SoK: Efficient Design and Implementation of Polynomial Hash Functions over Prime Fields*

Jean Paul Degabriele[1] Jan Gilcher[2] Jérôme Govinden[3] Kenneth G. Paterson[2]

[1] Technology Innovation Institute Cryptography Research Center, Abu Dhabi, United Arab Emirates
jeanpaul.degabriele@tii.ae
[2] Applied Cryptography Group, ETH Zurich, Switzerland
jan.gilcher@inf.ethz.ch, kenny.paterson@inf.ethz.ch
[3] CNS, Technische Universität Darmstadt, Germany
jerome.govinden@tu-darmstadt.de

Abstract

Poly1305 is a widely-deployed polynomial hash function. The rationale behind its design was laid out in a series of papers by Bernstein, the last of which dates back to 2005. As computer architectures evolved, some of its design features became less relevant, but implementers found new ways of exploiting these features to boost its performance. However, would we still converge to this same design if we started afresh with today's computer architectures and applications? To answer this question, we gather and systematize a body of knowledge concerning polynomial hash design and implementation that is spread across research papers, cryptographic libraries, and developers' blogs. We develop a framework to automate the validation and benchmarking of the ideas that we collect. This approach leads us to five new candidate designs for polynomial hash functions. Using our framework, we generate and evaluate different implementations and optimization strategies for each candidate. We obtain substantial improvements over Poly1305 in terms of security and performance. Besides laying out the rationale behind our new designs, our paper serves as a reference for efficiently implementing polynomial hash functions, including Poly1305.

1 Introduction

Universal hash functions are among the most fundamental and versatile tools in cryptography and computer science more generally. In the former domain, their most important application is in constructing Message Authentication Codes (MACs). In the latter area, they are widely used in probabilistic data structures. But not all hash functions are created equal. Choosing the right hash function for a given application is a challenge for practitioners, who seek the most efficient design without compromising on security.

Universal hash functions are relatively easy to implement and significantly faster than cryptographic hash functions. Their security is information-theoretic rather than computational. Consequently, they retain their security against quantum and computationally unbounded adversaries. Classes of universal hash functions are readily obtained from basic algebraic approaches, such as matrix multiplication, inner product, and polynomial evaluation. The class based on polynomial evaluation is prevalent in practice due to its shorter key sizes and perhaps because it is easy to understand and implement. We restrict ourselves to this class here.

To construct a MAC from a universal hash function generally entails evaluating the hash over the data, and either feeding the output to a blockcipher, or blinding the output with a pseudorandom value obtained by enciphering a nonce with a blockcipher. The former is called the hash-then-PRF construction. The latter is the Wegman-Carter MAC construction [WC81]. It is used in the two most popular Authenticated Encryption with Associated Data (AEAD) schemes in use on the Internet today: Galois-Counter-Mode (AES-GCM) and ChaCha20-Poly1305. These are the default choices in protocols like TLS, SSH, and Wireguard. Both constructions (hash-then-PRF and Wegman-Carter) have the properties that MAC security and performance depends closely, and in a provable manner, on corresponding properties of the underlying universal hash function.

*This is the full version of the work that appeared at IEEE S&P 2024.

On the surface, AES-GCM and ChaCha20-Poly1305 look very similar, but when viewed from the perspective of an implementer, a number of contrasting features emerge between the two schemes. At an abstract level, they both share the same high-level structure combining a nonce-based streamcipher (AES in Counter mode and ChaCha20) with a Wegman-Carter MAC instantiated with a polynomial evaluation hash (GHASH and Poly1305). However, the polynomials are instantiated over different finite fields, a binary field $\mathsf{GF}(2^n)$ in the case of GHASH and a prime field $\mathsf{GF}(q)$ in the case of Poly1305. As a result, the optimization techniques applicable to GHASH are very different from those for Poly1305. In terms of performance, AES-GCM is typically faster on hardware that natively supports AES instructions and carry-less multiplication, which are commonly supported by most 64-bit CPUs today. However, in the absence of such hardware support, ChaCha20-Poly1305 emerges as the faster option. In general, it enjoys relatively good performance across all architectures. Moreover, because it does not rely on hardware-specific instructions, it is more portable than AES-GCM, and it has been postulated that as the level of parallelization on modern CPUs increases, it might eventually outperform AES-GCM even on CPUs with native support [Don17]. Accordingly, ChaCha20-Poly1305 is generally perceived as the one-size-fits-all solution.

It is noteworthy that when ChaCha20-Poly1305 was proposed, AES-GCM was already the dominant AEAD scheme in use, and yet it managed to gain wide popularity relatively quickly. This serves to show that there is scope for deploying new and better cryptography when it becomes available. Our thesis is that if one were to design a replacement for Poly1305 today, one would end up with a different design. The two principal reasons, concerning both security and performance, are that Poly1305 was designed as a stand-alone primitive, and the characteristics of computer processors were quite different at the time of its design. Both ChaCha20 and Poly1305 were designed by Bernstein, but as independent primitives. In fact, Poly1305 was originally proposed in combination with AES as a Wegman-Carter MAC [Ber05]. The two were only combined into an AEAD scheme later on by Langley [NL18]. While the two match rather well in terms of their implementation characteristics, they have drastically mismatching security levels. ChaCha20 uses 256-bit keys and a block size of 512 bits, whereas the security of Poly1305 is roughly $(103 - \log_2(n))$ bits, where n is the message length measured in 128-bit blocks. This disparity shows up in the security analysis in [DGGP21], where it was shown that the security of ChaCha20-Poly1305 is heavily dominated by the security of Poly1305 and that Poly1305's security may not suffice in a multi-user setting with large amounts of data to be protected.

The design of Poly1305 was tailored for 32-bit architectures and driven by the idea of exploiting the IEEE-compliant Floating-Point Units (FPUs). A decade later, this design choice looks unfortunate. Today almost all implementations of Poly1305 use integer arithmetic because, on modern processors, this yields better speeds. Enabling Poly1305 to exploit IEEE-compliant FPUs required that 22 bit positions in its key be clamped (fixed to zero). This had the side-effect of reducing its security by 22 bits. Interestingly, key clamping can still be exploited (albeit in a different way) to gain speed when using integer arithmetic, but this is incompatible with fully exploiting instruction-level parallelism—which is likely to be more beneficial in general. As a result, today, key clamping significantly weakens the security of Poly1305 without providing an adequate payback in performance. Furthermore, while its choice of prime $(2^{130} - 5)$ works fairly well on 32-bit processors, requiring five 32-bit words to represent an element in $\mathsf{GF}(2^{130} - 5)$, one is forced to use three 64-bit words on a 64-bit processor. This is wasteful of the 192 bits that would otherwise be available, either to achieve higher speed or higher security. Given the above, there is a need and an opportunity to design a new generation of improved polynomial hash functions. We can aim to achieve better performance for the same security level, or better security for the same performance level (measured in cycles per byte), or possibly even strike a better tradeoff altogether.

There is more than meets the eye to this endeavour. The design space is fairly broad with multiple interactions that need to be taken into account. We need to identify and decide on the type of polynomial, the type of prime, the size of the prime, the key size, how to encode the key into a field element, whether to use just one polynomial or a combination of polynomials, and how to encode messages onto polynomials. In turn, each design choice affects the security and performance of the resulting hash function. The implications on security are rather straightforward to determine but the same cannot be said about performance. Each design choice can either make a certain optimization technique available to implementers or take it away from them. Moreover, certain optimization strategies are incompatible, and their efficacy generally depends on the targeted processor. Thus identifying the set of design choices and parameters which meet a certain security level while enabling the right combination of optimization tricks that result in the best performance, is a rather complex task. Furthermore, it is unlikely that there is a single design that is optimal in every aspect and every use case. To complicate matters even further, the body of knowledge relating to these design aspects and optimizations is scattered across the literature applied to different hash constructions and, in a number of cases, it exists only as folklore amongst implementers. This situation calls for an SoK.

1.1 Contributions

Our effort to identify a better alternative to Poly1305 starts by gathering and systematizing the relevant design choices and optimization techniques to get a clearer picture of the full design space. We then build a framework to explore and evaluate this design space and narrow down suitable candidates. Arguably, the characteristic features of Poly1305 and its appeal derive, for the most part, from its simplicity, in employing a plain univariate polynomial, and the ease of implementing prime-field arithmetic on generic hardware. Because our focus is to find a suitable candidates in the same category as Poly1305, we restricted the scope of our search to univariate polynomials over prime fields. That said, we do cover optimization techniques that were proposed in relation to hash functions outside of this category and adapt them to this setting. Our contributions, then, are three-fold:

Systematization of Knowledge. We gather and organize the design aspects relating to the fast implementation of polynomial hash functions based on the research literature, open-source software, and some of our own observations. We classify these into *design choices* and *implementation choices*. The design choices are fixed once and for all when the hash function is specified in a public document or standardized. On the other hand implementation choices can be revised at any point and can be tailored for the application at hand, taking into account the target hardware, complexity, and portability. A good design must be informed by both categories since the design choices shape the choices available to the implementer, which ultimately determine the performance of the scheme. Our exposition aims to develop an understanding of the inter-relationships and tradeoffs between these aspects. It then serves both to guide the design of new hash functions in this category as well as to inform developers on how to best implement them, including Poly1305 itself. In fact, we do identify novel and performant implementations of Poly1305 that we did not encounter in the wild.

Benchmarking Framework. While the above systematization serves to lay out the design space and identify compatible and conflicting choices, it only tells half the story. It serves us to make broad and high-level predictions, but it is difficult to determine precisely how a particular set of choices and parameters interact on different hardware and affect the overall performance. For this reason, we developed a software framework that enables us to quantitatively evaluate the insights derived from our systematization. Our framework provides a configurable implementation of a polynomial hash function with multiple sub-components in which we can specify any compatible combination of the above design and implementation choices and automatically generate executable code that we can readily benchmark. More concretely, we can specify the construction's dimensions, the prime field, how the data and key are encoded into field elements, the field arithmetic algorithms, and the evaluation strategy of the polynomial. The source code is made public so that other researchers can use it, validate it, and build upon it.

New Hash Function Designs. We propose and experimentally evaluate several new, attractive designs that result directly from this exploration, accompanied with performant implementations generated automatically using our framework. We target three categories: (a) very high security (following up on [DGGP21]), (b) higher security at the Poly1305 performance level, and (c) higher performance at the Poly1305 security level. For the second and third categories, we offer two alternatives that are fine-tuned either for 64-bit architectures, or 32-bit and 64-bit architectures simultaneously. We observe substantial improvement across all three categories. Moreover, our *fully-delayed, two-level* implementation emerges as the overall winner across most benchmarks.

2 Background

Naming and Notation. The empty string is denoted by ε. For any positive integer μ, $\{0,1\}^{\mu}$, $\{0,1\}^{\leq\mu}$ and $\{0,1\}^{*}$ denote respectively the set of binary strings of size μ, the set of binary strings of size less than or equal to μ, and the set of all finite strings. For any string x, $|x|$ denotes its size in bits, $|x|_{\mu} = \lceil |x|/\mu \rceil$ its size in μ-bit blocks, and $x[i{:}j]$ its substring spanning bits i to bit j inclusive, where $1 \leq i < j \leq |x|$. If $|x| = 8|x|_8$, x is said to be a byte string. For any two strings x and y, $x \| y$ and $x \oplus y$ denote their concatenation and bitwise XOR respectively. For any positive integer c such that $|x| = |y| = c \cdot \mu$, $x \overset{\mu}{+} y$ and $x \overset{\mu}{-} y$ denote respectively the strings obtained from interpreting their μ-bit substrings as unsigned integers and adding or subtracting individually modulo 2^{μ}.

For any finite set S and any positive integer n, $|S|$ denotes the set's cardinality, $[S]^n$ denotes its n-ary Cartesian power, $[S]^*$ denotes the set of all finite sequences of elements in S, and $y \leftarrow\!\!\$\ S$ denotes the process of sampling

uniformly at random an element from S and assigning it to y. If $x = (x_1, \cdots, x_n)$ is a finite tuple, then $x[i]$ denotes its i-th element, i.e., x_i, and $|x|$ denotes its size (i.e., n). For any non-empty string x, $x_1, \ldots, x_n \xleftarrow{\mu} x$ denotes its parsing into μ-bit blocks such that $|x_i| = \mu$ for all $1 \leq i \leq n-1$ and $0 < |x_n| \leq \mu$. For q a prime power, $\mathbb{F}_q$ denotes the finite field of order q and $\mathbb{F}_q^*$ denotes $\mathbb{F}_q \setminus \{0\}$. Our focus here is on prime fields (q prime) and binary fields (q a power of two). We construct prime fields via the integers modulo q and the corresponding arithmetic operations, which we denote by $\mathbb{Z}_q$. Then for any $x \in \{0,1\}^{\leq \lfloor \log_2(q) \rfloor}$, its natural mapping into $\mathbb{Z}_q$ as an unsigned integer is denoted by $[x]_q$.

In reference to computation on a processor, ω is an integer denoting the processor's *word size* in bits. Then representing an integer $a \in \mathbb{Z}_q$ where $log_2(q) \leq \omega \cdot \ell$ bits, requires at most ℓ ω-bit integers. We call one such ω-bit integer a *limb* and say that ℓ limbs are required to represent a. For two sequences $\{a_j\}$ and $\{b_k\}$, we denote the sum of products over all index pairs where the predicate $P(j,k)$ is true for $0 \leq j < l_1$ and $0 \leq k < l_2$, as:

$$\sum_{l_1 : l_2 : P(j,k)} a_j b_k \,.$$

2.1 Universal Hash Functions

Universal hash functions are keyed hash functions whose security properties are quantified through a parameter ϵ. We next present the formal definitions, explain how a universal hash function can be transformed into a MAC, and how the security of the resulting MAC relates to hash parameter ϵ. All hash functions are assumed to be deterministic.

Definition 1 (Universal Hash Functions)**.** *Let $H : \mathcal{R} \times \mathcal{M} \to \mathcal{T}$ be a keyed hash function with key space $\mathcal{R}$, domain $\mathcal{M}$ and digest space $\mathcal{T}$. We say that H is ϵ-almost universal if there exists a function $\epsilon : \mathcal{M} \times \mathcal{M} \to [0,1]$ such that for all distinct inputs $M, M' \in \mathcal{M}$, it holds that*

$$\Pr_{r \leftarrow\$ \mathcal{R}}[H_r(M) = H_r(M')] \leq \epsilon(M, M') \,.$$

Intuitively the universality property is a weak form of collision resistance where the key is secret and sampled independently of the message pair. A stronger notion, known as Δ-universality, generalizes this property further.

Definition 2 (Δ-Universal Hash Functions)**.** *Let $H : \mathcal{R} \times \mathcal{M} \to \mathcal{T}$ be a keyed hash function with key space $\mathcal{R}$, domain $\mathcal{M}$ and digest space $\mathcal{T}$ where $(\mathcal{T}, +)$ is an abelian group. We say that H is ϵ-almost Δ-universal if there exists a function $\epsilon : \mathcal{M} \times \mathcal{M} \to [0,1]$ such that for all distinct inputs $M, M' \in \mathcal{M}$, and all $z \in \mathcal{T}$ it holds that*

$$\Pr_{r \leftarrow\$ \mathcal{R}}[H_r(M) = H_r(M') + z] \leq \epsilon(M, M') \,.$$

Fixing z to be the group's identity element reduces the above notion to that of a universal hash function. When the abelian group $(\mathcal{T}, +)$ is $(\{0,1\}^\tau, \oplus)$, the hash function is also said to be almost XOR-universal (AXU). Other examples of concrete group instantiations are $(\mathbb{F}_q, +)$ and $(\{0,1\}^\tau, \overset{\tau}{+})$. When $\epsilon(M, M')$ is equal to $|\mathcal{T}|^{-1}$ for all distinct $M, M' \in \mathcal{M}$, H is said to be universal (resp. Δ-universal). When $\mathcal{M} = [S]^n$ for some finite set S and fixed positive integer n we say that H is a fixed-input-length hash; if $\mathcal{M} = [S]^*$ we say that H is a variable-input-length hash.

Wegman-Carter MAC. An ϵ-almost Δ-universal hash function can be combined with a pseudorandom function via the Wegman-Carter construction to yield a nonce-based MAC [WC81]. Namely, for a Δ-universal hash function $H : \mathcal{R} \times \mathcal{M} \to \mathcal{T}$ and a pseudorandom function $F : \mathcal{K} \times \mathcal{N} \to \mathcal{T}$, the corresponding Wegman-Carter MAC is defined as: $\mathsf{MAC}((r,k), N, M) = H(r, M) + F(k, N)$. A key benefit of this construction is its efficiency: the message only needs to be processed by the Δ-universal hash function; this is typically much faster than MAC constructions based on blockciphers or cryptographic hash functions. Here the pseudorandom function is only evaluated over a short nonce N, whose output serves to blind the hash output. This is essentially the MAC construction employed in GCM where the ϵ-almost Δ-universal hash is instantiated with GHASH and the pseudorandom function is instantiated with a single AES evaluation. ChaCha20-Poly1305 uses a variation of this construction where, in addition to deriving a blinding value s, the pseudorandom function is also used to derive a one-time hash key r for every nonce. That is, $\mathsf{MAC}(k, N, M) = H(r, M) + s$ where $(r, s) = F(k, N)$ and F is instantiated with the ChaCha20 block function. Various

security analyses exist for such constructions, which in turn depend on the exact instantiation and security model being considered. Without delving too much into the details, a typical security bound for these MAC constructions when N is sampled at random, is of the form $\delta_{prf} + q_a^2/|\mathcal{N}| + \epsilon \cdot q_v$. The term δ_{prf} denotes the probability of distinguishing F from a random function, q_a and q_v denote respectively the number of authenticated messages and the number of forgery attempts, and ϵ is an upper bound on $\epsilon(M, M')$ for some maximum message size. If on the other hand N is set to be a nonce the term $q_a^2/|\mathcal{N}|$ vanishes.

2.2 Composition

The following theorem establishes the security of composing two universal hash functions, extending a classical result by Stinson [Sti91] to Δ-universality while deriving a sharper bound. Composing universal hash functions can be done for multiple reasons: to transform a universal hash into a Δ-universal one, to compress the output of a universal hash as in UMAC [Kro06] and AES-GCM-SST [CMM24], for domain extension as in XPoly [DL20], or for "ramping-up"[1] as in PolyR [KR01]. The proof of the theorem is in App. C.

Theorem 1 (Composing AU with A(Δ)U)**.** *Consider two keyed hash functions $H_1 : \mathcal{R}_1 \times \mathcal{M}_1 \to \mathcal{T}_1$ and $H_2 : \mathcal{R}_2 \times (\mathcal{M}_2 \times \mathcal{T}_1) \to \mathcal{T}_2$ that are respectively ϵ_1-almost universal and ϵ_2-almost (Δ-)universal. Their composition is given by*

$$H((r_1, r_2), (M_1, M_2)) = H_2(r_2, (M_2, H_1(r_1, M_1)))\,.$$

Further let $\epsilon_3(\cdot, \cdot)$ be such that for all distinct $M_2, M_2' \in \mathcal{M}_2$ and distinct $T, T' \in \mathcal{T}_1$, $\epsilon_3((M_2, \cdot), (M_2', \cdot)) = 0$ and

$$\Pr_{r_2 \leftarrow\!\$ \mathcal{R}_2}[H_2(r_2, (M_2, T)) = H_2(r_2, (M_2, T'))] \leq \epsilon_3((M_2, T), (M_2, T'))\,.$$

Then H is ϵ-almost (Δ-)universal where

$$\epsilon((M_1, M_2), (M_1', M_2')) = \max_{r_1 \in \mathcal{R}_1} \left(\epsilon_2(\overline{H_1}, \overline{H_1}'), \epsilon_1(M_1, M_1') + \epsilon_3(\overline{H_1}, \overline{H_1}')\right)$$

where $\overline{H_1} = (M_2, H_1(r_1, M_1))$ and $\overline{H_1}' = (M_2', H_1(r_1, M_1'))$.

The above theorem holds both for the case when H_2 is universal and Δ-universal. As for the bound, note that, by definition, $\epsilon_3(\overline{H_1}, \overline{H_1}') \leq \epsilon_2(\overline{H_1}, \overline{H_1}')$ and thus replacing ϵ_3 by ϵ_2 and setting M_2 to be the empty string, the above bound reduces to Stinson's classical composition bound of $\epsilon_1 + \epsilon_2$ [Sti91]. However, when ϵ_3 is much smaller than ϵ_2, as is the case in the composed constructions XPoly and PolyR, the above bound is better than the classical composition bound. In addition, noting that injective functions are a special kind of universal hash function, injective domain and tag transformations can be handled as a straightforward application of this theorem.

Auxiliary Background Material. Additional background material on processor architecture can be found in App. A. We refer to [Ber05] for details of the design of Poly1305.

3 Polynomial Hash Design and Optimization

We distinguish the factors and strategies affecting the performance of a polynomial hash function by classifying them as either *design choices* or *implementation choices*.

3.1 Design Choices

Designing a polynomial hash function reduces to making three design choices: choosing a type of polynomial, choosing the finite field over which the polynomial is evaluated, and choosing how bit strings—the required format for the input/output of a hash function—are encoded as field elements and vice versa. Although we list these as separate choices they are certainly not independent of each other. As a result, picking the seemingly best option for each does not necessarily result in the best combination overall.

[1] This technique consists of processing short messages with a fast hash and, as the message gets longer, composing it with a slower hash that has a better maximum message length / key size ratio.

3.1.1 Choice of Polynomial Construction

The first aspect we consider is how to choose a polynomial hash function and construct the core of a universal hash function from it.

Many types of polynomials can be considered to construct a universal hash function; each has different characteristics in terms of security bound and efficiency of their evaluation algorithm. The multivariate polynomials MMH* and NMH* from Halevi and Krawczyk [HK97] have the significant drawback that they require a key of the same length as the message. This hinders their efficiency for longer messages, and so they are excluded from further consideration here. A more efficient choice is the classical univariate polynomial. It was introduced independently by den Boer [dB93], by Johansson, Kabatianskii and Smeets [JKS94], and by Taylor [Tay94]. It is the basis for most modern polynomial hash functions such as Poly1305 [Ber05], GHASH [MV04] and POLYVAL [GLL17]. Later, efficient univariate polynomials based on the BRW polynomial from [Ber07] were introduced [CGS17, GS19, BNS24].

All these univariate polynomial hash functions share a common feature: they obtain their universality property from their injectivity and maximum degree. We formalize this observation in Thm. 2. The theorem, which is a reformulation of an idea from Bernstein [Ber07, Section 5.7] and whose proof can be found in App. D.1, gives simplified conditions that a univariate polynomial needs to satisfy to be a universal hash function. Thus, choosing a suitable polynomial on which to build a universal hash function often reduces to choosing a polynomial satisfying the conditions from Thm. 2 and ensuring that it has an efficient evaluation algorithm.

Theorem 2 (Universality of Polynomial Hash)**.** *Let $P_x : \mathcal{M} \to \mathbb{F}_q[x]$ be an injective function over the domain $\mathcal{M}$ such that for any $M, M' \in \mathcal{M}$, $P_x(M) - P_x(M')$ is a univariate polynomial in the indeterminate x of degree at most $d(M, M')$. Then the keyed hash function $H : \mathbb{F}_q \times \mathcal{M} \to \mathbb{F}_q$ defined as $H(r, M) = P_r(M)$ (i.e. by evaluating the polynomial $P_x(M)$ at $x = r$) is ϵ-almost universal with $\epsilon(M, M') = \frac{d(M,M')}{q}$. Additionally, if the function $G_x : \mathcal{M} \to \mathbb{F}_q[x]$ defined as $G_x(M) = P_x(M) - P_0(M)$ is injective, then H is ϵ-almost Δ-universal.*

Notably, the theorem states that when P_x is injective with a constant coefficient of zero, then H is also a Δ-universal hash function. The domain $\mathcal{M}$ will be very often a subset of $[\mathbb{F}_q]^*$. In this case, the degree of $P_x(M)$ will depend on the tuple size of its input M. Thus, the complexity of the evaluation algorithm of H in terms of field element operations will also depend on this tuple size. We call any hash function that satisfies Thm. 2 a *polynomial universal hash function over the finite field $\mathbb{F}_q$*.

The Classical Polynomial. The classical polynomial is the simplest possible choice of polynomial for constructing a universal hash function. A univariate polynomial with indeterminate x and coefficients $M_1, \cdots, M_n$ can be constructed as $M_1 + M_2 \cdot x + M_3 \cdot x^2 + \cdots + M_n \cdot x^{n-1}$. However, this formulation cannot be evaluated in an online fashion using Horner's algorithm if M_1 is the first available message block. Accordingly, we will use the prevalent formulation where the message-block ordering is reversed, yielding:

$$H_{\text{poly}}(r, M) = \sum_{i=1}^{n} M_i \cdot r^{n-i} = M_1 \cdot r^{n-1} + M_2 \cdot r^{n-2} + \cdots + M_{n-1} \cdot r^1 + M_n.$$

We cover the different evaluation strategies for this hash function, including Horner's algorithm, in Sec. 3.2.4. By Thm. 2, we can enforce structure on the domain $\mathcal{M}$ in order to tune this construction to handle variable input lengths and attain the desired level of universality. The exact mapping of a message onto H_{poly} depends on the choice of data-to-field encoding (cf. Sec. 3.1.6), which in turn depends on the specific choice of finite field. Yet, assuming an injective data-to-field encoding, fixing the constant term be zero guarantees that H_{poly} achieves the stronger Δ-universality property, and fixing M_1 to be non-zero allows H_{poly} to be (Δ-)universal over a variable number of message blocks. If M_1 is allowed to be zero, then H_{poly} is only universal over a domain of some fixed number of message blocks n. These choices are summarised in Tab. 1. The last option shown there is a popular choice in practice. It is used in Poly1305 as well as our newly proposed designs.

In this paper we will focus on the classical polynomial as it is simple and general enough to present the main optimizations that are available to polynomial hash functions over finite fields. Most of these optimizations extend to other types of polynomials, but we defer the study of other types of polynomials (such as the BRW polynomial) to future work. Besides the choices listed in Tab. 1, there exist several other transformations that we can then apply to that polynomial in order to derive different hash functions with parameters that better suit our needs. We next discuss these transformations and how they impact security and efficiency.

$\mathcal{M}$	Universality Type	$\epsilon(M, M')$
$[\mathbb{F}_q]^n$	ϵ-almost universal	$\frac{n-1}{q}$
$\mathbb{F}_q^* \times [\mathbb{F}_q]^*$	ϵ-almost universal	$\frac{\max(\lvert M\rvert,\lvert M'\rvert)-1}{q}$
$[\mathbb{F}_q]^n \times \{0\}$	ϵ-almost Δ-universal	$\frac{n}{q}$
$\mathbb{F}_q^* \times [\mathbb{F}_q]^* \times \{0\}$	ϵ-almost Δ-universal	$\frac{\max(\lvert M\rvert,\lvert M'\rvert)-1}{q}$

Table 1: Universality properties of the classical polynomial H_{poly} for different domains $\mathcal{M}$.

3.1.2 Concatenation of Universal Hash Functions

Concatenating universal hash functions is typically used for two main reasons: to amplify security [Rog95, Proposition 3] and domain extension [Sti91, Theorem 5.4]. The resulting security amplification or domain extension come at the cost of an increased tag size and the penalty of evaluating an additional hash function. However when the two hash functions can be evaluated in parallel, the performance penalty is less significant.

Proposition 1 (Concatenation of AUs or AΔUs for Security Amplification)**.** *Let $H_1 : \mathcal{R}_1 \times \mathcal{M} \to \mathcal{T}_1$ be an ϵ_1-almost universal hash function (resp. ϵ_1-almost Δ-universal hash function) and $H_2 : \mathcal{R}_2 \times \mathcal{M} \to \mathcal{T}_2$ be an ϵ_2-almost universal hash function (resp. ϵ_2-almost Δ-universal hash function).*

- *Consider the family of keyed hash functions $H : (\mathcal{R}_1 \times \mathcal{R}_2) \times \mathcal{M} \to (\mathcal{T}_1 \times \mathcal{T}_2)$ defined as follows: for all $(r_1, r_2) \in \mathcal{R}_1 \times \mathcal{R}_2$ and $M \in \mathcal{M}$, $H((r_1, r_2), M) = (H_1(r_1, M), H_2(r_2, M))$. Then H is an ϵ-almost universal hash function (resp. ϵ-almost Δ-universal hash function) with $\epsilon(M, M') = \epsilon_1(M, M') \cdot \epsilon_2(M, M')$.*
- *When $\mathcal{R}_1 = \mathcal{R}_2$, consider the family of keyed hash functions $G : \mathcal{R}_1 \times \mathcal{M} \to (\mathcal{T}_1 \times \mathcal{T}_2)$ defined as follows: for all $r \in \mathcal{R}_1$ and $M \in \mathcal{M}$, $G(r, M) = (H_1(r, M), H_2(r, M))$. Then G is an ϵ-almost universal hash function (resp. ϵ-almost Δ-universal hash function) with $\epsilon(M, M') = \min(\epsilon_1(M, M'), \epsilon_2(M, M'))$.*

The above proposition considers the concatenation of two hash functions with independent keys and with the same key. As can be observed from the proposition, when the same key is used, there is no gain in terms of security. On the other hand, if we concatenate two independently-keyed polynomials over the fields $\mathbb{F}_{q_1}$ and $\mathbb{F}_{q_2}$, each of maximum degree d_1 and d_2, and satisfying the conditions from Thm. 2, we obtain a universal hash function with an improved security bound of $\frac{d_1 \cdot d_2}{q_1 \cdot q_2}$. For example, Degabriele et al. [DGGP21] proposed concatenating two independent instances of Poly1305 to realise a universal hash with higher security. Compared to the security of a polynomial hash function of maximum degree d designed over a single field of size $q_1 \cdot q_2$, the concatenation construction attains a better bound only when $d_1 \cdot d_2 \leq d$. However, for any given message length and data-to-field encoding, the resulting polynomial degree decreases as field size increases (cf. Sec. 3.1.6), meaning that d is smaller than both d_1 and d_2. Consequently, a polynomial designed over a single field of size $q_1 \cdot q_2$ yields better security than the concatenation construction. In cases where the degrees of the concatenated polynomials are small, this security loss can be acceptable if the efficiency of the concatenation construction is superior.

Proposition 2 (Concatenation of AUs or AΔUs for Domain Extension)**.** *Let $H_1 : \mathcal{R}_1 \times \mathcal{M}_1 \to \mathcal{T}_1$ be an ϵ_1-almost universal hash function (resp. ϵ_1-almost Δ-universal hash function) and $H_2 : \mathcal{R}_2 \times \mathcal{M}_2 \to \mathcal{T}_2$ be an ϵ_2-almost universal hash function (resp. ϵ_2-almost Δ-universal hash function). Consider the two families of keyed hash functions defined as follows:*

- *$H : (\mathcal{R}_1 \times \mathcal{R}_2) \times (\mathcal{M}_1 \times \mathcal{M}_2) \to (\mathcal{T}_1 \times \mathcal{T}_2)$ where for all $(r_1, r_2) \in \mathcal{R}_1 \times \mathcal{R}_2$ and $(M_1, M_2) \in \mathcal{M}_1 \times \mathcal{M}_2$,*

$$H((r_1, r_2), (M_1, M_2)) = (H_1(r_1, M_1), H_2(r_2, M_2)),$$

- *$G : \mathcal{R}_1 \times (\mathcal{M}_1 \times \mathcal{M}_2) \to (\mathcal{T}_1 \times \mathcal{T}_2)$ when $\mathcal{R}_1 = \mathcal{R}_2$ and for all $r \in \mathcal{R}_1$ and $(M_1, M_2) \in \mathcal{M}_1 \times \mathcal{M}_2$,*

$$G(r, (M_1, M_2)) = (H_1(r, M_1), H_2(r, M_2)).$$

Then both H and G are ϵ-almost universal hash functions (resp. ϵ-almost Δ-universal hash functions) with

$$\epsilon((M_1, M_2), (M_1', M_2')) = \max(\epsilon_1(M_1, M_1'), \epsilon_2(M_2, M_2')).$$

Prop. 2 states that reusing the same key across the two hash functions yields the same security bound as when using independent keys, making the former a more appealing choice when the goal is domain extension. Indeed this approach is used in UMAC [Kro06] and CLHASH [LK16]. Thus if we concatenate two polynomial universal hash functions over the fields $\mathbb{F}_{q_1}$ and $\mathbb{F}_{q_2}$, each of maximum degree d_1 and d_2, and satisfying the conditions from Thm. 2, we obtain a universal hash function with a larger domain and a security bound of $\max(\frac{d_1}{q_1}, \frac{d_2}{q_2})$. Yet, compared to a polynomial hash function defined over a field of size $q_1 \cdot q_2$, thus having a comparable tag length, the concatenation construction does not typically yield better security. For a given message length, and assuming that the amount of data bits embedded in a field element is proportional to the logarithm of the field size, it can be shown that $d \leq \max(d_1, d_2)$, where d is the maximum degree of the single polynomial construction. Accordingly, when $0 < d_1 < q_1$ and $0 < d_2 < q_2$, it follows that the security bound of the concatenation construction is always worse than the bound for the alternative single polynomial construction, i.e., $\frac{d}{q_1 \cdot q_2} \leq \frac{\max(d_1, d_2)}{q_1 \cdot q_2}$.

The ability to reuse keys in the concatenation construction would seem to indicate that it requires less randomness. However, it is also possible to use a single polynomial construction over a field of size $q_1 \cdot q_2$ with a reduced keyspace (using Prop. 4) to obtain a similar bound with the same amount of randomness. Thus, in terms of security, a single polynomial is always better; using concatenation for domain extension makes sense only if it results in better performance. In terms of performance, concatenating two polynomial hash functions over a smaller field does not generally affect performance one way or another, but it can degrade performance if more limbs are required to represent the same amount of data. In some cases, the evaluation of a single polynomial may also be parallelized, as is the case with the classical polynomial (cf. Sec. 3.2.4). Yet, for the concatenation construction, it is often easier to exploit parallelization (through AVX, for example) and without any overhead cost.

3.1.3 Composition of Polynomials

Composing polynomial hash functions may be used for the same reasons as when composing two general hash functions (cf. Sec. 2.2). However, in the general setting, we can only compose independently sampled hash functions. As stated in the following theorem, in the special case of polynomial hash functions over the same field, the same key can be reused in both polynomials at the cost of slightly more stringent requirements on the polynomials. The theorem relies on our alternative characterization of a univariate polynomial universal hash, i.e. a universal hash satisfying the requirements of Thm. 2 adapted from [Ber07]. Its proof can be found in App. D.4, uses the fact that the composition of injective functions is itself injective. This type of composition is implicit in multi-level single-key polynomial hash functions such as the ones described in [CGS17] and [BNS24].

Theorem 3 (Composition of Polynomial Hash Functions)**.** *Consider the family of keyed hash functions* $G : \mathbb{F}_q \times \prod_{i=0}^{n} \mathcal{M}_i \to \mathbb{F}_q$ *defined as follows. For any* $r \in \mathbb{F}_q$ *and* $(M_0, M_1, \cdots, M_n) \in \prod_{i=0}^{n} \mathcal{M}_i$*,*

$$G(r, (M_0, M_1, \cdots, M_n)) = H(r, (M_0, P_x^1(M_1), \cdots, P_x^n(M_n)))$$

where, for $i \in \{1, 2, \cdots, n\}$*,* $P_x^i : \mathcal{M}_i \to \mathbb{F}_q[x]$ *are injective functions and the function* $H : \mathbb{F}_q \times (\mathcal{M}_0 \times [\mathbb{F}_q[x]]^n) \to \mathbb{F}_q$ *is a polynomial hash function. Thus, there exist an injective function* $Q_y : \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n \to \mathbb{F}_q[y]$ *that satisfies:*

- *for any* $r \in \mathbb{F}_q$ *and* $M \in \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n$*,* $H(r, M) = Q_r(M)$ *(i.e. by evaluating the polynomial* $Q_y(M)$ *at* $y = r$*),*
- *for any* $M, M' \in \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n$*,* $Q_y(M) - Q_y(M')$ *is of total degree at most* $d(M, M')$*.*

Then G *is an* ϵ*-almost universal hash function with* $\epsilon((M_0, M_1, \cdots, M_n), (M_0', M_1', \cdots, M_n')) = \frac{d(\overline{M}, \overline{M}')}{q}$*, where* $\overline{M} = (M_0, P_x^1(M_1), \cdots, P_x^n(M_n))$ *and* $\overline{M}' = (M_0', P_x^1(M_1'), \cdots, P_x^n(M_n'))$*. If in addition, the function* $\overline{Q}_y : \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n \to \mathbb{F}_q[y]$ *defined as* $\overline{Q}_y(M) = Q_y(M) - Q_0(M)$ *is injective, then* G *is also* ϵ*-almost* Δ*-universal.*

If the functions P_r^i are all efficiently computable and H is efficiently computable when given efficiently computable polynomial hash functions as inputs, then G is also efficiently computable.

As a straightforward application of Thm. 3, one can transform any universal polynomial hash function H into a Δ-universal hash function through the composed function $G(r, M_1) = H_r(M_1) \cdot r$. In the two following subsections, we give two other applications of this theorem for domain extension.

3.1.4 Union of Domains

Transforming a set of polynomial hash functions H_r^i with domains $\mathcal{M}_i$ into a polynomial hash function with domain $\bigsqcup_i \mathcal{M}_i$ can be useful to obtain a variable-input-length hash from a set of fixed-input-length polynomial hashes. For almost universal hash functions, a simple technique used in the first version of UMAC and Badger [BCZ05] appends the message length to the hash output. This approach can be generalized to the disjoint union of domains $\bigsqcup_i \mathcal{M}_i$ by defining the hash function as $\overline{H}(r, M) = (H_r^j(M), j)$ when $M \in \mathcal{M}_j$. An obvious drawback of this technique is that it increases the tag size. To compress the tag size, we can compose the hash function $\overline{H}$ with a compressing hash function and apply either Thm. 1 or Thm. 3. In the first case, an extra hash key is required, but better security is achieved. In the second case, no additional key is needed but a slight security loss is incurred, as described in the following corollary.

Corollary 1 (Disjoint Union of Domains for Polynomial Hash)**.** *Let* $\textsc{Inj} : \{1, 2, \cdots, n\} \to \mathcal{M}$ *be an injective encoding function. Consider the family of keyed hash functions* $G : \mathbb{F}_q \times \bigsqcup_{i=1}^{n} \mathcal{M}_i \to \mathbb{F}_q$ *defined as follows. For any* $r \in \mathbb{F}_q$ *and* $M \in \mathcal{M}_j$*,*

$$G(r, M) = H(r, (\textsc{Inj}(j), P_x^j(M)))$$

where, for $i \in \{1, 2, \cdots, n\}$*,* $P_x^i : \mathcal{M}_i \to \mathbb{F}_q[x]$ *are injective functions and the function* $H : \mathbb{F}_q \times (\mathcal{M} \times \mathbb{F}_q[x]) \to \mathbb{F}_q$ *is a polynomial hash function. Thus, there exist an injective function* $P_y : \mathcal{M} \times \mathbb{F}_q[x] \to \mathbb{F}_q[y]$ *satisfying the following two conditions:*

- *for any* $r \in \mathbb{F}_q$ *and* $M \in \mathcal{M} \times \mathbb{F}_q[x]$*,* $H(r, M) = P_r(M)$*,*
- *for any* $M, M' \in \mathcal{M} \times \mathbb{F}_q[x]$*,* $P_y(M) - P_y(M')$ *is of total degree at most* $d(M, M')$*.*

Then G *is an* ϵ*-almost universal hash function where, for all* $(M, M') \in \mathcal{M}_{j_1} \times \mathcal{M}_{j_2}$*,*

$$\epsilon(M, M') = \frac{d((\textsc{Inj}(j_1), P_x^{j_1}(M)), (\textsc{Inj}(j_2), P_x^{j_2}(M')))}{q}.$$

If in addition, the function $\overline{P}_y : \mathcal{M} \times \mathbb{F}_q[x] \to \mathbb{F}_q[y]$ *defined as* $\overline{P}_y(M) = P_y(M) - P_0(M)$ *is injective, then* G *is also* ϵ*-almost* Δ*-universal.*

Applying this corollary to a set of polynomial hash functions H_r^i with domains $\mathcal{M}_i$, we can obtain an almost universal hash with domain $\bigsqcup_i \mathcal{M}_i$ by composing it with an efficient polynomial hash H. By instantiating H with the classical polynomial, we can obtain either an almost universal hash $G(r, M) = H_r^j(M) \cdot r + \textsc{Inj}(j)$ or an almost Δ-universal hash $G(r, M) = H_r^j(M) \cdot r^2 + \textsc{Inj}(j) \cdot r$, for $M \in \mathcal{M}_j$. If the functions H_r^i are also instantiated with classical polynomials over fixed-length domains, the above transformations both result in a variable-input-length polynomial hash of the form $H_{\text{poly}}(r, (M, \textsc{Inj}(j)))$. In practice, Inj would typically encode the message length to a field element. In the above two instantiations of the transformation G, the degree of the polynomial hash H_r^j is increased by at most 2, thereby adding only $\frac{2}{q}$ to its security bound.

3.1.5 Cartesian Product of Domains

Certain applications require a universal hash over a domain consisting of the Cartesian product of two sets. This is notably the case in AEAD constructions, such as in ChaCha20-Poly1305 and AES-GCM, where the input to a universal hash function is a pair of strings consisting of the message and associated data. Constructions of variable-input-length tweakable ciphers [DGGP24] typically employ a universal hash over a message and a tweak. Below we present a variety of techniques for obtaining a polynomial hash function with domain $\mathcal{M}_1 \times \mathcal{M}_2$. These results can be generalized to the product of more than two sets.

Injective Encoding. The first approach is to compose a polynomial hash H_r over the domain $\mathcal{M}$ with an injective encoding function $\textsc{Inj} : \mathcal{M}_1 \times \mathcal{M}_2 \to \mathcal{M}$. We discuss such encondings in more detail in § Data-to-Field Encoding, where the techniques described therein can be easily extended to pairs of strings. For improved security and efficiency, the encoding should aim to minimize the number of components in the output tuple. For example, to efficiently map an element from $\{0,1\}^* \times \{0,1\}^*$ to the domain $\mathbb{F}_q^* \times [\mathbb{F}_q]^* \times \{0\}$ of the classical polynomial (cf. Tab. 1) one can use the following injective encoding. We prepend two bits to each block to delineate the two input messages and mark

the last block of each, then in the last block of each input we add a stop-bit to mark its length, as shown in the following encoding

$$\textsc{Inj}(M, M') = [1\|M_1]_q, \ldots, [1\|M_{\lceil \frac{|M|}{\mu} \rceil}]_q,\ [11\|M'_1]_q, \ldots, [11\|M'_{\lceil \frac{|M'|}{\mu} \rceil - 1}]_q, [10^{\mu+1} \oplus (1\|M'_{\lceil \frac{|M'|}{\mu} \rceil})]_q, 0\,, \tag{1}$$

using a block size $\mu \leq \lfloor \log_2(q) \rfloor - 2$ where $M_1, \ldots, M_{\lceil \frac{|M|}{\mu} \rceil} \xleftarrow{\mu} M$ and $M'_1, \ldots, M'_{\lceil \frac{|M'|}{\mu} \rceil} \xleftarrow{\mu} M'$.

For that specific blocksize the above encoding has minimum output size, and it can be combined with all our new designs from Sec. 5 to extended their domains to $\{0,1\}^* \times \{0,1\}^*$. However, it is not directly applicable to Poly1305 as its blocksize μ is bigger than $\lfloor \log_2(q) \rfloor - 2$. In the case where $\mu = \lfloor \log_2(q) \rfloor$, an injective encoding from the domain $\{0,1\}^* \times \{0,1\}^*$ to $\mathbb{F}_q^* \times [\mathbb{F}_q]^* \times \{0\}$ is as follows:

$$\textsc{Inj}(M, M') = \begin{cases} [2|M| + 2]_q, [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q, [M'_1]_q, \ldots, [M'_{\lceil \frac{|M'|}{\mu} \rceil}]_q, 0 & \text{if } \mu \text{ divides } |M'|, \\ [2|M| + 3]_q, [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q, [M'_1]_q, \ldots, [M'_{\lceil \frac{|M'|}{\mu} \rceil - 1}]_q, [1\|M'_{\lceil \frac{|M'|}{\mu} \rceil}]_q, 0 & \text{otherwise.} \end{cases}.$$

To be injective, this encoding uses a stop-bit (if needed) in the last block of the second input M' to mark its length and encodes the presence of the stop-bit together with the bit length of the first input M in the first field element. At the same time, this encoding ensures that the first element is non-zero, as required by the classical polynomial for variable-length inputs. Moreover its overhead is limited to only one extra field element. Notably this encoding is employed in HCTR2 [CHB21].

When combined with the classical polynomial, an alternative encoding that uses a combination of stop-bit and length encoding is the following

$$\textsc{Inj}(M, M') = \begin{cases} [1]_q, [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q, [M'_1]_q, \ldots, [M'_{\lceil \frac{|M'|}{\mu} \rceil}]_q, [2|M| + 1]_q, 0 & \text{if } \mu \text{ divides } |M'|, \\ [1]_q, [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q, [M'_1]_q, \ldots, [M'_{\lceil \frac{|M'|}{\mu} \rceil - 1}]_q, [1\|M'_{\lceil \frac{|M'|}{\mu} \rceil}]_q, [2|M| + 2]_q, 0 & \text{otherwise.} \end{cases}.$$

Compared to the injective encoding used in HCTR2, in this encoding the length of the input M is only required at the end of the polynomial evaluation, which is preferable in practice.

Concatenation. An alternative that avoids the overhead incurred by injective encoding approach, is to use concatenation as described in Prop. 2. Namely two polynomial hash functions, H_r^1 and H_r^2, with domains $\mathcal{M}_1$ and $\mathcal{M}_2$, can be combined into a concatenated polynomial hash $(H_r^1(M_1), H_r^2(M_2))$ with domain $\mathcal{M}_1 \times \mathcal{M}_2$. The cost of this construction is at most the cost of processing both messages individually. While this solution allows parallel processing of the two input messages, it has the drawback of increasing the tag size. Once again, the tag size can be reduced through the composition theorem (Thm. 1 or Thm. 3) by composing the concatenated hash with another universal hash function. Using Thm. 3 requires fewer keys than Thm. 1, at the expense of a minor security loss. Below are two such constructions combining concatenation and composition with domain $\mathcal{M}_1 \times \mathcal{M}_2$:

$$H_r(M_1, M_2) = H_{r^2}^1(M_1) \cdot r + H_{r^2}^2(M_2)\,,$$
$$H_{r_1,r_2}(M_1, M_2) = H_{r_1}^1(M_1) \cdot r_2 + H_{r_1}^2(M_2)\,.$$

When instantiated with the classical polynomial, the security loss incurred by these transforms is, at most, a factor of 2. The first construction is an efficient solution in the case where the block size μ used to parse the bit-string data is equal to $\lfloor \log_2(q) \rfloor - 1$, as in Poly1305 where $\mu = 128$ and $q = 2^{130} - 5$. In this case, using the injective encoding from eq. (1) is not possible. However, it is possible to integrate the variable-length encoding described in § Data-to-Field Encoding, that only requires a stop-bit, with the above construction. In the special case where $H_{r_1}^1$ and $H_{r_2}^2$ are both ϵ-almost Δ-universal and keyed independently, the above constructions can be simplified. As shown in [CW77, Prop. 8], the following construction is also ϵ-almost Δ-universal:

$$H_{r_1,r_2}(M_1, M_2) = H_{r_1}^1(M_1) + H_{r_2}^2(M_2).$$

Note that this construction generalizes to the addition of three or more hash functions. In turn, we can further concatenate multiple instances of this construction and attain significantly higher security, without a corresponding increase in the key material, by applying the Toeplitz approach [BHK+99] as shown in the example below:

$$H_{r_1,r_2,r_3,r_4}(M_1, M_2, M_3) = H_{r_1}^1(M_1) + H_{r_2}^2(M_2) + H_{r_3}^3(M_3) \,||\, H_{r_2}^1(M_1) + H_{r_3}^2(M_2) + H_{r_4}^3(M_3).$$

Composition. Alternatively, the composition of two universal hash functions, through Thm. 1 or Thm. 3, readily gives us a way to extend their domains to $\mathcal{M}_1 \times \mathcal{M}_2$. That is, using Thm. 1, we obtain the following construction:

$$H_{r_1,r_2}(M_1, M_2) = H^2_{r_2}(M_2, H^1_{r_1}(M_1)).$$

This construction provides better security than injective encoding but requires two independent hash keys. Notably, this composition technique is used in XPoly to extend the domain of the classical polynomial with limited security loss, i.e., to obtain a different tradeoff between key size and security. Using Thm. 3 as a composition theorem requires fewer key material than Thm. 1, at the cost of a worse security bound.

Product of Two Disjoint Unions. The final technique that we present yields a universal hash whose domain is the product of two disjoint unions $\bigsqcup_i \mathcal{M}_i \times \bigsqcup_j \mathcal{M}_j$ by viewing it as the disjoint union of two products $\bigsqcup_{i,j} \mathcal{M}_i \times \mathcal{M}_j$, and then use one of the transforms for product of domains $\mathcal{M}_i \times \mathcal{M}_j$ and one of the transforms for the disjoint union of domains from Sec. 3.1.4.

A prominent example of where this is used over the domain $\{0,1\}^* \times \{0,1\}^*$ is GHASH, which uses the classical polynomial combined with the following encoding that appends a block containing the length of its two inputs M, M'

$$\textsc{Inj}(M, M') = [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q,\ [M'_1]_q, \ldots, [M'_{\lceil \frac{|M'|}{\mu} \rceil}]_q, [|M| \,\|\, |M'|]_q, 0\,.$$

3.1.6 Choice of Encodings

The polynomial universal hash functions we described in Sec. 3.1.1 are all based on polynomials over a finite field $\mathbb{F}_q$, and as such, their key space, digest space and often their domain, are cartesian products of this finite field. In practice, however, messages, keys, and tags are bit strings. We next show how to convert bit string messages and keys into finite field elements and convert tags that are finite field elements into bit strings.

Field-to-Tag Encoding. It is possible to convert an almost universal or Δ-universal hash function $H : \mathcal{R} \times \mathcal{M} \to \mathbb{F}_q$ into a hash function with a tag space $\{0,1\}^\tau$ by identifying a suitable transform function $f_t : \mathbb{F}_q \to \{0,1\}^\tau$ to compose it with.

Encoding for universal hash functions. If the function H is only ϵ-almost universal, according to Thm. 1, it is possible to transform it into a universal hash function with a tag space $\{0,1\}^\tau$ of size $2^\tau \geq q$ by composing it with an injective transform function f_t. Then, the composed function $f_t \circ H$ stays ϵ-almost universal. When $\mathbb{F}_q$ is a prime field, such an injective function can be defined by identifying $\mathbb{F}_q$ with the set of integers $\{0, \cdots, q-1\}$ and then mapping such integers to bit strings of length τ.

Encoding for Δ-universal hash functions. When the function H is ϵ-almost Δ-universal, we can use Prop. 3 below to perform a modulus switch from a prime field $\mathbb{Z}_q$ to the group $\mathbb{Z}_{2^\tau}$ and then naturally encode elements from $\mathbb{Z}_{2^\tau}$ in $\{0,1\}^\tau$. The proposition and its proof (deferred to App. D.5), reformulate the technique used for MMH and Poly1305 to obtain a Δ-universal hash function over bit strings.

Proposition 3 (Modulus Switching for AΔU Hash Function)**.** *Let $H : \mathcal{R} \times \mathcal{M} \to \mathbb{Z}_n$ be a family of keyed hash functions and $f_t : \mathbb{Z}_n \to \mathbb{Z}_m$ denote the function $f_t(x) = x \pmod m$. Define the family of keyed hash functions $H' : \mathcal{R} \times \mathcal{M} \to \mathbb{Z}_m$ as follows*

$$H'(r, M) = f_t(H(r, M)).$$

If H is ϵ-almost Δ-universal, then H' is ϵ'-almost Δ-universal with $\epsilon'(M, M') = \left(\lfloor \frac{2n-2}{m} \rfloor + 1\right) \cdot \epsilon(M, M')$.

Note that when $n \leq m$, f_t is an injection from $\mathbb{Z}_n$ to $\mathbb{Z}_m$ that does not require any computation. When $n > m$ and $m = 2^\tau$, $f_t(x) = x \pmod m$ can be efficiently implemented as the truncation of the highest bits of the binary representation of the integer x. If $n < \frac{m}{2} + 1$, there is no security loss when switching modulus in the proposition.

To summarize, when composing with a transform function $f_t : \mathbb{F}_q \to \{0,1\}^\tau$, there is almost always no security loss and computational overhead when $q \leq 2^\tau$. When $q > 2^\tau$, the bigger q is, the more significant the security loss with negligible computational overhead. Thus, the tag size τ needs to be chosen according to an acceptable tag size/security tradeoff.

Another notable approach for shortening the tag size involves multiplying the tag with a second key and then truncating. As this transformation requires a key, it cannot be classified as an encoding, but it is in fact a Δ-universal

hash function. Accordingly, the result is effectively a composition of two hash functions with a better overall security bound than the above modulus switching technique, as predicted by Thm. 1. In particular, with an appropriate choice of parameters, this technique can transform a polynomial hash function into a Δ-universal hash function, thereby yielding the best possible security ($\epsilon = 2^{-\tau}$) for that tag size at the cost of a mild expansion in the hash key [NGR05, CMM24].

Key-to-Field Encoding. We now discuss how to transform the key space of a universal hash function through encoding and the associated security loss. We then present two examples of such encoding functions and how restricting either the key size or the output format of these functions provides different security-performance tradeoffs.

Security loss due to encoding. Consider the composition of an almost universal or Δ-universal hash $H : \mathbb{F}_q \times \mathcal{M} \to \mathcal{T}$ with an encoding function $f_k : \{0,1\}^\rho \to \mathbb{F}_q$ to obtain a hash function with a key space $\{0,1\}^\rho$. The resulting security loss is quantified in Prop. 4. We say that the function f_k has bias β, if the min-entropy of the output distribution of f_k for random inputs is at least $\ln(1/\beta)$, or said differently, if for all $r \in \mathcal{R}$, $\Pr_{r' \leftarrow\$ \mathcal{R}'}[f_k(r') = r] \leq \beta$. The larger the bias β in the distribution of the outputs of f_k is, the more significant the security loss of the composed construction. The proof is presented in App. D.6.

Proposition 4 (Key Space Transformation of AU or AΔU)**.** *Let $H : \mathcal{R} \times \mathcal{M} \to \mathcal{T}$ be a family of keyed hash functions and f_k be an encoding function with bias β. Consider the hash function $H' : \mathcal{R}' \times \mathcal{M} \to \mathcal{T}$ defined as follows*

$$H'(r', M) = H(f_k(r'), M).$$

If H is ϵ-almost universal (resp. ϵ-almost Δ-universal), then H' is ϵ'-almost universal (resp. ϵ'-almost Δ-universal) with $\epsilon'(M, M') = \beta \cdot |\mathcal{R}| \cdot \epsilon(M, M')$.

Note that when f_k is an injective function it follows that $\beta = |\mathcal{R}'|^{-1}$, whereas if f_k is bijective the security of the hash function is preserved, i.e., $\epsilon' = \epsilon$.

Examples of encoding functions. For a key-to-field encoding function $f_k : \{0,1\}^\rho \to \mathbb{F}_q$, when $2^\rho \leq q$, a simple injective function f_k leads to a security loss by a factor of $\frac{q}{2^\rho}$ by Prop. 4. On the other hand, when $2^\rho > q$ we need an encoding function with a small bias to obtain the best security. One possibility is to associate $\{0,1\}^\rho$ with the set of integers $\{0, \cdots, 2^\rho - 1\}$, $\mathbb{F}_q$ with the set of integers $\{0, \cdots, q-1\}$, and letting $f_k(r) = r \bmod m$ for some $m \leq q$. The parameter m determines the efficiency at which f_k can be computed and also serves as an upper bound for the output of $f_k(r)$. When m is a power of two, the modulo reduction is equivalent to truncating the highest bits of r. When m is not a power of two, an alternative technique due to Lemire [Lem19] is to instead set $f_k(r) = \left\lfloor \frac{r \cdot m}{2^\rho} \right\rfloor$. This replaces the modulo operation with an integer multiplication and an integer division by a power of two (essentially a shift operation). Both encodings incur the same bias (β), and therefore the same security loss. This is stated formally in Prop. 5, whose proof can be found in App. D.7.

Proposition 5 (Random Integer Generation in an Interval)**.** *Let $m \leq q$ and $f_k : \{0, \cdots, 2^\rho - 1\} \to \{0, \cdots, q-1\}$ be defined as either*

- $f_k(r) = r \pmod m$*, or*
- $f_k(r) = \left\lfloor \frac{r \cdot m}{2^\rho} \right\rfloor$.

Then for all $\overline{r} \in \{0, \cdots, q-1\}$, $\Pr_{r \leftarrow\$ \{0, \cdots, 2^\rho - 1\}}[f_k(r) = \overline{r}] = \beta \leq \left\lceil \frac{2^\rho}{m} \right\rceil \cdot \frac{1}{2^\rho}$.

In the above proposition, the largest possible m, i.e., $m = q$, leads to the best bound but might not provide the best efficiency. Choosing $m = 2^{\lfloor \log_2(q) \rfloor}$ instead, leads to a very efficient modulo operation and a bound of $\left\lceil \frac{2^\rho}{2^{\lfloor \log_2(q) \rfloor}} \right\rceil \cdot \frac{1}{2^\rho}$, i.e., incurring a security loss of less than a single bit and thus striking a better tradeoff overall. Setting $\rho = \lfloor \log_2(m) \rfloor$ is typically a good choice since, by Prop. 5, a larger value can only reduce the bias by a single bit at best. Thus, letting $f_k(r)$ be the integer representation of the bit string $r[1{:}\min(\rho, \lfloor \log_2(q) \rfloor)]$ leads to the smallest security loss—$q \cdot 2^{-\min(\rho, \lfloor \log_2(q) \rfloor)} < 2$ when $2^\rho > q$.

Security-performance tradeoff due to key size. The efficiency of field arithmetic decreases as the number of limbs increases (cf. Sec. 3.2.2), and thus, a key size that requires less limbs is clearly preferable. Accordingly it is sometimes beneficial to reduce the key size from ρ bits to a smaller value $\overline{\rho}$, such that $2^{\overline{\rho}} \leq q$, and then use an

injective key-transform function $f_k : \{0,1\}^{\overline{\rho}} \to \mathbb{F}_q$ that encodes elements of $\mathbb{F}_q$ with fewer limbs. By Prop. 4, this incurs a security-loss factor of $q \cdot 2^{-\overline{\rho}}$ but it can improve performance significantly when the hash function evaluation involves frequent multiplications by the key.

Security-performance tradeoff due to key formatting. In a prime field $\mathbb{Z}_q$, when multiplying one field element with another, performance can be improved by enforcing some formatting on one of the field elements—namely setting specific bit positions to zero. We discuss how key formatting affects the efficiency of field multiplications in Sec. 3.3. This approach improves performance only when combined with specific polynomial-evaluation algorithms. PolyR and Poly1305 are two examples of polynomial universal hash functions that employ key formatting. There are two obvious approaches for transforming a bit string of size ρ into a key in the format-restricted space $\mathcal{R} \subset \mathbb{Z}_q$. The more efficient one is to compose one of the above key encoding functions $f_k : \{0,1\}^{\rho} \to \mathbb{Z}_q$ with a clamping function $\mathsf{clamp} : \mathbb{Z}_q \to \mathcal{R}$ that sets the required bits to zero by using a bit-masking operation. The alternative is to directly encode the key string into an element in $\mathcal{R}$ by stuffing zeros between key bits through bit shifting. This is less efficient but it may incur a smaller security loss, or none, if the key size is already small. In both cases the security loss factor is at most $q \cdot 2^{-\min(\rho, \lfloor \log_2(q) \rfloor) + b}$, where b is the number of bits required to be zero.

In a nutshell, specifying a polynomial universal hash function over a prime field with a key space of reduced size or restricted format has one main advantage: speed. This advantage comes, however, with several drawbacks. The first one is the security decrease that may be substantial compared to the same polynomial without key restrictions. The second drawback is the limited range of implementation choices that can exploit the restrictions for performance. Indeed, for the classical polynomial, only Horner's algorithm would benefit from them. Other evaluation algorithms (described in Sec. 3.2.4) of the classical polynomial would not, as they use multiplication by a power of the key and not only multiplication by the key itself. Moreover, for key formatting, only the Schoolbook multiplication algorithm in a specific radix representation would show performance gains. Key size reductions and key format restrictions are thus design choices that would benefit primarily only specific types of hardware.

Data-to-Field Encoding. The goal of this encoding is to transform the domain of the polynomial hash from a tuple of field elements to a string. Thm. 1 says that if we compose an ϵ-almost universal hash function with domain $\mathcal{M}$ with an injective function $\textsc{Inj} : \mathcal{M}' \to \mathcal{M}$ we obtain an ϵ'-almost universal hash function over the domain $\mathcal{M}'$ where $\epsilon'(M, M') = \epsilon(\textsc{Inj}(M), \textsc{Inj}(M'))$. Thus, from a universal hash function over $\mathbb{F}_q^n$, we can construct a fixed-input-length universal hash function over the domain $\{0,1\}^{\eta}$, where $2^{\eta} \leq n \cdot q$. Alternatively, from a universal hash function over $[\mathbb{F}_q]^*$, we can construct a variable-input-length universal $\textsc{Inj} : \{0,1\}^* \to [\mathbb{F}_q]^*$ hash function over $\{0,1\}^*$. Assuming we have settled on a specific finite field, we want to find an injective encoding that minimises the number of output field elements for any input string of a given length. More field elements imply a polynomial of higher degree, meaning that more arithmetic operations will be required for its evaluation (cf. Sec. 3.2.4 for the classical polynomial). It also means worse security, since the security bound of the classical polynomial hash depends on its maximum degree (cf. Thm. 2, and Tab. 1 for the classical polynomial).

Encoding for fixed-length messages. Let us first consider the case of fixed-length messages. For this we only need to parse a message $M \in \{0,1\}^{\eta}$ in blocks of size $\mu \leq \lfloor \log_2(q) \rfloor$ as $M_1, \ldots, M_{\lceil \frac{\eta}{\mu} \rceil} \xleftarrow{\mu} M$, and then encode them as field elements, that is,

$$\textsc{Inj}(M) = [M_1]_q, [M_2]_q, \ldots, [M_{\lceil \frac{\eta}{\mu} \rceil}]_q .$$

The most space-efficient choice is to choose a block size of $\mu = \lfloor \log_2(q) \rfloor$. However, when μ is not a multiple of the word size ω or not a multiple of 8, the blocks M_i will not be word- or byte-aligned, resulting in slower memory access. As a general rule, a block size that is a multiple of ω typically results in the speediest memory access. On the other hand, when the biggest multiple of ω less than or equal to $\lfloor \log_2(q) \rfloor$ is much smaller than $\lfloor \log_2(q) \rfloor$, the message will be mapped onto a larger tuple of field elements, resulting in a slower evaluation of the polynomial hash function. In such a case, it is typically better to opt for a blocksize equal to the largest multiple of 8 less than or equal to $\lfloor \log_2(q) \rfloor$, as otherwise the overhead in computation would outweigh the degradation in memory access.

Although not very common in practice, it is worth mentioning two techniques from [KR01] which permit the use of a block size where $\mu = \omega \cdot \left(\left\lfloor \log_2(q) \cdot \omega^{-1} \right\rfloor + 1 \right) > \lfloor \log_2(q) \rfloor$. One technique uses the DoubleTransform, where in-range blocks are encoded as single field elements and out-of-range blocks are mapped onto two field elements. The underlying assumption is that random data out-of-range blocks should be infrequent. A disadvantage of this encoding is that it is more complex to parse, and the data may not be random. The second approach employs a

non-injective randomised encoding through the use of an auxiliary random value. As this encoding is non-injective it requires a separate security analysis.

Encoding for variable-length messages. In order to adapt the classical polynomial to handle variable-length messages we have two main approaches depending on what structure is already imposed on its domain. As already noted in Sec. 3.1.1, if the highest-degree coefficient is restricted to be a non-zero field element then the classical polynomial is (Δ-)universal over tuples of variable number of field elements. One way to realise this is to set the highest degree coefficient to a constant field element and embed message blocks in the remaining coefficients. In that case we only need to ensure that our encoding is injective, especially over strings of different lengths that map to the same number of field elements. One such injective encoding is the stop-bit encoding, where the message is parsed into blocks of size $\mu \leq \lfloor \log_2(q) \rfloor$ where only the last block is allowed to be shorter than μ, prepend the last block with a '1' as a marker and translate every block into a field element. When $\mu + 1 \leq \lfloor \log_2(q) \rfloor$, this approach can be applied directly as follows

$$\textsc{Inj}(M) = [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil - 1}]_q, [1 \| M_{\lceil \frac{|M|}{\mu} \rceil}]_q \,.$$

If on the other hand $\mu = \lfloor \log_2(q) \rfloor$, $\textsc{Inj}(M)$ needs to be redefined as

$$= \begin{cases} [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil}]_q, 1 & \text{if } \mu \text{ divides } |M|, \\ [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil - 1}]_q, [1 \| M_{\lceil \frac{|M|}{\mu} \rceil}]_q & \text{otherwise.} \end{cases}$$

When the domain of the polynomial is of the form $[\mathbb{F}_q]^* \times \{0\}$ or $[\mathbb{F}_q]^*$ we can still adapt it to handle variable-length messages through a suitable choice of encoding. For the case where $\mu + 1 \leq \lfloor \log_2(q) \rfloor$ we can extend the prior approach by prepending a '1' bit to every block, i.e.

$$\textsc{Inj}(M) = [1 \| M_1]_q, \ldots, [1 \| M_{\lceil \frac{|M|}{\mu} \rceil - 1}]_q, [1 \| M_{\lceil \frac{|M|}{\mu} \rceil}]_q, 0 \,.$$

Note that the above encoding sets the highest-degree coefficient to be non-zero while simultaneously padding the last block in an injective way. Indeed this is the encoding used in Poly1305. Alternatively if $\mu = \lfloor \log_2(q) \rfloor$ we can pad the last block with zeros and append a block containing the length of the message as follows

$$\textsc{Inj}(M) = [M_1]_q, \ldots, [M_{\lceil \frac{|M|}{\mu} \rceil} \| 0^*]_q, [|M|]_q \,.$$

The justification of this last case is given in Sec. 3.1.4.

Security-Performance Tradeoff due to Block Size and Block Formatting. Similar to the case of key-to-field encoding, we can restrict the block size (i.e., use a smaller μ) and use block formatting to speed up field multiplication. To the best of our knowledge, we are the first to propose this optimization. The performance benefits are restricted to specific implementation choices, such as the two-level implementation of the classical polynomial (cf. Sec. 3.2.4). However, the security/efficiency tradeoff at play here differs from that for key size reduction and key format restriction. The performance gain is hindered by the lower data processing rate (i.e. a data tuple of bigger size). On the other hand, the decrease in security is less significant as it is only linear compared to the exponential degradation resulting from key size reduction and key formatting. This is due to the security bound $\frac{d(M,M')}{q} \cdot \frac{q}{2^\rho}$ of Thm. 2 combined with Prop. 4, which is inversely proportional to the block size and an exponential term in the key size.

As evidenced from the above discussion, the central parameter of the injective encoding affecting the efficiency and security of the hash is the block size μ which must be chosen in accordance with the word size ω and the field size q, which we discuss next.

3.1.7 Choice of Finite Fields

The polynomial universal hash functions constructed in Secs. 3.1.1 and 3.1.6 are defined over one or multiple finite fields $\mathbb{F}_q$. The format and size of q determine the computational aspects of the field $\mathbb{F}_q$, and in turn the security and efficiency of the polynomial hash function.

Format of q. A random prime q does not typically yield efficient arithmetic in $\mathbb{Z}_q$. While integer additions and multiplications are straightforward operations, modular reduction by a random prime is costly. Montgomery reduction [Mon85] is the standard modular reduction algorithm used when dealing with generic moduli. However, more straightforward and efficient modular reduction algorithms are known for special primes such as Mersenne primes of format $q = 2^\pi - 1$, Crandall primes [Cra92] of format $q = 2^\pi - \theta$ (for small θ), primes used in MMH and Square Hash of format $q = 2^\pi + \theta$, and Solinas primes [Sol99] of format $q = 2^{m \cdot d} - \sum_{i=0}^{d-1} c_i 2^{i \cdot m}$ where m is in general a multiple of the word size ω. For Crandall and $2^\pi + \theta$ primes, the choice of θ affects the efficiency of the modular reduction. It is usually chosen small, and such that multiplication by θ can be implemented efficiently (e.g. when θ is close to a small power of 2). Handling primes of the type $2^\pi + \theta$ is more complex than Crandall primes. Regarding their efficiency, at the same size, Mersenne primes have a faster reduction algorithm than Crandall and $2^\pi + \theta$ primes, which are themselves faster than Solinas primes. However, in terms of scarcity, the order is inverted. Notably, there are only twelve Mersenne primes less than 2^{512}, limiting the flexibility in choosing them. We give detailed implementation algorithms for Mersenne and Crandall prime arithmetic in Sec. 3.2.2, giving conditions for when θ is small enough to prevent overflow during modular multiplication. Modular reduction with Crandall and $2^\pi + \theta$ primes is slower than Mersenne primes due to the need for additional multiplications by θ. Yet, for polynomial evaluation algorithms that use several modular multiplications (simultaneous multiplication and modular reduction) by the same operand, most of these multiplications by θ can be computed only once and reused across the same operand, reducing the efficiency advantage of Mersenne primes for these polynomials. Most of the classical polynomial evaluation algorithms over Crandall prime fields can benefit from this trick. However, the benefit is smaller when a Solinas prime field is selected.

To summarize, for a specific prime size π, one would usually choose in order of preference a Mersenne, a Crandall, a $2^\pi + \theta$ or a Solinas prime, depending on their existence. The performance loss from one type of prime to the next can be small depending on the polynomial evaluation algorithm. When this performance loss is non-negligible, it might sometimes be preferable to choose a different prime size π for which there exists a better prime. However, as we will now describe, a different prime size π also comes with additional security/efficiency concerns.

Size of q. As already discussed in previous sections, the size of q (i.e. $\log_2(q)$) directly affects the security and efficiency of polynomial constructions over $\mathbb{F}_q$. Regarding security, polynomials defined over larger fields $\mathbb{F}_q$ usually provide better security—a property inherited from Thm. 2 which applies to all polynomial hash functions. However, this is not universally true as the choice of encoding for the key, data message and tag can degrade security (cf. Sec. 3.1.6). It is impossible for a univariate polynomial hash function to obtain better security than the tag size τ and the key size ρ. Reducing security through a particular choice of encoding is typically done to reduce randomness requirements, improve efficiency, or reduce bandwidth/storage consumption.

The efficiency of polynomial evaluation algorithms often depends on the tuple size of the data input and the efficiency of field additions and field multiplications (cf. Sec. 3.2.4). Specifically, the number of field additions and multiplications is linear in this tuple size. The tuple size is usually smaller for bigger fields, as bigger fields allow encoding more data in a single field element through a bigger block size μ (cf. Sec. 3.1.6). Specifically, a message of η bits is usually encoded in a tuple of length $\lceil \eta \cdot \mu^{-1} \rceil$. Thus, for polynomial evaluation algorithms, bigger fields $\mathbb{F}_q$ often mean fewer field operations. However, it is shown in Sec. 3.2.1 that bigger fields $\mathbb{F}_q$ also increase the number of limbs ℓ required to represent a field element and in Sec. 3.2.2, that the complexity of a finite field multiplication in terms of machine instructions increases quadratically with ℓ. Therefore, an increase in the size q often results in a polynomial evaluation with fewer field operations, but for which the field operations are more costly in terms of machine instructions. Overall, this results in a decrease in performance as both μ and ℓ are usually proportional to $\log_2(q)$ and the number of machine instructions is proportional to $\lceil \eta \cdot \mu^{-1} \rceil \cdot \ell^2$. Choosing a value of $\log_2(q)$ that is optimized to reduce the number of limbs ℓ (or cost of field operations) seems to be more worthwhile than increasing the block size μ. Then, for a specific range of security levels, a good tradeoff choice for $\log_2(q)$ is often to choose the largest one in the range with the smallest number of limbs.

We now consider more fine-grained aspects concerning the size of q. For efficient memory access, the parsing block size μ is ideally chosen to be a multiple of the word size, or less ideally a multiple of 8, and $\mu \leq \lfloor \log_2(q) \rfloor$ (cf. Sec. 3.1.6). Since the efficiency of prime field arithmetic decreases with $\log_2(q)$, for a fixed μ, choosing the smallest value of $\lfloor \log_2(q) \rfloor$ bigger than μ seems to be a good choice to increase efficiency. Thus, choosing the smallest $\lfloor \log_2(q) \rfloor$ slightly bigger than a multiple of the word size or of 8 seems to be, in general, a good design choice to ensure efficient memory access. To uniquely represent an element of $\mathbb{F}_q$ with ℓ limbs of at most λ bits, we need $\lceil \log_2(q) \rceil \leq \ell \cdot \lambda \leq \ell \cdot \omega$. How to make good choices for ℓ and λ is explained in Sec. 3.2.1. Since the efficiency of prime

field arithmetic depends on ℓ, for fixed values of ℓ and λ, choosing the biggest $\lceil \log_2(q) \rceil$ less than $\ell \cdot \lambda$ is good for increasing security without losing efficiency. Thus, selecting the biggest $\lceil \log_2(q) \rceil$ smaller or equal to a multiple of λ (and of ω) is also, in general, an advisable design choice to ensure a good security/efficiency tradeoff. To obtain an even better *balanced* representation, we need to choose $\lceil \log_2(q) \rceil$ to be precisely a multiple of λ. There is a notably different optimal choice of $\lceil \log_2(q) \rceil$ depending on whether we optimize for implementations using saturated (i.e. $\lambda = \omega$) or unsaturated limb representations.

On some architectures, increasing the number of machine instructions can result in code of the same performance due to the ability of processors to process similar independent instructions in parallel (cf. App. A.1). In the case of finite field operations, increasing the size of q may result in an increase in the number of similar independent instructions (cf. Sec. 3.2.2) with the runtime of each field operation staying the same, at the same time producing a polynomial evaluation requiring fewer field operations overall. This would result in a universal hash function with better security and improved efficiency. In the case of the classical polynomial, it is possible to use evaluation algorithms that leverage this parallel instruction capability without increasing the size of q (cf. Sec. 3.2.4).

3.2 Implementation Choices

Implementers of polynomial hash functions are faced with a variety of decisions that will affect the efficiency of their implementation. This section aims to cover these choices systematically along with the associated optimization techniques. Specifically, we cover the field-to-limb encoding required to represent field elements in memory as well as the implementation of the field arithmetic in the chosen representation, including carry-propagation and limb realignment strategies. We then discuss the different strategies for evaluating the classical polynomial, and how these strategies are affected by the other implementation choices.

3.2.1 Field-to-Limb Encoding

For prime fields, we generally represent the field elements by simply treating them as natural numbers and then use the usual ways to represent natural numbers. More precisely, let $B \in \mathbb{N}$ be some base and a an n-digit number in base B. That is, a can be represented as $\sum_{i=0}^{\ell-1} a_i B^i$, with $a_i < B$. In practice, our B is usually a power of two less or equal to 2^ω. Each integer a_i is called a limb, and ℓ is the number of limbs of a.

The types of encoding of field elements to integer limbs can then be classified based on two independent parameters. The encoding can be saturated, meaning that all bits in the integer limbs are used to uniquely represent a field element, or unsaturated, where we use strictly less than ω bits for each limb to represent each field element uniquely. Furthermore, the encoding can be balanced, meaning the unique representation uses the same number of bits in every limb, or unbalanced, where the number of bits per limb used in the unique representation varies per limb.

Saturated Representation. In a saturated representation, we have $B = 2^\omega$. To implement a saturated representation, we have to take care of carries that may occur during additions (including the additions required in a multiplication as discussed in Sec. 3.2.2). This can be implemented most efficiently when the target processor architecture provides an *add-with-carry* instruction (see App. A.1), which is the case on modern x86 and Arm architectures.

In general, a saturated representation introduces sequential dependencies between the additions since the carry status of the n-th addition is required as input for the $n+1$-th addition. This can significantly reduce processor pipeline utilization and, thus, overall processor utilization. On Modern x86-64 processors, we can instead use the ADCX and ADOX instructions, as done by [BNS24]. This allows for two parallel carry chains and thus increases utilization. To our knowledge, this instruction is not yet automatically used by non-specialized compilers and, therefore, requires manual assembly programming.

Unsaturated Representation. In an unsaturated representation, our basis consists of a set of $B_i = 2^{\lambda_i}$ with $\lambda_i < \omega$. When the λ_i are chosen properly, this allows field-level addition and multiplication without handling any intermediate carries and only carrying all bits once at the end, since up to $\omega - \lambda_i$ bits of carry can be stored in the i-th limb. Since this approach does not rely on the processor status flags, it removes most dependencies between most of the instructions and replaces them with one carry chain at the end. This can significantly increase processor

pipeline utilization. However, accumulating carry bits leads to a non-unique intermediate representation of field elements. For polynomial hashing, this does not pose any issue.

The downside of this approach is that it potentially increases the number of limbs and therefore, the cost of both multiplication and addition (as is evident from the discussion of field arithmetic below). Furthermore, the increased number of limbs, especially on 32-bit architectures, increases the likelihood of register spill,[2] as we not only need more registers due to the reduced word size, but also often have fewer registers to work with on 32-bit architectures.

Additionally, an unsaturated representation requires an explicit transformation from saturated to unsaturated, as field elements are usually implicitly encoded using a saturated representation after the key-to-field and data-to-field encoding (cf. Sec. 3.1.6). In practice, implementers usually combine all the necessary transformations from input data blocks to the final limb representation in one step in order to reduce copy operations and improve pipeline utilization.

(Un-)Balanced Representations. In full generality, unbalanced representations can have different values of λ_i for each limb. In practice, it is usually only the most significant limb whose bit length differs from the other limbs. An unbalanced representation comes with slight performance drawbacks, e.g. fast modular reduction tricks during multiplication usually require multiplication by additional factors (and thus more free limb space) when an unbalanced representation is used. This can be observed from eq. (7) as discussed below. In a saturated representation, the choice between a balanced and unbalanced representation is implicitly made as part of the design once the field size is chosen, since a balanced saturated representation is possible if and only if the wordsize of the CPU evenly divides the field size. When using an unsaturated representation, this opens up a tradeoff. Ideally, the field size is chosen to allow for a balanced representation using a minimal number of limbs on a wide number of target CPU architectures. When this is not done, a balanced representation can require many more limbs than an unbalanced representation. Since the complexity of multiplication depends on the number of limbs (as discussed in Sec. 3.2.2 below), this increase in limb count is usually not worth the slightly reduced cost of reduction and implementation complexity.

Summarizing all options, to uniquely represent an element of $\mathbb{Z}_q$ with ℓ limbs of at most λ bits, we need $\lceil \log_2(q) \rceil \leq \ell \cdot \lambda$ with $\lambda \leq \omega$. For a fixed λ, the smallest choice for ℓ is $\lceil \log_2(q) \cdot \lambda^{-1} \rceil$. Thus, choosing the biggest λ less or equal to ω seems to be a good choice to reduce the number of limbs and increase efficiency without decreasing security. However, for Crandall prime fields, when λ is too close to ω, it is not possible to do delayed carry propagation (cf. Sec. 3.2.3) with this representation. If delayed carry is unnecessary, then $\lambda = \omega$ (i.e. saturated representation) is the best choice to obtain the smallest number of limbs. There is no difference in the number of limbs between saturated and unsaturated representation when $\frac{\lceil \log_2(q) \rceil}{\lceil \log_2(q) \cdot \omega^{-1} \rceil} \leq \lambda$. Tab. 2 shows how large the results of Schoolbook multiplication can become, as a function of the limb index. If delayed carry is necessary, the biggest λ for which the bound in Tab. 2 are smaller than $2^{2\omega}$ (so that there is no overflow) when evaluated with the pair $(\lambda, \ell) = (\lambda, \lceil \log_2(q) \cdot \lambda^{-1} \rceil)$, gives the smallest number of limbs needed for delayed carry.For such a λ, a more balanced representation with this same smallest number of limbs is then obtained using the base $B = 2^{\overline{\lambda}}$, where $\overline{\lambda} = \left\lceil \frac{\log_2(q)}{\lceil \log_2(q) \cdot \lambda^{-1} \rceil} \right\rceil$.

3.2.2 Prime Field Arithmetic

This section summarises the main implementation strategies of prime field operations used in evaluating a polynomial hash function. Previous work by Nath et al. provides an alternative, formal description of these algorithms [NS22].

Addition. In an unsaturated representation, addition can usually be implemented by simply adding the limbs since any potential overflow bits can be stored in the unused upper bits. This enables individual additions to be computed in parallel, increasing the likelihood of fully filling CPU pipelines. Given λ bits per ω limb, we can do $2^{\omega-\lambda} - 1$ additions before we need to do a carry propagation.

A saturated representation does not allow this delay and so requires the use of Add-With-Carry (ADC) hardware instructions to propagate the carries from one limb to the next. This introduces dependencies between the individual instructions, which can reduce CPU utilization depending on the number of available adders. Modern Intel CPUs

[2] Register spill denotes the process when a computation needs more registers to store its working data and thus has to store it in memory instead.

allow for two independent addition chains through the use of two different ADC instructions (see App. A.1). If exploited, this can increase performance by increasing CPU utilization.

Modular Reduction. Efficiently implementing modular reduction is dictated by the chosen prime field size q.Let $R_q(x)$ be the function that maps x to its remainder modulo q. Then, for any Crandall prime $q = 2^\pi - \theta$, any $x \in \mathbb{Z}_q$, and any non-negative integer n, we have by the usual laws of remainder arithmetic that

$$R_q(x \cdot (2^\pi)^n) = R_q(x \cdot \theta^n), \tag{2}$$

which for Mersenne primes simplifies to

$$R_q(x \cdot (2^\pi)^n) = x. \tag{3}$$

Representing a number $z \in \mathbb{Z}$ in base 2^π allows reducing z modulo q via usually k multiplications and additions,[3] where k is the number of digits in the base 2^π representation of z.

Similar approaches can be used for Solinas primes $q = 2^{m \cdot d} - \sum_{i=0}^{d-1} c_i 2^{i \cdot m}$, using the more general form of the above equation.

$$R_q(x \cdot (2^{m \cdot d})^n) = R_q\left(x \cdot \left(\sum_{i=0}^{d-1} c_i 2^{i \cdot m}\right)^n\right) \tag{4}$$

As observed in [Sol99], these computations can be made using linear shift registers in practice.

To fully achieve constant-time modular reduction, the overall arithmetic must be designed to ensure that n is bounded by some fixed number (often 2). In the last step, we arrive at a number in the range $[0, 2q - 1]$. To compute the final reduction, we can subtract q and check if the result is negative.

Multiplication. We have several different algorithms to choose from when implementing finite field multiplication. In eq. (12), we evaluate Karatsuba multiplication. We show there that it outperforms Schoolbook multiplication in terms of total number of simple operations (i.e. integer multiplications, additions, and subtractions) only if we have more than 12 limbs.[4] Theoretically, this point can be lower if we weight multiplications higher than additions, e.g. if we assume the cost of a multiplication to be, on average, 1.5 times that of an addition, Karatsuba is better as soon as we have more than 7 limbs.[5] However, during preliminary testing, we observed that Karatsuba's algorithm does not yield performance benefits for the typical field sizes used in polynomial hash functions. We, therefore, omit a detailed discussion of asymptotically superior algorithms here and focus only on methods directly based on classical Schoolbook multiplication. An overview of the available subquadratic methods can be found in [Ber01].

We can multiply two numbers a, b as follows

$$a \cdot b = \sum_{i=0}^{2\ell-1} \sum_{\ell : \ell : j+k=i} a_j b_k B^i. \tag{5}$$

Given the quadratic complexity of this algorithm, we immediately see the number of limbs as being the most important factor for optimization.

To our knowledge, this naive version of Schoolbook multiplication is usually not implemented. Instead, an interleaved version of modular multiplication is used. Implementers usually use properties of the selected prime to keep the result number bounded and not require double the number of limbs. In the case of a prime field $\mathbb{Z}_q$ with a Crandall Prime $q = 2^\pi - \theta$, we can use a simplified form of eq. (2) for $x \in \mathbb{Z}_q$:

$$x \cdot 2^\pi = x \cdot \theta \mod (2^\pi - \theta) \tag{6}$$

to reduce the number of resulting limbs. Let $B = 2^\lambda$ and $\lambda' = \pi - (\ell - 1)\lambda > 0$. Isolating the powers larger than 2^π we can then rewrite eq. (5) using eq. (6) to obtain:

$$a \cdot b = \sum_{i=0}^{\ell-1} \left(2^\lambda\right)^i c_i \mod 2^\pi - \theta,$$

[3] Since addition and multiplication can overflow, this algorithm has to be applied iteratively.

[4] $c = 0, A = M$ gives $k \approx 3.59198$ in eq. (10) in eq. (12).

[5] $c = 0, 1.5A = M$ gives $k \approx 2.82657$ in eq. (10) in eq. (12).

Limb	Bound
ℓ	$(\ell-1)\cdot(2^\lambda-1)^2$ $+2\cdot(2^{\lambda+\lambda'}-2^\lambda-2^{\lambda'}+1)$
$\ell-1$	$(\ell-1)\cdot(2^\lambda-1)^2$ $+(2^{2\lambda'}-2^{\lambda'+1}+1)\cdot\theta\cdot 2^{\lambda-\lambda'}$
$0<i\leq\ell-2$	$i\cdot(2^\lambda-1)^2$ $+(2\cdot(2^{\lambda+\lambda'}-2^\lambda-2^{\lambda'}+1)$ $+(\ell-i-2)\cdot(2^\lambda-1)^2)\cdot\theta\cdot 2^{\lambda-\lambda'}$

Table 2: Maximum size of schoolbook multiplication limb results.

where each c_i has the form:

$$c_i = \sum_{\ell:\ell:j+k=i} a_j b_k + \sum_{\ell:\ell:j+k=i+\ell} a_j b_k \cdot 2^{\lambda-\lambda'}\cdot\theta\,. \tag{7}$$

While this does not reduce the number of operations, it allows us to store intermediate results in ℓ 2ω-bit integers instead of requiring 2ℓ such integers. During implementation, eq. (7) can be optimized further by factoring out terms independent of the summation indices. Furthermore, for simple polynomials, b_k is usually a limb value obtained from the key and thus constant throughout the hash computation, allowing the precomputation of partial products within the sum. From this, we can also derive some preliminary bounds for our limb choices relative to our prime. For each 2ω-bit integer we use to store the results, we must add ℓ numbers with up to 2λ bits. We then have to make sure that we choose λ in such a way as to have enough space for the $\log_2(\ell-1)$ potential carry bits; Tab. 2 shows more precise bounds on maximum limb sizes after Schoolbook multiplication.

For Mersenne and Solinas primes, we can similarly use eqs. (3) and (4) to directly partially reduce during multiplication. Of these three types of primes, Mersenne are the ones allowing for the fastest multiplication, and Solinas primes require the most additional operations.

Asymmetric schoolbook. Consider two elements $a=\sum_{i=0}^{\ell_1-1} a_i B_1^i$ and $b=\sum_{j=0}^{\ell_2-1} b_j B_2^j$, represented in different bases B_1, B_2. Their product can be evaluated as $a\cdot b=\sum_{i=0}^{\ell_1-1}\sum_{j=0}^{\ell_2-1} a_i b_j\cdot B_1^i B_2^j$, requiring $\ell_1\cdot\ell_2$ limb multiplications as opposed to ℓ^2 limb multiplications when using the same base for both operands. Unsaturated limb representation is used with schoolbook multiplication to allow delayed limb realignment (cf. Sec. 3.2.3), but this may require increasing the number of limbs ℓ. In such cases, asymmetric schoolbook multiplication allows delayed limb realignment while requiring less limb multiplications. Instead of using an unsaturated limb representation for both operands, asymmetric schoolbook uses a saturated representation for one operand and an unsaturated representation for the other. For example in the case of Poly1305 with 64-bit limbs this requires 6 limb multiplications instead of 9. In contrast to key and message formatting (cf Sec. 3.3), asymmetric schoolbook is a pure implementation optimization that does not reduce the security of the construction. However, this comes at the cost of a more complex and less efficient limb realignment strategy that requires more memory.

Squaring. While squaring can be implemented using general multiplication routines, when implementations make heavy use of squaring operations, it can be beneficial to implement optimized squaring operations. The performance benefit comes from dropping repeated multiplications. When using schoolbook multiplication for squaring, the limb multiplications $a_i a_j$ and $a_j a_i$ are evaluated separately—resulting in duplicate computation. A more efficient squaring operation can be obtained as follows: Let $\mathcal{S}_\ell=\{(i,j,h)\mid i,j\in\mathbb{Z}_\ell, h=i+j\}$, $\mathcal{S}_\ell^{\text{ev}}$ denote the set of elements in $\mathcal{S}_\ell$ such that $i\neq j$ and h is even, and $\mathcal{S}_\ell^{\text{od}}$ denote the set of elements in $\mathcal{S}_\ell$ where h is odd. It then follows that for any base B

$$\begin{aligned} a^2 = \sum_{(i,j,h)\in\mathcal{S}_\ell} a_i a_j B^h &= \sum_{k=0}^{\ell-1} a_k^2 B^{2k} + \sum_{(i,j,h)\in\mathcal{S}_\ell^{\text{ev}}} a_i a_j B^h + \sum_{(i,j,h)\in\mathcal{S}_\ell^{\text{od}}} a_i a_j B^h \\ &= \sum_{k=0}^{\ell-1} a_k^2 B^{2k} + 2\sum_{k=0}^{\ell-1}\sum_{0\leq i<k} a_i a_{2k-i} B^{2k} + 2\sum_{k=1}^{\ell-1}\sum_{0\leq i<k} a_i a_{2k-1-i} B^{2k-1}\,. \end{aligned} \tag{8}$$

```
1 : while ∃i ∈ {0, ℓ − 2}.a_i ≥ 2^λ ∨ a_{ℓ−1} ≥ 2^{λ'} :
2 :   for i in {0, ℓ − 2} :
3 :     a_{i+1} = a_{i+1} + ⌊a_i/2^λ⌋
4 :     a_i = a_i  mod 2^λ
5 :   a_0 = a_0 + ⌊a_{ℓ−1}/2^{λ'}⌋ ∗ θ
6 :   a_{ℓ−1} = a_{ℓ−1}  mod 2^{λ'}
7 : endwhile
```

Figure 1: Carry propagation for Crandall prime fields.

Through the above approach, computing a^2 with an ℓ-limb representation only requires computing $\binom{\ell+1}{2} = \frac{\ell^2+\ell}{2}$ limb multiplications. It does introduce two additional multiplications by two, but these can be implemented efficiently using addition or a shift by one bit position. While this does not give an asymptotic improvement, it always leads to less limb multiplications since $\frac{\ell^2+\ell}{2} \leq \ell^2$ for all $\ell \in \mathbb{N}$.

For binary fields, constructed as field extensions of $\mathbb{Z}_2$, we can optimize this further. Now multiplication by 2 corresponds to multiplication by 0, meaning that the second and third summations in equation 8 vanish, and thus

$$a^2 = \sum_{k=0}^{\ell-1} a_k^2 B^{2k}.$$

3.2.3 Limb Realignment and Carry Propagation

Delayed Carry Propagation. While a saturated representation requires the usage of techniques to immediately propagate carries between limbs, an unsaturated representation allows for carry bits to accumulate in the "unused" bits. For a Crandall prime field, this can be achieved using the algorithm in Fig. 1.

Proposition 6. *Consider a number a represented in an unsaturated, unbalanced limb representation, with $\forall 0 \leq i < \ell - 1$, $\lambda_i = \lambda$ and $\lambda_{\ell-1} = \lambda'$. Let $\pi = \sum_{i=0}^{\ell-1} \lambda_i$. We then have:*

$$a < 2^{(\ell-1)\lambda+\omega+1} = 2^{\pi-\lambda'+\omega+1}.$$

The proof of this proposition can be found in App. D.8. By Prop. 6 it follows that for any π-bit Crandall prime with sufficiently small θ and with $\pi > 2\omega$ we have to perform at most one modular reduction during carry propagation. Furthermore, for $\delta = \lceil \log_2 \theta \rceil$ and $j = \min \left\{ k < \ell \mid \sum_{i=0}^{k} \lambda_i > \omega - \lambda' + \delta + 1 \right\}$, we can represent a with at most 1 carry bit in the j-th limb. This gives rise to the improved constant-time algorithm in Fig. 2 that performs carry propagation with partial modulo reduction. In an unsaturated representation, this delayed carry thus allows for $2^{\omega-\lambda-1} - 2$ additions of partially reduced and carry propagated numbers before another carry propagation has to be performed.

Limb Realignment. Due to the nature of CPU multiplication instructions, simple application of the multiplication approaches discussed above yields intermediate results of the form: $\sum_{i=0}^{\ell-1} \left(2^{\lambda_i}\right)^i a_i$, with $a_i \in \mathbb{Z}_{2^{2\lambda_i}}$. Limb realignment essentially boils down to splitting a_i into $a_{i,u} \in \mathbb{Z}_{2^{\lambda_i}}$ and $a_{i,l} \in \mathbb{Z}_{2^{\lambda_i}}$, so that $a_i = a_{i,u} \cdot 2^{\lambda_i} + a_{i,l}$. As with carry propagation, we only discuss the case where the representation is only unbalanced in the most significant limb, i.e. $\forall 0 \leq i < \ell - 1.\lambda_i = \lambda$ and $\lambda_{\ell-1} = \lambda'$. We then add up corresponding $a_{i,u}$ and $a_{i,l}$, i.e. realign based on the following equation:

$$\sum_{i=0}^{\ell-1} 2^{\lambda i} a_i = a_{0,l} + \sum_{i=1}^{\ell-1} 2^{\lambda i}(a_{i,l} + a_{i-1,u}) + a_{\ell-1,u} \cdot 2^{\lambda \ell}.$$

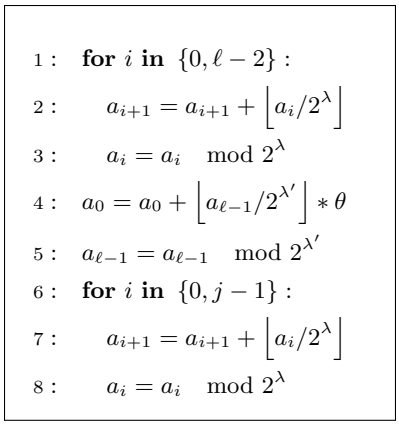


Figure 2: Constant-time carry propagation for Crandall prime fields.

In the final step, we now have to perform a simple modular operation to reduce $a_{\ell-1,u} \cdot (2^{\lambda})^{\ell}$.

This approach can be implemented using the same algorithm as for carry propagation. The major difference between implementing carry propagation and limb realignment is the types of variables involved: carry propagation operates over single words whereas limb realignment operates over a mix of single and double word variables. As with carry propagation, an unsaturated representation allows delaying limb realignment. However, since the cost of multiplication grows quadratically with the number of limbs, one usually uses the largest native integer size for the processor architecture to store each limb. This means double limbs have to be represented by two integers. Thus, the cost of multiplication of double-size limb numbers is four times that of single-size limb integers. Consequently, without limb realignment, the cost of each subsequent multiplication increases by a factor of 4, which can cancel out the performance benefit of delaying limb realignment. This makes delayed limb realignment trickier to exploit, leading to three different approaches:

Immediate Limb Realignment. As the name implies immediate realignment, performs realignment immediately after a double word size number is created, e.g. individual products appearing in the summation of field multiplication. This is excessive: it has no benefits for a saturated limb representation as carries still have to be propagated immediately; similarly, in an unsaturated representation, it would be more beneficial to simply let the carries accumulate on the larger limbs and then perform limb realignment and carry propagation. However, this is not always possible as this requires at least $\lceil \log_2 \ell \rceil$ free bits to store carries. If the chosen representation does not allow this, limbs must be realigned immediately.

Partially-Delayed Limb Realignment. By partially-delayed limb realignment, we mean performing limb realignment as the last operation of a field multiplication. That is, during the field multiplication, we temporarily work with double-sized limbs while adding up intermediate products. As discussed above, this requires choosing a representation that allows for at least $\lceil \log_2 \ell \rceil$ free bits per limb in this double-size representation. More precisely we require $\forall 0 \leq i < \ell.\ \lceil \log_2 \ell \rceil < 2\omega - \lambda_i$. This is the approach that implementers usually choose.

Fully-Delayed Limb Realignment. If we chose a representation that allows for significantly more carries than is required during a single multiplication, the question that arises is why not to delay realignment as much as possible? Depending on the chosen polynomial and the evaluation strategy used, this allows to sum up the results of several field multiplications and only perform limb realignment and carry propagation at the end. Since limb realignment is an operation that is not easily pipelined, reducing limb realignments can impact performance significantly. In particular, when combined with a polynomial evaluation strategy that does not involve a chain of dependent multiplications, we can reduce the number of limb realignments without incurring the aforementioned factor of 4 penalty in limb multiplications. One such case is the naive polynomial evaluation using precomputed key powers (employed in the 2-level polynomial evaluation strategy described below), although it still requires twice the number of limb additions. To our knowledge, this is usually not done in practice when working with prime fields. However, it is sometimes done with binary fields, and its potential benefits for prime fields have been pointed out in [BNS24].

$H_{\text{poly}}(r, (M_1, \cdots, M_n))$

1 : $T \leftarrow M_1$
2 : **for** $i = 2$ **to** n **do**
3 : $\quad T \leftarrow r \cdot T + M_i$
4 : **return** T

Figure 3: Sequential Horner's algorithm of H_{poly}.

$H_{\text{poly}}(r, (M_1, \cdots, M_n))$

1 : **for** $i = 1$ **to** B **do**
2 : $\quad T_{i-1} \leftarrow H_{\text{poly}}(r^B, (M_i, M_{i+B}, M_{i+2B}, \cdots))$
3 : **for** $i = 0$ **to** $B - 1$ **do**
4 : $\quad T_{n+i \bmod B} \leftarrow r^{B-i-1} \cdot T_{n+i \bmod B}$
5 : $T \leftarrow T_0 + T_1 + \cdots + T_{B-1}$
6 : **return** T

Figure 4: B-branch parallel evaluation algorithm of H_{poly}.

3.2.4 Classical Polynomial Evaluation Strategy

Various strategies exist for evaluating the classical polynomial H_{poly}. The naive method consists of calculating powers of the key and then multiplying each message block by one of these powers. For a message of n blocks, it requires roughly $2n$ field multiplications and allows fully-delayed limb realignment (cf. Sec. 3.2.3). If the key powers are precomputed, the remaining multiplications can be done in parallel. However, it is possible to use fewer field multiplications than this. Horner's algorithm, described in Fig. 3, is an online but sequential algorithm that requires only n multiplications. Due to the structure of the polynomial H_{poly}, it is possible to parallelize any evaluation algorithm of H_{poly} into a B-branch parallel algorithm (cf. Fig. 4) at the cost of precomputing the first B powers of the key. Thus, Horner's algorithm can be parallelized by using it as the subprocedure to the algorithm described in Fig. 4. However, neither Horner nor parallel Horner allows for fully-delayed limb realignment when combined with a limb representation filling more than half of the word size (which is typically the case). Another structural characteristic of H_{poly} is the possibility to abstract it as a 2-level polynomial as described in Fig. 5. It was initially proposed in [JL10] to speed up AES-GCM by lowering the number of polynomial reductions, and in [GG15] it was later adapted to Poly1305 to reduce the number of limb realignments .The algorithm processes the message by groups of B blocks through a lower-level polynomial $\overline{H}_{\text{poly}}$. Then, the concatenation of their outputs is fed as input to a higher-level polynomial $\widetilde{H}_{\text{poly}}$ keyed with the initial key to the power B. Any evaluation algorithm of H_{poly} can be used for $\overline{H}_{\text{poly}}$ and $\widetilde{H}_{\text{poly}}$. However, a considered choice allows the 2-level polynomial to benefit from properties of the lower and/or higher level polynomial. Typically, the higher-level polynomial $\overline{H}_{\text{poly}}$ is implemented using Horner or parallel Horner to make the 2-level construction, respectively, online or parallelizable. On the other hand, the lower-level polynomial $\widetilde{H}_{\text{poly}}$ is exclusively implemented using the naive evaluation algorithm in order to allow a fully-delayed limb realignment evaluation. To avoid double-limb multiplications, the first B powers of the key are precomputed and reused across each group of B blocks. Both parallel Horner and this 2-level construction using Horner at the top layer require $n + B$ field multiplications.

When evaluating the classical polynomial over a Crandall prime field using Horner, Parallel Horner or the 2-level implementation we just described, it is possible to further optimize modular multiplication by precomputing some of the limb multiplications corresponding to modular reduction (cf. Sec. 3.2.2). Still, only the last one allows for fully-delayed limb realignment. Both parallel Horner and the 2-level implementation allow for some level of parallelization of field multiplications that depend on the value B. This explains why they perform better than Horner in the benchmarks presented later, as they are better at filling up the CPU pipeline, even though they require more field multiplications.

$H_{\text{poly}}(r, (M_1, \cdots, M_n))$

1 : $n_B \leftarrow n \bmod B$, $n' \leftarrow \dfrac{n - n_B}{B}$

2 : $T \leftarrow \widetilde{H}_{\text{poly}}(r^B, (\overline{H}_{\text{poly}}(r, (M_1, \cdots, M_B)), \cdots,$

3 : $\qquad \overline{H}_{\text{poly}}(r, (M_{(n'-1)\cdot B+1}, \cdots, M_{n'\cdot B}))))$

4 : $T \leftarrow \overline{H}_{\text{poly}}(r, (T, M_{n-n_B+1}, \cdots, M_n))$

5 : **return** T

Figure 5: 2-level evaluation of H_{poly} grouped by B-blocks.

3.3 Key and Message Formatting

Field multiplication of two elements can be sped up by *clamping*, i.e. forcing a predefined set of bits in one of the operands to be zero. Technically, this is a design choice, but it is motivated by a specific implementation strategy and its benefits are limited to this implementation strategy. When the operand is the key, this reduces the keyspace, and when it is a message block, it reduces the block size. The key in Poly1305 was clamped to facilitate efficient implementations on floating-point units [Ber05]. While the use of floating-point units is no longer relevant today, this kind of format restriction can still yield speedups. Specifically, it can limit the size of intermediate limb products so as to allow delayed carry propagation in a saturated representation. The format-restricted operand must be such that it can be represented using a reduced number of (saturated) limbs, leading to fewer arithmetic operations during field multiplication. To our knowledge, this was first applied to Poly1305 in the monocypher library [web] and explained in [Vai]. We give an overview of this technique generalized for $\mathbb{Z}_q$ when q is a π-bit Crandall prime.

A saturated representation of field elements requires $\ell = \lceil \frac{\pi}{\omega} \rceil$ limbs. To ensure proper cache alignment, we further chose the formatted operand to be $\rho = \lfloor \frac{\pi}{\omega} \rfloor \cdot \omega$ bits in size, meaning that it uses either $\ell - 1$ limbs or, when the prime size is multiple of the word size, ℓ limbs. Therefore, we have to perform at most $\ell - 1$ additions of double-word sized limbs (cf. Sec. 3.2.2), requiring $\log_2 \ell$ bits of space to delay the carry. Thus, we can delay the carry only if the $\lceil \log_2 \ell \rceil$ most significant bits of each limb of the formatted operand are set to zero. In the case where our word size does divide the size of the prime, meaning we are in a balanced representation, this number of zero bits is sufficient for delayed carry. This leads to a reduction in the effective formatted operand (key or message) size by $\ell \cdot \lceil \log_2 \ell \rceil = \frac{\pi}{\omega} \cdot \lceil \log_2 \pi - \log_2 \omega \rceil$ bits. If the word size does not divide the prime size, i.e. we are in an unbalanced representation, applying eq. (7) leads to additional space requirements of $\omega - t$ bits per operand limb, where $t = \pi - \rho$. Thus it requires the $\lceil \log_2 \ell + \omega - t \rceil$ most significant bits of each operand limb to be zero. This can lead to significant key space or block size reductions when t is small. Instead, we can use the following equation to perform partial modular reduction:

$$x \cdot 2^{\rho} = x \cdot 2^{-t} \cdot 2^{\pi} = x \cdot 2^{-t}\theta \mod q. \tag{9}$$

If the t least-significant bits of x are zero, then division by 2^t can be done using a right shift by t bits. This motivates setting the $\lceil \log_2 \ell \rceil$ most-significant and the t least-significant bits in each limb of the formatted operand to zero, resulting in a clamping strategy with a smaller security or performance loss than simply applying eq. (7) as in the balanced case.

4 Benchmarking Framework

To assist our exploration of the design space we designed and implemented a framework[6] for rapidly testing different designs and implementations. Our framework follows a modular design reflecting our taxonomy in Sec. 3, with some violations of said modularity where the performance impact was too large to reflect real-world performance. We define a simple grammar for specifying: (a) key size, (b) output size, (c) block size, (d) prime field, (e) limb arrangement, (f) field-arithmetic algorithms, (g) target CPU and instruction set, (h) encoding for key, message, and hash output, (i) single polynomial or a concatenation, and (j) the polynomial evaluation strategy.

[6]The framework is continuously being improved. The most recent version is available here: `https://github.com/jangilcher/polynomial_hashing_framework`

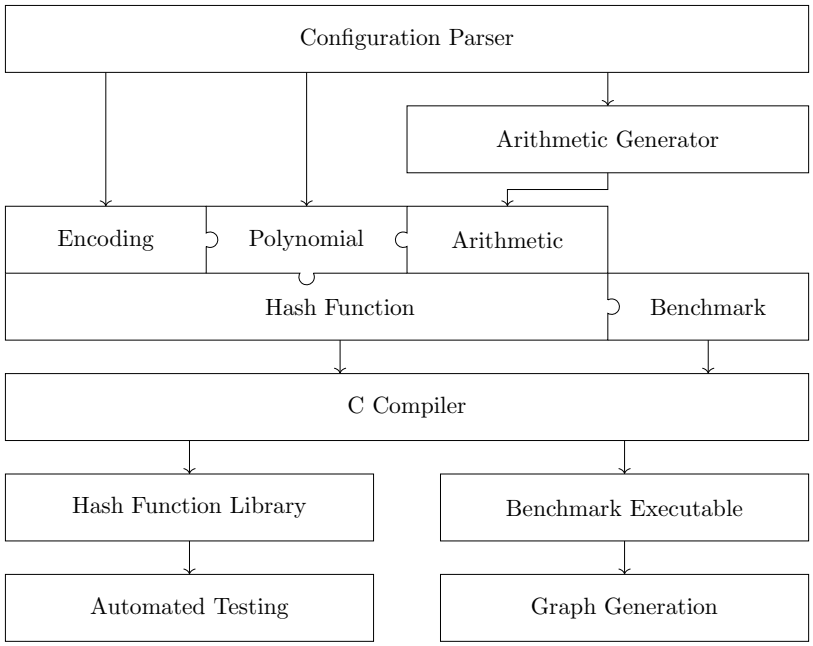


Figure 6: Schematic overview of our framework.

A schematic overview of how our framework is structured is shown in Fig. 6. After parsing the configuration it selects the appropriate C implementations for the polynomial and encoding functions, automatically generates the arithmetic in C and compiles and links these against a simple hash interface as well as a benchmarking suite. As a result we generate both a small library containing our hash function and a benchmarking executable. The arithmetic components of the library are then automatically tested on random data, and the benchmark is executed. In its final step it automatically creates individual and aggregate graphs for each configuration file. We emphasize that at this stage our framework should not be used to assemble production code.

Benchmarking Methodology. We measure the number of cycles, using the x86 instruction `rdtscp`, required to evaluate the hash function on a random message and key. For each data point we measure the total number of cycles when evaluating the hash function 1024 times in succession and then divide the total by 1024. Each hash iteration uses distinct random data that is loaded in the same memory location. This procedure is repeated a user-configurable number of times to compute the average and standard deviation for that sample which we use in the generated plots. We repeat these measurements for varying message sizes. By default, this happens at one-byte increments up to a maximum of 16 KB. Note that the framework does not automatically disable the CPU's dynamic frequency scaling.

Data-to-Field Encoding. Our original intention was to handle data-to-field encoding via an inline function call so as to support all possible types of encoding in a modular fashion. Unfortunately, this degraded performance substantially, most notably when messages are not consumed in a cache-friendly manner. This is because to truly support arbitrary mappings the function call requires an additional copy operation on the message blocks.

To circumvent this extra copy operation, we restricted ourselves to simple "inline" transformations and integrated them into the packing and unpacking operations of the field arithmetic library (which perform the transformation from and to the chosen field element representation). This is justifiable because a realistic and efficient design will naturally restrict itself to some encoding of this type which only adds one or two instructions. In our framework this is limited to the following two (non-exclusive) options:

- prepending or appending a byte to each block,
- applying an XOR-mask to each full limb.

Name	Prime	Message Block, Key & Tag Size	Security Level
Poly1305	$2^{130} - 5$	16 Byte	103
Poly2663	$2^{266} - 3$	32 Byte	245
Poly1743	$2^{174} - 3$	21 Byte	161
Poly1503	$2^{150} - 3$	18 Byte	137
Poly1223	$2^{122} - 3$	15 Byte	117
Poly1163	$2^{116} - 3$	14 Byte	107

Table 3: Parameters of Poly1305 and our designs.

Limitations. There are a number of avenues for expanding our framework's scope in future work. It currently lacks support for automatically implementing field arithmetic as vector operations exploiting the Intel AVX or the Arm Neon instruction set extensions. The currently supported multiplication algorithms are restricted to schoolbook and schoolbook with precomputation of the partial reduction in an unsaturated representation. As previously discussed, we omitted Karatsuba and other sub-quadratic algorithms. The framework uses the bounds displayed in Tab. 2 to automatically adapt the carry propagation so as to avoid integer overflow. To assist with this, a specially instrumented version of the code can be used to dynamically check for integer overflows during testing.

5 New Hash Function Designs

Informed by our survey we propose five new polynomial hash designs that serve as alternatives to Poly1305. Because the performance gains from key clamping only apply to niche use cases, our proposals do not rely on this approach. Our designs target three main categories that offer different tradeoffs between performance and security. The primary parameters of our designs are summarized in Tab. 3.

Very High Security Designs. We present Poly2663, a design that achieves extremely high security. It is intended as an alternative to the approach suggested in [DGGP21], which concatenates two independent invocations to Poly1305, resulting in a similarly high security level. Instead of concatenating two independent instances of Poly1305, we double the field size using the Crandall prime $q_0 = 2^{266} - 3$. By eschewing key clamping, it avoids the 44-bit security loss arising from the double application of Poly1305 in the approach of [DGGP21] and an additional security loss of $\log_2(n)$ bits due to the use of concatenation. At the same time Poly2663 requires less limb operations per message block, thereby resulting in better performance and roughly 25% more bits of security.

Higher Security at Poly1305 Performance. Here, we strive for designs which achieve higher security at a similar performance level to Poly1305. We achieve this by carefully selecting Crandall primes larger than $q = 2^{130} - 5$ which still require the same number of limbs as Poly1305. This results in a better utilization of these limbs. To increase security further, we eschew key clamping. The primes we propose are $q_1 = 2^{174} - 3$ and $q_2 = 2^{150} - 3$, and thus we call the corresponding hash functions Poly1743 and Poly1503. Both of these primes can be represented using a balanced, unsaturated representation on both 32 and 64-bit systems. The primary difference between these two proposals is that Poly1503 can be implemented using exactly the same number of limbs as Poly1305 on both 32-bit and 64-bit systems, whereas Poly1743 uses the same number of limbs on 64-bit systems but requires an additional limb on 32-bit systems. Thus Poly1743 sacrifices 32-bit performance for an additional 24 bits of security. Due to the larger field sizes, messages are split into fewer blocks than in Poly1305, and as a result they even outperform Poly1305 despite using the same number of limbs.

High Performance at Poly1305 Security. Here we strive for designs that improve performance while retaining a comparable security level to Poly1305. Our strategy here is to reduce the prime size so as to reduce the number of limbs, and then compensate for the resulting security loss by forgoing key clamping. Our favoured primes are $q_3 = 2^{122} - 3$ and $q_4 = 2^{116} - 3$, yielding Poly1223 and Poly1163. Analogous to the prior case, Poly1223 achieves maximum security and peak performance on 64-bit processors by sacrificing performance on 32-bit processors, whereas Poly1163

Polynomial	Wordsize	Horner Branches	Degree of Inner Polynomial	Limb Arrangement
Poly1163	32-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[29, 29, 29, 29]
	64-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[58, 58]
Poly1223	32-bit	1, 2, 3, 4, 8	1	[25, 25, 25, 25, 22]
	64-bit	1, 2, 3, 4, 8	1	[61, 61]
Poly1305	32-bit	1, 2, 3, 4, 8	1, 2, 3	[26, 26, 26, 26, 26]
	64-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[44, 44, 42]
Poly1503	32-bit	1, 2, 3, 4, 8	1	[30, 30, 30, 30, 30]
	64-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[50, 50, 50]
Poly1743	32-bit	1, 2, 3, 4, 8	1, 2, 3	[29, 29, 29, 29, 29, 29]
	64-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[58, 58, 58]
Poly2663	32-bit	1, 2, 3, 4, 8	1, 2, 3	[27, 27, 27, 27, 27, 27, 27, 27, 27, 23]
	64-bit	1, 2, 3, 4, 8	1, 2, 3, 7	[54, 54, 54, 54, 50]

Table 4: Parameters for each implementation strategy. A Horner Branch parameter of 1 corresponds to Classical Horner, greater than 1 corresponds to Parallel Horner. *Degree of inner polynomial* refers to the inner polynomial of the 2-level polynomial strategy. Limb arrangements are specified from least significant limb to most significant limb.

achieves slightly lower security but excellent performance on both 32-bit and 64-bit processors. Poly1163 can easily be represented using a balanced, unsaturated representation. For Poly1223, a balanced representation can only be reasonably achieved on 64-bit processors.

5.1 Implementation

We used our framework to implement and benchmark all of our designs as well as Poly1305. For each design, we have at least three variants: a classical Horner's rule implementation, a parallel Horner's rule implementation, and a 2-level polynomial evaluation that makes use of fully-delayed limb realignment. For the latter two options, we ran the experiment with different parameters, i.e. different numbers of parallel branches for parallel Horner and different numbers of blocks processed by the inner polynomial in the 2-level polynomial. An overview of the parameters is given in Tab. 4. While combining a parallel Horner's and 2-level implementation is conceivable, for our implementations we treat these as two distinct, exclusive options.[7] For Poly1305, Poly1503, and Poly1743, we also added a variant with custom field arithmetic that utilizes key clamping as outlined in Secs. 3.1.6 and 3.3. This is not yet automatically generated by our framework. The key clamping scheme used is the same for all three of these alternative variants. However, it is not the one originally used in Poly1305. Instead, we only set bits 60-65 and 123-127 to zero, reducing the security level by only 10 bits. This is therefore denoted as "10-bit Key Clamping" in our benchmarks. Unless otherwise specified, all implementations use a partially-delayed realignment strategy.

5.2 Evaluation

To evaluate our designs we used our benchmarking framework to generate them as discussed in the previous section. Based on the benchmarking methodology outlined in Sec. 4 we ran our experiments on a range of different machines. For our experiments we choose a message size of up to 16 KB, and the measurements for each data point were repeated 15 times. We disabled dynamic frequency scaling and simultaneous multithreading technologies (like Intel Turbo Boost and Hyper-Threading) on all but two machines due to administrative limitations on these machines. The configurations of our machines are outlined in Tab. 5.

[7] Or alternatively the number of Horner branches for a 2-level implementation is always one and conversely the degree of the inner polynomial of a parallel Horner's implementation is always zero.

CPU	Frequency	Turbo Boost	Operating System	Compiler
AMD EPYC 7742	3.4 GHz	Yes	Ubuntu 20.04 LTS	GCC 11.2.0
Intel Xeon Gold 6258R	4 GHz	Yes	Ubuntu 20.04 LTS	GCC 11.2.0
Intel Core i7-8565U	1.8 GHz	No	Debian 12	GCC 12.2.0
Intel Core i7-4790K	4 GHz	No	Debian 12	GCC 12.2.0
Intel Core i7-940	2.93 GHz	No	Debian 12	GCC 12.2.0
Intel Core 2 Duo T5800	2 GHz	No	Debian 12	GCC 12.2.0
Arm Cortex-A76	2.4 GHz	No	Debian 12	GCC 12.2.0

Table 5: Overview of the systems used for evaluation.

CPU	Poly1163		Poly1223		Poly1305		Poly1503		Poly1743		Poly2663	
	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
AMD EPYC 7742	2	3	2	2	2	3	2	2	2	2	2	2
Intel Xeon Gold 6258R	2	2	2	2	2	2	2	2	2	2	2	2
Intel Core i7-8565U	2	2	2	2	2	2	2	2	2	2	2	2
Intel Core i7-4790K	2	2	2	2	2	2	2	2	2	2	2	2
Intel Core i7-940	3	2	4	2	3	2	2	2	2	2	2	2
Intel Core 2 Duo T5800	2	3	2	3	2	2	2	2	2	2	2	2
Arm Cortex-A76	3	8	4	4	3	2	2	2	2	2	2	2

Table 6: Optimal number of branches for parallel Horner's implementations for each of CPU.

App. E contains the benchmarking results for our designs across different CPUs measured in cycles per byte. Note that for each implementation strategy we only depict the performance of the parameter set that resulted in the best performance on that particular CPU, as depicted in Tabs. 6 and 7. For this purpose, we interpret *better performance* in the following way. If the peak throughput of two parameter sets is significantly different, we pick the parameters that require fewer cycles per byte. If there is no significant difference in peak throughput, we pick the parameters for which the convergence to the peak throughput is faster, i.e. that achieves peak throughput at a lower message size. We consider this approach as a realistic performance tuning for the following reasons. Firstly, there are two different kinds of implementers, providers of general-purpose cryptographic libraries such as OpenSSL, and specialized libraries targeting very specific systems such as the internal cryptographic providers of AWS. For the latter, this kind of performance tuning is reasonable — especially at large scale — and already actively being done [KBH+24]. For the former, we observe from Tabs. 6 and 7 that for a specific polynomial implementation targeting a specific word size the best parameters do not vary a lot across different CPUs from the same manufacturer implementing the same architecture (we conjecture that this is due to the similarities in micro-architecture), and in fact are in some cases even remarkably stable even across CPU architectures. As libraries of this category usually already support multiple implementations not only for different CPU architectures, but also for different sets of e.g. instruction set extensions (like AVX2), they might — in cases where the performance impact is large — opt to also support some micro-architecture optimized implementations (e.g. to distinguish between AMD's and Intel's implementation of the x86-64 architecture). While it is possible to pick a parameter that is too large and causes performance degradation, this effect usually stays very small as long as the parameter is still close to the optimal one. As an example, picking the inner polynomial degree to be 2 instead of 1 for a 32-bit implementation of Poly1163 on our Intel Core i7-4790K does not significantly affect the peak performance. It only delays achieving that performance from messages of about 3.5 KB to slightly below 4 KB. However, picking a degree of 7 would result in the implementation requiring about 15% more cycles per byte at peak performance. Thus, with the relatively small spread of optimal parameters, implementers of these libraries should be able to pick reasonable compromise parameters, that work across wide ranges of CPUs with negligible performance differences.

In the following we will now discuss the results for each of our categories defined above, before discussing effects of CPUs across all designs as well as a comparison with state-of-the-art Poly1305 implementations.

High Performance at Poly1305 Security. Looking across Figs. 9, 11, 13, 15, 19 and 21, in the 64-bit case we can clearly observe that Poly1163 vastly outperforms Poly1305 with speed-up of up to 2 at slightly higher security

CPU	Poly1163		Poly1223		Poly1305		Poly1503		Poly1743		Poly2663	
	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
AMD EPYC 7742	2	7	1	1	1	3	1	3	1	3	1	7
Intel Xeon Gold 6258R	2	7	1	1	1	7	1	7	1	7	1	7
Intel Core i7-8565U	2	7	1	1	1	7	1	7	1	7	1	1
Intel Core i7-4790K	1	7	1	1	1	7	1	3	1	3	1	2
Intel Core i7-940	2	7	1	1	1	7	1	7	1	7	2	1
Intel Core 2 Duo T5800	2	7	1	1	1	7	1	3	1	3	3	2
Arm Cortex-A76	2	2	1	1	1	2	1	1	1	1	2	1

Table 7: Optimal degree of inner Polynomial for 2-Level implementations for each of CPU.

levels (see Tab. 3). Poly1223 achieves speed-ups of up to 1.3 in the best case but fails to achieve this on most machines. This is caused by our choice of limb sizes. With 61-bit limbs, the degree to which we can delay the reduction in the Poly1223 implementation is limited compared to Poly1163 and Poly1305. The effect of this limitation becomes more pronounced, the more instruction level parallelism the CPU can handle. In the 32-bit case, as seen in Figs. 10, 12, 14, 16, 20 and 22, we see similar performance for Poly1223 and Poly1305 (which is expected given the identical number of limbs), but a speed-up of about 1.3 for Poly1163.

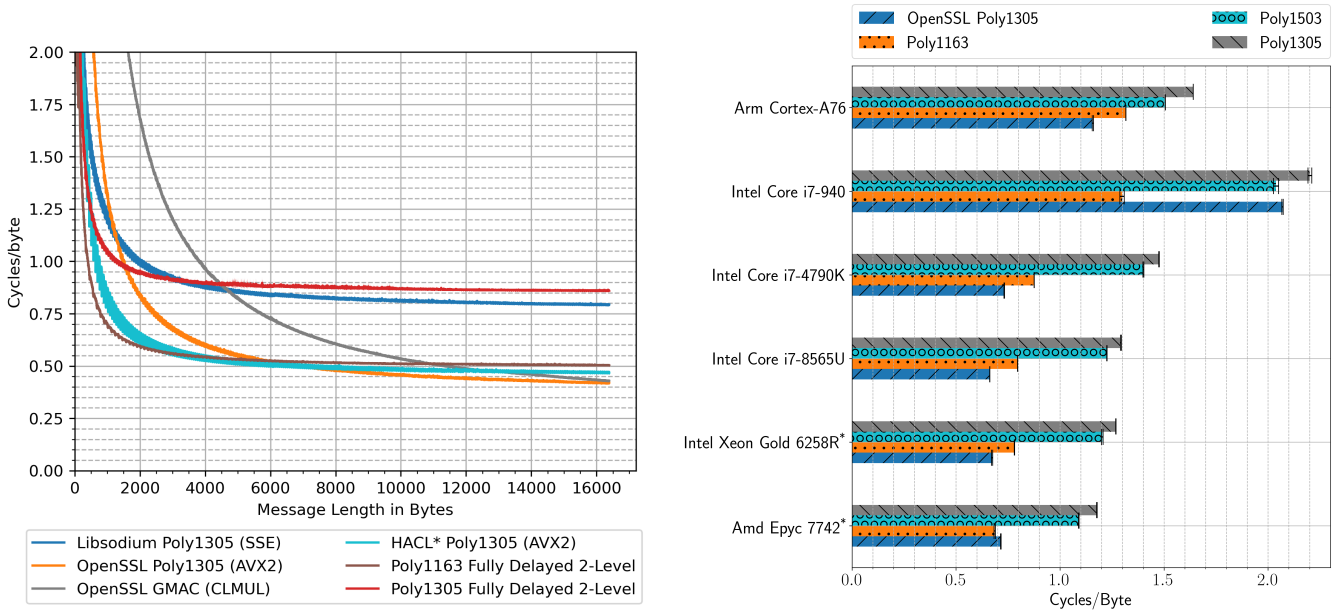
Given the overall better performance of Poly1163 on both modern and older CPUs, Poly1163 seems like a clear best candidate at the current Poly1305 security level.

Higher Security at Poly1305 Performance. For our designs from Sec. 5 we can see a slight speed-up for our designs compared to Poly1305 in the 64-bit case, about 1.1 for Poly1503 and about 1.3 for Poly1743, but with about 50% more bits of security. This is caused by the increased block size and therefore increased processing rate at the same number of limbs for these two designs. For the 32-bit version there is a marginal speed-up for Poly1503 whereas Poly1743 is slightly slower than Poly1305 due to the additional limb. As in the 64-bit case the speed-up of Poly1503 over Poly1305 is caused by the increased processing rate due to the larger field. In summary our data shows that we can therefore not only easily increase the security by about 50%, but even gain performance on both 64- and 32-bit platforms. If 32-bit platforms are of no concern, we can even achieve about 70% higher security with 20% higher performance.

Very High Security Designs. Lastly for our high security design from Sec. 5, Poly2663, we see a significant drop in performance by a factor of about 2 compared to our lower security designs. This is expected, however, since we are working in a field of double the size. In the 32-bit case the cost of Poly2663 becomes prohibitive, as pipeline utilization depends on the number of limbs, not their size. Therefore, 32-bit implementation are likely to reach full CPU utilization at roughly half the field size for the same implementation strategy.

The Case of Arm. If we take a closer look at Figs. 21 and 22 we observe a seemingly surprising behaviour: on Arm the 32-bit implementations consistently outperform their 64-bit counterparts. This is despite generally requiring about twice the number of limbs, which should lead to about four times the number of low level arithmetic operations. The explanation for this behaviour comes down to the exact performance characteristics of the Armv8-A architecture on this specific processor. The Armv8-A architecture has fused multiply-and-add instructions [Arma] that for three 32-bit inputs a, b, c compute the 64-bit value $a \cdot b + c$ in the same number of cycles as a normal multiplication, which allows saving several cycles otherwise spent on additions. However, there is no variant of this instruction taking 3 64-bit inputs and producing a 128-bit output. In fact, to compute the 128-bit result of $a \cdot b$ the lower and upper 64-bit have to be computed via two separate instructions. Moreover, on this specific Arm CPU, the instruction to compute the upper 64-bit takes four times the amount of time as a normal multiplication [Armb]. In combination, these two effects lead to a significant speed-up of the 32-bit version compared to the 64-bit version on this specific Arm processor. While the most modern and high-performance Arm CPUs might come with faster upper 64-bit multiplication, this nonetheless highlights the need for 32-bit support in new designs, as every day appliances are increasingly outfitted with Arm processors with these performance characteristics.

State-of-the-Art Poly1305 Implementations. Fig. 7 compares our fastest Poly1305 and Poly1163 versions to state-of-the-art libraries. Based on Fig. 7a we observe that our implementations are competitive even with



(a) Comparison for varying message length on an Intel Xeon Gold 6258R with Turbo Boost enabled.

(b) Comparison across CPUs for 16 KB messages.
*Numbers adjusted to correct for the effect of Turbo Boost.

Figure 7: Comparison of our proposed designs with various library implementations of polynomial hash functions.

implementations that utilize specialized hardware features like `CLMUL` and vectorized instructions. Specifically for messages below 8 KB in size, Poly1163 outperforms OpenSSL's AVX2 implementation of Poly1305 as well as its GMAC implementation utilizing `CLMUL`, and is roughly equal to the performance of the Poly1305 AVX2 implementation in HACL*. Note, though that this is partially caused by these library implementations converging very slowly towards their performance limit and not reaching it for the message sizes here. On the other hand, 8 KB far exceeds the typical size of network packets. Furthermore, from Fig. 7b we observe that our implementations outperform OpenSSL when features like AVX2 are not available, as is the case on the Intel Core i7-940. However, even in the presence of AVX2 our designs can still outperform existing implementations, when the effects of pipeline utilization are more impactful than vectorization, as on the AMD EPYC 7742. We conclude that a well-crafted AVX2 implementation of Poly1163 is likely to strongly outperform the existing state-of-the-art.

Summary. Looking across high level specifications, we see that, for all our polynomials, the impact of increasing pipeline utilization is to produce large performance improvements for parallel Horner. As an implementation strategy, parallel Horner is outperformed by the fully delayed reduction strategy, which is due to the fact that reduction and limb realignment together act as a soft "barrier", since these operations introduce sequential dependency between the computations. While fully delayed limb realignment generally produces better peak performance, it performs worse for smaller messages than parallel Horner does.

Furthermore, we note that 32-bit implementations are still relevant as our Arm data shows that on this platform the 32-bit versions outperform their 64-bit counterparts. In addition, comparing between our 32-bit and 64-bit implementations, we can see the importance of considering possible implementation strategies as some of our 32-bit implementations might be limited by the available space in the limbs when considering a 2-level implementation strategy (cf. Tab. 7).

In general, our designs and implementations are competitive with state-of-the-art implementations as deployed in commonly used libraries like OpenSSL as well as those reported in [BNS24]. Overall, we find that these benchmarks validate our intuition to a good degree, and in particular, our proposed designs achieve the expected gains. Notably, the best implementations of Poly1163 perform twice as fast as the best implementation of Poly1305 on a 64-bit processor. Alternatively, Poly1743 and Poly1503 increase the security by 34 bits and 58 bits respectively without compromising on performance. For even higher security, Poly2663 or a concatenation of Poly1223 are both far better options than a concatenation of Poly1305 (as proposed in [DGGP21]), both being twice as fast as the concatenation

option. Indeed, we believe that these gains are substantial enough to warrant replacing Poly1305 in ChaCha20-Poly1305 with one of our designs. Since they follow the same structure and provide the same type of security, minimal surgery would be required.

6 Conclusion

Our systematization of knowledge has enabled us to design five new polynomial hash functions, four of which easily outperform Poly1305 in both security and performance and the fifth which offers a very high security hash function that can compete with Poly1305 on 64-bit platforms. With our benchmarking framework, we were able to not only easily evaluate several different designs but were also able to evaluate different implementation strategies and find optimal parameters for these strategies. Furthermore, we can easily run our benchmarking framework on different systems to guide us towards the best implementation strategies for that specific system. Concretely, we propose Poly1163 and Poly1503 as drop-in replacements for Poly1305, both of which are able to outperform Poly1305 at higher security on both 32- and 64-bit systems. Additionally, we further propose three designs that drop 32-bit support in favor of higher security. Our work highlights both the ongoing relevance of 32-bit support for good performance on many Arm processors, as well as the importance of considering various implementations strategies when choosing the parameters that define our hash functions.

Future Work. In future work we plan to investigate more complex polynomial constructions such as BRW-polynomials that can be used to further increase security bounds at a given performance level. For this we also plan to extend our discussion to binary fields, and the specialized hardware instructions necessary to efficiently implement them.

A further future direction is the consideration of vectorized instructions such as AVX. Some of these instruction sets come with additional limitations e.g. in regard to the maximum bit length supported by the vectorized multiplier.

Our current approach is limited to using a large amount of fuzz-testing for correctness in combination with dynamic analysis via ctgrind to ensure the absence of timing-based side channels. Therefore, another open problem for our framework is automatically proving the formal correctness of the generated implementations as is done e.g. in [EPG+19]. There, the code generator only provides the field arithmetic itself without addressing the polynomial construction nor aspects like delayed reduction and the additional space requirements in the limbs that supporting this optimization requires.

References

[Amd] *AMD64 Architecture Programmer's Manual*. `https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5`. 33

[Arma] *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. `https://developer.arm.com/documentation/den0024/a/Introduction`. 28, 33

[Armb] *ARM Cortex-A76 Software Optimization Guide*. `https://developer.arm.com/documentation/PJDOC-466751330-7215/r4p1/`. 28

[BCZ05] Martin Boesgaard, Thomas Christensen, and Erik Zenner. Badger - a fast and provably secure MAC. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05International Conference on Applied Cryptography and Network Security*, volume 3531 of *LNCS*, pages 176–191. Springer, Berlin, Heidelberg, June 2005. 9

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001. `https://cr.yp.to/papers.html#m3`. 18

[Ber05] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, Berlin, Heidelberg, February 2005. 2, 5, 6, 23

[Ber07] Daniel J. Bernstein. Polynomial evaluation and message authentication. *Unpublished manuscript*, 2007. `http://cr.yp.to/papers.html#pema`. 6, 8

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: Fast and secure message authentication. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 216–233. Springer, Berlin, Heidelberg, August 1999. 10

[BNS24] Sreyosi Bhattacharyya, Kaushik Nath, and Palash Sarkar. Polynomial hashing over prime order fields. *Advances in Mathematics of Communications*, 2024. 6, 8, 16, 21, 29

[CGS17] Debrup Chakraborty, Sebati Ghosh, and Palash Sarkar. A fast single-key two-level universal hash function. *IACR Trans. Symm. Cryptol.*, 2017(1):106–128, 2017. 6, 8

[CHB21] Paul Crowley, Nathan Huckleberry, and Eric Biggers. Length-preserving encryption with HCTR2. Cryptology ePrint Archive, Report 2021/1441, 2021. 10

[CMM24] Matt Campagna, Alexander Maximov, and John Preuß Mattsson. Galois Counter Mode with Secure Short Tags (GCM-SST). Internet-Draft draft-mattsson-cfrg-aes-gcm-sst-03, Internet Engineering Task Force, March 2024. Work in Progress. 5, 12

[Cra92] Richard E. Crandall. Method and apparatus for public key exchange in a cryptographic system, US Patent 5 159 632, October 1992. 15

[CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 106–112, New York, NY, USA, 1977. Association for Computing Machinery. 10

[dB93] Bert den Boer. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security*, 2:65–71, 1993. 6

[DGGP21] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. The security of ChaCha20-Poly1305 in the multi-user setting. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1981–2003. ACM Press, November 2021. 2, 3, 7, 25, 29

[DGGP24] Jean Paul Degabriele, Jan Gilcher, Jérôme Govinden, and Kenneth G Paterson. Universal hash designs for an accordion mode. NIST Workshop on the Requirements for an Accordion Cipher Mode 2024, 2024. 9

[DL20] Sébastien Duval and Gaëtan Leurent. Lightweight macs from universal hash functions. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, pages 195–215, Cham, 2020. Springer International Publishing. 5

[Don17] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *NDSS 2017*. The Internet Society, February / March 2017. 2

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE Computer Society Press, May 2019. 30

[GG15] Martin Goll and Shay Gueron. Vectorization of poly1305 message authentication code. In *Proceedings of the 2015 12th International Conference on Information Technology - New Generations*, ITNG '15, page 145–150, USA, 2015. IEEE Computer Society. 22

[GLL17] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Specification and analysis. Cryptology ePrint Archive, Report 2017/168, 2017. 6

[GS19] Sebati Ghosh and Palash Sarkar. Evaluating bernstein–rabin–winograd polynomials. *Designs, Codes and Cryptography*, 87(2):527–546, 2019. 6

[HK97] Shai Halevi and Hugo Krawczyk. MMH: Software message authentication in the Gbit/second rates. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 172–189. Springer, Berlin, Heidelberg, January 1997. 6

[Int] *Software Developer Manuals for Intel® 64 and IA-32 Architectures*. `https://www.intel.com/content/www/us/en/support/articles/000006715/processors.html`. 33

[JKS94] Thomas Johansson, Gregory Kabatianskii, and Ben Smeets. On the relation between A-codes and codes correcting independent errors. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 1–11. Springer, Berlin, Heidelberg, May 1994. 6

[JL10] Krzysztof Jankowski and Pierre Laurent. Packed AES-GCM algorithm suitable for AES/PCLMULQDQ instructions. *IEEE Transactions on Computers*, 60(1):135–138, 2010. 22

[KBH+24] Dusan Kostic, Hanno Becker, John Harrison, Juneyoung Lee, Nevine Ebeid, and Torben Brandt Hansen. Adoption of high-assurance and highly performant cryptographic algorithms at AWS. Real World Crypto Symposium 2024, 2024. 27

[KR01] Ted Krovetz and Phillip Rogaway. Fast universal hashing with small keys and no preprocessing: The PolyR construction. In Dongho Won, editor, *ICISC 00*, volume 2015 of *LNCS*, pages 73–89. Springer, Berlin, Heidelberg, December 2001. 5, 13

[Kro06] Ted Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, March 2006. 5, 8

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), jan 2019. 12

[LK16] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering*, 6(3):171–185, September 2016. 8

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985. 15

[MV04] David A. McGrew and John Viega. The security and performance of the Galois/counter mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355. Springer, Berlin, Heidelberg, December 2004. 6

[NGR05] Kaisa Nyberg, Henri Gilbert, and Matt Robshaw. Galois MAC with forgery probability close to ideal. June 2005. Public comment on NIST modes of operation, `https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/Comments/general-comments/papers/Nyberg_Gilbert_and_Robshaw.pdf`. 12

[NL18] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. 2

[NS22] Kaushik Nath and Palash Sarkar. Efficient arithmetic in (pseudo-)mersenne prime order fields. *Advances in Mathematics of Communications*, 16(2):303–348, 2022. 17

[Rog95] Phillip Rogaway. Bucket hashing and its application to fast message authentication. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 29–42. Springer, Berlin, Heidelberg, August 1995. 7

[Sol99] Jerome A. Solinas. Generalized mersenne numbers. Technical Report CORR 99-39, 1999. 15, 18

[Sti91] Douglas R. Stinson. Universal hashing and authentication codes. In *Annual International Cryptology Conference*, pages 74–85. Springer, 1991. Preliminary version of [Sti94]. 5, 7

[Sti94] Douglas R. Stinson. Universal hashing and authentication codes. *Designs, Codes and Cryptography*, 4(3):369–380, 1994. 32

[Tay94] Richard Taylor. An integrity check value algorithm for stream ciphers. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 40–48. Springer, Berlin, Heidelberg, August 1994. 6

[Vai] Loup Vaillant. The design of poly1305. https://loup-vaillant.fr/tutorials/poly1305-design. 23

[WC81] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981. 1, 4

[web] monocypher. https://monocypher.org/. 23

A Extended Background

A.1 Relevant Processor Characteristics

In the following, we give a brief overview of modern processor design as it affects the performance of polynomial hash functions. Implementers should independently familiarize themselves with the details of their target architectures, e.g. using the available documentation from CPU manufacturers [Int, Amd, Arma]. Modern processors don't execute every instruction sequentially. Instead each processor core can independently reorder incoming instructions and schedule them to be executed on different pipelines. Figure 8 shows a very simplified overview of this design.

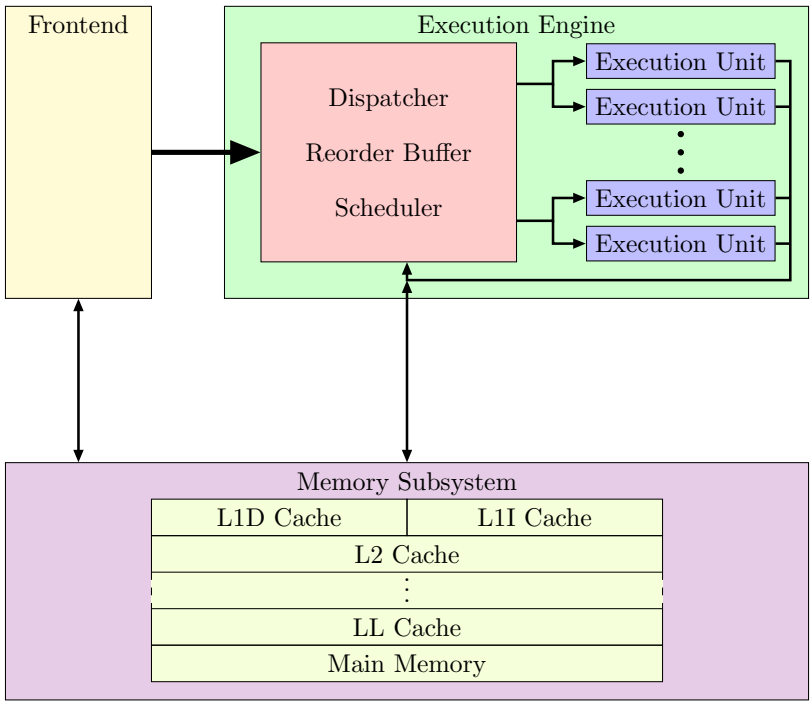


Figure 8: Simplified overview of a modern CPU core.

The frontend fetches the program code from memory and decodes the instructions into a set of micro instructions (µops). They are then sent to the execution engine, which eventually schedules them on the different execution units. The execution units of the processor (e.g. its ALU) are usually grouped into different sets which are available via different execution ports, which execute independently from each other. Overall this results in deep pipelines, and peak throughput is only achieved when these pipelines are saturated. Thus, for peak performance care has to be taken to write code that allows the processor to use this. For example an Intel Skylake processor has 8 execution ports, 4 of which contain an integer ALU.

Memory Subsystem. Figure 8 also shows a very simplified overview of the memory hierarchy. On modern high performance CPUs we are usually dealing with L1D cache sizes of at least 32 KB. This size is divided into a fixed number of cache lines, commonly 64 B in size. The CPU always fetches memory in cache line parts. Therefore it is general beneficial for performance to arrange data locally in memory (and not spread it out). Furthermore it is beneficial to not have data span more cache lines than necessary, as this increases the amount of data that needs to

be read. This can be achieved by properly aligning data in memory, as the location is cache (lines) is derived from the global memory address.

Special Registers and Instructions. Both the Armv8-A and the x86-64 architectures utilize a special CPU *Flags* register. Among others these contain status flags,[8] which are set and cleared as result of executing arithmetic instructions to indicate some additional result of the last instruction that has been executed. Relevant for this work are the *Carry flag*, which is set, if and only if the result of an unsigned-integer arithmetic operation results in an overflow, and the *Overflow flag*, which is set, if and only if the result of a signed-integer arithmetic operation results in an overflow.

In cryptographic implementations their most important use is in special arithmetic instructions. Most notably the *add-with-carry* (ADC) instruction, which adds the value of the carry flag as an additional input to its two operands. This allows implementing integer arithmetic for arbitrary large integers. Modern Intel processors also come with an additional pair of add-with-carry instructions, ADCX and ADOX, which use the carry or overflow flag, respectively, as an additional input to the addition (similar to ADC), but do *not* set the overflow or carry flag, respectively, (unlike ADC which sets *both* the carry and overflow flag). Use of these two instructions allows for two simultaneous carry chains.

B Runtime Analysis of Karatsuba Multiplication

Let $n = 2^k$ be the number of basic integers that represent a long integer (limbs). Let A be the cost of one basic integer addition (or subtraction) and M be the cost of one basic integer multiplication. Then schoolbook multiplication of two $n = 2^k$ digit numbers has a cost of

$$T_S(2^k) = 2^{2k}M + \left(2^{2k} - 1\right) A. \tag{10}$$

The cost of for Karatsuba multiplication of two $n = 2^k$ digit numbers, switching to the Schoolbook method once the numbers reach 2^c digits, is given by the recursion

$$T_K(2^k) = \begin{cases} 3T_k\left(2^{k-1}\right) + 3 \cdot 2^k A & \text{if } k > c \\ 2^{2c}M + \left(2^{2c} - 1\right) A & \text{otherwise.} \end{cases} \tag{11}$$

It follows that

$$\begin{aligned} T_K(2^k) =& 3^{k-c} \cdot 2^{2c} \cdot M + \\ & \left(3^{k-c} \cdot \left(2^{2c} - 1\right) + 3^{k-c+1} \cdot 2^{c+1} - 3 \cdot 2^{k+1}\right) \cdot A. \end{aligned} \tag{12}$$

For brevity we omit the simple induction proof.

C Proof of Thm. 1 (Composition Theorem)

Let $(M_1, M_2), (M_1', M_2')$ be distinct elements of $\mathcal{M}_1 \times \mathcal{M}_2$ and $z \in \mathcal{T}_2$. Assume that H_1 is an ϵ_1-almost universal hash function, and H_2 is an ϵ_2-almost Δ-universal hash function. Further, let $\epsilon_3(\cdot, \cdot)$ be such that for all distinct $M_2, M_2' \in \mathcal{M}_2$ and distinct $T, T' \in \mathcal{T}_1$, $\epsilon_3((M_2, \cdot), (M_2', \cdot)) = 0$ and

$$\Pr_{r_2 \leftarrow\$ \mathcal{R}_2}[H_2(r_2, (M_2, T)) = H_2(r_2, (M_2, T'))] \leq \epsilon_3((M_2, T), (M_2, T'))\,.$$

We want to bound the probability that $H_2(r_2, (M_2, H_1(r_1, M_1))) = H_2(r_2, (M_2', H_1(r_1, M_1'))) + z$. To that end, we consider three cases:

- **Case $M_2 \neq M_2'$.** In this case, clearly $(M_2, H_1(r_1, M_1)) \neq (M_2', H_1(r_1, M_1'))$ and thus we can use the Δ-universality property of H_2 to obtain

$$\begin{aligned} &\Pr[H_2(r_2, (M_2, H_1(r_1, M_1))) = H_2(r_2, (M_2', H_1(r_1, M_1'))) + z] \\ &\leq \max_{r_1 \in \mathcal{R}_1} \epsilon_2((M_2, H_1(r_1, M_1)), (M_2', H_1(r_1, M_1'))). \end{aligned}$$

[8] This is according to Intel and AMD naming conventions. Nomenclature and implementation details differ on Arm platforms. For simplicity we will use Intel's nomenclature.

- **Case $M_2 = M_2'$ and $z = 0$.** As $(M_1, M_2), (M_1', M_2')$ are distinct and $M_2 = M_2'$, we need that $M_1 \neq M_1'$. Depending on the value of r_1 sampled, there are two possibilities, either $H_1(r_1, M_1) \neq H_1(r_1, M_1')$ or $H_1(r_1, M_1) = H_1(r_1, M_1')$. Thus

$$\begin{aligned}
&\Pr[H_2(r_2, (M_2, H_1(r_1, M_1))) = H_2(r_2, (M_2, H_1(r_1, M_1')))] \\
&= \Pr[(H_2(r_2, H_1(r_1, M)) = H_2(r_2, H_1(r_1, M'))) \wedge (H(r_1, M) \neq H(r_1, M'))] \\
&\qquad + \Pr[(H_2(r_2, H_1(r_1, M)) = H_2(r_2, H_1(r_1, M'))) \wedge (H(r_1, M) = H(r_1, M'))] \\
&\leq \Pr[(H_2(r_2, (M_2, H_1(r_1, M_1))) = H_2(r_2, (M_2, H_1(r_1, M_1')))) \wedge (H_1(r_1, M_1) \neq H_1(r_1, M_1'))] \\
&\qquad + \Pr[H_1(r_1, M_1) = H_1(r_1, M_1')] \\
&\leq \max_{r_1 \in \mathcal{R}_1} \epsilon_3((M_2, H_1(r_1, M_1)), (M_2, H_1(r_1, M_1'))) + \epsilon_1(M_1, M_1')\,.
\end{aligned}$$

- **Case $M_2 = M_2'$ and $z \neq 0$.** In this remaining case, when $H_1(r_1, M_1) = H_1(r_1, M_1')$ then

$$z = H_2(r_2, (M_2, H_1(r_1, M_1))) - H_2(r_2, (M_2', H_1(r_1, M_1'))) = 0.$$

  Thus, $H_1(r_1, M_1) \neq H_1(r_1, M_1')$ is the only possibility, and using the Δ-universality property of H_2, we obtain

$$\begin{aligned}
&\Pr[H_2(r_2, (M_2, H_1(r_1, M_1))) = H_2(r_2, (M_2', H_1(r_1, M_1'))) + z] \\
&\leq \max_{r_1 \in \mathcal{R}_1} \epsilon_2((M_2, H_1(r_1, M_1)), (M_2', H_1(r_1, M_1'))).
\end{aligned}$$

As the three cases are mutually exclusive, taking the maximum of the three bounds obtained, gives us the value of $\epsilon((M_1, M_2), (M_1', M_2'))$ such that H is ϵ-almost Δ-universal. When H_2 is only an ϵ_2-almost universal hash function, the proof is similar and excludes the last case.

D Proofs of Sec. 3 (Polynomial Hash Design and Optimization)

D.1 Proof of Thm. 2

Let M, M' be distinct elements of $\mathcal{M}$. If $H(r, M) - H(r, M') = 0$, then r is a root of $P_x(M) - P_x(M')$. As P_x is injective, $P_x(M) - P_x(M')$ is a non-zero polynomial and, by assumption, of degree at most $d(M, M')$. Thus, this polynomial has at most $d(M, M')$ roots. Hence, when sampling r uniformly at random from $\mathbb{F}_q$, the probability that r is a root of $P_x(M) - P_x(M')$ is at most $\frac{d(M,M')}{q}$. Consequently, H is an ϵ-almost universal hash function with $\epsilon(M, M') = \frac{d(M,M')}{q}$.

Let $z \in \mathbb{F}_q$ and assume additionally that the function $G_x : \mathcal{M} \to \mathbb{F}_q[x]$ defined as $G_x(M) = P_x(M) - P_0(M)$ is injective. Note here that $G_x(M)$ is a polynomial with a constant coefficient of zero. As G_x is injective, $G_x(M) - G_x(M')$ is a non-zero polynomial whose constant coefficient is zero. Thus it is a non-zero polynomial of degree at least one and $P_x(M) - P_x(M') - z = G_x(M) - G_x(M') + P_0(M) - P_0(M') - z$ is also a non-zero polynomial of degree at least one. By assumption, $P_x(M) - P_x(M') - z$ is of degree at most $d(M, M')$. Similarly as before, the probability that r is a root of $P_x(M) - P_x(M') - z$ is at most $\frac{d(M,M')}{q}$, and therefore H is an ϵ-almost Δ-universal hash function with $\epsilon(M, M') = \frac{d(M,M')}{q}$.

D.2 Proof of Prop. 1

We first prove the result on H. Let M, M' be distinct elements of $\mathcal{M}$ and $(z_1, z_2) \in \mathcal{T}_1 \times \mathcal{T}_2$. Moreover, let H_1 be an ϵ_1-almost Δ-universal hash function and H_2 be an ϵ_2-almost Δ-universal hash function. Then,

$$\begin{aligned}
&\Pr[H((r_1, r_2), M) = H((r_1, r_2), M') + (z_1, z_2)] \\
&= \Pr[(H_1(r_1, M) = H_1(r_1, M') + z_1) \wedge (H_2(r_2, M) = H_2(r_2, M') + z_2)] \\
&\leq \epsilon_1(M, M') \cdot \epsilon_2(M, M')\,.
\end{aligned}$$

We now prove the result on G. Let M, M' be distinct elements of $\mathcal{M}$ and $(z_1, z_2) \in \mathcal{T}_1 \times \mathcal{T}_2$. Moreover, let H_1 be an ϵ_1-almost Δ-universal hash function and H_2 be an ϵ_2-almost Δ-universal hash function. Then,

$$\begin{aligned}
&\Pr[G(r, M) = G(r, M') + (z_1, z_2)] \\
&= \Pr[(H_1(r, M) = H_1(r, M') + z_1) \wedge (H_2(r, M) = H_2(r, M') + z_2)] \\
&\leq \min(\epsilon_1(M, M'), \epsilon_2(M, M'))\,.
\end{aligned}$$

When H_1 and H_2 are only almost universal hash functions, the proof is similar, with the difference of removing "$+z_1$", "$+z_2$" and "$+(z_1, z_2)$" in the equations.

D.3 Proof of Prop. 2

We only prove the result of almost universality for H as the proofs for Δ-universality and the function G are similar. Let $(M_1, M_2), (M_1', M_2')$ be distinct elements of $\mathcal{M}_1 \times \mathcal{M}_2$, H_1 be an ϵ_1-almost universal hash function and H_2 be an ϵ_2-almost universal hash function. Assume $H((r_1, r_2), (M_1, M_2)) = H((r_1, r_2), (M_1', M_2'))$, then both $H_1(r_1, M_1) = H_1(r_1, M_1')$ and $H_2(r_2, M_2) = H_2(r_2, M_2')$. Thus if $M_1 \neq M_1'$, the probability that $H((r_1, r_2), (M_1, M_2)) = H((r_1, r_2), (M_1', M_2'))$ is bounded by $\epsilon_1(M_1, M_1')$. Similarly, if $M_2 \neq M_2'$ then this probability is bounded by $\epsilon_2(M_2, M_2')$. As $\epsilon_1(M_1, M_1) = 0$ and $\epsilon_2(M_2, M_2) = 0$, we obtain that

$$\Pr[H((r_1, r_2), (M_1, M_2)) = H((r_1, r_2), (M_1', M_2'))] \leq \max(\epsilon_1(M_1, M_1'), \epsilon_2(M_2, M_2')).$$

D.4 Proof of Thm. 3

Let $G : \mathbb{F}_q \times \prod_{i=0}^{n} \mathcal{M}_i \to \mathbb{F}_q$ be a keyed hash function defined as follows. For all $r \in \mathbb{F}_q$ and $(M_0, M_1, \cdots, M_n) \in \prod_{i=0}^{n} \mathcal{M}_i$, $G(r, (M_0, M_1, \cdots, M_n)) = H(r, (M_0, P_x^1(M_1), \cdots, P_x^n(M_n)))$. We want to prove that G is an ϵ-almost universal hash function with

$$\epsilon((M_0, M_1, \cdots, M_n), (M_0', M_1', \cdots, M_n')) = \frac{d(\overline{M}, \overline{M}')}{q}.$$

As $H : \mathbb{F}_q \times (\mathcal{M}_0 \times [\mathbb{F}_q[x]]^n) \to \mathbb{F}_q$ is a polynomial hash function (cf. Thm. 2), there exists an injective function $Q_y : \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n \to \mathbb{F}_q[y]$ that satisfies:

- for any $r \in \mathbb{F}_q$ and $M \in \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n$, $H(r, M) = Q_r(M)$ (i.e. by evaluating the polynomial $Q_y(M)$ at $y = r$),
- for any $M, M' \in \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n$, $Q_y(M) - Q_y(M')$ is of total degree at most $d(M, M')$.

Let $R_y : \prod_{i=0}^{n} \mathcal{M}_i \to \mathbb{F}_q[y]$ be the function defined as follows. For all $(M_0, M_1, \cdots, M_n) \in \prod_{i=0}^{n} \mathcal{M}_i$,

$$R_y(M_0, M_1, \cdots, M_n) = Q_y(M_0, P_x^1(M_1), \cdots, P_x^n(M_n)).$$

Then $G(r, (M_0, M_1, \cdots, M_n)) = R_r(M_0, M_1, \cdots, M_n)$ (i.e. evaluating the polynomial $R_y(M_0, M_1, \cdots, M_n)$ at $y = r$). As Q_y is an injective function and for all i, $P_x^i : \mathcal{M}_i \to \mathbb{F}_q[x]$ are injective functions, the composed function R_y is also injective. Moreover, for all $(M_0, M_1, \cdots, M_n), (M_0', M_1', \cdots, M_n') \in \prod_{i=0}^{n} \mathcal{M}_i$, $R_y(M_0, M_1, \cdots, M_n) - R_y(M_0', M_1', \cdots, M_n')$ is a univariate polynomial in the indeterminate y of degree at most $d(\overline{M}, \overline{M}')$. Thus, using Thm. 2, G is an ϵ-almost universal hash function.

If, in addition, the function $\overline{Q}_y : \mathcal{M}_0 \times [\mathbb{F}_q[x]]^n \to \mathbb{F}_q[y]$ defined as $\overline{Q}_y(M) = Q_y(M) - Q_0(M)$ is injective, using again Thm. 2 applied to R_y, we obtain that G is also ϵ-almost Δ-universal.

D.5 Proof of Prop. 3

For any integer q, let $\mathbb{Z}_q$ be the set of all congruence classes of integers modulo q and, for any integer a, let $[a]_q$ denote the congruence class of a modulo q, i.e., $[a]_q = \{z \in \mathbb{Z} | \exists u \in \mathbb{Z}, a - z = u \cdot q\}$. Then, $f_t : \mathbb{Z}_n \to \mathbb{Z}_m$ is defined as follows: for any $z \in \mathbb{Z}_n$, $f_t(z) = [a]_m$ where a is the only integer in $\{0, \ldots, n-1\}$ such that $z = [a]_n$.

Let $z' \in \mathbb{Z}_m$ and M, M' be distinct elements from $\mathcal{M}$. We want to bound the probability that $H'(r, M) = H'(r, M') + z'$, but first, let assume there exists r such that $f_t(H(r, M)) = f_t(H(r, M')) + z'$. Thus, there exist $a_H, a_{H'} \in \{0, \ldots n-1\}$ and $a_{z'} \in \{0, \ldots m-1\}$ such that $[a_H]_n = H(r, M)$, $[a_{H'}]_n = H(r, M')$, $[a_{z'}]_m = z'$ and

$$[a_H]_m - [a_{H'}]_m = [a_{z'}]_m. \tag{13}$$

Let $Z' = \{z \in \mathbb{Z} | (z = a_{z'} \pmod m) \wedge (-n < z < n)\}$ be the set of integers between $-n$ and n that is congruent modulo m to $a_{z'}$. As this set is finite, there exist only $|Z'|$ distinct values $z_1, \ldots, z_{|Z'|} \in \mathbb{Z}$ that is congruent modulo m to $a_{z'}$. Coming back to eq. (13), over $\mathbb{Z}$, we have that $a_H - a_{H'} = a_{z'} \pmod m$, or said differently, $a_H - a_{H'}$ is in Z'. Thus, there exist $i \in \{1, \ldots, |Z'|\}$ such that $a_H - a_{H'} = z_i$. Switching to $\mathbb{Z}_n$, there exist $i \in \{1, \ldots, |Z'|\}$ such that $H(r, M) - H(r, M') = [z_i]_n$. Using this result, we can therefore claim that

$$\Pr_{r \leftarrow\$ \mathcal{R}}[f_t(H(r, M)) = f_t(H(r, M')) + z'] \leq \Pr_{r \leftarrow\$ \mathcal{R}}\left[\bigcup_{i=1}^{|Z'|} H(r, M) - H(r, M') = [z_i]_n\right].$$

By using a union bound and noting that $|Z'| = \left\lceil \frac{n - a_{z'}}{m} \right\rceil - \left\lceil \frac{1 - n - a_{z'}}{m} \right\rceil \leq \left\lceil \frac{2n-1}{m} \right\rceil = \left\lfloor \frac{2n-2}{m} \right\rfloor + 1$, we then obtain

$$\begin{aligned}
\Pr_{r \leftarrow\$ \mathcal{R}}[f_t(H(r, M)) = f_t(H(r, M')) + z'] &\leq \sum_{i=1}^{|Z'|} \Pr_{r \leftarrow\$ \mathcal{R}}[H(r, M) - H(r, M') = [z_i]_n] \\
&\leq |Z'| \cdot \epsilon(M, M') \\
&\leq \left(\left\lfloor \frac{2n-2}{m} \right\rfloor + 1\right) \cdot \epsilon(M, M').
\end{aligned}$$

D.6 Proof of Prop. 4

Let $z \in \mathcal{T}$ and M, M' be distinct elements from $\mathcal{M}$. Assume that H is an ϵ-almost Δ-universal hash function. Then,

$$\begin{aligned}
&\Pr_{r' \leftarrow\$ \mathcal{R}'}[H'(r', M) = H'(r', M') + z] = \Pr_{r' \leftarrow\$ \mathcal{R}'}[H(f_k(r'), M) = H(f_k(r'), M') + z] \\
&= \sum_{r \in \mathcal{R}} \Pr[H(r, M) = H(r, M') + z \mid f_k(r') = r] \cdot \Pr_{r' \leftarrow\$ \mathcal{R}'}[f_k(r') = r]
\end{aligned}$$

Using the fact that f_k has bias β and $\Pr_{\overline{r} \leftarrow\$ \mathcal{R}}[r = \overline{r}] = |\mathcal{R}|^{-1}$, we obtain

$$\begin{aligned}
&\leq \beta \cdot |\mathcal{R}| \cdot \sum_{r \in \mathcal{R}} \Pr[H(r, M) = H(r, M') + z \mid f_k(r') = r] \cdot \Pr_{\overline{r} \leftarrow\$ \mathcal{R}}[r = \overline{r}] \\
&= \beta \cdot |\mathcal{R}| \cdot \sum_{r \in \mathcal{R}} \Pr[H(r, M) = H(r, M') + z \mid r = \overline{r}] \cdot \Pr_{\overline{r} \leftarrow\$ \mathcal{R}}[r = \overline{r}] \\
&\leq \beta \cdot |\mathcal{R}| \cdot \epsilon(M, M').
\end{aligned}$$

When H is only an ϵ-almost universal hash function, the proof is similar, with the difference of removing "$+z$" in the equations.

D.7 Proof of Prop. 5

Let $\overline{r} \in S_2$. We first consider the case where $f_t(r) = r \pmod m$. If there exists $r \in S_1$ such that $\overline{r} = r \pmod m$, then there exists $k \in \mathbb{Z}$ such that $0 \leq \overline{r} + k \cdot m \leq 2^\rho - 1$. Thus $\frac{-\overline{r}}{m} \leq k \leq \frac{2^\rho - 1 - \overline{r}}{m}$, and there are at most $\left\lceil \frac{2^\rho}{m} \right\rceil$ possible values for k. Therefore, there are at most $\left\lceil \frac{2^\rho}{m} \right\rceil$ possible values for $r \in S_1$ such that $\overline{r} = r \pmod m$, and as r is sampled uniformly at random, we obtain the bound stated.

We now consider the case where $f_t(r) = r \pmod m$. If there exists $r \in S_1$ such that $\overline{r} = \left\lfloor \frac{r \cdot m}{2^\rho} \right\rfloor$, then $\frac{\overline{r} \cdot 2^\rho}{m} \leq r < \frac{(\overline{r}+1) \cdot 2^\rho}{m}$. Therefore, there are at most $\left\lceil \frac{2^\rho}{m} \right\rceil$ possible values for $r \in S_1$ such that $\overline{r} = \left\lfloor \frac{r \cdot m}{2^\rho} \right\rfloor$, and as r is sampled uniformly at random, we obtain the bound stated.

D.8 Proof of Prop. 6

The maximum value a that can be represented using this representation with ω-bit integers is

$$\begin{aligned} a &= \sum_{i=0}^{\ell-1} (2^{\omega}-1)\cdot 2^{i\lambda} = (2^{\omega}-1)\cdot\left(\frac{1-2^{\ell\lambda}}{1-2^{\lambda}}\right) \\ &= \left(2^{\ell\lambda}-1\right)\cdot\left(\frac{2^{\omega}-1}{2^{\lambda}-1}\right). \end{aligned}$$

Since $\omega > \lambda$ it follows that

$$\begin{aligned} a &= \left(2^{\ell\lambda}-1\right)\cdot\left(\frac{2^{\omega}-1}{2^{\lambda}-1}\right) \\ &\leq \left(2^{\ell\lambda}-1\right)\cdot 2\cdot\frac{2^{\omega}}{2^{\lambda}} < 2^{(\ell-1)\lambda+\omega+1}. \end{aligned}$$

The last equality in the proposition follows directly from the definitions of π and λ_i.

E Benchmarks

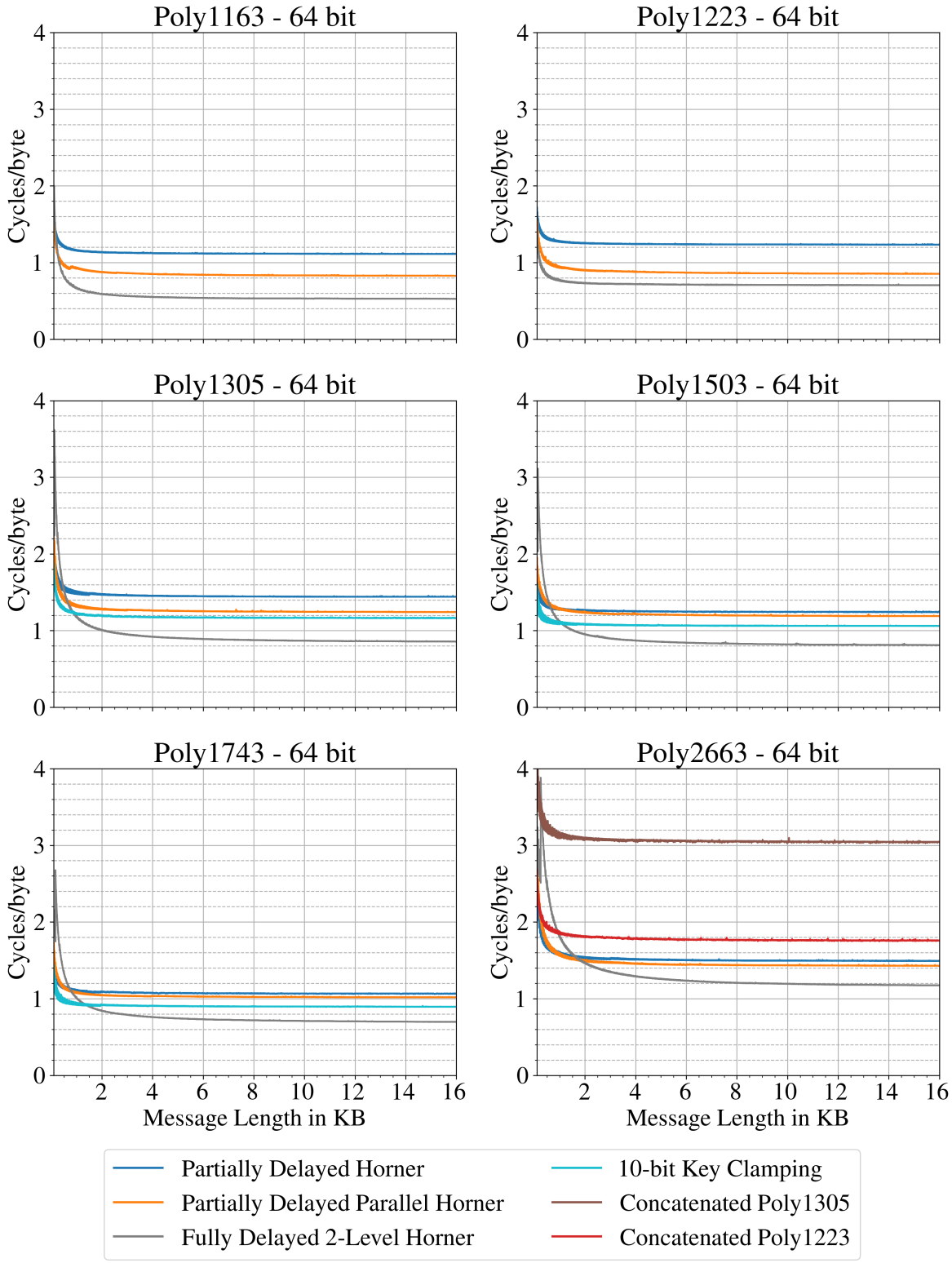


Figure 9: Benchmarking results for 64-bit implementations on Intel Xeon Gold 6258R with Turbo Boost up-to 4 GHz.

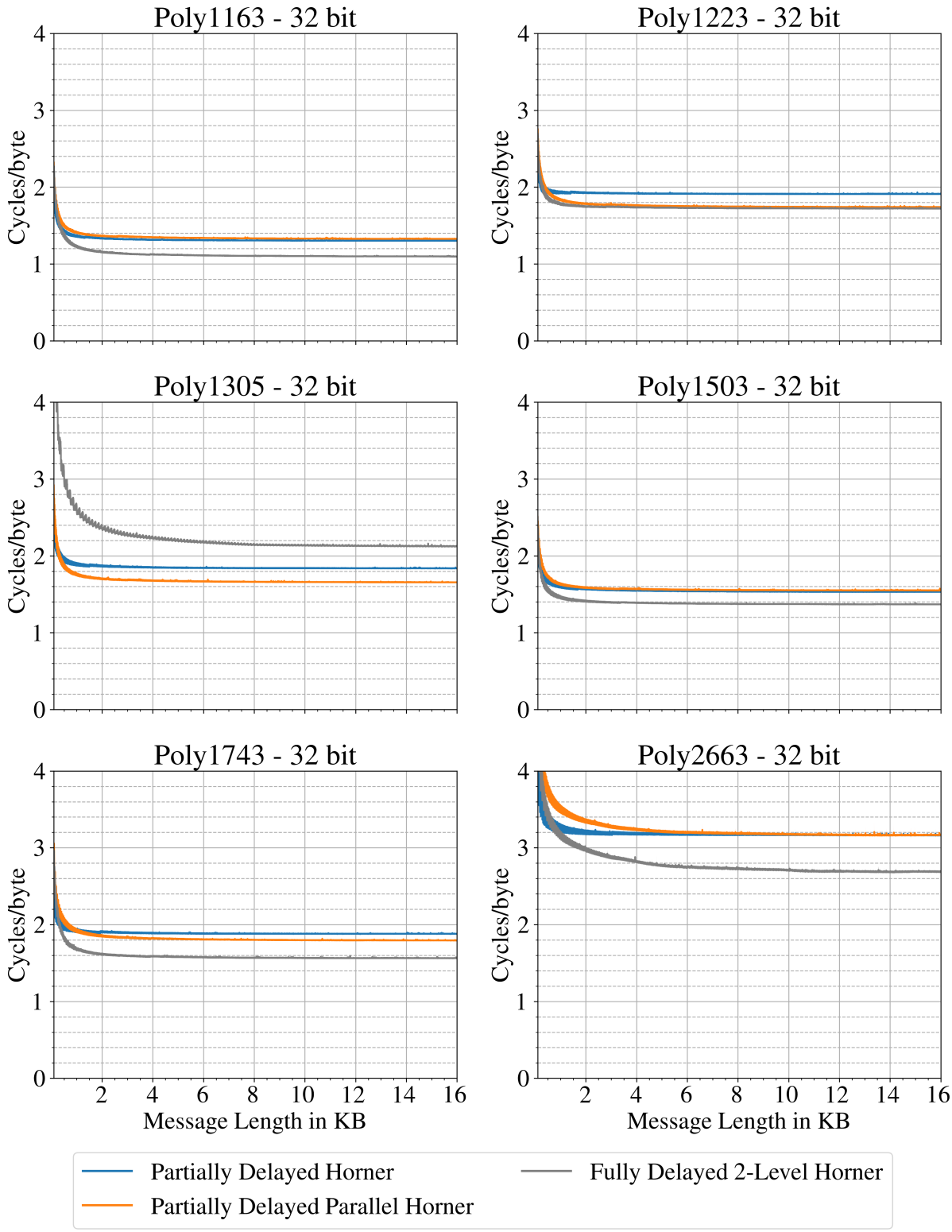


Figure 10: Benchmarking results for 64-bit implementations on Intel Xeon Gold 6258R with Turbo Boost up-to 4 GHz.

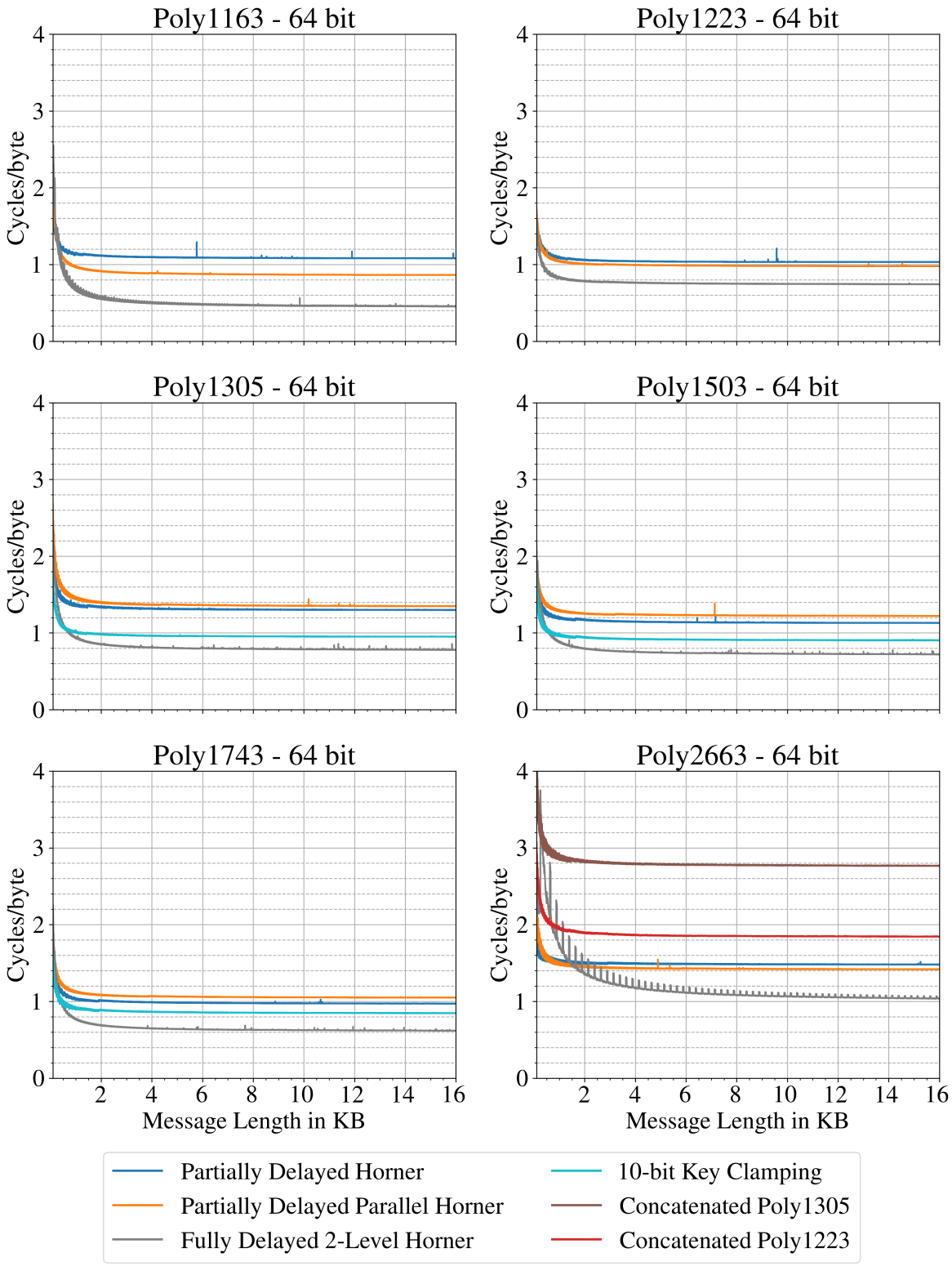


Figure 11: Benchmarking results for 64-bit implementations on AMD EPYC 7742 with Turbo Boost up-to 3.4 GHz.

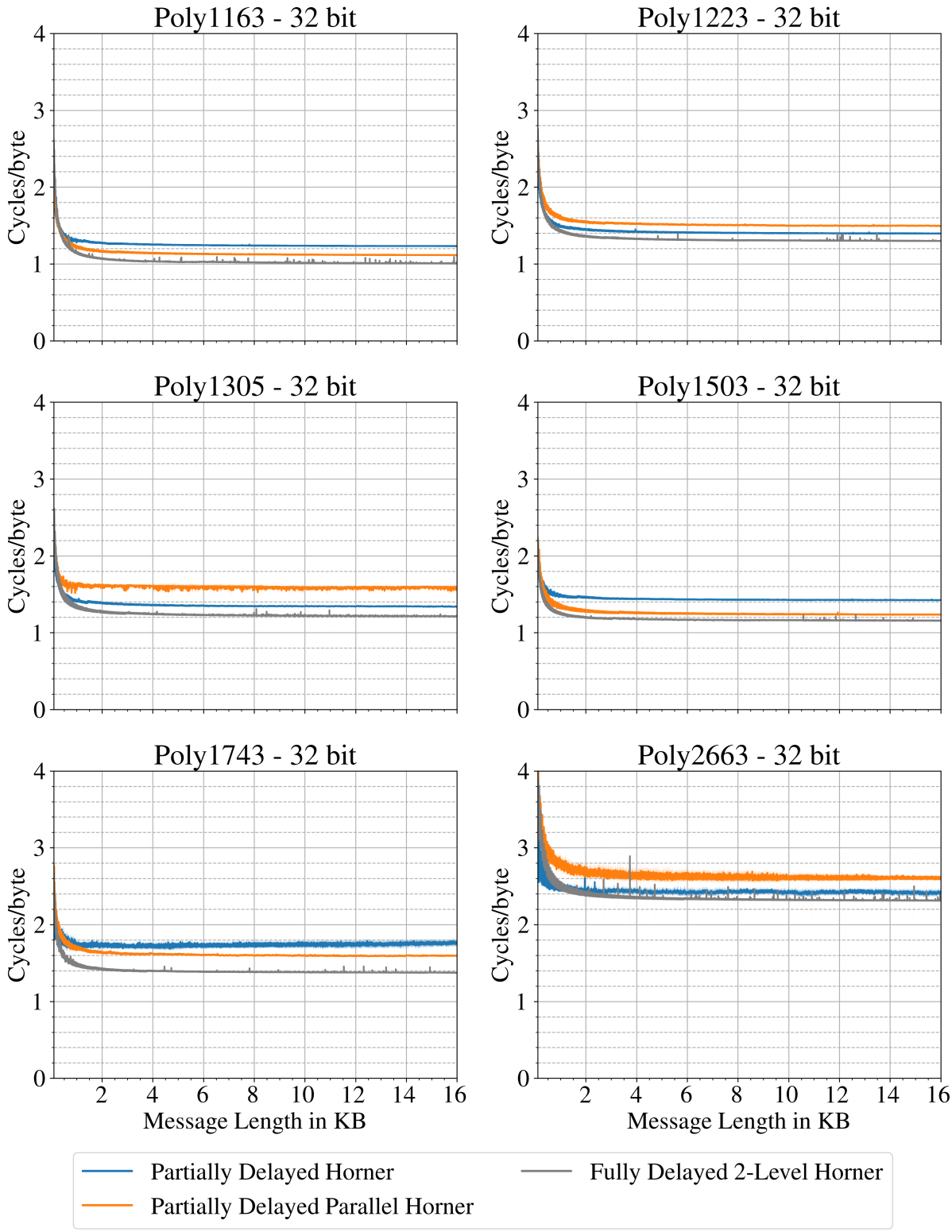


Figure 12: Benchmarking results for 64-bit implementations on AMD EPYC 7742 with Turbo Boost up-to 3.4 GHz.

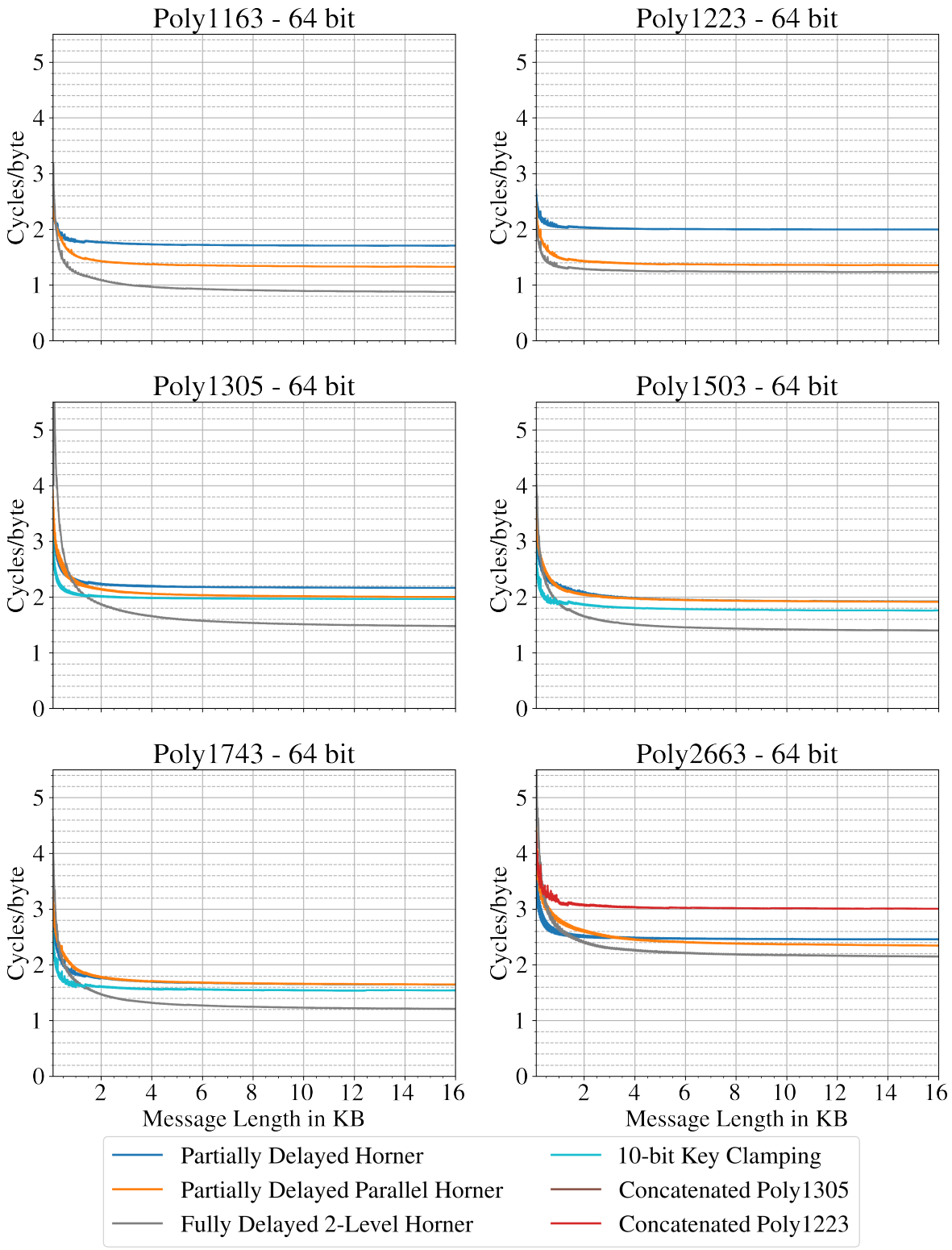


Figure 13: Benchmarking results for 64-bit implementations on Intel Core i7-4790K running at 4 GHz without Turbo Boost.

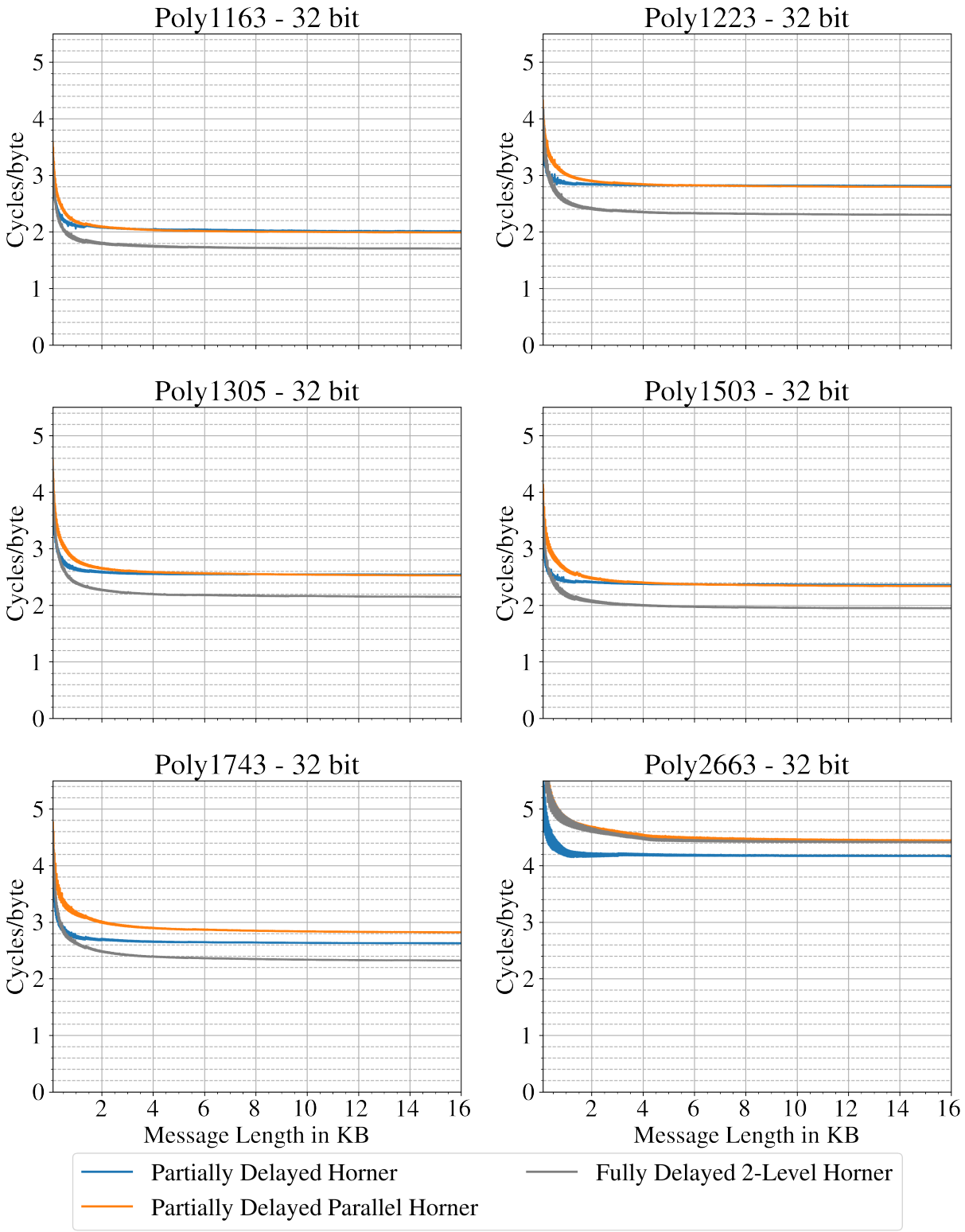


Figure 14: Benchmarking results for 32-bit implementations on Intel Core i7-4790K running at 4 GHz without Turbo Boost.

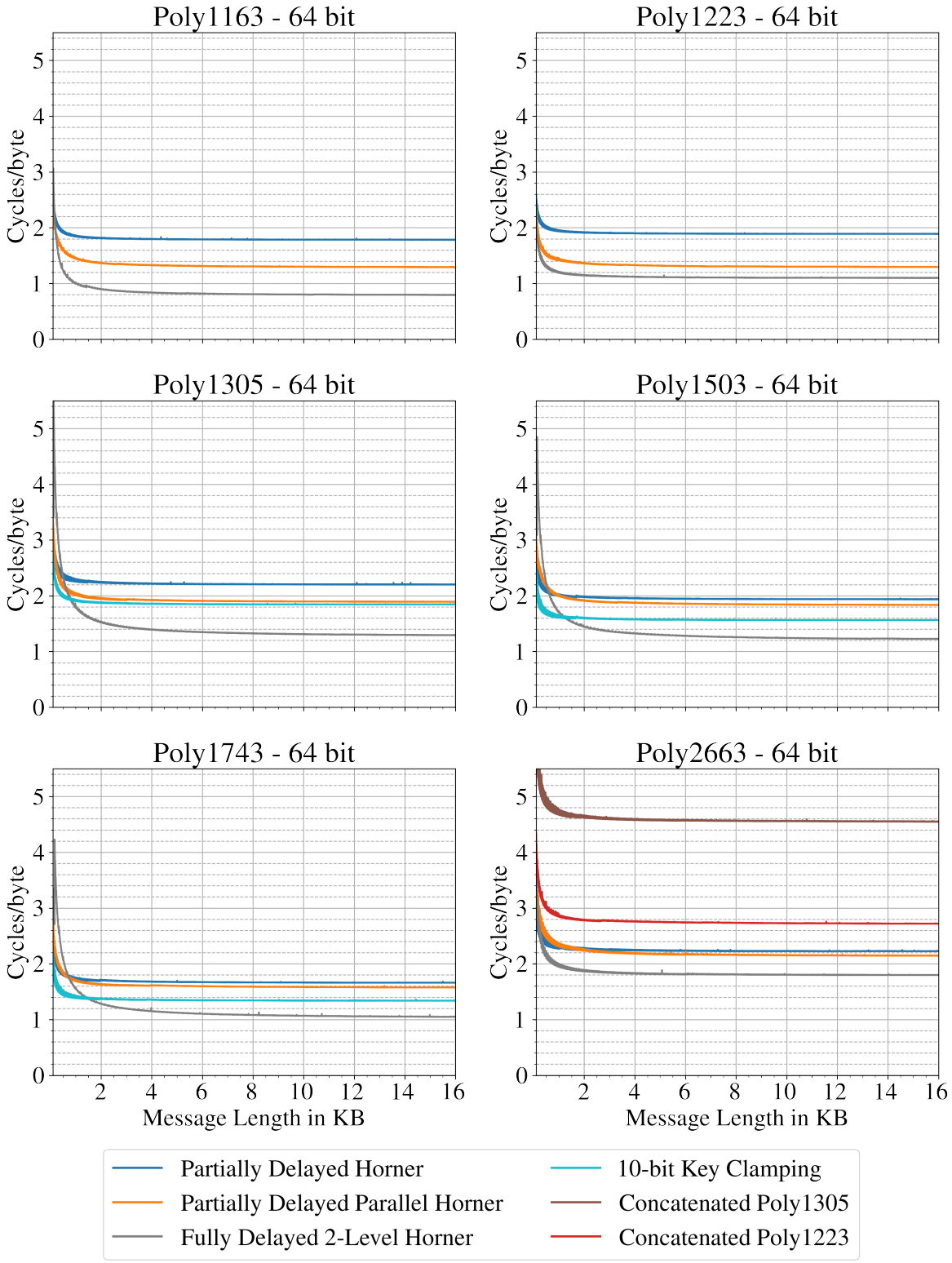


Figure 15: Benchmarking results for 64-bit implementations on Intel Core i7-8565U running at 1.8 GHz without Turbo Boost.

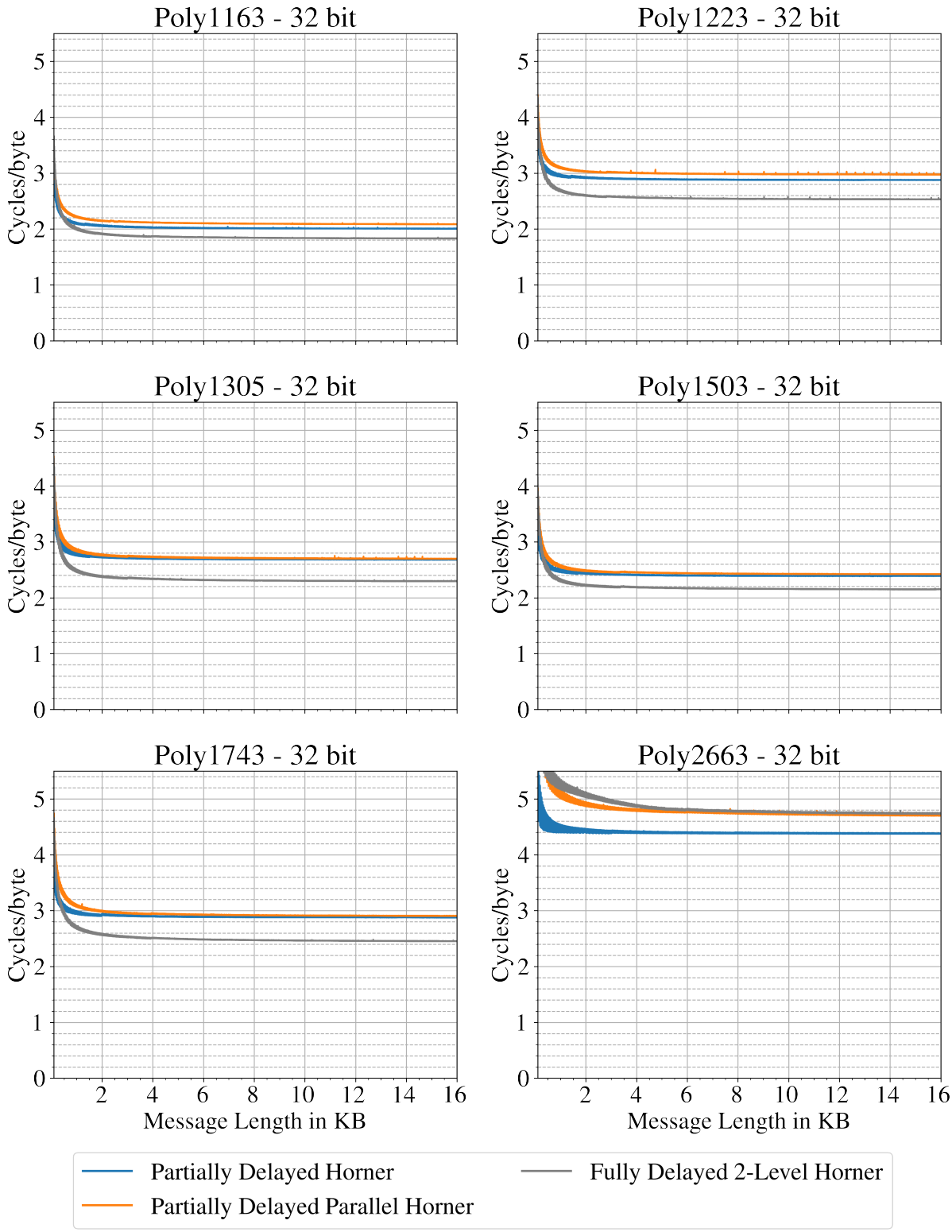


Figure 16: Benchmarking results for 32-bit implementations on Intel Core i7-8565U running at 1.8 GHz without Turbo Boost.

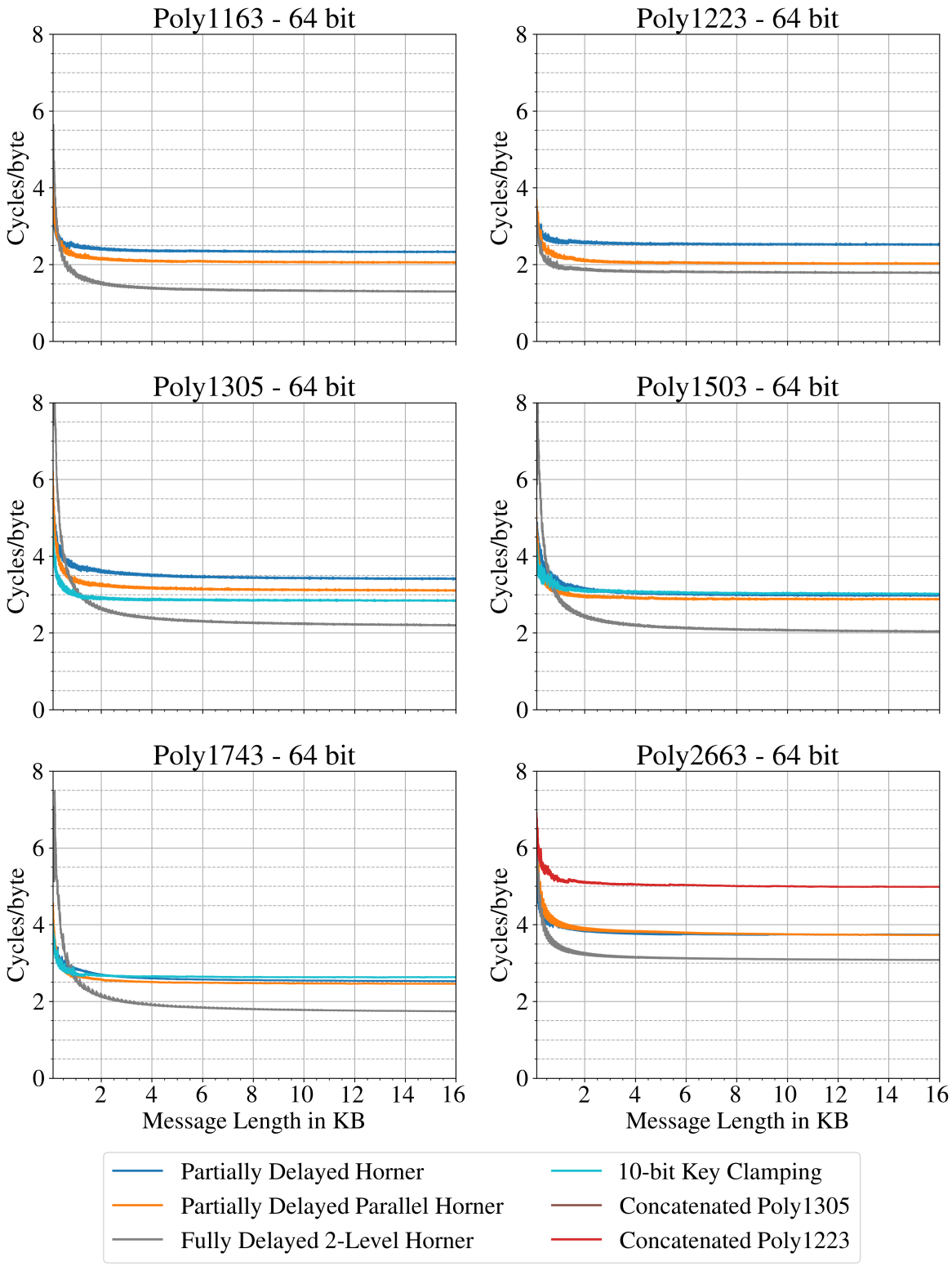


Figure 17: Benchmarking results for 64-bit implementations on Intel Core i7-940 running at 2.93 GHz without Turbo Boost.

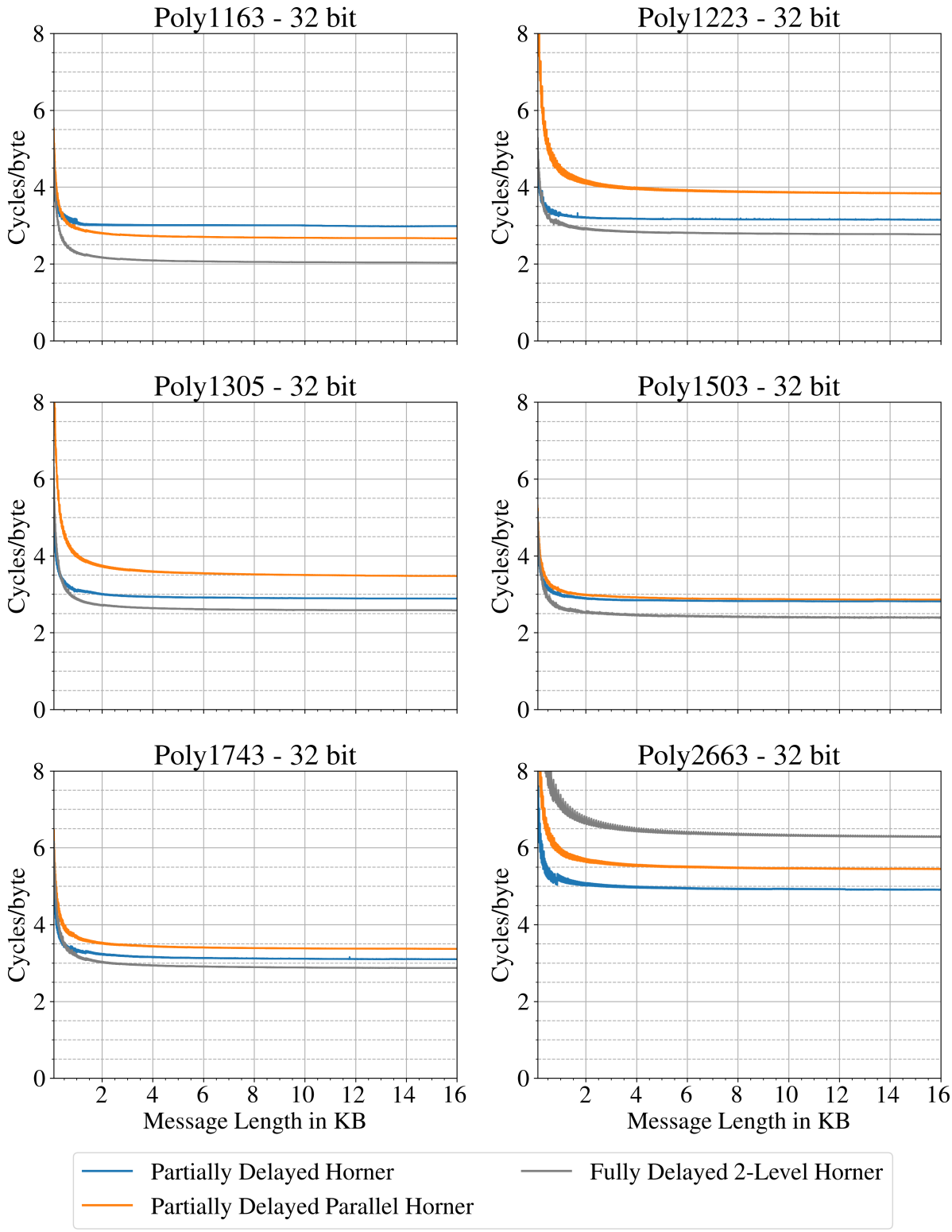


Figure 18: Benchmarking results for 32-bit implementations on Intel Core i7-940 running at 2.93 GHz without Turbo Boost.

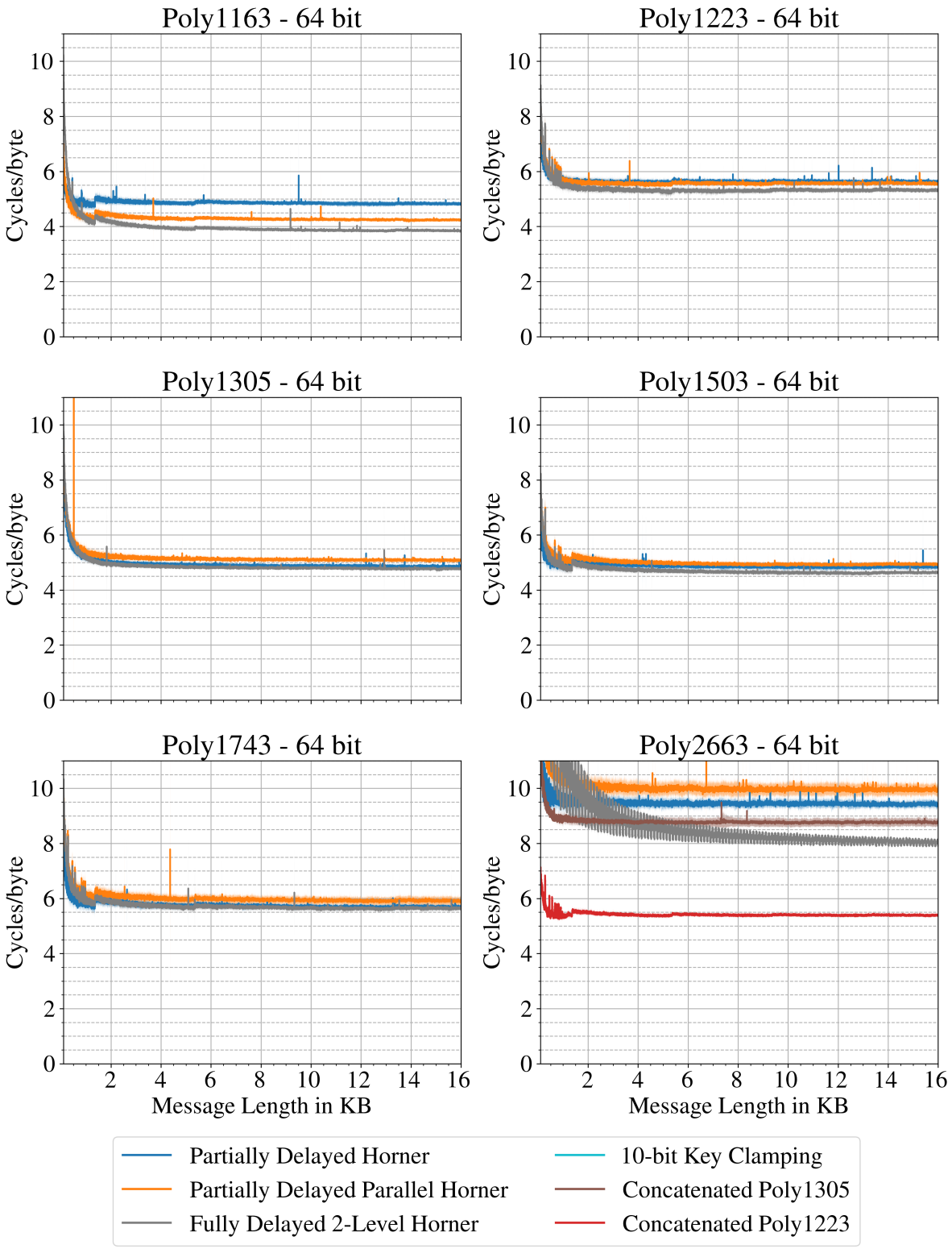


Figure 19: Benchmarking results for 64-bit implementations on Intel Core 2 Duo T5800 running at 2 GHz without Turbo Boost.

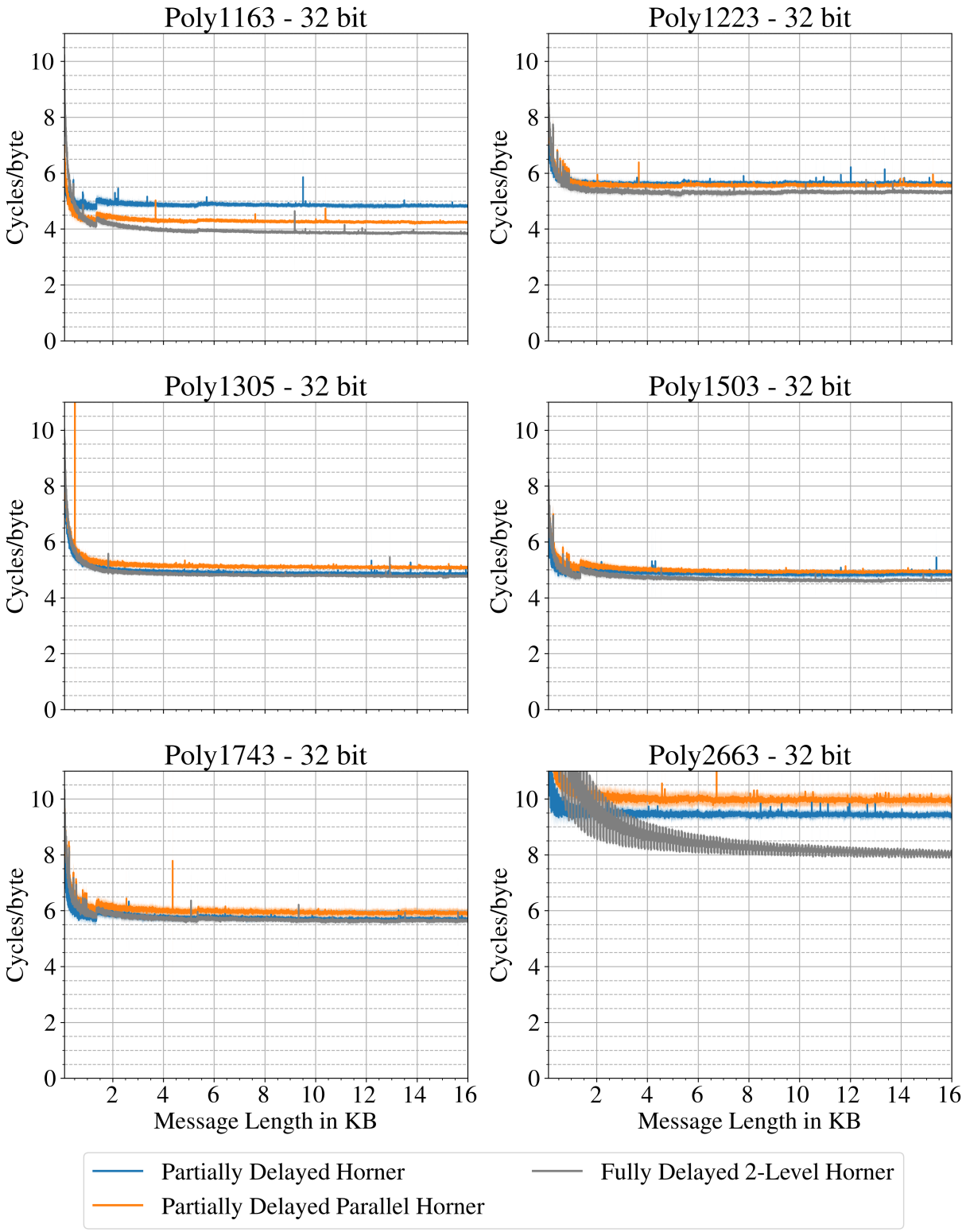


Figure 20: Benchmarking results for 32-bit implementations on Intel Core 2 Duo T5800 running at 2 GHz without Turbo Boost.

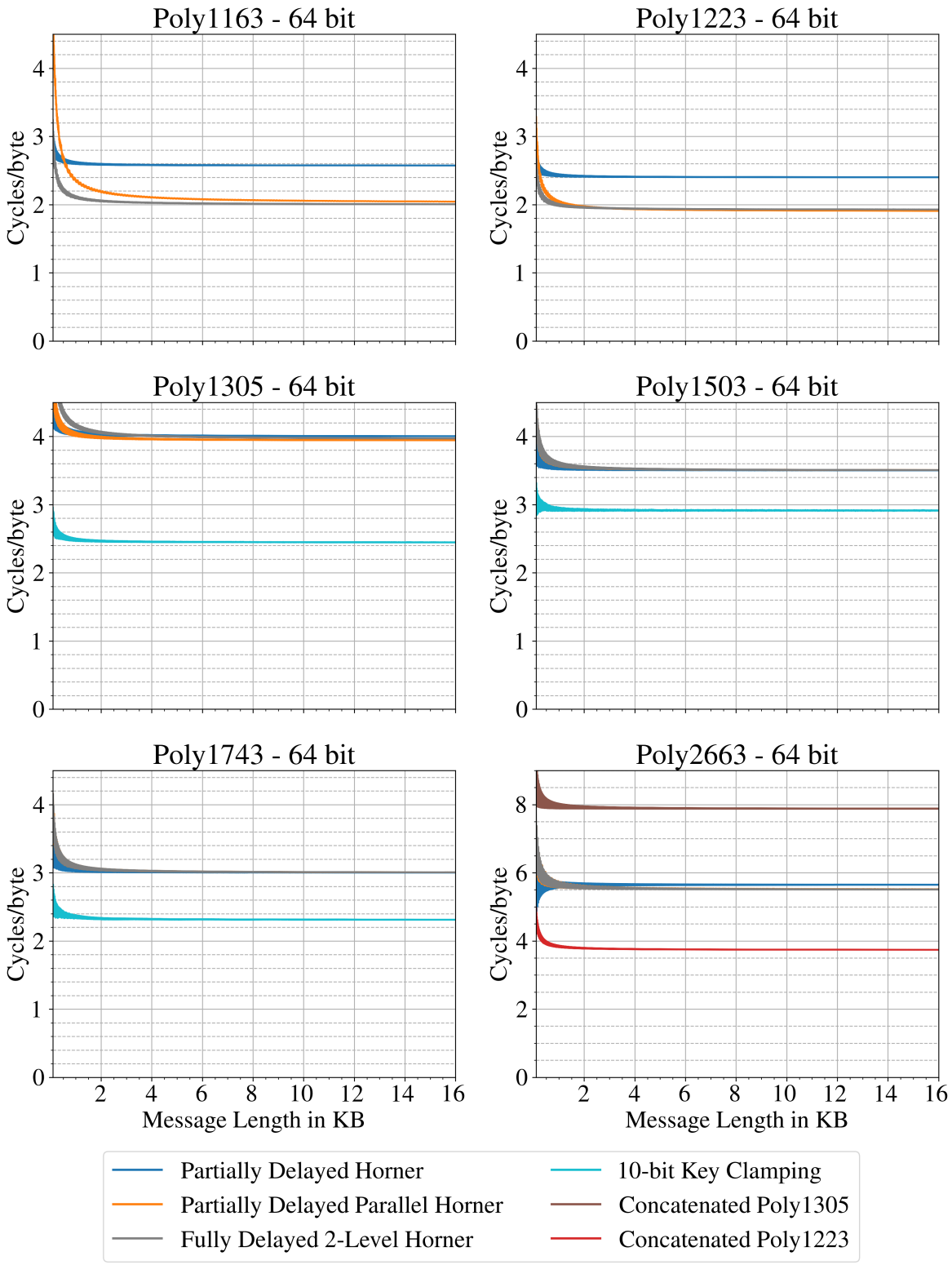


Figure 21: Benchmarking results for 64-bit implementations on Arm Cortex-A76 running at 2.4 GHz. *Please note the change in y-axis scaling for* Poly2663.

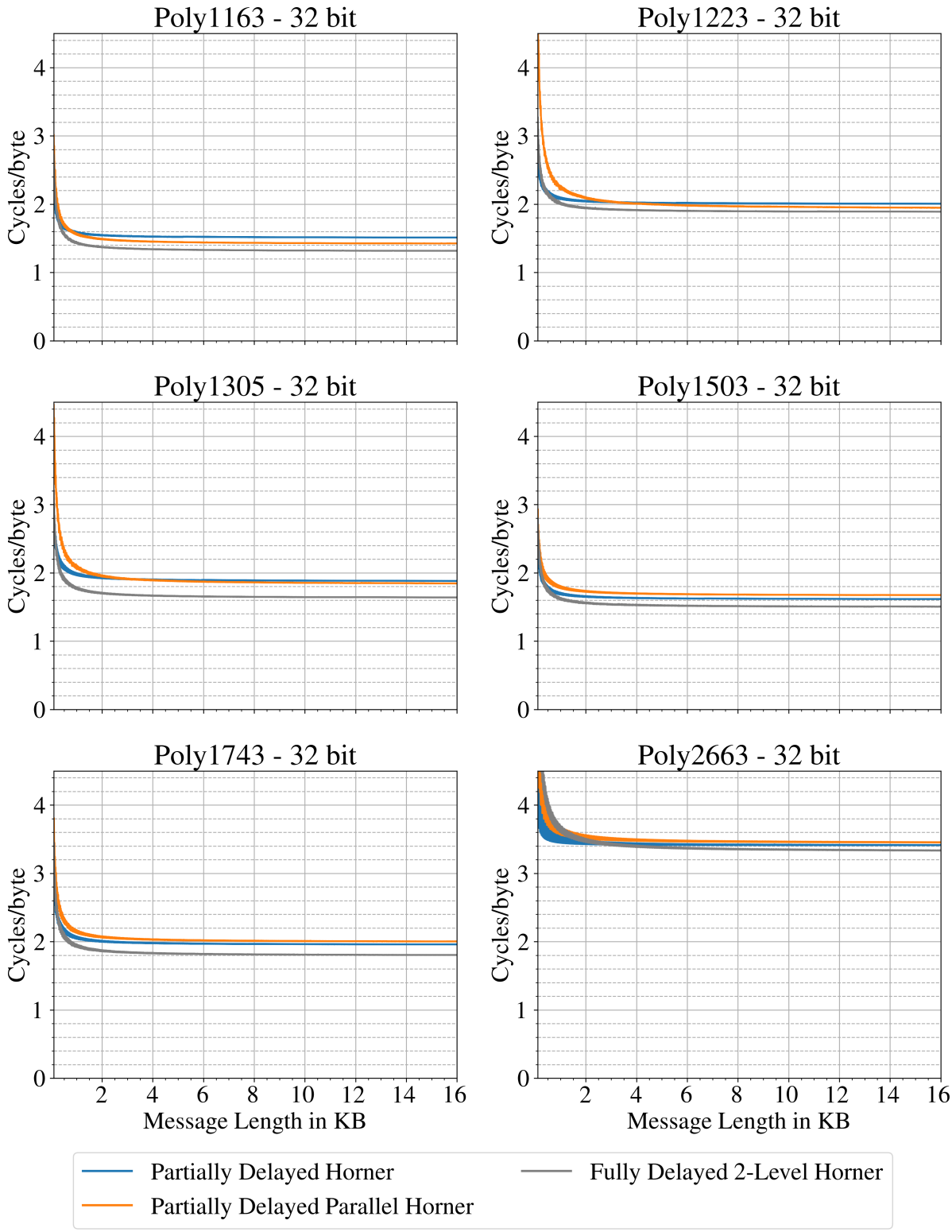


Figure 22: Benchmarking results for 32-bit implementations on Arm Cortex-A76 running at 2.4 GHz.