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

Author(s):
Degabriele, Jean Paul; Gilcher, Jan; Govinden, Jérôme; Paterson, Kenneth

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000654260

Originally published in:
https://doi.org/10.1109/SP54263.2024.00132

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

Jean Paul Degabriele*, Jan Gilcher†, Jérôme Govinden‡, Kenneth G. Paterson†
*Technology Innovation Institute, jeanpaul.degabriele@tii.ae
†ETH Zurich, {jan.gilcher, kenny.paterson}@inf.ethz.ch
‡TU Darmstadt, 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 [1]. 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.

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 [2]. 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 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. While ChaCha20 and Poly1305 were both designed by Bernstein, but as independent primitives. In fact, Poly1305 was originally proposed in combination with AES as a Wegman-Carter MAC [3]. The two were only combined into an AEAD scheme later on by Langley [4]. 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 [5], 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 [5]), (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 infinite 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 [6]. 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 sm 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 is of the form $\delta_{prf} + q_a^2/|\mathcal{N}| + \epsilon \cdot q_v$ where 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.

2.2. Composition

The following theorem establishes the security of composing two universal hash functions, extending a classical result by Stinson [7] 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 [8], for domain extension as in XPoly [9], or for "ramping-up"[1] as in PolyR [10]. The proof of the theorem is in the full version.

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*

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

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.

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

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

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

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

$\overline{H_1} = (M_2, H_1(r_1, M_1))$, 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 or Δ-universal. As for the bound, note that $\epsilon_3(\overline{H_1}, \overline{H_1}') \leq \epsilon_2(\overline{H_1}, \overline{H_1}')$ always. In some cases, as in the composition theorem of XPoly and PolyR, ϵ_3 is much smaller than ϵ_2, resulting in a bound $\max(\epsilon_2, \epsilon_1 + \epsilon_3)$ that is better than the classical bound $\epsilon_1 + \epsilon_2$ for composition. As 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 [3] 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.

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 et al. [11] 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 [12], by Johansson, Kabatianskii and Smeets [13], and by Taylor [14]. It is the basis for most modern polynomial hash functions such as Poly1305 [3], GHASH [15] and POLYVAL [16]. Later, efficient univariate polynomials based on the BRW polynomial from [17] were introduced [18], [19], [20].

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 [17, Section 5.7] and whose proof can be found in the full version, 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 by $G_x(M) = P_x(M) - P_0(M)$ 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 $\mathbb{F}_q$*.

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: $H_{\text{poly}}(r, M) = M_1 \cdot r^{n-1} + M_2 \cdot r^{n-2} + \cdots + M_{n-1} \cdot r^1 + M_n$. 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. 3.1.2), which in turn depends on the specific choice of finite field. Yet, assuming an injective data-to-field encoding, fixing M_1 to be non-zero allows H_{poly} to be (Δ-)universal over a variable number of message blocks, and fixing the constant term be zero guarantees that H_{poly} achieves the stronger Δ-universality property. 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.

$\mathcal{M}$	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 $\mathcal{M}$.

We will focus on the classical polynomial as it is simple and general enough to present the main optimizations available to polynomial universal hash function over finite fields. Most of these optimizations extend to other types of polynomials. The study of other types of polynomials (such as the BRW polynomials) is left to future work.

Transforming Polynomial Hash Functions. Once a polynomial is chosen, it might only have some of the properties we desire for our hash function. It might not be secure enough, not be Δ-universal, its domain too small, etc. In that case we might be able to apply some efficient transformations to obtain the desired properties. App. B lists possible transformations and how they affect security and efficiency.

3.1.2. 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 $\text{TR} : \mathbb{F}_q \to \{0,1\}^\tau$ to compose it with. 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 such an injective transform function TR. Then, the composed function $\text{TR} \circ H$ 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.

If the function H is also a Δ-universal hash function, we can use Prop. 1 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 the full version), reformulate the technique used for MMH and Poly1305 to obtain a Δ-universal hash function over bit strings.

Proposition 1 (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 $\mathrm{TR} : \mathbb{Z}_n \to \mathbb{Z}_m$ denote the function $\mathrm{TR}(x) = x \pmod m$. Define the family of keyed hash functions $H' : \mathcal{R} \times \mathcal{M} \to \mathbb{Z}_m$ as $H'(r, M) = \mathrm{TR}(H(r, M))$. If H is ϵ-almost Δ-universal, then H' is ϵ'-almost Δ-universal with $\epsilon'(M, M') = \left(\left\lfloor \frac{2n-2}{m} \right\rfloor + 1\right) \cdot \epsilon(M, M')$.*

Note that when $n \leq m$, TR is an injection from $\mathbb{Z}_n$ to $\mathbb{Z}_m$ that does not require any computation. When $n > m$ and $m = 2^\tau$, $\mathrm{TR}(x) = x \pmod m$ 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 $\mathrm{TR} : \mathbb{F}_q \to \{0,1\}^\tau$, 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.

Key-to-Field Encoding. According to Prop. 2 below, to transform an almost universal or Δ-universal hash $H : \mathbb{F}_q \times \mathcal{M} \to \mathcal{T}$ into a hash function with a key space $\{0,1\}^\rho$, it suffices to compose a transform function $\mathrm{TR} : \{0,1\}^\rho \to \mathbb{F}_q$ with H. When given random strings as input and compared to a uniform distribution, the larger the bias in the distribution of the outputs of TR is, the more significant the security loss of the composed construction. The proof can be found in the full version.

Proposition 2 (Key Space Transformation of AU or AΔU)**.** *Let $H : \mathcal{R} \times \mathcal{M} \to \mathcal{T}$ be an ϵ-almost universal hash function (resp. ϵ-almost Δ-universal hash function) and $\mathrm{TR} : \mathcal{R}' \to \mathcal{R}$ be a function satisfying $\Pr_{r' \leftarrow\$ \mathcal{R}'}[\mathrm{TR}(r') = r] \leq N$ for all $r \in \mathcal{R}$. Consider the family of keyed hash functions $H' : \mathcal{R}' \times \mathcal{M} \to \mathcal{T}$ defined by $H'(r', M) = H(\mathrm{TR}(r'), M)$ for all $r' \in \mathcal{R}'$ and $M \in \mathcal{M}$. Then H' is ϵ'-almost universal (resp. ϵ'-almost Δ-universal) with $\epsilon'(M, M') = N \cdot |\mathcal{R}| \cdot \epsilon(M, M')$.*

Note that when TR is an injective function, then one can choose $N = |\mathcal{R}'|^{-1}$ in the above proposition, thereby preserving the value of ϵ.

Defining $\mathrm{TR}(r)$ as the integer representation of the $\lfloor \log_2(q) \rfloor$-bit string $r[1\colon \min(\rho, \lfloor \log_2(q) \rfloor)] \| 0^*$ leads to the smallest security loss of $q \cdot 2^{-\min(\rho, \lfloor \log_2(q) \rfloor)}$, which is less than 2 when $2^\rho > q$. This is proved in the full version. We now describe two choices of key-to-field encoding which imply different tradeoffs between security and efficiency: key size reduction and key format restriction.

Key Size Reduction. As the efficiency of field arithmetic decreases with the number of limbs (cf. Sec. 3.2.2), mapping to a bit string with a smaller number of limbs is better. To increase the efficiency of a polynomial hash function, we might want to reduce the key size from ρ bits to a smaller $\overline{\rho}$ bits with $2^{\overline{\rho}} \leq q$ and use an injective key transform function $\mathrm{TR} : \{0,1\}^{\overline{\rho}} \to \mathbb{F}_q$ that encodes elements of $\mathbb{F}_q$ with fewer limbs. According to Prop. 2, the security loss factor of such a transform is $q \cdot 2^{-\overline{\rho}}$. However, this approach would only lead to a performance improvement when the evaluation algorithm of the polynomial hash function involves multiple multiplications with the key.

Key Format Restriction. In a prime field $\mathbb{Z}_q$, multiplying a field element with another one of a particular format can be more efficient than multiplying two general field elements. Examples of polynomial universal hash functions that use key format restrictions are PolyR and Poly1305. We discuss one particular key format, key clamping, in detail in Sec. 3.3. Another format based on bit shifting is described in the full version. Both techniques have a security loss factor of at most $q \cdot 2^{-\min(\rho, \lfloor \log_2(q) \rfloor) + b}$, where b is the number of bits required to be zero per limb.

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 format restrictions, 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 $\mathrm{INJ} : \mathcal{M}' \to \mathcal{M}$ we obtain an ϵ'-almost universal hash function over the domain $\mathcal{M}'$ where $\epsilon'(M, M') = \epsilon(\mathrm{INJ}(M), \mathrm{INJ}(M'))$. 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 $\mathrm{INJ} : \{0,1\}^* \to [\mathbb{F}_q]^*$ 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).

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 the overhead in computation will outweigh the degradation in memory access.

Although not very common, it is worth mentioning two techniques from [10] 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 block 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.

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

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

Similarly to the case of key-to-field encoding, we can employ block size restriction (i.e., use a smaller μ) and block format restriction in order to speed up field multiplication. To the best of our knowledge, we are the first to propose this optimisation. 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 security decrease is less significant as it is only linear compared to the exponential degradation resulting from key size reduction and key format restriction. This is due to the security bound $\frac{d(M,M')}{q} \cdot \frac{q}{2^\rho}$ of Thm. 2 combined with Prop. 2, 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.

3.1.3. Choice of Finite Fields. The polynomial universal hash functions constructed in Secs. 3.1.1 and 3.1.2 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 [21] 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 [22] 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 [23] 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. 3.1.2). 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. 3.1.2). 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. 3.1.2). 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

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 [20]. 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.

(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. (6) 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 necessary for delayed

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.

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 describes the main implementation strategies of prime field operations used in evaluating a polynomial hash function.

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 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 $q = 2^\pi - \theta$ and $x \in \mathbb{Z}_q$ we have by the usual laws of remainder arithmetic that

$$R_q(x \cdot (2^\pi)^n) = R_q(x \cdot \theta^n), \tag{1}$$

which for Mersenne primes simplifies to

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

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.

$$R_q(x \cdot (2^{m \cdot d})^n) = R_q\left(x \cdot \left(\sum_{i=0}^{d-1} c_i 2^{i \cdot m}\right)^n\right) \tag{3}$$

As observed in [23], 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.

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

Multiplication. We have several different algorithms to choose from when implementing finite field multiplication. In App. C, we evaluate Karatsuba multiplication. We show there 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 [24].

We can multiply two numbers a, b as follows

$$a \cdot b = \sum_{i=0}^{2\ell-1} \sum_{\ell:\ell:j+k=i} a_j b_k B^i. \tag{4}$$

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. (1) for $x \in \mathbb{Z}_q$:

$$x \cdot 2^\pi = x \cdot \theta \mod (2^\pi - \theta) \tag{5}$$

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. (4) using eq. (5) to obtain:

$$\begin{aligned} a \cdot b &= \sum_{i=0}^{\ell-1} \left(2^\lambda\right)^i c_i \mod 2^\pi - \theta, \\ \text{with } c_i &= \sum_{\ell:\ell:j+k=i} a_j b_k + \sum_{\ell:\ell:j+k=i+\ell} a_j b_k \cdot 2^{\lambda-\lambda'} \cdot \theta \end{aligned} \tag{6}$$

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. (6) 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

4. $c = 0, A = M$ gives $k \approx 3.59198$ in eq. (8) in App. C.
5. $c = 0, 1.5A = M$ gives $k \approx 2.82657$ in eq. (8) in App. C.

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.

1: **while** $\exists i\in\{0,\ell-2\}.a_i\geq 2^\lambda\vee a_{\ell-1}\geq 2^{\lambda'}$:
2: **for** i **in** $\{0,\ell-2\}$:
3: $a_{i+1}=a_{i+1}+\left\lfloor a_i/2^\lambda\right\rfloor$
4: $a_i=a_i \mod 2^\lambda$
5: $a_0=a_0+\left\lfloor a_{\ell-1}/2^{\lambda'}\right\rfloor*\theta$
6: $a_{\ell-1}=a_{\ell-1} \mod 2^{\lambda'}$
7: **endwhile**

Figure 1: Carry propagation for Crandall prime fields

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. (2) and (3) 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.

In the full version, we also consider an asymmetric version of Schoolbook multiplication as well as optimizations for squaring.

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 3. *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 the full version.

By Prop. 3 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 a variant of the algorithm in Fig. 1 that does only one full iteration of the while loop. In the second iteration of the while loop it terminates after executing line 4 in the j-th iteration of the for loop. 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:

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

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 with the same algorithm as for carry propagation. The major difference in the implementation between carry propagation and limb realignment is the types of the involved variables: in carry propagation all our variables are single-word integers, while in limb realignment we start with double-word sized integers. As with carry propagation, an unsaturated representation allows to delay 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, the cost of multiplication increases by a factor of 4 with each multiplication without limb realignment. This leads to three different approaches to schedule limb realignment:

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

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

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

Figure 2: Sequential Horner's algorithm of H_{poly}.

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, this reduction of limb realignments can have significant performance impacts. 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 [20].

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 carry (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. 2, 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. 3) 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. 3. However, neither Horner nor parallel Horner allow for longer delayed carry propagation. Another structural characteristic of H_{poly} is the possibility to abstract it as a 2-level polynomial as described in Fig. 4 (and introduced in [25], [26]). The algorithm processes

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

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** $i = 1$ **to** B **do**
4 : $\quad T_{n+i \bmod B} \leftarrow T_{n+i \bmod B} \cdot r^{B-i}$
5 : $T \leftarrow T_0 + T_1 + \cdots + T_{B-1}$
6 : **return** T

Figure 3: B-branch parallel evaluation algorithm of H_{poly}.

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

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 4: 2-level evaluation of H_{poly} grouped by B-blocks.

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 $\widetilde{H}_{\text{poly}}$ uses Horner or parallel Horner so that the 2-level construction inherits their online or parallel characteristics. The lower-level polynomial $\overline{H}_{\text{poly}}$ almost exclusively uses the naive evaluation algorithm to employ its delayed-carry strategies while only needing to precompute the first B powers of the key (these can be reused across each group of B blocks). This 2-level construction and parallel Horner both require $n + B$ field multiplications.

When evaluating the classical polynomial over a Crandall prime field, a characteristic of Horner, Parallel Horner or the 2-level implementation choice we just described is the possibility to 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 longer delayed carry propagation. Both parallel Horner and the 2-level implementation allow for some level of parallelization of field multiplications that depend on the value B. This notably explains why they perform better than Horner in the benchmarks presented later, as they allow to fill the CPU pipeline, even though they require more field multiplications.

3.3. Key and Message Format Restriction

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 [3]. 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 [27] and explained in [28]. 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 $\ell = \left\lceil \frac{\pi}{\omega} \right\rceil$ limbs. To ensure proper cache alignment, we further chose the formatted operand to be $\rho = \left\lfloor \frac{\pi}{\omega} \right\rfloor \cdot \omega$ bits in size, meaning it uses either $\ell - 1$ or in the case that 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. (6) 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:

$$x \cdot 2^{\rho} = x \cdot 2^{-t} \cdot 2^{\pi} = x \cdot 2^{-t}\theta \mod q. \tag{7}$$

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. (6) 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.

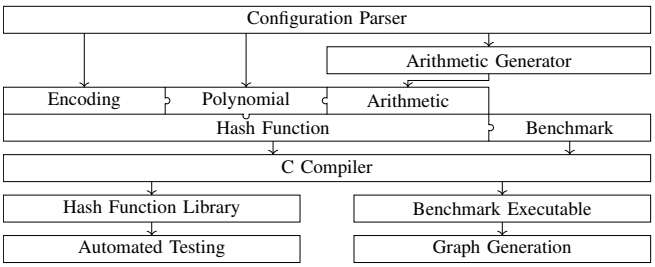

Figure 5: Schematic overview of our framework

A schematic overview of how our framework is structured is shown in Fig. 5. 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 25 times to compute the average and standard deviation for that sample which we use in our plots. We repeat these measurements for varying message sizes at one-byte increments up to a maximum of 8000 bytes. Note that the framework does not automatically CPU disable 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.

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

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.

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 options:

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

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 [5], 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 [5] and an additional security loss of $\log_2(n)$ 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 $q_2 = 2^{116} - 3$, 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 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 and Evaluation

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 realignment. For the latter two, we ran the experiment with different hyperparameters, 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. For Poly1305, Poly1503, and Poly1743, we also added a configuration with custom field arithmetic that utilizes key clamping as outlined in Secs. 3.1.2 and 3.3. This is not yet automatically generated by our framework. The key clamping scheme used is the same for all three. 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 10 bits. Unless otherwise specified, all implementations use a partially-delayed realignment strategy. All experiments were run on a 4 GHz Intel Xeon Gold 6258R configured to run in 64-bit mode with Turbo Boost enabled and the compiler used was gcc-9.4.0.

6. Benchmarking Results and Conclusions

Our benchmarks are summarized in Fig. 8 in the appendix. It shows the processing rate in cycles per byte for the implementations generated by our framework. We observe that our Poly1305 implementations are competitive with the state-of-the-art implementations reported in [20]. 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 Poly1223 and 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 [5]), 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.

Fig. 6 compares our fastest Poly1305 and Poly1163 versions to state-of-the-art libraries (with Turbo Boost enabled). We observe that our implementations are competitive even with implementations that utilize specialized hardware features like `CLMUL` and vectorized instructions. Specifically for messages below 8KB in size, Poly1163 outperforms OpenSSL's AVX2 implementation of Poly1305 as well as its GMAC implementation utilizing `CLMUL`, and also 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, 8KB far exceeds the typical size of network packets. We conclude that a well-crafted AVX2 implementation of Poly1163 is likely to strongly outperform the existing state-of-the-art.

Unfortunately, due to space limitations we are only able to include benchmarks for one processor type here, but we will make more benchmarks available in the full version. In future work, we plan to expand our scope to polynomials over binary fields and hash functions that employ more sophisticated polynomials, such as BRW. Finally adding further features and functionality to our framework, such as automatically proving the correctness of the code it generates with respect to a mathematical specification, is in and of itself an interesting research direction.

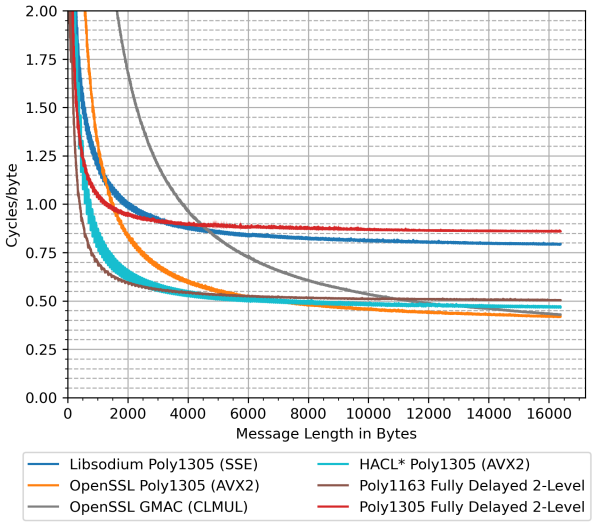

Figure 6: Comparison of our proposed designs with various library implementations of polynomial hash functions.

References

[1] M. N. Wegman and L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, vol. 22, pp. 265–279, 1981.

[2] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *NDSS 2017*. The Internet Society, Feb. / Mar. 2017.

[3] D. J. Bernstein, "The poly1305-AES message-authentication code," in *FSE 2005*, ser. LNCS, H. Gilbert and H. Handschuh, Eds., vol. 3557. Springer, Heidelberg, Feb. 2005, pp. 32–49.

[4] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 8439, Jun. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8439

[5] J. P. Degabriele, J. Govinden, F. Günther, and K. G. Paterson, "The security of ChaCha20-Poly1305 in the multi-user setting," in *ACM CCS 2021*, G. Vigna and E. Shi, Eds. ACM Press, Nov. 2021, pp. 1981–2003.

[6] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of computer and system sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[7] D. R. Stinson, "Universal hashing and authentication codes," in *Annual International Cryptology Conference*. Springer, 1991, pp. 74–85, preliminary version of [29].

[8] T. Krovetz, "UMAC: Message Authentication Code using Universal Hashing," RFC 4418, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4418

[9] S. Duval and G. Leurent, "Lightweight macs from universal hash functions," in *Smart Card Research and Advanced Applications*, S. Belaïd and T. Güneysu, Eds. Cham: Springer International Publishing, 2020, pp. 195–215.

[10] T. Krovetz and P. Rogaway, "Fast universal hashing with small keys and no preprocessing: The PolyR construction," in *ICISC 00*, ser. LNCS, D. Won, Ed., vol. 2015. Springer, Heidelberg, Dec. 2001, pp. 73–89.

[11] S. Halevi and H. Krawczyk, "MMH: Software message authentication in the Gbit/second rates," in *FSE'97*, ser. LNCS, E. Biham, Ed., vol. 1267. Springer, Heidelberg, Jan. 1997, pp. 172–189.

[12] B. den Boer, "A simple and key-economical unconditional authentication scheme," *Journal of Computer Security*, vol. 2, pp. 65–71, 1993.

[13] T. Johansson, G. Kabatianskii, and B. Smeets, "On the relation between A-codes and codes correcting independent errors," in *EUROCRYPT'93*, ser. LNCS, T. Helleseth, Ed., vol. 765. Springer, Heidelberg, May 1994, pp. 1–11.

[14] R. Taylor, "An integrity check value algorithm for stream ciphers," in *CRYPTO'93*, ser. LNCS, D. R. Stinson, Ed., vol. 773. Springer, Heidelberg, Aug. 1994, pp. 40–48.

[15] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," in *INDOCRYPT 2004*, ser. LNCS, A. Canteaut and K. Viswanathan, Eds., vol. 3348. Springer, Heidelberg, Dec. 2004, pp. 343–355.

[16] S. Gueron, A. Langley, and Y. Lindell, "AES-GCM-SIV: Specification and analysis," Cryptology ePrint Archive, Report 2017/168, 2017, https://eprint.iacr.org/2017/168.

[17] D. J. Bernstein, "Polynomial evaluation and message authentication," *Unpublished manuscript*, 2007, http://cr.yp.to/papers.html#pema. [Online]. Available: http://cr.yp.to/papers.html#pema

[18] D. Chakraborty, S. Ghosh, and P. Sarkar, "A fast single-key two-level universal hash function," *IACR Trans. Symm. Cryptol.*, vol. 2017, no. 1, pp. 106–128, 2017.

[19] S. Ghosh and P. Sarkar, "Evaluating bernstein–rabin–winograd polynomials," *Designs, Codes and Cryptography*, vol. 87, no. 2, pp. 527–546, 2019.

[20] S. Bhattacharyya, K. Nath, and P. Sarkar, "Polynomial hashing over prime order fields," *Cryptology ePrint Archive*, 2023, https://eprint.iacr.org/2023/634. [Online]. Available: https://eprint.iacr.org/2023/634

[21] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

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

[23] J. A. Solinas, "Generalized mersenne numbers," Tech. Rep. CORR 99-39, 1999.

[24] D. J. Bernstein, "Multidigit multiplication for mathematicians," 2001, https://cr.yp.to/papers.html#m3. [Online]. Available: https://cr.yp.to/papers.html#m3

[25] K. Jankowski and P. Laurent, "Packed aes-gcm algorithm suitable for aes/pclmulqdq instructions," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 135–138, 2010.

[26] M. Goll and S. Gueron, "Vectorization of poly1305 message authentication code," in *Proceedings of the 2015 12th International Conference on Information Technology - New Generations*, ser. ITNG '15. USA: IEEE Computer Society, 2015, p. 145–150. [Online]. Available: https://doi.org/10.1109/ITNG.2015.28

[27] "monocypher," https://monocypher.org/. [Online]. Available: https://monocypher.org/

[28] L. Vaillant, "The design of poly1305," https://loup-vaillant.fr/tutorials/poly1305-design. [Online]. Available: https://loup-vaillant.fr/tutorials/poly1305-design

[29] D. R. Stinson, "Universal hashing and authentication codes," *Designs, Codes and Cryptography*, vol. 4, no. 3, pp. 369–380, 1994.

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

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

[32] *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, https://developer.arm.com/documentation/den0024/a/Introduction. [Online]. Available: https://developer.arm.com/documentation/den0024/a/Introduction

[33] P. Rogaway, "Bucket hashing and its application to fast message authentication," in *CRYPTO'95*, ser. LNCS, D. Coppersmith, Ed., vol. 963. Springer, Heidelberg, Aug. 1995, pp. 29–42.

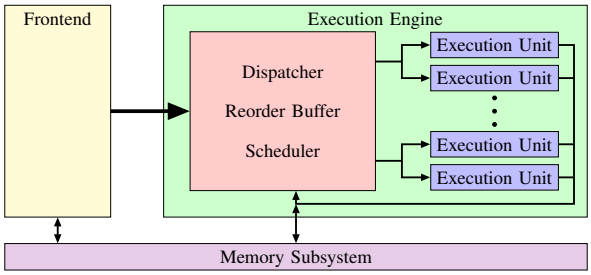

Figure 7: Simplified overview of a modern CPU core.

Appendix 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 [30], [31], [32]. 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 7 shows a very simplified overview of this design.

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.

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,[7] 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

7. 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.

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.

Appendix B. Transforming Polynomial Hash Functions

In this section, we give a brief description of the concatenation and composition transformations and how they affect security and efficiency. In the full version, we also give efficient transformations for the union and product of domains and a more in-depth analysis of all them.

Concatenation. Concatenating polynomial universal hash functions can be done for two main reasons: increased security [33, Proposition 3] or domain extension [7, Theorem 5.4]. In both cases, the two hash functions concatenated can be evaluated in parallel, but the tag space of the construction is increased. When parallel evaluation of the two polynomials is not possible, this improved security or domain increase is obtained at the cost of efficiency, as we also need to evaluate a second polynomial hash function.

The concatenation of two polynomial hash functions over two small fields is often less secure than using a single polynomial universal hash function over a bigger field (of size the sum of the two small fields) [5, Theorem 7.4]. Thus, concatenation is only useful when the construction is more efficient. For example, concatenating two polynomial hash functions over a smaller field with a good ratio of speed over field size would result in faster evaluation than a single polynomial over a bigger field (of size twice the size of the small field) with a less good ratio. While concatenation allows parallel evaluation of the hash functions, it is also possible to evaluate a single polynomial in parallel, as in the case of the classical polynomial (cf. Sec. 3.2.4). Yet, with the concatenation construction, implementing parallelization (for example, with AVX) would be easier and without any overhead cost.

Composition. The composition of polynomial hash functions can be considered for the same reasons as composition of two general hash functions (cf. Sec. 2.2). Yet, composing general hash functions can only be done with two independently sampled hash functions. As stated in the following theorem, in the case of polynomial hash functions over the same field, the same key can be reused by both polynomials at the cost of a slightly stronger assumption. The theorem is an example of the application of our new definition of univariate polynomial universal hash, i.e. a universal hash satisfying Thm. 2. The proof of the theorem which can be found in the full version, uses the fact that the composition of injective functions is injective. It helps in building polynomial hash functions with additional properties without needing another key. This type of composition is implicitly used by some multi-level and single key polynomial hash functions such as the ones in [18] and [20].

Theorem 3 (Composition of Polynomial Hash)**.** *Consider the family of keyed hash functions $G : \mathbb{F}_q \times (\mathcal{M} \times \prod_{i=1}^{n} \mathcal{M}_i) \to \mathbb{F}_q$ defined as follows. For any $r \in \mathbb{F}_q$ and $(M, M_1, \cdots, M_n) \in \mathcal{M} \times \prod_{i=1}^{n} \mathcal{M}_i$, $G(r, (M, M_1, \cdots, M_n)) = H(r, (M, P_x^1(M_1), \cdots, P_x^n(M_n)))$ 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 $H : \mathbb{F}_q \times (\mathcal{M} \times (\mathbb{F}_q[x])^n) \to \mathbb{F}_q$ is a polynomial hash function. Thus, there exist an injective function $P_y : \mathcal{M} \times (\mathbb{F}_q[x])^n \to \mathbb{F}_q[y]$ that satisfies: for any $r \in \mathbb{F}_q$ and $M, M' \in \mathcal{M} \times (\mathbb{F}_q[x])^n$, $H(r, M) = P_r(M)$ and $P_y(M) - P_y(M')$ is of total degree at most $d(M, M')$. Then G is an ϵ-almost universal hash function with $\epsilon((M, M_1, \cdots, M_n), (M', M_1', \cdots, M_n')) = \frac{d(\overline{M}, \overline{M'})}{q}$, where $\overline{M} = (M, P_x^1(M_1), \cdots, P_x^n(M_n))$ and $\overline{M}' = (M', P_x^1(M_1'), \cdots, P_x^n(M_n'))$. If in addition, the function $\overline{P}_y : \mathcal{M} \times (\mathbb{F}_q[x])^n \to \mathbb{F}_q[y]$ defined by $\overline{P}_y(M) = P_y(M) - P_0(M)$ is injective, then G is also an ϵ-almost Δ-universal hash function.*

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. For simplicity, we state in the theorem that P_y needs to be injective over $\mathcal{M} \times (\mathbb{F}_q[x])^n$. But the theorem also holds when P_y is only injective over the image $\mathcal{M} \times \prod_{i=1}^{n} P_x^i(\mathcal{M}_i)$. Also, for simplicity, the theorem is stated only for tuples of fixed length n but it can be extended to variable input lengths.

Appendix C. 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

$$T_S(2^k) = 2^{2k}M + \left(2^{2k} - 1\right) A. \tag{8}$$

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

$$T_K(2^k) = \begin{cases} 3T_k\left(2^{k-1}\right) + 3 \cdot 2^k A & \text{if } k > c \\ 2^{2c}M + \left(2^{2c} - 1\right) A & \text{otherwise.} \end{cases} \tag{9}$$

It follows that

$$\begin{aligned} T_K(2^k) = & 3^{k-c} \cdot 2^{2c} \cdot M + \\ & \left(3^{k-c} \cdot \left(2^{2c} - 1\right) + 3^{k-c+1} \cdot 2^{c+1} - 3 \cdot 2^{k+1}\right) \cdot A. \end{aligned} \tag{10}$$

For brevity we omit the simple induction proof.

Appendix D.

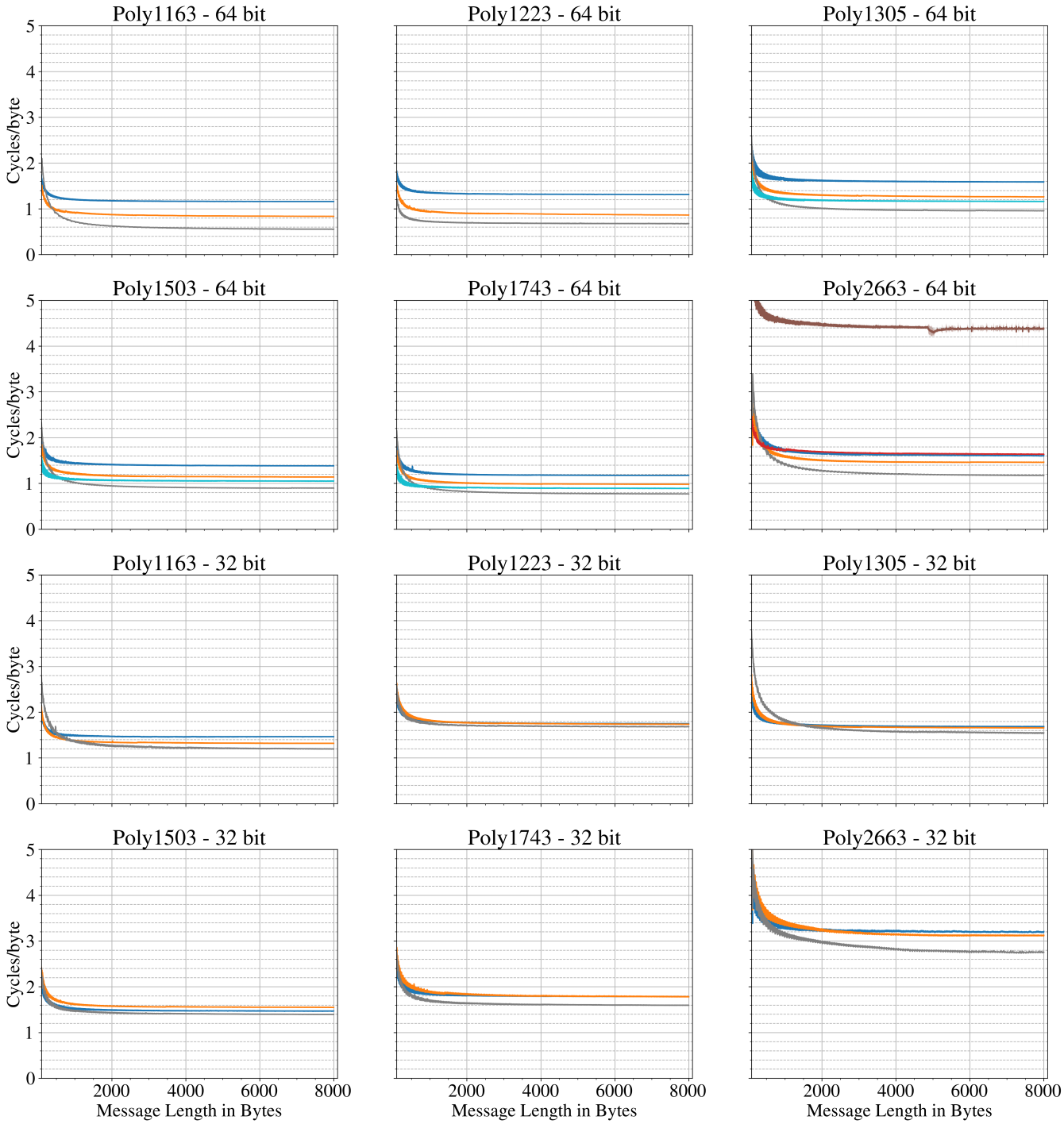

Figure 8: Benchmarking results for IntelXeon Gold 6258R with Turbo Boost up-to 4 GHz.

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

The paper investigates the impact of design and implementation choices on the performance and security of universal hash functions of a particular type that are constructed using univariate polynomial over a prime field with modulus of a type that allows efficient modular reduction, in short, that are variants of Poly1305. The authors use the insights they share with the reader to propose several variants of Poly1305 that either have better performance or better security and present a benchmarking tool and benchmarking results.

E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

E.3. Reasons for Acceptance

1) The authors create a new benchmarking framework that will allow future developers to test designs, optimizations, and how they interact.
2) This paper provides a valuable step forward in improving this central cryptographic operation, which requires understanding the interactions between security goals, the design space of mathematical functions under consideration, the design space of algorithms to compute those functions, and computer architecture.