 4\}$. For BRWHash and 4-decBRWHash1305, timing results are provided for different values of the look-ahead parameter t, with $t \in \{2, 3, 4, 5\}$. Further, for each case two sets of timings are provided, one for the situation where the key powers are pre-computed, and the other for the situation where the key powers are computed on the fly. Based on the timing results in Appendix B, here we provide some plots which illuminate the various types of comparisons.

Comparison between `maax` and `avx2` implementations of polyHash1305. This comparison is shown in Figure 2. From the figure we note that for the case of pre-computed key powers, `avx2` implementation is faster than the `maax` implementations for 16 or more blocks. For the case of key powers computed on the fly, the two implementations are within 0.02 cycles/byte for 16 to 19 blocks, and the `avx2` implementation is faster for 20 or more blocks.

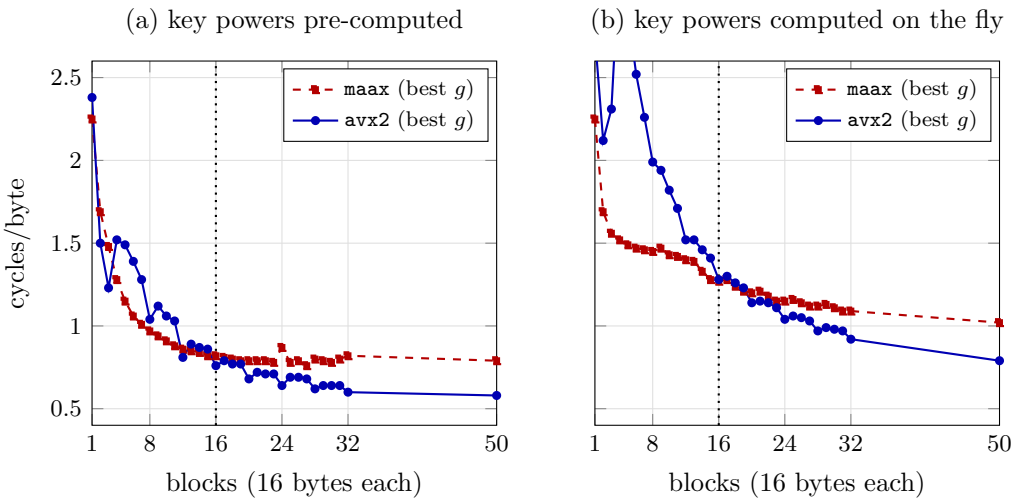

Figure 2: Comparison of `maax` and `avx2` implementations of polyHash1305. The dotted line denotes 16 blocks.

Comparison between avx2 implementation of polyHash1305 and previous avx2 implementations of Poly1305. Implementations of Poly1305 using avx2, avx512, and avx512_ifma instruction sets were reported in [28]. Measurements of the number of cycles required were obtained on the Intel Sapphire Rapids processor and are available from [29]. Measurements of the number of cycles required by HACL* [37] and OpenSSL [30] implementations of Poly1305 on the Intel Sapphire Rapids processor are also available from [29]. For comparison with our avx2 implementations, we focus on the avx2 measurements reported at [29].

From the measurements in [29] of each of avx2 implementations of SPHGen [28], HACL* [37] and OpenSSL [30], for every value of the number of bytes, we took the minimum value of cycles/byte. We converted the number of bytes to the number of blocks by dividing with 16. The plots of the obtained cycles/byte against the number of blocks are shown in Figure 3. For the avx2 implementation of polyHash1305, from the tables in Appendix B, for every value of the number of blocks, we took the minimum value of cycles/byte. The corresponding plot of the cycles/byte against the number of blocks is shown in Figure 3.

Our measurements were made on the Tiger Lake processor, while the measurements reported in [29] were made on the later generation Sapphire Rapids processor. Before commenting on the nature of the comparison in Figure 3, we would like to clarify the issue of comparing measurements on two different processors. It is important to note that both our measurements and those in [29] are of the *number of cycles* and not of the actual time. For the comparison to be meaningful, it is required to consider the latency and throughput of the various individual instructions. The avx2 version (working on ymm registers) of each of the instructions vpmuludq, vpaddq, vpand, vpxor, vpsllq, vpsrlq, has the same latency and throughput, and can be executed on the same number of ports on both Tiger Lake and Sapphire Rapids processors. The avx2 version of vmovdqa has the same latency on both Tiger Lake and Sapphire Rapids processors for both register-to-register, and memory-to-register moves; however, the throughput of vmovdqa for both register-to-register, and memory-to-register moves is better in Sapphire Rapids than in Tiger Lake. So our avx2 implementation of polyHash1305 will require the same or less number of cycles on Sapphire Rapids than on Tiger Lake. In view of this, it is meaningful to compare the measurements of our avx2 implementations of polyHash1305 on Tiger Lake with the measurements of Poly1305 made on Sapphire Rapids.

Considering Figure 3, we see that our avx2 implementation of polyHash1305 outperforms the avx2 implementations of Poly1305 in SPHGen [28], HACL* [37] and OpenSSL [30]. The lowest value of cycles/byte among the 300 data points for the avx2 implementation of Poly1305 as part of SPHGen [28] which was reported in [29] is 0.497894 and was obtained for 24704 bytes, i.e. 1544 blocks. In comparison, our avx2 implementation of polyHash1305 runs at about 0.43 cycles/byte for 1500 block messages, which is about a 14% speed-up over the SPHGen [28] implementation of Poly1305.

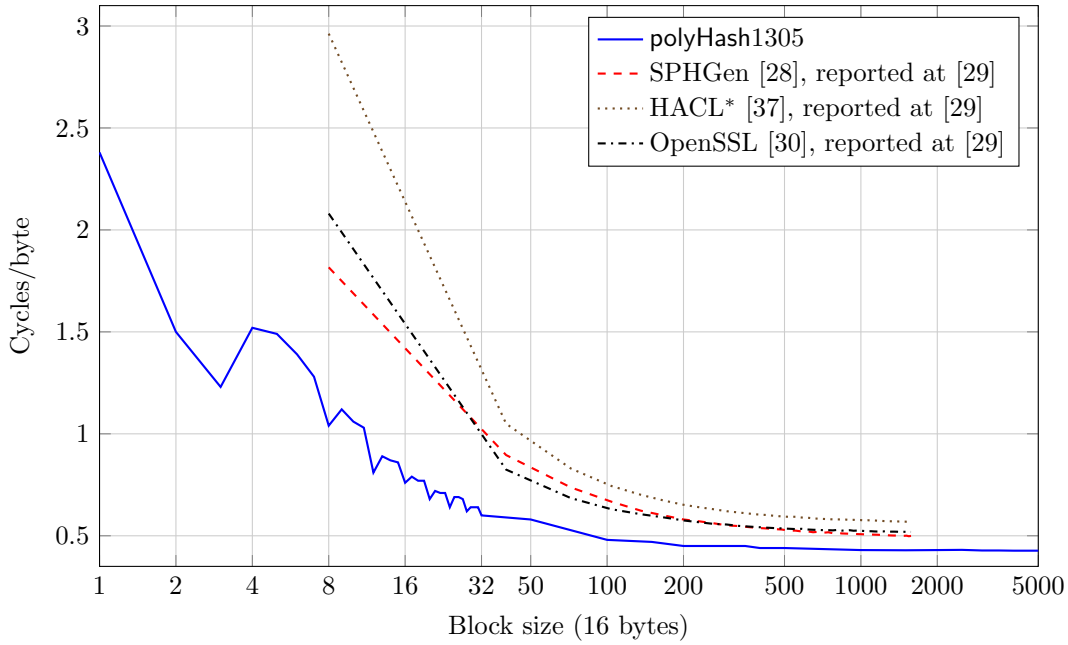

Figure 3: Comparison of the `avx2` implementation of polyHash1305 with the `avx2` implementations of Poly1305 in SPHGen [28], HACL* [37], and OpenSSL [30]. The measurements for polyHash1305 are from the tables in Appendix B, while the measurements for the three implementations of Poly1305 are from [29].

Comparison between the `maax` implementation of BRWHash1305 and the `avx2` implementation of 4-decBRWHash1305. We chose $t = 5$ for both the hash functions. Figure 4 shows the comparison. For both the cases of pre-computed and on the fly computation of key powers, from 100 blocks onwards 4-decBRWHash1305 is clearly faster. At around 50 blocks, for the case of pre-computed key powers, BRWHash1305 is about 3% faster, while for the case of on the fly computation of key powers, both are level.

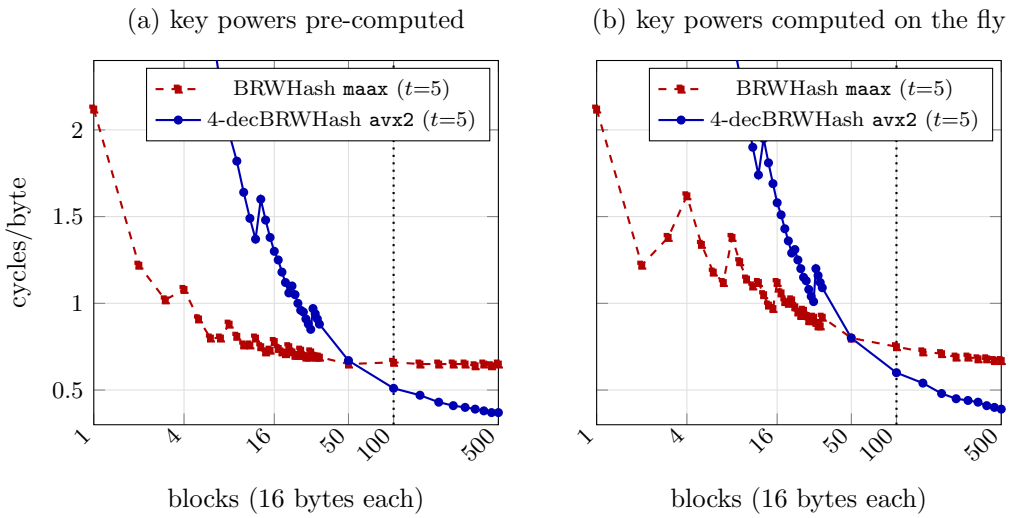

Figure 4: Comparison between the maax implementation of BRWHash1305 and the avx2 implementation of 4-decBRWHash1305 The dotted line denotes 100 blocks.

Comparison between maax implementation of BRWHash1305 and avx2 implementation of polyHash1305. For BRWHash1305, we chose $t = 5$ and for the avx2 implementation of polyHash1305 we chose $g = 4$. The comparison is shown in Figure 5. The avx2 implementation of polyHash1305 is eventually faster. For the case of pre-computed key powers, polyHash1305 becomes faster somewhere between 32 and 50 blocks, while for the case of key powers computed on the fly, polyHash1305 becomes faster somewhere between 50 and 100 blocks.

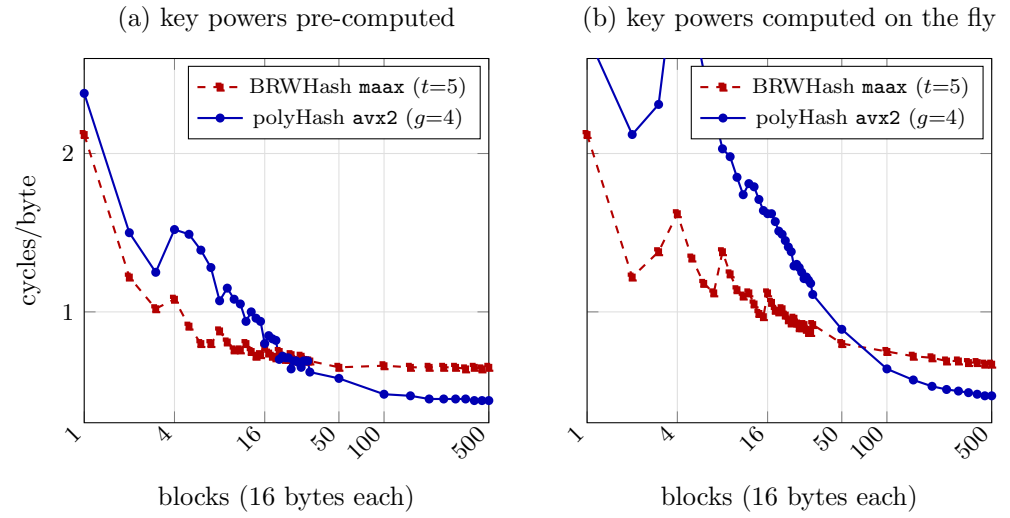

Figure 5: Comparison between maax implementation of BRWHash1305 and avx2 implementation of polyHash1305.

Comparison among choices of g for polyHash1305, and choices of t for 4-decBRWHash1305, both for avx2 implementations. The comparison is shown in Figure 6. For polyHash1305 the situation is the following: with pre-computed key powers, $g = 4$ is always faster than the other

options for 50 blocks or more, and for on the fly computation of key powers, $g = 4$ is slower than $g = 2$ or $g = 3$ up to 100 blocks, tied at 150–200 blocks, and is faster from 250 or more blocks. For 4-decBRWHash1305, the situation is the following: the other values of t provide speeds which are at most 5.4% slower than $t = 5$; the values of $t = 4$ and $t = 5$ are almost the same for messages with more than 100 blocks.

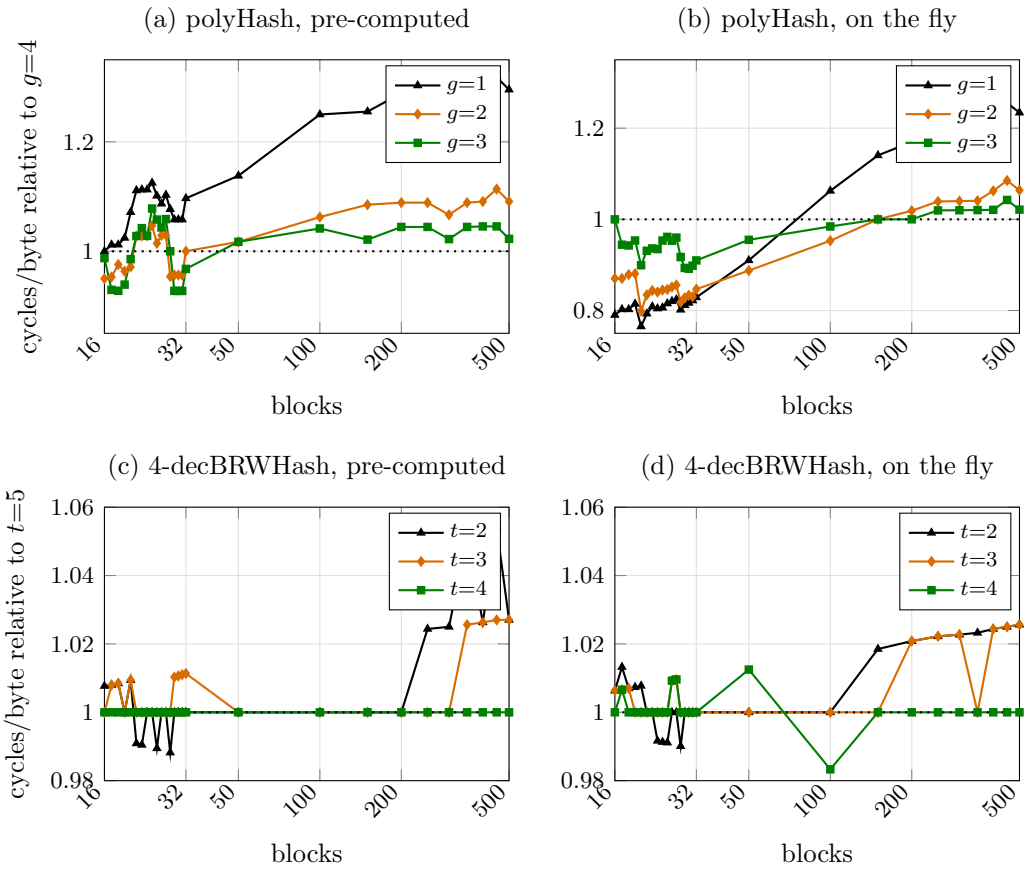

Figure 6: For avx2 implementations, comparison among the choices of g for polyHash1305, and the choices of t for 4-decBRWHash1305. For polyHash1305, the curves for the different values of g are relative to the reference value $g = 4$. For 4-decBRWHash1305, the curves for the different values of t are relative to the reference value $t = 5$. In both cases, a value above 1 means that the corresponding value is slower than the reference.

Comparison between the timings with pre-computed key powers and on the fly computation of key powers. We consider the avx2 implementations. The comparison is shown in Figure 7. The y-axis represents $100\,(t_{\text{fly}} - t_{\text{pre}})/t_{\text{pre}}$. The overhead is about 5%–7% at 500 blocks, 4% or less from 1000 blocks, and about 0.2%–0.3% at 2^{15} blocks.

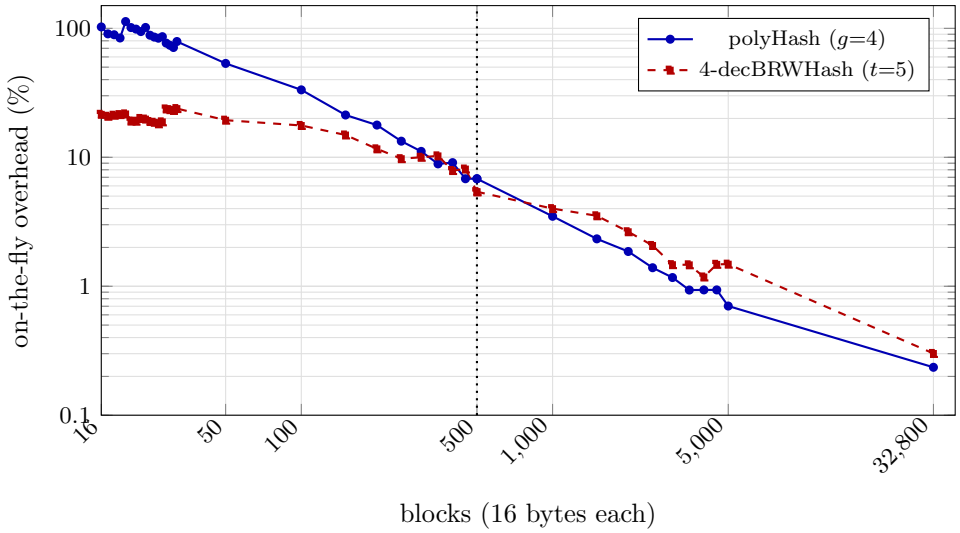

Figure 7: Comparison between the timings with pre-computed key powers and on the fly computation of key powers. The dotted line represents 500 blocks.

Speed-up of 4-decBRWHash1305 **over** polyHash1305 **for** `avx2` **implementations.** This is the main point of comparison and the central focus of the present paper. We choose $g = 4$ for the `avx2` implementation of polyHash1305, and $t = 5$ for 4-decBRWHash1305. The comparison is shown in Figure 8. The speed-up percentage is measured as $(t_1 - t_2)/t_1$, where t_1 is the cycles/byte for polyHash1305, and t_2 is the cycles/byte for 4-decBRWHash1305. The summary is as follows.

Pre-computed key powers. For 150 or more blocks (equivalently, 2400 or more bytes), the timing results show that 4-decBRWHash1305 is faster than polyHash1305. For messages having 200 to 500 blocks (equivalently, 3200 to 8000 bytes), the speed-up is in the range of 4% to 16%; for messages having 1000 to 5000 blocks (equivalently, 16 KB to 80 KB), the speed-up is in the range of 18% to 21%; and for messages with 32768 blocks (equivalently, 2^{19} bytes), the speed-up is about 22%.

Key powers computed on the fly. For 16 or more blocks (equivalently, 256 or more bytes) 4-decBRWHash1305 is faster than polyHash1305. For messages having 50 to 500 blocks (equivalently, 800 to 8000 bytes), the speed-up is mostly in the range of 10% to 17%; for messages having 1000 to 5000 blocks (equivalently, 16 KB to 80 KB), the speed-up is in the range of 18% to 20%; and for messages with 32768 blocks (equivalently, 2^{19} bytes), the speed-up is about 22%.

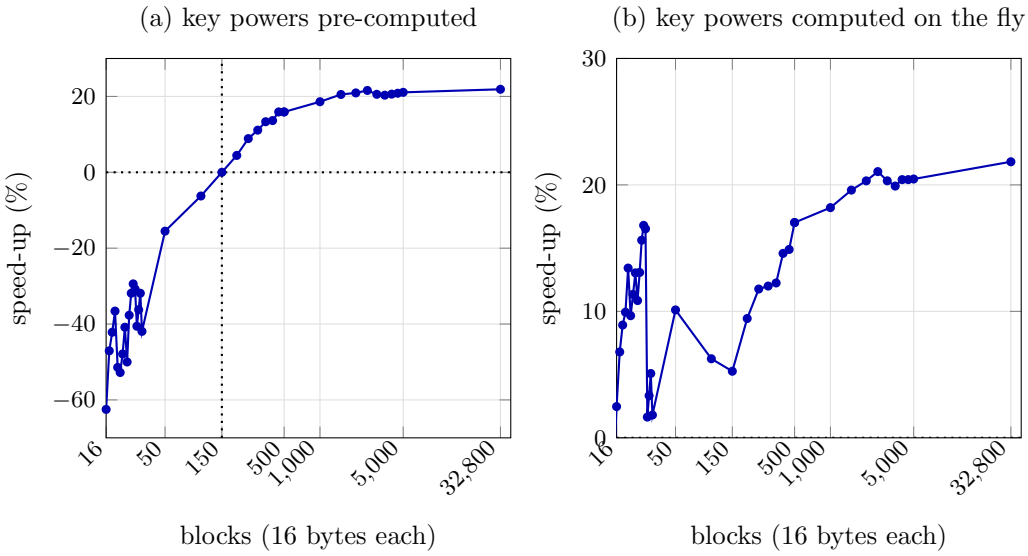

Figure 8: Speed-up of 4-decBRWHash1305 over polyHash1305 for avx2 implementations.

To summarise, for avx2 implementations with key powers computed on-the-fly, timing results show that 4-decBRWHash1305 is faster than polyHash1305 for messages of lengths 256 bytes or more, achieves a speed-up of about 16% for messages which are a few kilobytes long, and the speed-up improves to about 22% for messages which are a few megabytes long. In typical file systems [18], text files are usually a few kilobytes long while media files such as pictures, audio and video files, are about a few megabytes long. For such files, i.e. both text files and media files, hashing using 4-decBRWHash1305 will be substantially faster than hashing using polyHash1305.

7 Conclusion

We proposed a new AXU hash function based on BRW polynomials. The hash function is a generalisation of the hash function based on BRW polynomials, with the generalisation permitting efficient SIMD implementations. For the prime $2^{130} - 5$, SIMD implementations of the new hash function using avx2 instructions on modern Intel processors show that the new hash function is faster than the well known Poly1305 hash function for messages longer than a few hundred bytes achieving a speed-up of about 16% for message lengths in kilobyte range to about 22% for message lengths in the megabyte range. This makes the new hash function an attractive alternative to Poly1305 for use in authentication and authenticated encryption systems for general files found in typical file systems.

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. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 965–982. IEEE, 2020. 25

[2] Daniel J. Bernstein. Floating-point arithmetic and message authentication, 2004. `https://cr.yp.to/papers.html#hash127`. 3

[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. 1, 3, 6, 24

[4] Daniel J. Bernstein. Polynomial evaluation and message authentication, 2007. `http://cr.yp.to/papers.html#pema`. 1, 2, 3, 4

[5] Daniel J. Bernstein. ChaCha, a variant of Salsa20. `https://cr.yp.to/papers.html#chacha`, 2008. 1

[6] Daniel J. Bernstein and Kaushik Nath. lib1305: Poly1305 one-time authenticator microlibrary. `https://lib1305.cr.yp.to/index.html`. Accessed: 9 November 2026. 25

[7] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012. 7, 18, 25

[8] Sreyosi Bhattacharyya, Kaushik Nath, and Palash Sarkar. Polynomial hashing over prime order fields. *Advances in Mathematics of Communications*, 19(1):337–378, 2025. 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 17, 18, 21, 22, 23, 24, 25, 26, 35

[9] Sreyosi Bhattacharyya and Palash Sarkar. Improved SIMD implementation of Poly1305. *IET Inf. Secur.*, 14(5):521–530, 2020. 19, 25

[10] Jürgen Bierbrauer, Thomas Johansson, Gregory Kabatianskii, and Ben J. M. Smeets. On families of hash functions via geometric codes and concatenation. In Stinson [35], pages 331–342. 3

[11] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. 3

[12] 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. 3, 17

[13] 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. 2, 3

[14] 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. 3

[15] Jean Paul Degabriele, Jan Gilcher, Jérôme Govinden, and Kenneth G. Paterson. SoK: Efficient design and implementation of polynomial hash functions over prime fields. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 3128–3146. IEEE, 2024. 3, 12, 18, 23, 24, 25

[16] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 463–482. IEEE Computer Society, 2017. 25

[17] Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, 2:65–72, 1993. 3

[18] Jesse David Dinneen and Ba Xuan Nguyen. How big are peoples' computer files? file size distributions among user-managed collections. In *Information: Equity, Diversity, Inclusion, Justice, and Relevance - Proceedings of the 84th ASIS&T Annual Meeting, ASIST 2021, Salt Lake City, UT, USA, October 30 - November 2, 2021*, volume 58 of *Proc. Assoc. Inf. Sci. Technol.*, pages 425–429. Wiley, 2021. 3, 32

[19] Sebati Ghosh and Palash Sarkar. Evaluating Bernstein-Rabin-Winograd polynomials. *Designs, Codes, and Cryptography*, 87(2-3):527–546, 2019. 2, 3, 5

[20] 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. 1, 7, 12, 13, 14, 15, 17, 18, 25

[21] 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. 3

[22] Linux Kernel Developers. Linux Kernel: x86 cryptographic implementations. `https://github.com/torvalds/linux/tree/master/lib/crypto/x86`. Source code repository. Accessed: 9 November 2026. 25

[23] Andrew Moon. poly1305-opt: Optimized implementations of Poly1305. `https://github.com/floodyberry/poly1305-opt`. GitHub source code repository. Accessed: 9 November 2026. 25

[24] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)Mersenne prime order fields. *Advances in Mathematics of Communications*, 16(2):303–348, 2022. 13

[25] Kaushik Nath and Palash Sarkar. An update to "Polynomial Hashing over Prime Order Fields". Cryptology ePrint Archive, Paper 2025/1224, 2025. 2, 3, 5, 8, 13, 17, 22, 26, 36

[26] 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. 13

[27] 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. 13

[28] Tommaso Pegolotti, Jan Gilcher, Kenneth G. Paterson, and Markus Püschel. SPHGen: A program generator for fast polynomial hash functions. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2026(1):161–184, 2026. 3, 12, 14, 18, 23, 25, 26, 27, 28

[29] Tommaso Pegolotti, Jan Gilcher, Kenneth G. Paterson, and Markus Püschel. SPHGen and Poly1305 benchmark results. `https://github.com/tpegolotti/SPHGen/tree/main/paper_results`, 2026. SPHGen project, GitHub repository. 24, 26, 27, 28

[30] Andy Polyakov. Poly1305 for x86_64. `https://android.googlesource.com/kernel/common/+/d15bb1f6dabe1/arch/x86/crypto/poly1305-x86_64-cryptogams.pl`. Perl-based assembly generator for the Linux kernel. Originally written for the OpenSSL project. Accessed: 9 November 2026. 25, 26, 27, 28

[31] Michael O. Rabin and Shmuel Winograd. Fast evaluation of polynomials by rational preparation. *Communications on Pure and Applied Mathematics*, 25:433–458, 1972. 3

[32] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4759, 2009. 1, 4

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

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

[35] 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. 33, 35

[36] Richard Taylor. An integrity check value algorithm for stream ciphers. In Stinson [35], pages 40–48. 3

[37] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1789–1806. ACM, 2017. 26, 27, 28

A Algorithm for Computing BRW Polynomials

Algorithm 1 describes the algorithm given in [8] to compute $\mathsf{BRW}(\tau; M_1, \ldots, M_l)$, where l is a non-negative integer. The call $\mathsf{unreducedBRW}$ in the algorithm performs the following computation. There are two calls to $\mathsf{unreducedBRW}$ in Algorithm 1, in Steps 10 and 19. The call in Step 10 is on exactly $2^t - 1$ blocks, while the call in Step 19 is on at most $2^t - 1$ blocks. Since t is a fixed value, both of these calls to $\mathsf{unreducedBRW}$ are implemented using straight line codes.

- $\mathsf{unreducedBRW}(\tau;) = 0$;
- $\mathsf{unreducedBRW}(\tau; M_1) = M_1$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2) = \mathsf{unreducedMult}(M_1, \tau) + M_2$;

- $\mathsf{unreducedBRW}(\tau; M_1, M_2, M_3) = \mathsf{unreducedMult}((\tau + M_1), (\tau^2 + M_2)) + M_3$;
- $\mathsf{unreducedBRW}(\tau; M_1, M_2, \ldots, M_k)$
  $= \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{unreducedBRW}(\tau; M_1, \ldots, M_{k-1})), (\tau^k + M_k))$,
  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 $k < l < 2k$.

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

function $\mathsf{ComputeBRW}(\tau, M_1, \ldots, M_l)$
  $\mathsf{keyPow}[0] \leftarrow \tau$
  if $l > 2$ **then**
    for $j \leftarrow 1$ to $\lfloor \lg l \rfloor$ **do**
      $\mathsf{keyPow}[j] \leftarrow \mathsf{keyPow}[j-1]^2$
    end for
  end if
  $\mathsf{top} \leftarrow -1$
  for $i \leftarrow 1$ to $\lfloor l/2^t \rfloor$ **do**
    $\mathsf{tmp} \leftarrow \mathsf{unreducedBRW}(\tau; M_{2^t(i-1)+1}, \ldots, M_{2^t \cdot i-1})$;
    $k \leftarrow \mathsf{ntz}(i)$
    for $j \leftarrow 0$ to $k-1$ **do**
      $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{stack}[\mathsf{top}]$; $\mathsf{top} \leftarrow \mathsf{top} - 1$
    end for
    $\mathsf{tmp} \leftarrow \mathsf{unreducedMult}(\mathsf{reduce}(\mathsf{tmp}), M_{2^t \cdot i} + \mathsf{keyPow}[t+k])$
    $\mathsf{top} \leftarrow \mathsf{top} + 1$; $\mathsf{stack}[\mathsf{top}] \leftarrow \mathsf{tmp}$
  end for;
  $r \leftarrow l \bmod 2^t$;
  $\mathsf{tmp} \leftarrow \mathsf{unreducedBRW}(\tau; M_{l-r+1}, \ldots, M_l)$;
  $i \leftarrow \mathsf{wt}(\lfloor l/2^t \rfloor)$
  for $j \leftarrow 0$ to $i-1$ **do**
    $\mathsf{tmp} \leftarrow \mathsf{tmp} + \mathsf{stack}[\mathsf{top}]$; $\mathsf{top} \leftarrow \mathsf{top} - 1$
  end for
  return $\mathsf{reduce}(\mathsf{tmp})$;
end function.

B Detailed Timing Results

The timing results for the avx2 implementation of polyHash1305 and 4-decBRWHash1305 are shown in Tables 5 to 8 for messages having 1 to 32 blocks, and in Table 9 for messages having 50 to 500 blocks. For comparison, in these tables we also present timing results from [25] for the maax implementation of polyHash1305 and BRWHash1305 obtained on the above platform. In Table 10, we present the timing results for the avx2 implementation of polyHash1305 with $g = 4$ and 4-decBRWHash1305 with $t = 5$ for messages having 1000 to 5000 blocks.

For all the tables, the figures in the cells of the tables denote the number of cycles per byte

		# msg blks							
		1	2	3	4	5	6	7	8
polyHash1305 (maax)	$g = 1$	2.31	1.69	1.56	1.52	1.49	1.47	1.46	1.45
	$g = 4$	2.38	1.84	1.48	1.38	1.40	1.23	1.16	1.15
		2.38	1.84	1.88	2.05	2.14	1.86	1.71	1.63
	$g = 8$	2.38	1.84	1.48	1.28	1.15	1.06	1.01	1.03
		2.38	1.84	1.88	1.94	1.94	1.95	1.98	2.09
	$g = 16$	2.25	1.84	1.48	1.28	1.15	1.06	1.01	0.97
		2.25	1.84	1.88	1.94	1.94	1.95	1.98	2.03
	$g = 32$	2.31	1.84	1.48	1.28	1.15	1.06	1.01	0.97
		2.31	1.84	1.88	1.92	1.94	1.98	1.99	2.02
polyHash1305 (avx2)	$g = 1$	2.38	1.50	1.23	1.55	1.56	1.41	1.30	1.04
		2.75	2.12	2.33	3.27	2.94	2.55	2.29	1.99
	$g = 2$	2.38	1.50	1.23	1.55	1.56	1.42	1.30	1.05
		2.81	2.12	2.31	3.28	2.95	2.55	2.29	2.02
	$g = 3$	2.44	1.50	1.23	1.55	1.52	1.41	1.30	1.06
		2.75	2.12	2.31	3.25	2.92	2.53	2.27	2.03
	$g = 4$	2.38	1.50	1.25	1.52	1.49	1.39	1.28	1.07
		2.75	2.12	2.31	3.25	2.90	2.52	2.26	2.03
BRWHash1305 (maax)	$t = 2$	2.06	1.22	0.96	1.16	0.93	0.81	0.81	0.91
		2.06	1.22	1.38	1.73	1.40	1.21	1.14	1.40
	$t = 3$	2.12	1.19	0.98	1.09	0.90	0.80	0.80	0.89
		2.12	1.19	1.35	1.62	1.35	1.17	1.11	1.38
	$t = 4$	2.19	1.22	0.98	1.08	0.90	0.81	0.80	0.89
		2.19	1.22	1.35	1.62	1.34	1.18	1.11	1.38
	$t = 5$	2.12	1.22	1.02	1.08	0.91	0.80	0.80	0.88
		2.12	1.22	1.38	1.62	1.34	1.18	1.12	1.38
4-decBRWHash1305 (avx2)	$t = 2$	11.19	5.62	3.73	2.80	3.17	2.65	2.27	1.98
		13.31	6.66	4.44	3.33	4.08	3.40	2.91	2.55
	$t = 3$	11.25	5.62	3.73	2.81	3.16	2.64	2.26	1.98
		13.31	6.66	4.44	3.33	4.06	3.39	2.90	2.55
	$t = 4$	11.25	5.62	3.75	2.80	3.16	2.64	2.26	1.98
		13.31	6.66	4.44	3.33	4.08	3.40	2.91	2.55
	$t = 5$	11.25	5.62	3.73	2.81	3.16	2.64	2.26	1.98
		13.31	6.66	4.44	3.33	4.08	3.40	2.91	2.55

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

required to compute the digest by the corresponding hash function with the stated value of the parameter. Each cell has two figures, the figure on the top denotes the number of cycles per byte when the required key powers are pre-computed and stored (i.e. the time for generating the key powers are not included in the time for hashing), while the number on the bottom denotes the number of cycles per byte when the required key powers are computed on the fly (i.e. the time for generating the key powers are included in the time for hashing). Recall from Table 1 that the block size n is 128 bits for the prime $2^{130} - 5$, and so the number of blocks mentioned in the tables can be converted to number of bytes by multiplying with 16.

		# msg blks							
		9	10	11	12	13	14	15	16
polyHash1305 (maax)	$g = 1$	1.47	1.43	1.45	1.47	1.48	1.47	1.47	1.46
	$g = 4$	1.19	1.12	1.07	1.07	1.11	1.06	1.03	1.03
		1.60	1.49	1.42	1.40	1.39	1.33	1.28	1.27
	$g = 8$	1.09	1.02	0.98	0.96	0.94	0.91	0.90	0.92
		2.11	1.94	1.83	1.73	1.65	1.57	1.51	1.50
	$g = 16$	0.94	0.91	0.88	0.87	0.85	0.84	0.82	0.85
		2.00	1.98	2.01	2.04	2.06	2.08	2.10	2.15
	$g = 32$	0.94	0.91	0.88	0.86	0.85	0.84	0.82	0.82
		2.01	1.99	2.01	2.03	2.06	2.08	2.10	2.12
polyHash1305 (avx2)	$g = 1$	1.12	1.06	1.03	0.88	0.95	0.92	0.91	0.80
		1.94	1.82	1.71	1.52	1.52	1.46	1.41	1.28
	$g = 2$	1.12	1.07	1.03	0.81	0.89	0.87	0.86	0.76
		1.96	1.84	1.72	1.69	1.68	1.61	1.54	1.41
	$g = 3$	1.13	1.08	1.04	0.93	0.99	0.96	0.94	0.79
		1.97	1.84	1.73	1.78	1.77	1.69	1.62	1.62
	$g = 4$	1.15	1.08	1.05	0.94	1.00	0.96	0.94	0.80
		1.98	1.85	1.74	1.81	1.79	1.71	1.64	1.62
BRWHash1305 (maax)	$t = 2$	0.81	0.78	0.76	0.83	0.77	0.74	0.74	0.79
		1.25	1.16	1.12	1.16	1.08	1.03	1.01	1.15
	$t = 3$	0.81	0.74	0.74	0.80	0.76	0.72	0.71	0.77
		1.23	1.14	1.10	1.13	1.05	1.00	0.97	1.12
	$t = 4$	0.81	0.76	0.76	0.80	0.76	0.72	0.72	0.76
		1.24	1.14	1.10	1.12	1.05	0.99	0.97	1.11
	$t = 5$	0.81	0.76	0.76	0.80	0.75	0.72	0.73	0.78
		1.24	1.14	1.10	1.12	1.05	0.99	0.97	1.12
4-decBRWHash1305 (avx2)	$t = 2$	1.83	1.64	1.49	1.37	1.62	1.50	1.40	1.31
		2.32	2.09	1.90	1.74	1.96	1.82	1.70	1.59
	$t = 3$	1.83	1.64	1.49	1.37	1.61	1.49	1.39	1.30
		2.33	2.09	1.90	1.74	1.95	1.81	1.69	1.59
	$t = 4$	1.82	1.64	1.49	1.36	1.61	1.49	1.39	1.30
		2.33	2.09	1.90	1.74	1.95	1.81	1.69	1.58
	$t = 5$	1.82	1.64	1.49	1.37	1.60	1.48	1.38	1.30
		2.32	2.09	1.90	1.74	1.95	1.81	1.69	1.58

Table 6: 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 (maax)	$g = 1$	1.40	1.40	1.41	1.41	1.41	1.41	1.41	1.40
	$g = 4$	1.06	1.03	1.00	1.01	1.03	1.05	0.99	0.99
		1.28	1.24	1.21	1.20	1.21	1.18	1.15	1.15
	$g = 8$	0.95	0.92	0.90	0.90	0.93	0.87	0.86	0.88
		1.49	1.43	1.39	1.36	1.32	1.29	1.26	1.26
	$g = 16$	0.89	0.86	0.85	0.84	0.85	0.83	0.88	0.87
		2.16	2.06	1.99	1.92	1.88	1.84	1.76	1.71
	$g = 32$	0.81	0.80	0.79	0.79	0.79	0.79	0.78	0.88
		2.08	2.10	2.11	2.12	2.12	2.15	2.17	2.19
polyHash1305 (avx2)	$g = 1$	0.86	0.84	0.84	0.75	0.80	0.79	0.79	0.72
		1.30	1.26	1.23	1.14	1.15	1.14	1.11	1.04
	$g = 2$	0.81	0.81	0.79	0.68	0.74	0.73	0.73	0.67
		1.41	1.38	1.33	1.19	1.21	1.19	1.16	1.09
	$g = 3$	0.79	0.77	0.77	0.69	0.74	0.74	0.73	0.69
		1.53	1.48	1.44	1.34	1.35	1.32	1.29	1.23
	$g = 4$	0.85	0.83	0.82	0.70	0.72	0.71	0.71	0.64
		1.62	1.57	1.51	1.49	1.45	1.41	1.38	1.29
BRWHash1305 (maax)	$t = 2$	0.75	0.72	0.74	0.78	0.74	0.72	0.73	0.76
		1.08	1.05	1.03	1.06	1.01	0.98	0.97	0.99
	$t = 3$	0.73	0.70	0.70	0.74	0.71	0.70	0.70	0.73
		1.06	1.02	1.00	1.02	0.98	0.95	0.94	0.97
	$t = 4$	0.72	0.70	0.69	0.73	0.70	0.68	0.68	0.71
		1.04	1.00	0.99	1.01	0.96	0.94	0.93	0.95
	$t = 5$	0.74	0.72	0.71	0.75	0.72	0.70	0.70	0.73
		1.06	1.01	1.00	1.02	0.98	0.95	0.93	0.96
4-decBRWHash1305 (avx2)	$t = 2$	1.26	1.19	1.12	1.07	1.09	1.04	1.00	0.96
		1.53	1.44	1.37	1.30	1.31	1.25	1.19	1.14
	$t = 3$	1.26	1.19	1.12	1.07	1.10	1.05	1.00	0.96
		1.52	1.44	1.36	1.29	1.31	1.25	1.20	1.15
	$t = 4$	1.25	1.18	1.12	1.06	1.10	1.05	1.00	0.96
		1.52	1.43	1.36	1.29	1.31	1.25	1.20	1.15
	$t = 5$	1.25	1.18	1.12	1.06	1.10	1.05	1.00	0.96
		1.51	1.43	1.36	1.29	1.31	1.25	1.20	1.15

Table 7: 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 (maax)	$g = 1$	1.40	1.40	1.41	1.40	1.40	1.40	1.40	1.40
	$g = 4$	1.01	0.99	0.98	0.98	1.00	0.98	0.97	0.97
		1.16	1.14	1.12	1.12	1.13	1.11	1.09	1.09
	$g = 8$	0.90	0.88	0.87	0.87	0.86	0.85	0.85	0.86
		1.27	1.24	1.22	1.20	1.18	1.16	1.14	1.14
	$g = 16$	0.89	0.88	0.85	0.80	0.79	0.78	0.85	0.82
		1.68	1.63	1.60	1.57	1.54	1.52	1.51	1.47
	$g = 32$	0.78	0.79	0.76	0.88	0.81	0.79	0.80	0.92
		2.17	2.20	2.19	2.23	2.24	2.24	2.25	2.31
polyHash1305 (avx2)	$g = 1$	0.76	0.75	0.75	0.70	0.73	0.73	0.73	0.68
		1.06	1.05	1.03	0.97	0.99	0.98	0.97	0.92
	$g = 2$	0.70	0.71	0.70	0.62	0.66	0.66	0.66	0.62
		1.10	1.09	1.07	0.99	1.01	1.00	0.98	0.94
	$g = 3$	0.73	0.72	0.72	0.65	0.64	0.64	0.64	0.60
		1.25	1.22	1.20	1.11	1.09	1.07	1.06	1.01
	$g = 4$	0.69	0.69	0.68	0.65	0.69	0.69	0.69	0.62
		1.30	1.28	1.25	1.21	1.22	1.20	1.18	1.11
BRWHash1305 (maax)	$t = 2$	0.73	0.72	0.72	0.75	0.72	0.72	0.72	0.75
		0.96	0.94	0.93	0.96	0.92	0.91	0.90	0.97
	$t = 3$	0.71	0.69	0.69	0.73	0.69	0.69	0.68	0.71
		0.93	0.91	0.90	0.92	0.89	0.87	0.87	0.95
	$t = 4$	0.69	0.68	0.68	0.71	0.68	0.67	0.67	0.70
		0.92	0.89	0.88	0.90	0.87	0.85	0.85	0.93
	$t = 5$	0.70	0.69	0.69	0.72	0.70	0.69	0.69	0.69
		0.93	0.90	0.90	0.92	0.89	0.87	0.87	0.92
4-decBRWHash1305 (avx2)	$t = 2$	0.94	0.91	0.88	0.84	0.97	0.94	0.91	0.88
		1.12	1.08	1.04	1.00	1.20	1.16	1.12	1.09
	$t = 3$	0.95	0.91	0.88	0.85	0.98	0.95	0.92	0.89
		1.13	1.09	1.05	1.01	1.20	1.16	1.12	1.09
	$t = 4$	0.95	0.91	0.88	0.85	0.97	0.94	0.91	0.88
		1.13	1.09	1.05	1.01	1.20	1.16	1.12	1.09
	$t = 5$	0.95	0.91	0.88	0.85	0.97	0.94	0.91	0.88
		1.13	1.08	1.04	1.01	1.20	1.16	1.12	1.09

Table 8: Cycles/byte measurements for 25 to 32 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks									
		50	100	150	200	250	300	350	400	450	500
polyHash1305 (maax)	$g = 1$	1.38	1.37	1.37	1.36	1.36	1.36	1.36	1.36	1.36	1.36
	$g = 4$	0.95	0.94	0.93	0.93	0.93	0.93	0.92	0.92	0.92	0.92
		1.03	0.97	0.95	0.94	0.94	0.94	0.93	0.93	0.93	0.93
	$g = 8$	0.84	0.83	0.81	0.81	0.81	0.81	0.81	0.81	0.81	0.80
		1.02	0.91	0.87	0.85	0.85	0.84	0.83	0.83	0.82	0.82
	$g = 16$	0.79	0.77	0.76	0.75	0.76	0.76	0.75	0.74	0.75	0.74
		1.22	0.97	0.89	0.85	0.84	0.82	0.81	0.80	0.79	0.78
	$g = 32$	0.86	0.84	0.84	0.81	0.81	0.81	0.81	0.79	0.79	0.79
		1.79	1.29	1.14	1.03	0.98	0.95	0.94	0.91	0.89	0.88
polyHash1305 (avx2)	$g = 1$	0.66	0.60	0.59	0.58	0.59	0.58	0.58	0.57	0.58	0.57
		0.81	0.68	0.65	0.62	0.61	0.60	0.60	0.59	0.59	0.58
	$g = 2$	0.59	0.51	0.51	0.49	0.49	0.48	0.49	0.48	0.49	0.48
		0.79	0.61	0.57	0.54	0.53	0.52	0.51	0.51	0.51	0.50
	$g = 3$	0.59	0.50	0.48	0.47	0.47	0.46	0.47	0.46	0.46	0.45
		0.85	0.63	0.57	0.53	0.52	0.51	0.50	0.49	0.49	0.48
	$g = 4$	0.58	0.48	0.47	0.45	0.45	0.45	0.45	0.44	0.44	0.44
		0.89	0.64	0.57	0.53	0.51	0.50	0.49	0.48	0.47	0.47
BRWHash1305 (maax)	$t = 2$	0.70	0.71	0.71	0.71	0.70	0.71	0.70	0.70	0.70	0.70
		0.85	0.80	0.77	0.76	0.75	0.75	0.74	0.73	0.73	0.73
	$t = 3$	0.68	0.67	0.66	0.67	0.67	0.67	0.66	0.67	0.66	0.66
		0.82	0.77	0.74	0.72	0.71	0.71	0.70	0.70	0.69	0.69
	$t = 4$	0.66	0.66	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65
		0.81	0.75	0.72	0.71	0.69	0.69	0.68	0.68	0.68	0.68
	$t = 5$	0.65	0.66	0.65	0.65	0.65	0.65	0.64	0.65	0.64	0.65
		0.80	0.75	0.72	0.71	0.69	0.69	0.68	0.68	0.67	0.67
4-decBRWHash1305 (avx2)	$t = 2$	0.67	0.51	0.47	0.43	0.42	0.41	0.41	0.39	0.39	0.38
		0.80	0.60	0.55	0.49	0.46	0.45	0.44	0.42	0.41	0.40
	$t = 3$	0.67	0.51	0.47	0.43	0.41	0.40	0.40	0.39	0.38	0.38
		0.80	0.60	0.54	0.49	0.46	0.45	0.43	0.42	0.41	0.40
	$t = 4$	0.67	0.51	0.47	0.43	0.41	0.40	0.39	0.38	0.37	0.37
		0.81	0.59	0.54	0.48	0.45	0.44	0.43	0.41	0.40	0.39
	$t = 5$	0.67	0.51	0.47	0.43	0.41	0.40	0.39	0.38	0.37	0.37
		0.80	0.60	0.54	0.48	0.45	0.44	0.43	0.41	0.40	0.39

Table 9: Cycles/byte measurements for 50 to 500 blocks for the various hash functions based on the prime $2^{130} - 5$.

		# msg blks								
		1000	1500	2000	2500	3000	3500	4000	4500	5000
polyHash1305 (avx2)	$g = 4$	0.430	0.429	0.430	0.431	0.428	0.428	0.427	0.427	0.427
		0.445	0.439	0.438	0.437	0.433	0.432	0.431	0.431	0.430
4-decBRWHash1305 (avx2)	$t = 5$	0.350	0.341	0.340	0.338	0.340	0.341	0.339	0.338	0.337
		0.364	0.353	0.349	0.345	0.345	0.346	0.343	0.343	0.342

Table 10: Cycles/byte measurements for 1000 to 5000 blocks for avx2 implementation of polyHash1305 with $g = 4$ and 4-decBRWHash1305 with $t = 5$.