New Designs of Multivariate-Polynomial Universal Hash Functions*

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

[1] CRC, Technology Innovation Institute, Abu Dhabi, United Arab Emirates
jeanpaul.degabriele@tii.ae, jerome.govinden@tii.ae
[2] Applied Cryptography Group, ETH Zurich, Switzerland
jan.gilcher@inf.ethz.ch, kenny.paterson@inf.ethz.ch

Abstract

Universal hash functions (UHFs) are basic building blocks in cryptography, making the topic of designing secure, fast UHFs of longstanding interest. This paper presents an exploration of the design space for multivariate UHFs, that is UHFs that involve the evaluation of a multivariate polynomial over a finite field. We focus on two-level designs, wherein a lower-level hash function produces intermediate values that are consumed by a higher-level one, and where both hash functions are based on either univariate or multivariate polynomials. This approach allows designs to benefit from the desirable features of both components and thereby strike new trade-offs between key size, security level, and amenability to optimization techniques. We extend the recent UHF code generation and benchmarking framework of Degabriele *et al.* (IEEE S&P 2024) to accommodate our multivariate designs (and also to support binary field arithmetic). We then use the framework to study the performance of a large collection of new two-level designs. This is done by first conducting a statistical factor analysis to determine which design features (and combinations of those features) most influence performance, and then using it to identify particular combinations of lower-level and higher-level hash functions offering particularly good performance/security trade-offs. We present new designs for both binary and prime fields, at two different security levels (corresponding to roughly 128 and 256 bits of security). Our best designs have performance that significantly outperforms state-of-the-art UHFs in the research literature and as deployed in mainstream cryptography libraries by up to 25%, resulting in 0.3 cycles/byte for 128-bit binary fields. We expect further gains from optimizations such as vectorization, as our benchmarks rely purely on auto-generated code from the extended framework, while state-of-the-art implementations typically use hand-optimized implementation strategies. We conclude with a brief inquiry into the performance implications of employing our best UHF design in the AEAD and Accordion mode designs currently under consideration for standardization by NIST.

1 Introduction

Universal hash functions (UHFs) are lightweight keyed hash functions satisfying a weak security notion. Security is only guaranteed as long as the key and hash digests are hidden from the adversary. However, their security is typically based on combinatorial techniques and thus holds even against computationally unbounded adversaries. Today, UHFs constitute an essential component in symmetric cryptography. Through the classical construction of Carter and Wegman they can be combined with a single block cipher call to yield very fast MAC designs over variable-length messages [37]. Such MAC constructions can be found in the two most widely deployed AEAD schemes today: GCM and ChaCha20-Poly1305, which use the UHFs GHASH and Poly1305 respectively. UHFs are also found in several more advanced MAC and

*This is the full version of the work that will appear at ACM CCS 2026.

AEAD designs. One notable case is in Accordion mode designs (variable-input-length tweakable ciphers), which NIST is currently planning to standardize [31]. An Accordion mode can be easily morphed into an AEAD scheme with the strongest possible security [22] or applied directly for disk encryption and other applications requiring encryption without ciphertext expansion, also providing the best possible security. Accordion mode constructions typically use two UHF instances and one encryption layer, e.g. AES-CTR, as in the currently favored NIST candidate HCTR2 [31]. Thus, in Accordion constructions, UHFs are even more crucial to the overall performance of the scheme.

Given their prominent role in the design of practical and efficient encryption schemes, there has been a substantial body of work proposing several optimized UHF designs. The vast majority rely on algebraic structures over some finite field within the following three categories.

Classical Polynomial. The most popular construction is the classical (univariate) polynomial construction [13, 25, 36]. Both GHASH and Poly1305 use this design, where GHASH is a polynomial over a 128-bit binary field and Poly1305 is a polynomial over the prime field corresponding to $2^{130} - 5$. Here the message is mapped to the polynomial coefficients, and the digest is computed evaluating the polynomial on the secret key. Its benefits are that it is conceptually simple, requires very short keys (the size of a single field element), supports online evaluation (using Horner's rule), and is parallelizable [10].

(Pseudo) Inner Product. A less known alternative is the inner product construction, already proposed by Carter and Wegman in 1979 [8] and later employed in the MMH construction by Halevi and Krawczyk [21]. Here the message and key are treated as two equal-length vectors over a finite field and the hash digest is their inner product. A related construction, also proposed by Carter and Wegman, is the pseudo inner product construction behind the NMH design [21] and UMAC [27]. It uses a similar algebraic structure but requires only half the number of multiplications as the inner product and classical polynomial. This comes at the cost of an equal increase in the number of additions but, since multiplication is typically more computationally intensive, it still results in significant speedups. The main limitation of MMH and NMH is that they require a key as long as the message.

BRW Polynomial. UHF designs with large key sizes are not as practical as the overhead in generating the key material often outweighs their performance gains. They also incur higher CPU cache consumption. This motivated Bernstein to ask whether it was possible to combine the short key size of the classical polynomial with the reduced multiplication complexity of the pseudo inner product construction. Building on ideas by Rabin and Winograd, he proposed such a construction that came to be known as the Bernstein-Rabin-Winograd (BRW) polynomial [3]. Although the BRW polynomial manages to combine the best of both worlds, it also introduces new limitations. Unlike the prior two, no parallelizable algorithm is known for evaluating the BRW polynomial. Furthermore, it is incompatible with popular field-arithmetic optimization techniques, such as delayed limb realignment, delayed carry, and delayed reduction which enable significant speedups in typical implementations [10].

A Unified View. Stepping back for a moment, we observe that, the classical polynomial, the pseudo inner product, and the BRW polynomial, are all instances of multivariate polynomials. More specifically, they represent edge-case instances with extreme traits in the full space of multivariate polynomials, e.g., single variable with full degree (classical polynomial) vs maximum number of variables with degree one (inner product). On a more practical level, if we consider the following three goals, **(a)** short keys, **(b)** reduced multiplication complexity, and **(c)** compatibility with delayed-operation optimizations and parallelizability, we observe that each construction satisfies exactly two of these properties but not all three. This poses the following research question:

Can we identify new universal hash designs that strike better trade-offs and attain better performance?

For instance, in ChaCha20-Poly1305, the key material for the hash is obtained via a separate call to the ChaCha20 block function where currently only 16 bytes of the available 48 bytes are used. A UHF design exploiting this free key material would clearly be beneficial in that setting. As existing designs lie at the extremities in the space of multivariate polynomials, it is likely that UHF designs with more balanced trade-offs can be found within the interior of this space. Accordingly, we set out to study and systematically explore the space of multivariate-polynomials in order to improve the state of the art in UHF design.

Our search through the space of multivariate polynomials is of course not exhaustive. Instead, it is informed by the current state of the art in UHF design and their optimized implementations. Our exploration is based on a two-level hash structure composing two simpler polynomials from a carefully selected set of candidates having unique and attractive features. When both components are multivariate polynomials, the two-level structure will yield another multivariate polynomial. By employing this two-level structure we obtain hybrid constructions that can benefit from the advantageous features of both components. This allows us to strike different trade-offs between key size, security level, and amenability to various optimizations that we cannot achieve using a single atomic polynomial. For example, we can employ multiple instances of a highly efficient and parallelizable polynomial at the lower level, and combine their outputs at the higher level with another polynomial that is less parallelizable but which yields a better security bound with less key material. One immediate benefit of this two-level structure is that the key material can be reused across the lower-level polynomial instances, thereby reducing the overall key size.

This two-level structure was originally proposed to speed up implementations of classical polynomial hash functions, by decomposing a classical polynomial into a combination of two classical polynomials [24, 18, 10]. It was also used in two recent hash designs, Hash2L and d-2LHash, combining the BRW polynomial with the classical polynomial [9, 5]. We explore this two-level structure more thoroughly than prior work, considering several different candidates at each level and evaluating the security and performance of each composition. We are interested in the security of each composition as an almost-Δ-universal hash function and its performance measured in the customary metric of cycles per byte. To assess the performance of the different options, we build on the framework of Degabriele *et al.* [10], extending its functionality to support our new constructions and using it to compare and contrast different design choices. Through this approach we make several novel contributions, as explained below.

1.1 Our Contributions

Security of Multivariate Polynomials. We start by formally analyzing the security of multivariate polynomial hash functions. Namely, we establish the almost-universality and almost-Δ-universality of multivariate polynomials over any finite field under an injectivity assumption and derive security bounds in terms of the total degree of the multivariate polynomial. This result is stated in Thm. 4. Polynomial hash functions satisfying the conditions of our theorem have the additional benefit of admitting variants with fewer keys.

New Multivariate-Polynomial Hash Designs. Next, we identify suitable candidate polynomials to insert at the lower and higher levels of the two-level hash construction. In such construction, most of the computation happens at the lower level, and thus, an efficient choice of polynomial is crucial here. We consider four distinct lower-level polynomials MMH^+, SQH^+, NMH^+, HKM, where the first three are variants of prior works [21, 16] and HKM is a new design. To the best of our knowledge, no prior work has used these polynomials to instantiate the two-level construction. Each strikes a different trade-off between parallelizability, support for delayed limb realignment, key size, and number of field arithmetic operations

needed. All require relatively large keys when compared to the alternative choice of BRW that is employed in Hash2L and d-2LHash, but they are simpler to implement. Moreover, in the two-level construction, their effective key size is significantly reduced due to the reuse of keys across instances.

At the higher level, our focus shifts to optimizing the trade-off between the amount of key material and obtaining a good security bound for long messages without degrading performance. Here we consider five choices of polynomials: tMMH, tSQH, tNMH, t̂BRW, MHP. The first three are tree-based constructions instantiated with MMH^+, SQH^+, and NMH^+ respectively. The instantiations with SQH^+ and NMH^+ appear to be new. The remaining two constructions, t̂BRW and MHP, are novel multivariate generalizations of BRW and the classical univariate polynomial. Here, t̂BRW also has the added benefit of being more parallelizable than BRW. Their key sizes grow logarithmically in the maximum message length, except for MHP which only requires a constant key size. Additionally, leveraging our security theorem for multivariate polynomials, we present variants of tMMH and t̂BRW that also admit constant key size. In Sec. 4, we describe each of these constructions in detail and analyze their security.

Benchmarking Framework. With the above choices we now have 4×5 distinct hash designs to compare. While their security properties can be readily derived from our exposition in Sec. 4 and Thm. 4, objectively comparing their performance is much more challenging. For this task, we build on the open-source framework for code generation and benchmarking of Degabriele *et al.* [10]. Besides adding our constituent polynomials to the framework, we also extend it to cover binary field arithmetic and multivariate polynomials. The framework does not yield highly optimized implementations of the kind one would obtain by writing directly in assembly. However, it allows us to assess the performance of different hash designs with different parameter choices in a systematic and objective manner.

Factor Analysis. We run benchmarks for all of our two-level hash designs over a small set of parameter choices and over varying message sizes. To make sense of this large data set and filter through it to identify the best-performing designs, we used a statistical factor analysis, see App. E. This analysis examines the variability in performance resulting from changes in parameter choices, exposing the parameters with the highest weight and highest level of interaction. This step was important to validate our intuition on the performance of the designs and to see whether there were any unexpected interactions. As we posited earlier, we confirm that the choice of lower-level polynomial has a much higher influence on the overall performance of the two-level construction than the higher-level one, and this becomes more pronounced as the message length increases. Superblock size and delay parameter emerge as the two parameters with the highest weight.

Outperforming the State of the Art. We conclude by presenting benchmarks for our best-performing multivariate hash designs. The top-performing designs turn out to be MHP-NMH^+, t̂BRW-NMH^+ and tMMH-NMH^+, both over binary fields and prime fields. We compare these to state-of-the-art implementations such as Poly1305 implementations in OpenSSL and HACL* with AVX2, Haberdashery's GMAC, and Hash2L [9], as well as our implementation of d-2LHash [5] over identical prime fields.

For binary fields, our best construction MHP-NMH^+ outperforms all others by up to 25% for message lengths exceeding 4 kB in 128-bit fields and by up to 15% past 6 kB messages in 256-bit fields, while offering around 122 and 250 bits of security, respectively. For messages above 500 bytes, it also offers up to 5 additional bits of security. Note that our auto-generated implementations incur overhead due to the framework's modularity, yet our new designs still significantly outperform state-of-the-art dedicated implementations.

For prime fields, a similar picture emerges. MHP-NMH^+ outperforms even the highly optimized implementation of Poly1305 in OpenSSL by 10%. Over the same field, it also outperforms d-2LHash by up

to 25% while increasing security by 10 bits. Similar effects occur in a larger field.

Finally, we assess the performance impact of instantiating AEAD and Accordion-mode designs with our fastest UHF, MHP-NMH$^+$. For AES-GCM, we expect essentially no measurable improvement. In contrast, for Rijndael256-based AEAD and Accordion modes, using MHP-NMH$^+$ yields a clear speedup in the 5-26% range, depending on the specific construction.

2 Background

Naming and Notation. For any positive integer μ, $\{0,1\}^\mu$ denotes the set of binary strings of length μ. For any string x, $|x|$ denotes its size in bits. If $|x|$ is divisible by 8, x is said to be a byte string. For any two strings x and y, $x \oplus y$ denotes their bitwise XOR. 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, then reinterpreting the results as strings of length $c \cdot \mu$.

If $x = (x_1, \cdots, x_n)$ is a finite tuple, then $|x|$ denotes its length (i.e., n). Furthermore, the tuple $(x_j, \cdots, x_k)$ is denoted by $(x_i)_j^k$. For any finite set S and any positive integer n, $|S|$ denotes the set's cardinality, $[S]^n$ denotes its n-ary Cartesian power, $[S]^{\leq n}$ denotes the set of all tuples of length less than or equal to n over S, $[S]^*$ denotes the set of all finite sequences of elements in S, and $y \leftarrow\!\$\ S$ denotes the process of sampling uniformly at random an element from S and assigning it to y. 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). For any integer $x \in \{0, 1, \ldots, q-1\}$, we denote by $[x]_q$ an injective encoding of x into $\mathbb{F}_q$. We define $\mathrm{odd}(x)$ to be $1 \in \mathbb{F}_q$ if x is odd and $0 \in \mathbb{F}_q$ otherwise. $\log_b$ denotes the base b logarithm, with the convention that $\log_b(0) = 0$.

Universal Hash Functions. Universal hash functions are keyed hash functions whose security properties are quantified through a function ϵ. We next present the formal definitions. 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 $M, M' \in \mathcal{M}$,*

$$\mathrm{Pr}_{r \leftarrow\!\$ \mathcal{R}}\big[H_r(M) = H_r(M')\big] \leq \epsilon(M, M')\,.$$

Intuitively the universality property is a weak form of collision resistance where the key is secret and sampled independently of the message pair. Classically, the probability of such a collision is bounded by a fixed parameter that depends on the maximum length of messages in the domain $\mathcal{M}$. By contrast, we employ a more fine-grained approach replacing this parameter with a function ϵ, which we refer to as the *security bound*, that allows more precise bounds for different pairs of inputs. Smaller values for the function ϵ indicate a lower collision probability and, hence, a more secure universal hash function. Consequently, when we speak of the *security* of such a function, we are referring to how large (or small) the values of ϵ can be. A stronger notion, known as Δ-universality, generalizes the universality 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 $z \in \mathcal{T}$,*

$$\mathrm{Pr}_{r \leftarrow\!\$ \mathcal{R}}\big[H_r(M) = H_r(M') + z\big] \leq \epsilon(M, M')\,.$$

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 $(\{0,1\}^\tau, \overset{\tau}{+})$. If $\epsilon(M, M') = |\mathcal{T}|^{-1}$ 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 $n \in \mathbb{N} \setminus \{0\}$ we say that H is a fixed-input-length hash; if $\mathcal{M} = [S]^*$ we call H a variable-input-length hash.

Hash Function Associated with a Polynomial. The universal hash functions we consider below are all based on polynomials over a finite field $\mathbb{F}_q$, and as such, their key space, digest space and domain, are cartesian products of this field. Let $P : \mathcal{M} \to \mathbb{F}_q[x_1, \ldots, x_v]$ be a function that associates to each message a polynomial over $\mathbb{F}_q$ in v indeterminates. Consider the keyed hash function $H : [\mathbb{F}_q]^v \times \mathcal{M} \to \mathbb{F}_q$ where $H((r_1, \ldots, r_v), M)$ is defined as the evaluation of the polynomial $P(M)$ at $(x_1, \ldots, x_v) = (r_1, \ldots, r_v)$. We call H the hash function *associated with* P. While the above syntax of a universal hash function takes as input only one key, when the key space is $[\mathbb{F}_q]^v$, we refer to v as being the number of keys of H.

Yet in practice, messages, keys, and tags are bit strings. We use the techniques described in [10] to securely convert messages and tags between bit strings and prime fields.

Monomial Ordering. For practical implementation, we impose the ordering $x_v \succ x_{v-1} \succ \cdots \succ x_1$ on the variables and use the graded reverse lexicographic order (*grevlex*) for monomials in v variables. Specifically, in this ordering, monomials are first compared by total degree, with smaller degrees preceding larger ones. Among monomials of the same degree, ordering follows the reverse lexicographic rule, meaning that for two monomials, it compares exponents of x_i starting from $i = 1$ to $i = v$, declaring the monomial with the smaller exponent in the first differing position to be the larger monomial. Most of our arguments are applicable to any other graded ordering. For $(r_i)_1^v \in [\mathbb{F}_q]^v$, we denote by R_i^v the i-th non-constant monomial according to this ordering evaluated at $(x_i)_1^v = (r_i)_1^v$. We refer to a polynomial as *monic* if its leading coefficient – with respect to the chosen monomial order – is equal to 1.

Composition. The theorem below provides a bound on the security of composing two universal hash functions. The composition of universal hash functions serves various purposes, such as converting a universal hash function into a Δ-universal one or reducing the output size of a universal hash function.

Theorem 1 (Composing AU with A(Δ)U)**.** *[34, 10] Consider two keyed hash functions $H_1 : \mathcal{R}_1 \times \mathcal{M} \to \mathcal{T}_1$ and $H_2 : \mathcal{R}_2 \times \mathcal{T}_1 \to \mathcal{T}_2$ that are respectively ϵ_1-almost universal and ϵ_2-almost (Δ-)universal. Their composition is given by*

$$H((r_1, r_2), M) = H_2(r_2, H_1(r_1, M)) .$$

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

$$\epsilon(M, M') = \epsilon_1(M, M') + \max_{r_1 \in \mathcal{R}_1} \epsilon_2(H_1(r_1, M), H_1(r_1, M')).$$

Concatenation. Concatenating polynomial-based universal hash functions can be motivated by two primary objectives: enhancing security or extending the domain. The following theorem focuses on the latter, specifically addressing domain extension.

Theorem 2 (Concatenation of AUs)**.** *[34, 10] Consider two keyed hash functions $H_1 : \mathcal{R} \times \mathcal{M}_1 \to \mathcal{T}_1$ and $H_2 : \mathcal{R} \times \mathcal{M}_2 \to \mathcal{T}_2$ that are respectively ϵ_1-almost universal and ϵ_2-almost universal. Their concatenation is given by*

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

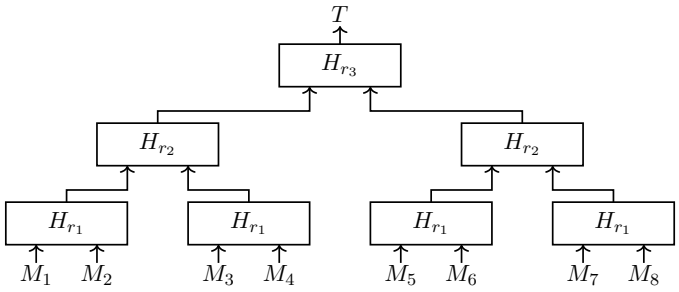


Figure 1: Tree hash function associated with $H : \mathcal{R} \times [\mathcal{M}]^{\leq 2} \to \mathcal{M}$ for the input message $\left(M_1, \ldots, M_8\right) \in [\mathcal{M}]^8$ and keys $(r_1, r_2, r_3) \in [\mathcal{R}]^3$.

Then H is ϵ-almost universal where

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

Tree Construction. The tree construction, first introduced by Wegman and Carter [37, Section 3], is a well-known method for extending the domain of a fixed-input-length almost universal hash function. It is particularly appealing, as it requires only a logarithmic number of independent hash keys for an exponential increase in domain size. We now present a variant of the tree construction from the Badger scheme [7], along with an analysis of its security.

For a hash function $H : \mathcal{R} \times [\mathcal{M}]^{\leq C} \to \mathcal{M}$, the associated tree hash function $G : [\mathcal{R}]^{\lfloor \log_C(n-1) \rfloor + 1} \times [\mathcal{M}]^n \to \mathcal{M}$ is defined recursively as follows:

- For $n \leq C$, $G\left(r_1, (M_j)_1^n\right) = H\left(r_1, (M_j)_1^n\right)$.
- For $n > C$, $G\left((r_j)_1^{v+1}, (M_j)_1^n\right) = H\left(r_{v+1}, \overline{M}\right)$, where

$$\begin{cases} \overline{M} = \left(G\left((r_j)_1^v, (M_j)_1^i\right), G\left((r_j)_1^v, (M_j)_{i+1}^{2i}\right), \ldots, G\left((r_j)_1^v, (M_j)_{\lfloor \frac{n-1}{i} \rfloor \cdot i+1}^n\right)\right), \\ v = \lfloor \log_C(n-1) \rfloor, \\ i = C^v. \end{cases}$$

Note that the tuple size of $\overline{M}$ is less than or equal to C.

At a high level, given a universal hash function H with domain $[\mathcal{M}]^{\leq C}$, the tree hash function G is capable of processing messages of length $n > C$. The function begins by partitioning the message into groups of C elements, which are processed in parallel using H. The resulting outputs are concatenated to form a smaller intermediate message. This reduction step is repeated iteratively, applying H at each iteration with a new independently sampled key, until a single element of $\mathcal{M}$ remains. We illustrate this construction in Fig. 1, considering the case where $C = 2$ and $n = 8$.

Theorem 3 (Tree Hash is AU)**.** *If H is ϵ-almost universal over the domain $[\mathcal{M}]^j$ for all $j \leq C$, then the associated tree hash function G is ϵ'-almost universal with*

$$\epsilon'(M, M') = (\lfloor \log_C(n-1) \rfloor + 1) \cdot \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}').$$

Proof. The proof is done by induction on the length n of the message. For $n \leq C$, G is ϵ'-almost universal over the domain $[\mathcal{M}]^n$ by the assumption on H. Now, let $n > C$, define $v = \lfloor \log_C(n-1) \rfloor$ and assume

that for all $j \leq C^v < n$, G is $\overline{\epsilon}$-almost universal over the domain $[\mathcal{M}]^j$ with

$$\begin{aligned}\overline{\epsilon}(M, M') &= (\lfloor \log_C(j-1) \rfloor + 1) \cdot \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}') \\ &\leq v \cdot \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}').\end{aligned}$$

The last inequality holds because $j - 1 < C^v$ implies $\lfloor \log_C(j-1) \rfloor + 1 < \lfloor \log_C(C^v) \rfloor + 1 = v + 1$, which directly gives $\lfloor \log_C(j-1) \rfloor + 1 \leq v$.

Let $\overline{G}$ be the hash function that maps $(r_1, \ldots, r_v)$ and $(M_1, \ldots, M_n)$ to

$$\begin{aligned}(\, G((r_1, \ldots, r_v), (M_1, \ldots, M_i)), G((r_1, \ldots, r_v), (M_{i+1}, \ldots, M_{2i})), \ldots \\ G((r_1, \ldots, r_v), (M_{\lfloor \frac{n-1}{i} \rfloor \cdot i+1}, \ldots, M_n))\,).\end{aligned}$$

Since $\overline{G}$ is formed by concatenating multiple instances of G, each applied to messages of length at most C^v and individually satisfying $\overline{\epsilon}$-almost universality, it follows from Thm. 2 that $\overline{G}$ itself remains $\overline{\epsilon}$-almost universal.

Finally, noting that the tree hash function G over the domain $[\mathcal{M}]^n$ consists of composing the ϵ-almost universal hash function H with the $\overline{\epsilon}$-almost universal hash function $\overline{G}$, using Thm. 1, we obtain that G over the domain $[\mathcal{M}]^n$ is ϵ'-almost universal with

$$\begin{aligned}\epsilon'(M, M') &\leq \overline{\epsilon}(M, M') + \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}') \\ &\leq (v+1) \cdot \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}'),\end{aligned}$$

where the last inequality follows from

$$\overline{\epsilon}(M, M') \leq v \cdot \max_{\overline{M}, \overline{M}' \in [\mathcal{M}]^{\leq C}} \epsilon(\overline{M}, \overline{M}').$$

□

3 Universality of Multivariate Polynomials

This section describes how to build a universal hash function from a multivariate polynomial over a finite field $\mathbb{F}_q$, and the security bound obtained. In particular, we quantify what security can be expected from a polynomial with v variables. As our designs mainly consider fixed-input-length hash functions, we also describe how to efficiently transform them into variable-input-length (VIL) hash functions in App. A.

Building Universal Hash Functions from Polynomials. Degabriele et al. [10] demonstrated that the universality property of a hash function derived from a univariate polynomial P stems from the injectivity and maximum degree of P. In Thm. 4, we extend this result to multivariate polynomials, replacing the requirement on the degree of a univariate polynomial with a condition on the total degree of a multivariate polynomial. Previous security analyses of multivariate polynomial hash functions originally required a dedicated analysis for each specific polynomial. Thm. 4 provides sufficient conditions for a multivariate polynomial to be a universal hash function along with its corresponding security bound. While this bound is not always tight, injectivity is a necessary condition – without it, there exist distinct messages whose hashes collide for every key. This result provides a novel, unified perspective on the

various polynomial-based universal hash functions over finite fields found in the literature. Specifically, it encompasses both multivariate constructions – such as MMH*, NMH* [21], SQH* [16], CLNH [28], tree-based schemes from [32], XPoly [15] – as well as univariate ones, including BRW [3], Hash2L [9], d-2LHash [5] and various instantiations of the classical polynomial hash [13, 25, 36, 29, 1, 2, 20, 10, 5].

Theorem 4 (Universality of Multivariate Polynomial Hash)**.** *Let $P : \mathcal{M} \to \mathbb{F}_q[x_1, \ldots, x_v]$ be such that for any $M, M' \in \mathcal{M}$, $P(M) - P(M')$ is a polynomial of total degree at most $d(M, M')$. If P is injective on $\mathcal{M}$, then the hash function associated with P is ϵ-almost universal with $\epsilon(M, M') = \frac{d(M,M')}{q}$.*
Additionally, let $Q : \mathcal{M} \to \mathbb{F}_q[x_1, \ldots, x_v]$ be defined as follows: $Q(M)$ is equal to the polynomial $P(M)$ with its constant coefficient set to zero. If Q is injective on $\mathcal{M}$, then the hash function associated with P is ϵ-almost Δ-universal.

Proof. Let M, M' be distinct elements of $\mathcal{M}$. As P is injective on $\mathcal{M}$, $P(M) - P(M')$ is a non-zero multivariate polynomial and, by assumption, of total degree at most $d(M, M')$. Using the Schwartz-Zippel lemma [12, 33, 38], when sampling $r_1, \ldots, r_v$ independently and uniformly at random from $\mathbb{F}_q$, the probability that $(r_1, \ldots, r_v)$ is a root of $P(M) - P(M')$ is at most $\frac{d(M,M')}{q}$. Consequently, the hash function associated with P is ϵ-almost universal with $\epsilon(M, M') = \frac{d(M,M')}{q}$.

Let $z \in \mathbb{F}_q$ and assume that Q is injective on $\mathcal{M}$. As Q is injective, $Q(M) - Q(M')$ is a non-zero polynomial whose constant coefficient is zero. Thus, it is a non-zero polynomial of degree at least one. From the definition of Q, we obtain that $P(M) - P(M') - z$ is also a non-zero polynomial of degree at least one. By assumption, $P(M) - P(M') - z$ is of total degree at most $d(M, M')$. Then, similarly as before, the probability that $(r_1, \ldots, r_v)$ is a root of $P(M) - P(M') - z$ is at most $\frac{d(M,M')}{q}$, and therefore the hash function associated with P is ϵ-almost Δ-universal. □

The theorem highlights that an important parameter to consider when constructing a multivariate polynomial hash function is its total degree. As security degrades with an increasing total degree, the goal in designing such hash functions should be to select polynomials with a low total degree. While not explicitly stated in the theorem, another factor influencing security is the number of variables v.

Consider the scenario where we fix a domain $\mathcal{M} = \mathbb{F}_q^n$ and aim to construct a polynomial P that achieves optimal security. For this, P must injectively map $\mathbb{F}_q^n$ into a set of polynomials with the smallest possible total degree. Since the number of monomials in v variables of degree at most d is $\binom{v+d}{v}$, the number of polynomials in v variables with degree at most d is $q^{\binom{v+d}{v}}$. By the injectivity requirement of P, the minimum degree d is the smallest value that satisfies $q^n \leq q^{\binom{v+d}{v}}$, which simplifies to $n \leq \binom{v+d}{v}$. Using the inequality chain $\left(\frac{d}{v}\right)^v \leq \left(\frac{v+d-1}{v}\right)^v \leq \binom{v+d-1}{v} < n$, we derive the bound $d < v \cdot n^{1/v}$. Thus, to hash a message of n field elements using a polynomial with v variables, the best achievable security according to the theorem is approximately $\frac{v \cdot n^{1/v}}{q}$. Notably, the theorem confirms the intuition that increasing the number of variables allows for higher security. In some cases (e.g., ChaCha20-Poly1305), the construction supplies the extra variables/keys at no additional cost.

Reducing Keys of Multivariate Polynomial Hash. Thm. 4 enables constructing a universal hash function with key space $[\mathbb{F}_q]^v$ from an injective function $P : \mathcal{M} \to \mathbb{F}_q[x_1, \ldots, x_v]$. From P, one can derive another universal hash function with a reduced key space $[\mathbb{F}_q]^{v-1}$ by first constructing a second injective function $\overline{P} : \mathcal{M} \to \mathbb{F}_q[x_1, \ldots, x_{v-1}]$, followed by an application of Thm. 4 to $\overline{P}$.

The construction is as follows. For all $M, M' \in \mathcal{M}$, if $P(M) - P(M')$ has degree at most d in the variable x_{v-1}, define $\overline{P}(M)$ as the polynomial $P(M)$, with the variable x_v replaced by x_{v-1}^{d+1}. This transformation from P to $\overline{P}$ is known as Kronecker substitution [26]. The resulting $\overline{P}(M)$ is a polynomial in $v-1$ variables.

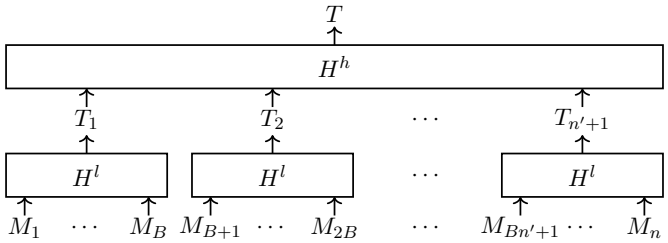


Figure 2: A basic two-level hashing approach: the input message $(M_1, \ldots, M_n) \in [\mathbb{F}_q]^n$ is partitioned into $n'+1$ superblocks, each comprising B elements of $\mathbb{F}_q$ (except the last superblock, which may be shorter).

If P is injective, then $\overline{P}$ is also injective, since this substitution is injective on polynomials whose degree in x_{v-1} is at most d.

By combining this transformation with Thm. 4, one can construct a universal hash function with a smaller key space than $[\mathbb{F}_q]^v$ starting from a polynomial P in v variables. However, the reduction in key space comes with a trade-off, as the total degree increases, resulting in reduced security compared to the original hash function. By iterating this transformation, it is possible to derive a single-key universal hash function based on a multivariate polynomial.

4 Two-Level Polynomial Hash Designs

In this section, we outline the rationale for our new universal hash designs and formally describe their structure, along with efficient evaluation strategies. Our exploration of multivariate polynomial hashing focuses on polynomials with a two-level structure, as illustrated in Fig. 2. Originally introduced to accelerate classical univariate polynomial evaluation [24, 18], the two-level structure was later adapted in designs combining the BRW polynomial with the classical univariate polynomial [9, 5]. Here, we extend this framework to a broader class of polynomial instantiations – particularly multivariate constructions – that offer improved security. As we explain below, this focus on two-level constructions is motivated by their ability to strike a better trade-off between key length, number of multiplications, delayed limb realignment, and parallelizability.

The Two-Level Design Approach. The two-level approach described in Fig. 2 operates by first processing the input message $(M_1, \ldots, M_n) \in [\mathbb{F}_q]^n$ through multiple parallel instances of a lower-level hash function H^l, where each instance processes a fixed, small number B of field elements, except for the last instance, which may process fewer elements. Each group of B field elements is referred to as a superblock. The outputs from these parallel hashes are concatenated and then processed through another independent hash function H^h, at the higher level. According to Thm. 2 , if H^l is ϵ_1-almost universal, the concatenated hash functions from the lower level will also be ϵ_1-almost universal. Moreover, leveraging Thm. 1 on the composition of universal hash functions, if H^h is ϵ_2-almost universal, the overall two-level construction achieves $(\epsilon_1 + \epsilon_2)$-almost universality.

This two-level design offers several advantages. By judiciously selecting H^l and H^h – namely, opting for a highly efficient H^l paired with an H^h that provides good security bounds with short keys – we can harness the strengths of both functions. An alternative structure, Fig. 3 in Sec. 4.2, introduces a different trade-off that we will explore later. We now detail the criteria guiding our selection of candidates for H^l at the lower level and H^h at the higher level, and explain how we arrived at the particular choices described in Secs. 4.1 and 4.2, from among the broader range of possibilities.

Lower-Level Polynomial Selection. The lower level of our two-level hash constructions (see Figs. 2 and 3),

	MMH^+	SQH^+	NMH^+	HKM
# keys	$B-1$	$B-1$	B	$B/2+1$
Total Degree	1	2	2	$B/2+1$
FIL AU bound	$1/q$	$1/q$	$1/q$	$B/(2q)$
# field multiplications	$B-1$	-	$B/2$	$B/2$
# field squarings	-	$B-1$	-	-
# field additions	$B-1$	$2B-2$	$3B/2-1$	B
Fully-Delayed Limb Realignment	yes	yes	yes	no
Parallelizable	yes	yes	yes	no

Table 1: Properties of lower-level polynomials, for messages consisting of B field elements in $\mathbb{F}_q$.

where most of the data is processed, demands an efficient evaluation algorithm for H^l. Ideally, H^l should require few multiplications, support parallelism, and allow for optimizations such as delayed limb realignment. A key aspect of the two-level structure is that the same key can be reused across all parallel instances of H^l, thus amortizing the cost of generating long keys when processing long messages. Additionally, since H^l processes only small messages of B field elements, this ensures that any potential drawbacks of H^l related to long keys or weaker security bounds for long messages, have limited impact on the overall two-level design. Consequently, these factors should not overly constrain the selection of candidates for H^l. We therefore select lower-level polynomials by the following criteria: (i) few field multiplications per message block, (ii) compatibility with delayed limb realignment and parallelizability. From the literature we retained all polynomials meeting best our criteria – MMH^+, SQH^+, and NMH^+ (variants of [21, 16]) – and introduced a new polynomial, HKM. We detail these in Sec. 4.1. We do not use the classical polynomial or BRW as lower-level choices: the former is equivalent to our reduced-key variant of MMH^+ and does not reduce the number of field multiplications, while the latter is less compatible with the delayed-operation and parallel-evaluation techniques that dominate optimized implementations. We note that BRW was already analyzed at the lower level of a two-level construction in [9, 5], so we still compare these designs with ours in Sec. 5.

Higher-Level Polynomial Selection. For the higher level of our two-level constructions (cf. Figs. 2 and 3), the requirements are slightly different than for the lower level. Since less data is processed at this stage, performance is less critical. Still, a polynomial H^h that supports parallel evaluation is appealing, as it enables full parallelism across the two-level structure. Because H^h processes a number of field elements that scales with the length n of the input message $\big(M_1, \ldots, M_n\big) \in [\mathbb{F}_q]^n$, it is crucial for H^h to have short keys and offer a good security bound ϵ_2. Here the dominant criteria are instead: (i) small key material for long messages, (ii) a security bound that grows slowly with the number of superblocks. Among existing designs, we again selected and adapted the polynomials that meet our criteria: tMMH, tSQH, tNMH, t̂BRW, and MHP. The first three are tree-based constructions instantiated with MMH^+, SQH^+, and NMH^+ respectively. The remaining two are multivariate generalizations of BRW and the classical univariate polynomial. Full details appear in Sec. 4.2.

4.1 Polynomial designs at the lower level

We identified four polynomial candidates for the lower level: MMH^+, SQH^+, NMH^+, and HKM. Their key characteristics are summarized in Tab. 1. The first three polynomials support fully-delayed limb realignment and parallelization, while the last three focus on reducing the number of field multiplications. A notable aspect shared by all these polynomials is that achieving these properties necessitates a number

of keys that is linear with respect to the superblock size B. Deriving the almost universality bound from Tab. 1 for messages of fixed-input-length size B, is a straightforward application of Thm. 4 to each polynomial.

MMH$^+$. We define the polynomial hash MMH$^+$ as follows: for a message $M = (M_i)_1^B \in [\mathbb{F}_q]^B$ and a key $(r_i)_1^{B-1} \in [\mathbb{F}_q]^{B-1}$,

$$\mathsf{MMH}^+((r_i)_1^{B-1}, M) = \sum_{i=1}^{B-1} M_i \cdot r_i + M_B \,.$$

When $M_B = 0$, MMH$^+$ is equivalent to the inner product construction MMH* from [21]. Its main advantage lies in its high parallelizability and its ability to fully delay limb realignment, up to the limit imposed by its limb representation.

Reduced-Key Variant. Using the technique described in Sec. 3, we can define variants of MMH$^+$ that require a fixed number v of keys, independent of the superblock size B. However, in this case, the degree and security bound would scale linearly with B. To address this, we propose v-MMH$^+$ as an alternative. It is defined as follows:

$$\begin{aligned} v\text{-}\mathsf{MMH}^+((r_i)_1^v, M) &= \mathsf{MMH}^+((R_1^v, R_2^v, \ldots, R_{B-1}^v), M) \\ &= M_1 \cdot r_1 + \cdots + M_v \cdot r_v + M_{v+1} \cdot r_1^2 + M_{v+2} \cdot r_1 \cdot r_2 + M_{v+3} \cdot r_1 \cdot r_3 + \cdots \,. \end{aligned}$$

Concretely, we produce the $B-1$ keys for MMH$^+$ by taking the first $B-1$ monomials in v variables (listed in grevlex order) and evaluating them at $\big(x_1, \ldots, x_v\big) = \big(r_1, \ldots, r_v\big)$ (see Sec. 2). Since the monomials R_i^v are ordered according to a graded ordering, the total degree of v-MMH$^+$ is the smallest possible for a message of B field elements, and thus bounded by $v \cdot B^{1/v}$ (cf. Sec. 3). By Thm. 4, v-MMH$^+$ is ϵ-almost universal with $\epsilon \leq \frac{v \cdot B^{1/v}}{q}$. Compared to MMH$^+$, this construction requires $B - v$ additional multiplications to compute the monomials R_i^v. However, in a two-level construction, these extra operations are amortized for long messages, as they can be reused across instances of the lower-level hash H^l. A retained benefit of MMH$^+$ is that, once the monomials R_i^v are precomputed, v-MMH$^+$ supports high parallelizability and fully delayed limb realignment.

SQH$^+$. Since squaring is generally faster than multiplication, a natural approach to accelerate MMH$^+$ is to replace multiplications with squarings. SQH$^+$ does this in finite fields with characteristic different from 2, and is defined as follows:

$$\mathsf{SQH}^+((r_i)_1^{B-1}, M) = \sum_{i=1}^{B-1} (M_i + r_i)^2 + M_B \,.$$

This function is the same as SQH* [16] when $M_B = 0$. The SQH$^+$ construction retains the parallelizability and fully delayed limb realignment benefits of MMH$^+$, aiming to improve efficiency by substituting squarings for multiplications, albeit with an increase in the number of additions required.

Reduced-Key Variant. Similarly as for MMH$^+$, we can define a variant of SQH$^+$ with fewer keys, that is ϵ-almost universal with $\epsilon \leq \frac{v \cdot B^{1/v}}{q}$, as follows:

$$v\text{-}\mathsf{SQH}^+((r_i)_1^v, M) = \mathsf{SQH}^+((R_1^v, R_2^v, R_3^v, \ldots), M) \,.$$

NMH$^+$. One approach to reduce the number of multiplications in MMH$^+$ is to employ a pseudo inner product construction. NMH$^+$ requires half as many multiplications as MMH$^+$, at the cost of slightly more additions. The definition of NMH$^+$ is as follows:

$$\mathsf{NMH}^+((r_i)_1^B, M) = \sum_{i=1}^{\lfloor B/2 \rfloor} (M_{2i-1} + r_{2i-1}) \cdot (M_{2i} + r_{2i}) + \mathrm{odd}(B) \cdot M_B.$$

Like MMH$^+$ and SQH$^+$, NMH$^+$ supports parallelization and fully delayed limb realignment. The case where B is even corresponds to the NMH* construction in [21].

Reduced-Key Variant. Following the approach for MMH$^+$, we define a reduced-key variant v-NMH$^+$, which is ϵ-almost universal with $\epsilon \leq \frac{v \cdot B^{1/v}}{q}$:

$$v\text{-}\mathsf{NMH}^+((r_i)_1^v, M) = \mathsf{NMH}^+((R_1^v, R_{2\lfloor B/2 \rfloor}^v, R_2^v, R_{2\lfloor B/2 \rfloor - 1}^v, R_3^v, R_{2\lfloor B/2 \rfloor - 2}^v, \ldots), M).$$

In the construction of v-NMH$^+$, compared to v-MMH$^+$ and v-SQH$^+$, the ordering of the monomials R_i^v has been adjusted to lower the total degree of v-NMH$^+$.

HKM. The last lower-level polynomial hash we consider is a new design, denoted HKM (Half Key / Half Multiply). Compared to the previous constructions, HKM uses only half as many keys and uses only $\lfloor B/2 \rfloor$ multiplications with relatively few additions. However, this improved number of operations comes with trade-offs: namely, a weaker security bound and no support for fully delayed limb realignment and parallelization. The recursive definition of HKM is as follows:

- $\mathsf{HKM}(r_1, M_1) = r_1 + M_1$
- $\mathsf{HKM}((r_1, \ldots, r_{\lfloor B/2 \rfloor + 1}), (M_1, \ldots, M_B)) =$

$$\mathsf{HKM}((r_1, \ldots, r_{\lfloor B/2 \rfloor}), (M_1, \ldots, M_{2\lfloor B/2 \rfloor - 1})) \cdot (r_{\lfloor B/2 \rfloor + 1} + M_{2\lfloor B/2 \rfloor}) + \begin{cases} 0 & \text{if } B \text{ even,} \\ M_B & \text{otherwise.} \end{cases}$$

While it is possible to define a reduced-key variant of this hash function, it would offer only a degraded security bound.

4.2 Polynomial designs at the higher level

We consider five candidate constructions: tMMH, tSQH, tNMH, t̂BRW, and MHP, with their properties summarized in Tab. 2. These constructions follow the framework shown in Fig. 2, with the exception of t̂BRW, which employs the alternative structure depicted in Fig. 3. Although this alternative construction involves processing more data through the higher-level function H^h, it introduces a compelling efficiency trade-off, particularly advantageous when paired with t̂BRW.

At the higher level, the functions tMMH, tSQH, and tNMH follow the tree construction analyzed in Thm. 3, built respectively from MMH$^+$, SQH$^+$, and NMH$^+$. The last two constructions, t̂BRW and MHP, are multivariate generalizations of Bernstein's univariate BRW polynomial and the classical univariate polynomial. The tree-based structures of the first four constructions require only a number of keys logarithmic in the message length n, while MHP requires a fixed number v. When the number of keys required by a polynomial hash function depends on the message length n, we refer to it as a *Variable-Key-Length* (VKL) polynomial. This property characterizes our four tree-based constructions except MHP. For tMMH and t̂BRW, this number of keys can be reduced even further to a fixed number v, independent of n, using the method in Sec. 3. Though similar reductions are possible for tSQH and tNMH, their high degree yields weak bounds, so we instead apply Thm. 3 for these cases. Bounds for all other constructions are derived using Thm. 4.

	tMMH	tSQH	tNMH	$\hat{\text{t}}$BRW	MHP
# keys	$\lfloor \log_2(n') \rfloor + 1$	$\lfloor \log_2(n') \rfloor + 1$	$2(\lfloor \log_3(n') \rfloor + 1)$	$\lfloor \log_2(\overline{n}) \rfloor + 1$	v
Total Degree	$\lfloor \log_2(n') \rfloor + 1$	$2^{\lfloor \log_2(n') \rfloor + 1}$	$2^{\lfloor \log_3(n') \rfloor + 1}$	$\lfloor \log_2(\overline{n}) \rfloor + 1$	less than $v(n'+1)^{\frac{1}{v}}$
FIL AU bound	$\frac{\lfloor \log_2(n') \rfloor + 1}{q}$	$\frac{\lfloor \log_2(n') \rfloor + 1}{q}$	$\frac{\lfloor \log_3(n') \rfloor + 1}{q}$	$\frac{\lfloor \log_2(\overline{n}) \rfloor + 1}{q}$	$\frac{v(n'+1)^{\frac{1}{v}}}{q}$
# field multiplications	n'	-	$n'/2 + \log_3(n')$	$\overline{n}$	n'
# field squarings	-	n'	-	-	-
# field additions	n'	$2n'$	$3n'/2 + \log_3(n')$	$2\overline{n}$	n'
Fully-Delayed Limb Realignment	yes	no	no	no	yes
Parallelizable	yes	no	no	partially	yes

Table 2: Properties of higher-level polynomials, excluding the cost of the lower-level hash H^l. All designs follow the two-level framework from Fig. 2 with $n' = \lceil \frac{n}{B} \rceil - 1$, except for $\hat{\text{t}}$BRW that follows Fig. 3 with $\overline{n} = \left\lfloor \frac{n}{B+1} \right\rfloor$.

Parameters and Analysis. To support a two-level construction with one of the lower-level polynomial hash from Sec. 4.1, we parameterize our higher level hash functions with a generic polynomial H^l at the lower level. This formulation allows us to isolate how properties of H^l influence the overall construction. The lower-level polynomial H^l operates on superblocks of at most B field elements and requires b keys $r_1, \ldots, r_b$.

Since we aim to compare only the higher-level polynomials in Tab. 2, we exclude the cost of H^l from the table. Here, we outline that additional cost. For tMMH, tSQH, tNMH, and MHP, the cost of running parallel instances of H^l is approximately $\lceil \frac{n}{B} \rceil$ times the cost of H^l on a single superblock. For $\hat{\text{t}}$BRW, it is instead multiplied by $\left\lceil \frac{n}{B+1} \right\rceil$. The additional keys required is limited to b, as these keys can be reused across H^l's parallel instances.

In terms of security, the bound of the two-level construction, which depends on the polynomials used at each level, is simply the sum of the individual bounds for each hash function from Tabs. 1 and 2. Finally, to ensure universality and security for variable input lengths, one of the transforms from App. A should be added.

tMMH. The polynomial hash tMMH is constructed following the tree-based approach of Thm. 3, instantiated with MMH$^+$ and $C = 2$, thereby forming a binary tree. Each leaf in this tree corresponds to the hash of a superblock through a lower-level polynomial hash function H^l. We denote this composition by tMMH$[H^l]$, defined as follows:

- For $n \leq B$,
$$\text{tMMH}[H^l]\left((r_j)_1^b, (M_j)_1^n\right) = H^l\left((r_j)_1^b, (M_j)_1^n\right).$$
- For $n > B$,
$$\text{tMMH}[H^l]\left((r_j)_1^v, (M_j)_1^n\right) = \text{tMMH}[H^l]\left((r_j)_1^{v-1}, (M_j)_1^i\right) \cdot r_v + \text{tMMH}[H^l]\left((r_j)_1^{v-1}, (M_j)_{i+1}^n\right),$$
where $v = \lfloor \log_2(n') \rfloor + b + 1$, $i = B \cdot 2^{\lfloor \log_2(n') \rfloor}$ and $n' = \lceil \frac{n}{B} \rceil - 1 = \lfloor \frac{n-1}{B} \rfloor$.

In terms of computational efficiency, evaluating tMMH$[H^l]$ involves, beyond the cost of computing the individual H^l instances, an additional n' field multiplications and n' field additions, where $n'+1$ represents

$\mathrm{tMMH}[H^l]((r_1, \ldots, r_v), (M_1, \ldots, M_n))$

$n' \leftarrow \left\lfloor \frac{n-1}{B} \right\rfloor$

// processing message by blocks of B

for $i = 0$ **to** $n' - 1$ **do**

  $T_0 \leftarrow H^l_{r_1,\ldots,r_b}(M_{B\cdot i+1}, \ldots, M_{B\cdot(i+1)})$

  // exponent of highest power of 2 dividing $i+1$

  $s \leftarrow \mathrm{ExpHighest2Power}(i+1)$

  $T_{s+1} \leftarrow \left(\sum_{j=0}^{s} T_j\right) \cdot r_{b+s+1}$

// processing remaining blocks

$T_0 \leftarrow H^l_{r_1,\ldots,r_b}(M_{B\cdot n'+1}, \ldots, M_n)$

// summing the result in a single accumulator

for $i = 1$ **to** $\lfloor \log_2(n') \rfloor + 1$ **do**

  // check if i-th bit of n' is 1

  if $\mathrm{bit}(i, n') = 1$ **then**

    $T_0 \leftarrow T_0 + T_i$

return T_0

Algorithm 1: Sequential evaluation algorithm of $\mathrm{tMMH}[H^l]$.

the total number of superblocks. Compared to the simple polynomial hash functions described in Sec. 4.1, which require only fixed small memory, the tree structure of $\mathrm{tMMH}[H^l]$ necessitates logarithmic space in n' to store intermediate results. Although $\mathrm{tMMH}[H^l]$ is defined recursively here for clarity, both sequential and parallel evaluation methods are also available in Algs. 1 and 2. The sequential approach employs the ExpHighest2Power function, which calculates the exponent of the highest power of two that divides its input—equivalent to counting trailing zeros in binary form, a task efficiently handled by compiler-intrinsic functions. To obtain delayed limb alignment, one can combine lower levels in the tree and precompute key powers for these grouped levels. A key property of tMMH is its linearity: for $r \in \mathbb{F}_q$ and H_1, H_2 being two functions:

$$\mathrm{tMMH}[r \cdot H_1 + H_2] = r \cdot \mathrm{tMMH}[H_1] + \mathrm{tMMH}[H_2].$$

This linearity facilitates parallel evaluation of $\mathrm{tMMH}[H^l]$, if H^l itself is expressible as a sum of functions, as in the cases of MMH^+, SQH^+, and NMH^+. For further parallelism, beyond that allowed by H^l, or when H^l cannot be decomposed, we can leverage the recursive form of tMMH:

$$\mathrm{tMMH}[H^l] = \overline{\mathrm{tMMH}}[r_{b+1} \cdot H^l + H^l] = r_{b+1} \cdot \overline{\mathrm{tMMH}}[H^l] + \overline{\mathrm{tMMH}}[H^l],$$

where tMMH uses keys $(r_1, \ldots, r_v)$, while $\overline{\mathrm{tMMH}}$ employs keys $(r_1, \ldots, r_b, r_{b+2}, \ldots, r_v)$. This property underlies the parallel evaluation algorithm in Alg. 2.

While a transform from App. A can securely adapt tMMH for VIL messages, Δ-universality for VIL messages can also be directly achieved by enforcing a structure on the messages, to satisfy the injectivity conditions of Thm. 4. Specifically, for $\mathrm{tMMH}[H^l]$, this entails restricting the domains as follows: $[\mathbb{F}_q]^* \times \mathbb{F}_q^* \times \{0\}$ when the lower-level polynomial H^l is SQH^+, NMH^+, or HKM, and $[\mathbb{F}_q^*]^* \times \{0\}$ when it is MMH^+.

Reduced-Key Variant. Using the reduced-key transformation from Sec. 3, we introduce a variant of tMMH that operates with a fixed number v of keys, independent of the message length n. For the lower-level hash

$\mathrm{tMMH}[H^l]((r_1,\ldots,r_v),(M_1,\ldots,M_n))$

$n' \leftarrow \left\lfloor \frac{n-1}{B} \right\rfloor$

// split message in superblocks of size B

for $i = 1$ **to** n' **do**

  $\overline{M}_i \leftarrow M_{(i-1)\cdot B+1},\ldots,M_{i\cdot B}$

$\overline{M}_{n'+1} \leftarrow M_{n'\cdot B+1},\ldots,M_n$

// 2^t independent branches that can be evaluated in parallel

for $j = 1$ **to** 2^t **do**

  $T_j \leftarrow \mathrm{tMMH}[H^l]((r_1,\ldots,r_b,r_{b+t+1},\ldots,r_v),$
  $(\overline{M}_j,\overline{M}_{j+2^t},\overline{M}_{j+2\cdot 2^t},\overline{M}_{j+3\cdot 2^t},\ldots))$

// accumulating results from each branch

$T \leftarrow \mathrm{tMMH}[\mathsf{Id}]_{r_{b+1},r_{b+2},\ldots,r_{b+t}}(T_1,\ldots,T_{2^t})$

return T

Algorithm 2: 2^t-branch parallel evaluation algorithm for $\mathrm{tMMH}[H^l]$, where Id is the identity function.

H^l, we require that for any $M, M' \in [\mathbb{F}_q]^B$, the difference $H^l(M) - H^l(M')$ is a multivariate polynomial of total degree at most d_l, and that $v \geq b$. Observing that the degree of the polynomial $\mathrm{tMMH}[H^l]$ in each variable $r_{b+1},\ldots,r_v$ is 1, we provide the following definition:

$$v\text{-}\mathrm{tMMH}[H^l]((r_1,\ldots,r_v),M) = \mathrm{tMMH}[H^l]((r_1,\ldots,r_v,r_1^{d_l+1},\ldots,r_v^{d_l+1},r_1^{2\cdot(d_l+1)},\ldots,r_v^{2\cdot(d_l+1)},\\ r_1^{4\cdot(d_l+1)},\ldots,r_v^{4\cdot(d_l+1)},\ldots),M).$$

Evaluating v-$\mathrm{tMMH}[H^l]$ incurs an additional cost beyond that of $\mathrm{tMMH}[H^l]$, specifically for computing key powers required by the definition. This involves a number of multiplications and squarings that grows logarithmically with the message length n. For many of the lower-level polynomials described in Sec. 4.1, the first d_l powers of the keys are already computed during the lower-level evaluation, enabling their direct reuse at the upper level. By construction, the total degree of this polynomial is at most $2 \cdot v \cdot (d_l+1) \cdot {n'}^{\frac{1}{v}}$. Applying Thm. 4, we deduce that v-$\mathrm{tMMH}[H^l]$ is a FIL ϵ-almost universal hash with $\epsilon \leq \frac{2\cdot v\cdot(d_l+1)\cdot {n'}^{\frac{1}{v}}}{q}$.

tSQH. The polynomial hash function tSQH is based on the tree construction outlined in Thm. 3, similarly to tMMH, but it uses SQH^+ with domain $\mathbb{F}_q^2$ as its core operation. Consequently, the construction is secure only over fields whose characteristic is not equal to two. The construction is specified as follows:

- For $n \leq B$,
$$\mathrm{tSQH}[H^l]\left((r_j)_1^b,(M_j)_1^n\right) = H^l\left((r_j)_1^b,(M_j)_1^n\right).$$
- For $n > B$,
$$\mathrm{tSQH}[H^l]\left((r_j)_1^v,(M_j)_1^n\right) = \left(\mathrm{tSQH}[H^l]\left((r_j)_1^{v-1},(M_j)_1^i\right) + r_v\right)^2 + \mathrm{tSQH}[H^l]\left((r_j)_1^{v-1},(M_j)_{i+1}^n\right),$$
where $v = \lfloor \log_2(n') \rfloor + b + 1$, $i = B \cdot 2^{\lfloor \log_2(n') \rfloor}$ and $n' = \left\lceil \frac{n}{B} \right\rceil - 1$.

The tree structure of tSQH aligns with that of tMMH, enabling a similar sequential evaluation algorithm. However, tSQH replaces each multiplication with squaring and addition operations, which can offer computational advantages due to the typically lower cost of squaring. Unlike MMH^+, SQH^+ lacks a linearity property, which prevents tSQH from being parallelized.

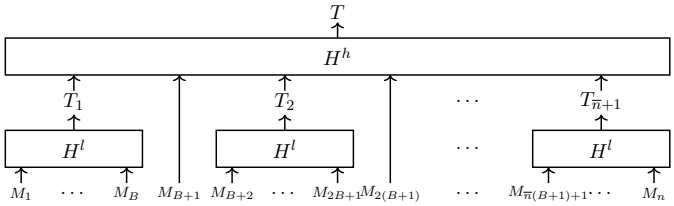


Figure 3: Refined Two-Level Hash for $\hat{\mathsf{t}}\mathsf{BRW}$: the input message $(M_1, \ldots, M_n) \in [\mathbb{F}_q]^n$ is partitioned into $\overline{n}+1$ superblocks, each comprising B elements of $\mathbb{F}_q$ (except the last superblock, which may be shorter).

From a security perspective, tSQH qualifies as a polynomial hash, suggesting that the security bound could be derived using Thm. 4. Yet, due to its high total degree, the bound produced by this approach is relatively loose. Applying Thm. 3 directly yields a tighter bound, as listed in Tab. 2. This high degree also prevents an effective reduced-key version of tSQH; attempting a reduced-key approach via Thm. 4 would result in an unacceptably weak security bound.

tNMH. To improve the efficiency of tMMH, a natural approach is to replace MMH^+ with NMH^+ in the tree construction with $C = 3$. This yields a ternary tree, rather than a binary one, with a lower-level hash H^l at the leaves. The resulting hash function tNMH$[H^l]$ is defined as follows:

- For $n \le B$,
$$\mathrm{tNMH}[H^l]\left((r_j)_1^b, (M_j)_1^n\right) = H^l\left((r_j)_1^b, (M_j)_1^n\right).$$
- For $n > B$,
  - if $n \le 2 \cdot i$,
$$\mathrm{tNMH}[H^l]\left((r_j)_1^v, (M_j)_1^n\right) = \left(\mathrm{tNMH}[H^l]\left((r_j)_1^{v-2}, (M_j)_1^i\right) + r_{v-1}\right) \cdot \left(\mathrm{tNMH}[H^l]\left((r_j)_1^{v-2}, (M_j)_{i+1}^n\right) + r_v\right),$$
  - if $n > 2 \cdot i$,
$$\mathrm{tNMH}[H^l]\left((r_j)_1^v, (M_j)_1^n\right) = \mathrm{tNMH}[H^l]\left((r_j)_1^{v-2}, (M_j)_{2i+1}^n\right) + \left(\mathrm{tNMH}[H^l]\left((r_j)_1^{v-2}, (M_j)_1^i\right) + r_{v-1}\right) \cdot \left(\mathrm{tNMH}[H^l]\left((r_j)_1^{v-2}, (M_j)_{i+1}^{2i}\right) + r_v\right),$$

  where $v = 2 \cdot \lfloor \log_3(n') \rfloor + b + 2$, $i = B \cdot 3^{\lfloor \log_3(n') \rfloor}$ and $n' = \lceil \frac{n}{B} \rceil - 1$.

Since tNMH adopts a structure analogous to that of tMMH, its sequential evaluation can be carried out similarly to the one depicted for tMMH in Alg. 1. The use of NMH^+ in tNMH reduces the number of required field multiplications. However, this advantage is offset by the ternary structure of the tree, which introduces additional complexity. The efficient ExpHighest2Power function, pivotal in handling the binary-tree structure of tMMH, is replaced with the less efficient ExpHighest3Power, which computes the largest power of three dividing a given input. Additionally, the final summation across tree levels in tNMH is more intricate, leading to reduced overall efficiency.

Similar to tSQH, tNMH is not parallelizable, and its high polynomial degree precludes the application of Thm. 4 for constructing a reduced-key variant.

$\hat{\mathsf{t}}$BRW. The polynomial tNMH, while reducing the number of field multiplications by half compared to tMMH, comes with notable disadvantages: it requires a key twice as large, employs a ternary tree structure,

lacks support for parallelization, and has a high total degree. The construction of $\hat{\mathsf{tBRW}}$ addresses these limitations by adopting a more balanced design that lies between tMMH and tNMH. Specifically, it modifies the two-level hash from Fig. 2 to the structure described in Fig. 3, integrating a tailored variant of the tree construction for H^h. This approach retains the efficiency of reducing field multiplications, achieves a lower total degree, and aligns more closely with the structure of a binary tree. To further enhance efficiency, $\hat{\mathsf{tBRW}}$ reduces key requirements and allows partial parallelization, all while maintaining applicability of Thm. 4. However, this necessitates that H^l be a monic polynomial hash of degree at least one, thereby excluding generic universal hashes at the lower level. Most polynomials from Sec. 4.1 meet the requirements for H^l when $B > 1$. The notable exceptions are MMH^+ and v-MMH^+, which require restricting the message input M_{B-1} to the multiplicative identity in $\mathbb{F}_q$, in order to fulfill the conditions imposed by the construction.

The formal specification of $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ is as follows:

- For $n \leq B$,

$$\hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^b, (M_j)_1^n\right) = H^l\left((r_j)_1^b, (M_j)_1^n\right).$$

- For $n > B$,

$$\hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^v, (M_j)_1^n\right) = \\ \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, (M_j)_1^{i-1}\right) \cdot (r_v + M_i) + \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, (M_j)_{i+1}^n\right),$$

  where $v = \lfloor \log_2(\overline{n}) \rfloor + b + 1$, $i = (B+1) \cdot 2^{\lfloor \log_2(\overline{n}) \rfloor}$ and $\overline{n} = \left\lfloor \frac{n}{B+1} \right\rfloor$.

At first glance, the higher level of $\hat{\mathsf{tBRW}}$ might appear to require a similar number of multiplications as tMMH (see Tab. 2). However, examining the construction in Fig. 3 reveals that $\hat{\mathsf{tBRW}}$ processes twice as many inputs at the higher level. These additional inputs do not need to be processed at the lower level, and $\hat{\mathsf{tBRW}}$ handles them efficiently at the higher level, requiring $\overline{n}$ multiplications and $2\overline{n}$ additions to process $2\overline{n}$ input elements. This design effectively reduces the number of multiplications compared to tMMH.

The higher level of $\hat{\mathsf{tBRW}}$ follows a hybrid tree construction that is in between a binary and ternary tree. Its sequential evaluation algorithm, shown in Alg. 3, achieves efficiency comparable to a binary tree. Moreover, like tMMH, $\hat{\mathsf{tBRW}}$ can benefit from the structure of H^l to enable parallelization: for $r \in \mathbb{F}_q$ and functions H_1, H_2, $\hat{\mathrm{t}}\mathrm{BRW}[r \cdot H_1 + H_2] = r \cdot \hat{\mathrm{t}}\mathrm{BRW}[H_1] + \hat{\mathrm{t}}\mathrm{BRW}[H_2]$. Despite these advantages, $\hat{\mathsf{tBRW}}$ introduces a challenge for full parallelization when H^l cannot be decomposed into two additive components. Unlike tMMH, the higher level of $\hat{\mathsf{tBRW}}$ directly processes some message elements, limiting the algorithm's parallelizability. While closely following the analysis of BRW [3], the injectivity property of $\hat{\mathsf{tBRW}}$, required for applying Thm. 4, is proven in App. B.1.

Reduced-Key Variant. Analogous to tMMH, the reduced-key transformation from Sec. 3 allows the construction of a variant of $\hat{\mathsf{tBRW}}$ that employs a fixed number v of keys. This variant requires that the lower-level hash H^l be a monic polynomial hash, with a total degree bounded between one and d_l, and that $v \geq b$. The construction is described as follows:

$$v\text{-}\hat{\mathrm{t}}\mathrm{BRW}[H^l]((r_1, \ldots, r_v), M) = \hat{\mathrm{t}}\mathrm{BRW}[H^l]((r_1, \ldots, r_v, r_1^{d_l+1}, \ldots, r_v^{d_l+1}, r_1^{2\cdot(d_l+1)}, \ldots, r_v^{2\cdot(d_l+1)}, \\ r_1^{4\cdot(d_l+1)}, \ldots, r_v^{4\cdot(d_l+1)}, \ldots), M).$$

Since the total degree of this polynomial is bounded by $2 \cdot v \cdot (d_l + 1) \cdot \overline{n}^{1/v}$, applying Thm. 4, we conclude that v-$\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ achieves FIL ϵ-almost universality, where $\epsilon \leq \frac{2\cdot v\cdot(d_l+1)\cdot\overline{n}^{1/v}}{q}$.

The univariate BRW polynomial proposed by Bernstein [3] is a particular instantiation of v-$\hat{\mathrm{t}}\mathrm{BRW}[H^l]$, obtained by setting $B = 3$, $v = b = 1$, $H^l = 1$-NMH^+, and $d_l = 3$.

$\hat{\text{t}}\text{BRW}[H^l]((r_1,\ldots,r_v),(M_1,\ldots,M_n))$

$\overline{n} \leftarrow \left\lfloor \dfrac{n}{B+1} \right\rfloor$

// processing message by blocks of $B+1$

for $i = 0$ **to** $\overline{n}-1$ **do**

  $T_0 \leftarrow H^l_{r_1,\ldots,r_b}(M_{(B+1)\cdot i+1},\ldots,M_{(B+1)\cdot i+B})$

  // exponent of highest power of 2 dividing $i+1$

  $s \leftarrow \text{ExpHighest2Power}(i+1)$

  $T_{s+1} \leftarrow \left(\sum_{j=0}^{s} T_j\right) \cdot (r_{b+s+1} + M_{(B+1)\cdot(i+1)})$

// processing remaining blocks

$T_0 \leftarrow H^l_{r_1,\ldots,r_b}(M_{(B+1)\cdot\overline{n}+1},\ldots,M_n)$

// summing the result in a single accumulator

for $i = 1$ **to** $\lfloor \log_2(\overline{n}) \rfloor + 1$ **do**

  // check if i-th bit of $\overline{n}$ is 1

  if $\text{bit}(i,\overline{n}) = 1$ **then**

    $T_0 \leftarrow T_0 + T_i$

return T_0

Algorithm 3: Sequential evaluation algorithm of $\hat{\text{t}}\text{BRW}[H^l]$.

MHP. The MHP (Multivariate Horner Style Polynomial) hash is a multivariate generalization of the classical univariate polynomial hash. Unlike the other constructions above, it requires only a fixed number of keys $v - b > 0$, independent of the number of superblocks $n' + 1$. A distinctive feature of MHP is its evaluation algorithm, described in Alg. 4, which has similar characteristics to Horner's algorithm. This algorithm operates online, requires a fixed amount of memory independent of the input message, and needs n' field additions and multiplications. Furthermore, MHP achieves the smallest possible total degree for a polynomial in $v - b$ variables handling $n' + 1$ superblocks.

Various multivariate generalizations of Horner's algorithm exist, typically based on recursively decomposing a multivariate polynomial into several univariate polynomials, each evaluated via Horner's method. Notably, these decompositions can be performed in multiple ways. As far as we know, MHP is the first construction to combine all the above described properties, making it a compelling choice for a polynomial hash function.

The recursive definition of MHP is as follows:

- For $n \leq B$,

$$\text{MHP}[H^l]\left((r_j)_1^b, (M_j)_1^n\right) = H^l\left((r_j)_1^b, (M_j)_1^n\right)$$

- For $n > B$,

$$\text{MHP}[H^l]\left((r_j)_1^v, (M_j)_1^n\right) = r_v \cdot \text{MHP}[H^l]\left((r_j)_1^v, (M_j)_1^m\right) + \text{MHP}[H^l]\left((r_j)_1^{v-1}, (M_j)_{m+1}^n\right)$$

  where $m = B \cdot \max_i \left\{ \binom{v-b+i-1}{v-b} : \binom{v-b+i-1}{v-b} < \left\lceil \frac{n}{B} \right\rceil \right\}$.

When $v - b = 1$, MHP reduces to the classical univariate polynomial hash, and its evaluation algorithm, as detailed in Alg. 4, coincides with Horner's method. As already noted, MHP has the property of having

$\mathrm{MHP}[H^l]((r_1, \ldots, r_v), (M_1, \ldots, M_n))$

1 : $i \leftarrow 0$
  // Computing v variables polynomial
2 : $T_v \leftarrow H^l_{r_1,\ldots,r_b}(M_{i+1}, \ldots, M_{\min(i+B,n)})$,
3 : $i \leftarrow \min(i+B, n)$, $d_v \leftarrow 0$
4 : **while** $(i < n)$ **do**
5 : $T_v \leftarrow T_v \cdot r_v$, $d_v \leftarrow d_v + 1$
  // Computing temporary $v-1$ variables polynomial
6 : $T_{v-1} \leftarrow H^l_{r_1,\ldots,r_b}(M_{i+1}, \ldots, M_{\min(i+B,n)})$,
7 : $i \leftarrow \min(i+B, n)$, $d_{v-1} \leftarrow 0$
8 : **while** $(i < n)\&\&(d_{v-1} < d_v)$ **do**
9 : $T_{v-1} \leftarrow T_{v-1} \cdot r_{v-1}$, $d_{v-1} \leftarrow d_{v-1} + 1$
    $\vdots$ // Similar nested loops until temporary $b+1$ variables polynomial
    // Computing temporary $b+1$ variables polynomial
10 : $T_{b+1} \leftarrow H^l_{r_1,\ldots,r_b}(M_{i+1}, \ldots, M_{\min(i+B,n)})$,
11 : $i \leftarrow \min(i+B, n)$, $d_{b+1} \leftarrow 0$
12 : **while** $(i < n)\&\&(d_{b+1} < d_{b+2})$ **do**
13 : $T_{b+1} \leftarrow T_{b+1} \cdot r_{b+1}$, $d_{b+1} \leftarrow d_{b+1} + 1$
    // Accumulating temporary $b+1$ variables polynomial
14 : $T_{b+1} \leftarrow T_{b+1} + H^l_{r_1,\ldots,r_b}(M_{i+1}, \ldots, M_{\min(i+B,n)})$,
15 : $i \leftarrow \min(i+B, n)$
    // Accumulating temporary $b+2$ variables polynomial
16 : $T_{b+2} \leftarrow T_{b+2} + T_{b+1}$
    $\vdots$ // Similar accumulations until temporary v variables polynomial
  // Accumulating temporary v variables polynomial
17 : $T_v \leftarrow T_v + T_{v-1}$
18 : **return** T_v

Algorithm 4: Sequential evaluation algorithm of $\mathrm{MHP}[H^l]$.

the smallest possible degree, i.e. the smallest value d_h that satisfies $\lceil \frac{n}{B} \rceil \leq \binom{v-b+d_h}{v-b}$ (cf. Sec. 3).[1] This follows from its construction, which increases d_h only when the number of superblocks $\lceil \frac{n}{B} \rceil$ exceeds the number of monomials of degree at most d_h. This ensures that superblocks can still be injectively mapped to polynomials of degree bounded by d_h, as required by Thm. 4. The constraint $v > b$ can be lifted by applying the key-reduction technique from Sec. 3. The injectivity of MHP is proven in App. B.2.

Like tMMH, MHP has a linearity property that allows it to leverage the structure of H^l or its own recursive structure to enable parallel evaluation. Furthermore, Δ-universality for VIL messages can also be directly achieved by imposing the same structure on the messages as described for tMMH.

[1] $\binom{v-b+d_h}{v-b}$ is the number of monomials in $v-b$ variables of degree at most d_h.

5 Benchmarking of New Designs

We will now discuss the performance of our new designs, starting by briefly covering important implementation aspects and our choice of reference polynomials used to evaluate performance. We then discuss our benchmarking methodology and setup, followed by a discussion of the results, including an analysis of the impact of various parameters on the performance of our implementations. Instructions to reproduce our results are given in App. H.

5.1 Implementation

To implement and benchmark our designs we use the polynomial hashing framework developed by [10]. This allows for automatic generation of the field arithmetic and dynamically linking it against different polynomial implementations, followed by testing and benchmarking. However, the framework [10] is limited to prime fields and only has support for non-nested polynomials. We thus add support for binary fields, message-dependent key lengths, and nesting polynomials in a modular fashion. To handle message-dependent key lengths, we implement a key generator, utilizing a ChaCha20-based DRBG, using Libsodium's ChaCha20 implementation. As shown in App. E this is a bottleneck for our schemes whose key size depends on the message size. Our updated version of the framework of [10] is available at `https://github.com/jangilcher/polynomial_hashing_framework` and is also archived on Zenodo [11].

With these extensions we implement the polynomials from Sec. 4.1 to generate both stand-alone implementations and for use as building blocks for the designs of Sec. 4.2. We parametrize our polynomial implementations to enable testing and benchmarking of different fields using different limb realignment and carry propagation strategies, similarly to the approach used for the classical polynomial in [10]. However, especially for the higher-level polynomials, the adaptability of our implementations likely does not yield peak performance. In particular, our higher-level polynomials do not currently use loop unrolling, due to the complexity of combining this with the current generic composition approach. We do use this and other optimizations for lower-level polynomials. Note that none of our prime field implementations make use of vectorized instruction set extensions as these are currently unsupported by the framework of [10]. We emphasize that all implementations generated by the framework are full hash functions over byte strings and so the performance and security discussions below include all the costs included in lifting our constructions from their theoretical description in Sec. 4 to working over byte strings. This also includes always using fresh key material and thus not amortizing internal key-generation costs over different invocations.

5.2 Reference Implementations

We compare our designs with the GMAC implementations from Haberdashery [19] and OpenSSL, Poly1305 implementations from Libsodium and OpenSSL, Poly1163 and Poly2663 from [10], as well as Hash2L [9] and d-2LHash [5]. In addition, we also benchmarked the classical polynomial over $\mathsf{GF}(2^{226}-5)$, $\mathsf{GF}(2^{128})$, and $\mathsf{GF}(2^{256})$. We had to write a wrapper around the implementations of Hash2L [9] to make it fit the methodology used in the benchmarking framework. This wrapper is not fully transparent, and is likely to impact the measurements; see App. D for further details.

As Hash2L is a hand-optimized implementation, we also benchmark Hash2L as generated by the framework, i.e. using the framework's field arithmetic and modular two-level implementation. This allows us to compare designs at the same level of optimization. To disambiguate the two implementations, we call this framework-generated version of Hash2L "Poly-BRW", reflecting its two-level design. As [5] does not provide an implementation of d-2LHash over $\mathsf{GF}(2^{116}-3)$ and $\mathsf{GF}(2^{226}-5)$, we instead use a version of d-2LHash as generated by the framework for a like-for-like comparison. Note that [5] actually improves on these results due to hand-optimized field arithmetic. In App. C we provide a comparison between the implementations from [5] and the framework. Similar gains should be achievable for our generated implementations.

5.3 Benchmarking Methodology

Our approach is similar to the one chosen by [10]. We evaluate each polynomial hash function for messages of up to 16 kB in length. For each message length we collect 25 samples, each consisting of 1000 measurements. All measurements are taken with Turboboost and dynamic frequency scaling disabled using a system with an Intel Xeon Gold 6258R running at 2.7 GHz. For the Variable-Key-Length (VKL) constructions we include the key-generation time in the benchmarks. For constructions that use a fixed number of keys, like Hash2L, d-2LHash, and constructions using MHP at the higher level, we do not include key-generation time.

We instantiate our constructions over both prime and binary fields, with 100–128-bit and 200–256-bit security levels in each case, using the following four fields: $\mathsf{GF}(2^{116}-3)$, $\mathsf{GF}(2^{226}-5)$, $\mathsf{GF}(2^{128})$, and $\mathsf{GF}(2^{256})$. The choice of our prime fields is motivated by the observation in [10] that an optimal security-performance trade-off is achieved when the limbs in the underlying field element representation are as saturated as possible while still allowing delayed carry propagation. In particular [10] identified $\mathsf{GF}(2^{116}-3)$ as promising candidate at the 128-bit level. For the high-security target we choose $\mathsf{GF}(2^{226}-5)$ instead of $\mathsf{GF}(2^{266}-3)$, due to its increased support for delayed carry propagation while still allowing to achieve security levels high enough for any authentication needs. When considering all possible combinations of inner and outer polynomials from Sec. 4 including the range of possible parameters for each of the polynomials, we are faced with a very large number of configurations to run. Thus, we run benchmarks with reduced sampling rates only measuring for message lengths that are multiples of 100 bytes. Tabs. 4 to 6 in App. F list the security levels of all schemes considered during benchmarking.

To better understand the design trade-offs across the many choices we perform a factor analysis on the data obtained from our benchmarks, to estimate the contribution and importance of individual parameters. The factor analysis and details on its methodology can be found in App. E. The factor analysis confirms our intuitions for the performance of the two-level designs. The choice of lower-level polynomial almost always accounts for more than half of the total variation in performance, while the higher-level only accounts for at most 5% of the variation. The remaining variation is mostly dominated by superblock size and the delay parameter.

5.4 Implementation Complexity and Memory Footprint

We now briefly discuss implementation complexity and memory usage. This is particularly relevant for our two-level designs, since they use more explicit key material and precomputation than the evaluation of the classical polynomial.

Regarding implementation complexity, it is worth noting that efficient implementations of standard classical polynomials are quite complex themselves [10]. High-performance implementations commonly use two-level evaluation, precomputed powers of the hash key, delayed reductions, and other optimizations to achieve high performance. Thus, while our multivariate polynomials are more complex at the specification level, this gap in complexity is much less at the implementation level.

The main memory cost in our implementations is the accumulators and key/precomputation storage. Similarly to when implementing the classical polynomial using a two-level approach, in our designs, keys are usually precomputed, requiring memory scaling with the superblock size for the lower-level polynomial. Additionally, for the accumulators and keys of the higher-level polynomials, our implementations require memory that is at most linear in the key size. In our constructions, VKL designs store accumulators and keys that grow logarithmically with the message length, whereas in MHP, they have a fixed size. Alternatively, for the reduced-key variants of our lower- and higher-level polynomial designs, if the memory cost is too prohibitive and at the expense of efficiency, one can also opt to recompute the key powers instead of storing them, thereby requiring memory storage smaller than the superblock size. For example,

(a) Binary Fields.

Scheme	Heap	Stack	Total
$\hat{\mathsf{t}}$BRW-NMH$^+$ 20SB 128	432	1472	1904
tMMH-NMH$^+$ 24SB 128	496	1792	2288
1-MHP-1-NMH$^+$ 36SB 128	-	1312	1312
5-MHP-1-NMH$^+$ 36SB 128	-	1744	1744
Classical Polynomial 128	-	736	736
Poly-BRW 128	-	1168	1168
$\hat{\mathsf{t}}$BRW-NMH$^+$ 24SB 256	960	1712	2672
tMMH-NMH$^+$ 24SB 256	960	1856	2816
1-MHP-1-NMH$^+$ 24SB 256	-	2656	2656
5-MHP-1-NMH$^+$ 28SB 256	-	3536	3536
Classical Polynomial 256	-	1424	1424
Poly-BRW 256	-	2112	2112

(b) Prime Fields.

Scheme	Heap	Stack	Total
Poly1163	-	928	928
$\hat{\mathsf{t}}$BRW-NMH$^+$ 24SB 1163	434	1792	2226
tMMH-NMH$^+$ 24SB 1163	434	1728	2162
1-MHP-1-NMH$^+$ 36SB 1163	-	1952	1952
5-MHP-1-NMH$^+$ 32SB 1163	-	6192	6192
d-2LHash1163	-	1344	1344
$\hat{\mathsf{t}}$BRW-NMH$^+$ 28SB 2265	952	2848	3800
tMMH-NMH$^+$ 28SB 2265	952	2656	3608
1-MHP-1-NMH$^+$ 36SB 2265	-	4832	4832
5-MHP-1-NMH$^+$ 36SB 2265	-	8704	8704
Poly2663	-	11408	11408
Poly2265	-	6576	6576
d-2LHash2265	-	2512	2512

Table 3: Total memory estimates in Byte for 16 KB messages (excluding the cost to store the initial message input).

an implementation of 1-MHP-1-NMH$^+$ can require storing only two keys when key powers are recomputed, rather than memory proportional to the superblock size. Note that larger key sizes imply better security, and thus there is also a memory-efficiency/security trade-off. Tab. 3 provides indicative memory usage of our implementations for 16KB messages. We note however that our implementations are not optimized for memory-efficiency and sacrifice memory efficiency for performance and security gains. For our concrete constructions, the lower level requires at most 1.125 KB to store the keys (based on worst-case analysis requiring 36 256-bit keys). For our MHP variants, total key storage stays below 1.3 KB.

5.5 Performance Results

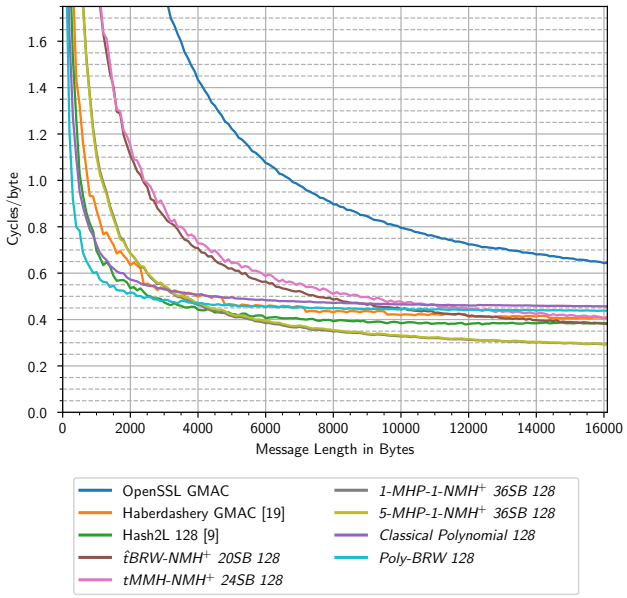
Tabs. 7 and 8 in App. G list the peak throughput achieved by the different combinations of higher and lower-level polynomials for our VKL constructions.

Notably across all fields, choices with NMH$^+$ at the lower level perform significantly better than all other versions. This aligns with expectations as NMH$^+$ uses half as many field multiplications as the other polynomials, at the cost of adding cheaper-to-implement field additions, and the bulk of computations occurs at the leaves of the tree constructions. While requiring the same number of multiplications as NMH$^+$, HKM is not able to compete with the other choices for the lower level, due to its lack of support for delayed carry propagation. This reinforces observations from [10] about the importance of delayed carry propagation and limb realignment.

At the higher level $\hat{\mathsf{t}}$BRW is faster than the remaining VKL options, though results are generally within 0.02 cycles/byte of each other when NMH$^+$ is used as lower-level polynomial. This suggests using tMMH over $\hat{\mathsf{t}}$BRW as tMMH's design is better suited to parallelization and vectorization.

With MHP at the higher level, peak performance is again achieved with NMH$^+$ at the lower level (cf. Tabs. 9 and 10). For prime fields, we obtain a speed-up of roughly 10% compared to the VKL constructions. For binary fields, the speed-up is 25% in the best case. This speed-up is gained by a reduction of the security level by 4–5 bits depending on the number of keys used for MHP and the message length. Following our factor analysis in App. E we note that once the more important parameters have been chosen properly, the effect of the number of keys on performance becomes noticeable, albeit small.

(a) Standard Security Level (∼128 bits).



(b) High Security Level (∼256 bits).

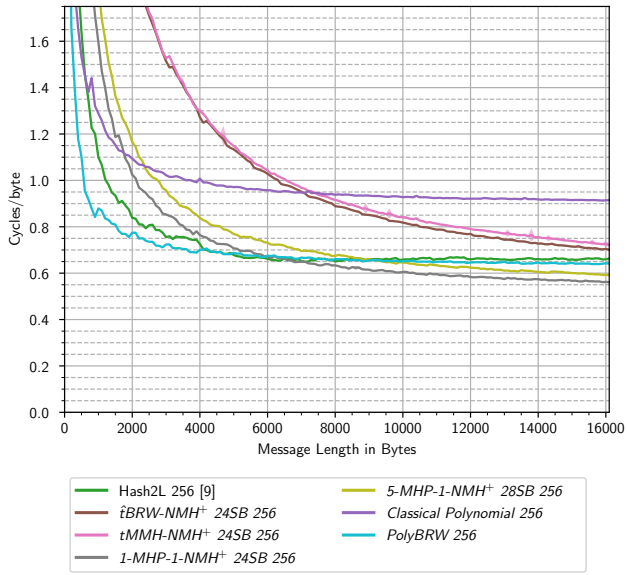


Figure 4: Performance of our polynomials over binary fields compared to SotA. New implementations generated by the framework in *italics*.

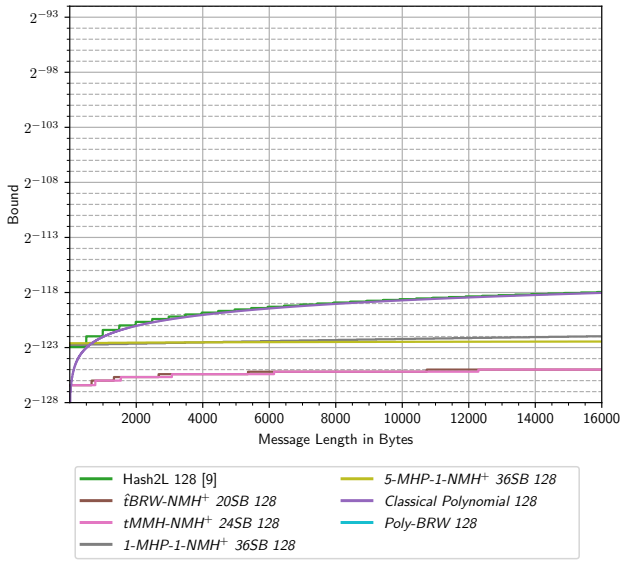
Binary Fields. Our results for binary fields are depicted in Fig. 4 (with corresponding security levels depicted in Fig. 5). At both security levels the choice of 1-NMH$^+$ over BRW at the lower level in 1-MHP -1-NMH$^+$ leads to a significant increase in performance of about 0.1 cycles/byte. This amounts to a 25% improvement compared to 128-bit Hash2L's runtime and a 15% improvement compared to the 256-bit case. At 122 and 250 bits for 16 kB, respectively, this performance improvement is further backed by a slight increase in security for messages over 500 bytes, rising to a maximum of 4 bits over the range of message lengths considered in our benchmarks when compared to Hash2L and the classical polynomial. When going up to 5 keys for MHP-NMH$^+$, this comes at a performance cost of less than 0.01 cycles/byte at the 128-bit security level and about 0.04 cycles/byte at the 256-bit security level with no significant increase in security.

When more security is needed, the VKL constructions provide a better alternative. At 126–125 and 254–253 bits, they achieve 3–5 bits more security than MHP-NMH$^+$ and up to 7 bits more compared to Hash2L. But this comes at the cost of lower performance than Hash2L in all cases except ĤBRW-NMH$^+$ over GF(2^{128}), which achieves the same performance as Hash2L for 16 kB messages.

Nonetheless, our benchmarks indicate that MHP-NMH$^+$ implementations over GF(2^{256}) as well as the VKL construction over both binary fields have not yet reached peak performance within the range of message lengths we considered. This is shown by the negative curve slopes in Fig. 4. This indicates that the VKL constructions in particular might be a good choice for very long messages, especially as they have better security bounds in this setting.

We further see that the optimized implementation of Hash2L from [9] and Poly-BRW are close in performance, with the optimized 128-bit version being 0.05 cycles/byte faster and the two 256-bit versions performing near identically. However, at about 0.37 cycles/byte this small difference for the GF(2^{128}) version amounts to about 14% more cycles for Poly-BRW. We observe that Poly-BRW128 performs similarly to our classical polynomial, which seems to confirm the 10–15% speed-up of Hash2L over GHASH claimed in [9] at equivalent security levels. However, we also note that the highly optimized Haberdashery GMAC

(a) Standard Security Level (∼128 bits).



(b) High Security Level (∼256 bits).

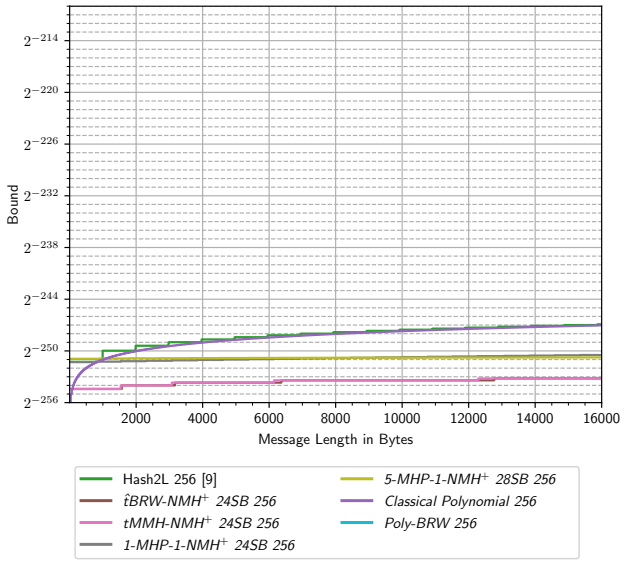


Figure 5: Security levels of our polynomials over binary fields compared to SotA; lower is better. New implementations generated by the framework in *italics*.

implementation [19] is within 0.03 cycles of Hash2L reducing the speed-up to about 8%.[2] Looking at GF(2^{256}), we see both Hash2L and Poly-BRW performing at 0.65 cycles/byte. In contrast to our classical polynomial over GF(2^{256}) at 0.91 cycles/byte this constitutes a performance increase of 28% at the same security level. This provides a reference point for 256-bit Hash2L that is missing from [9].

Prime Fields. Fig. 6 depicts the performance of t̂BRW-NMH$^+$, tMMH-NMH$^+$ and MHP-NMH$^+$ alongside the reference polynomials from Sec. 5.2 for prime fields. In addition, Fig. 7 depicts how the security levels of these schemes vary across the range of message lengths we considered in our benchmarks.

Starting with the lower security level in Fig. 6a, we observe that d-2LHash1163 achieves similar performance to Poly1305 from HACL* at about 0.7 cycles/byte, albeit at 3 bit higher security level of 106–96 bits as message size increases.

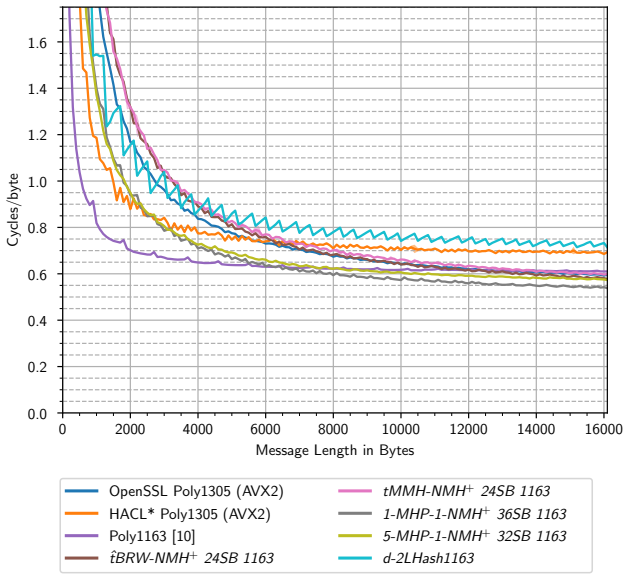
At about 0.6 cycles/byte t̂BRW-NMH$^+$ and tMMH-NMH$^+$ perform similarly to OpenSSL's Poly1305, but offer an up-to 16-bit increase in security over Poly1305's 103–93 bits. With a 10% increase in performance MHP-NMH$^+$ comes out as the fastest design at 0.54 cycles/byte. However, at 107–106 bits of security this increase in performance comes at the price of losing up to 4 bits of security for messages of up to 1600 bytes compared to t̂BRW-NMH$^+$'s and tMMH-NMH$^+$'s 111–109 bits.

Lastly, we highlight that the much simpler Poly1163 from [10] does extremely well in terms of performance. While its 112 bits offers 5 bits more security than MHP-NMH$^+$ for very short messages, it is generally up to 5 bits behind MHP-NMH$^+$ once messages exceed 500 bytes, achieving only 102 bits at 16 kB. Still, one might argue that the relative simplicity of its design and implementation is worth that loss in security for real-world deployments.

At the higher security level depicted in Fig. 6b we can see a massive performance increase over Poly2663 [10] for all the remaining hash functions. This is as expected because of the larger field size. At 256—245 bits, this also comes with a security increase of 32 bits compared to Poly2265.

[2]Note that GMAC contains an additional AES call compared to GHASH. 8% thus constitutes a strict upper bound on the speed-up.

(a) Standard Security Level (∼128 bits).



(b) High Security Level (∼256 bits).

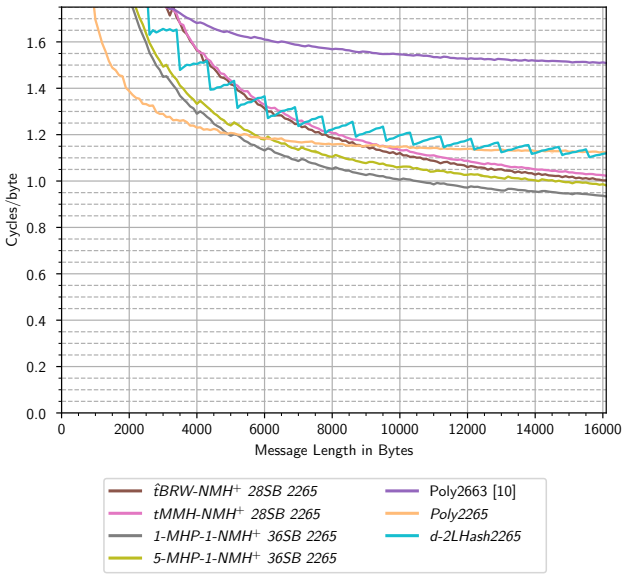
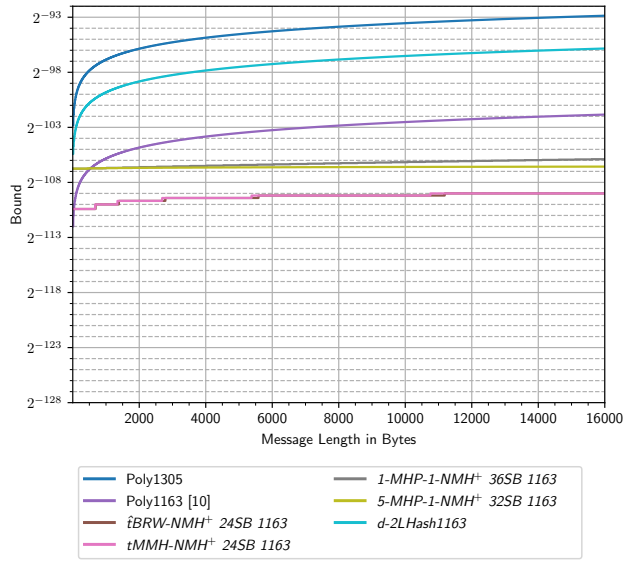


Figure 6: Performance of our polynomials over prime fields compared to SotA. New implementations generated by the framework in *italics*.

(a) Standard Security Level (∼128 bits).



(b) High Security Level (∼256 bits).

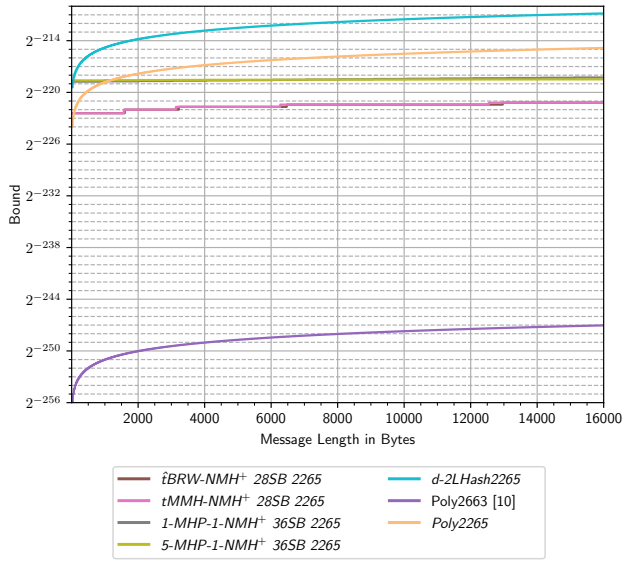


Figure 7: Security levels of our polynomials over prime fields compared to SotA; lower is better. New implementations generated by the framework in *italics*. Note that the security level for Poly1305 considers it as standardized, i.e. including the loss in security incurred from key clamping.

d-2LHash2265 performs similarly for large messages compared to Poly2265, though there is a very significant performance gap in favor of Poly2265 for messages below 8 kB. Furthermore, d-2LHash2265 achieves this performance at a 4 bit lower security level of 219–211 bits than Poly2265.

For messages longer than about 7 kB the use of 1-NMH$^+$ starts compensating for the increased key precomputation required by the increased superblock size of MHP-NMH$^+$ compared to d-2LHash, leading to a performance increase of about 0.12 cycles/byte for the 1 key variant. Notably this performance increase is also accompanied by a 3–4 bit increase in security up-to 218 bits. A further 3–4 bits of security can be gained by using one of our VKL constructions. This however comes with performance loss of about 0.06 cycles/byte compared to 1-MHP-1-NMH$^+$, placing it evenly between the former and d-2LHash2265 in terms of performance. However, due to the relatively large key generation time, the VKL schemes only reach their top speed very slowly.

Summary. Our new VKL multivariate two-level designs compete with the state-of-the-art. But their increased security has a clear performance cost. In many cases protocol designers might prefer the increased performance of MHP-NMH$^+$, which generally loses less than 10 bits of security compared to the VKL constructions over the message size range we considered. We show that using BRW at the lower level limits performance, and we are able to improve upon prior work [9, 5], both in terms of performance and security, using the much simpler 1-NMH$^+$ at the lower level instead.

While our prime field implementations still leave room for optimization, e.g. via vectorized instruction set extensions, we note that arithmetic in binary fields with hardware support for carryless multiplication still massively outperforms prime fields at consistently higher security levels. As these instructions are widely available on x86 platforms, and also increasingly on Arm processors, binary fields seem preferable. Even when prime field implementations can be sped up significantly by using newer hardware instructions for two independent carry chains as in [9, 5], the benefits gained from the choice of prime fields over binary fields seem small, since (as far as we know) carryless multiplication is also always available on these platforms.

5.6 Performance Impact on AEAD Schemes

We now briefly evaluate the impact of our most promising design, MHP-NMH$^+$, on the performance of AEAD and Accordion modes. An extended evaluation can be found in [17, Chapter 5].

AES-GCM. We do not expect any gain here because in highly optimized AES-GCM implementations the bottleneck is exclusively the AES encryption. This is evident by comparing the performance of AES-GCM implementations from [19] with the top performance of AES-CTR implementations in SUPERCOP [4] and the theoretical bound based on AES-NI instruction throughput. For AES 128 and AES 256 respectively, AES-GCM achieves 0.66[3] and 0.88[4] cycles/byte, while AES-CTR achieves 0.63 cycles/byte and 0.87 cycles/byte[5]; the theoretical bounds are 0.625 and 0.875 cycles/byte. Thus, performance improvements in the UHF would have only a tiny impact for AES-GCM.

Rijndael256. NIST plans to standardize a 256-bit-block variant of Rijndael [30], as well as new block cipher modes, potentially including a 256-bit authenticator. An optimized Rijndael256 implementation exploiting AES-NI and using additional blendshuffles is given in [14]. The necessary instructions to implement this

[3] https://github.com/facebookincubator/haberdashery/blob/main/benchmark_data/md/haberdashery_aes128gcm_skylakex.md

[4] https://github.com/facebookincubator/haberdashery/blob/main/benchmark_data/md/haberdashery_aes256gcm_skylakex.md

[5] https://bench.cr.yp.to/results-stream/amd64-samba.html

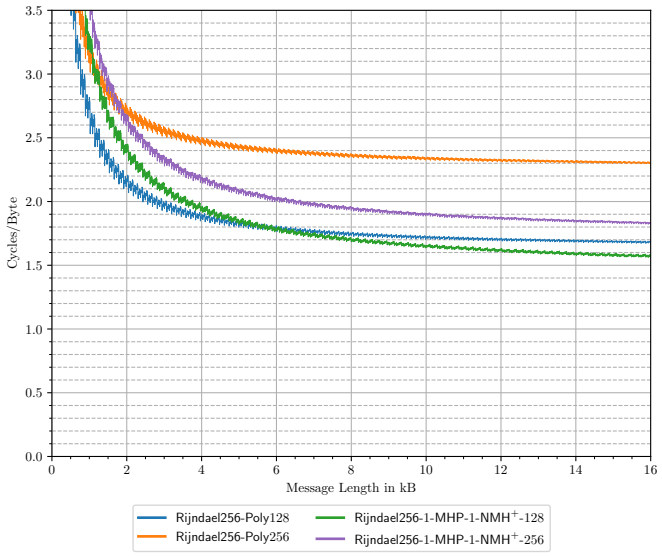


Figure 8: Performance of Rijndael256-based AEAD Schemes.

blendshuffle compete for execution ports with both the `aesenc` and `pclmulqdq` instructions, thus breaking the previously carefully designed resource partition between the encryption and hashing instructions. This competition for resources makes reduced UHF costs beneficial for performance.

We use the field arithmetic generated by our framework to implement four different AEAD schemes, Rijndael256-Poly128 and Rijndael256-Poly256, which utilize the classical polynomial, and Rijndael256-1-MHP-1-NMH$^+$-128 and Rijndael256-1-MHP-1-NMH$^+$-256, all of which are based on the design of AES-GCM. Our Rijndael implementation follows the approach of [14]. The resulting AEAD implementations utilize an online design, where encryption and UHF computation are interleaved.

We benchmark our AEAD implementations with a similar methodology as used for the UHFs in Sec. 5.3, i.e. with plaintexts of up to 16 kB in length and without AD. The results are depicted in Fig. 8. We observe that Rijndael256-1-MHP-1-NMH$^+$-128 outperforms Rijndael256-Poly128 by 5.4% (1.58 and 1.67 cycles/byte). As the cost of field multiplication grows quadratically with the field size, for the larger 256-bit field, field multiplication is substantially more expensive. Since 1-MHP-1-NMH$^+$-256 performs fewer field multiplications than the corresponding classical polynomial hash, we observe higher efficiency gains at this field size, with a 20.9% performance increase for Rijndael256-1-MHP-1-NMH$^+$-256 over Rijndael256-Poly256 (1.82 and 2.30 cycles/byte).

Accordion mode. NIST also plans to standardize an Accordion mode [31], for AES and Rijndael256, from which AEAD with stronger security properties can easily be realised. Typical constructions, as the currently preferred candidate HCTR2, employ two instances of a universal hash and an encryption component consisting of AES or Rijndael256 in CTR mode. Here one hash instance can be evaluated in parallel with the encryption component while the other is inherently sequential and must be evaluated before or after. Thus irrespective of whether AES or Rijndael256 is used, it is always beneficial to switch to a faster hash due to the hash instance which is evaluated sequentially (1-MHP-1-NMH$^+$-256 is 37.8% faster than a classical 256-bit polynomial). In addition, when Rijndael256 is used we would expect an additional performance boost, similar to that of Rijndael256-1-MHP-1-NMH$^+$-256 (20.9%), when evaluating the hash and encryption component in parallel, resulting in an estimated 25.6% overall increase in performance.

6 Conclusion

In this paper, we explored the design space of multivariate polynomial UHFs via a two-level approach inspired by [9, 5]. Extending the code-generation and benchmarking framework of [10], and complementing it with a factor analysis, we identified attractive lower- and higher-level polynomial combinations, yielding new UHF designs whose performance matches or surpasses the state of the art in leading cryptographic libraries and the implementations of [9]. We highlight two complementary designs: MHP-NMH$^+$, which delivers very high performance and improved security over the state of the art for all but very short messages; and tMMH-NMH$^+$, which, for any given field, attains the highest security – with minimal loss as message length increases – at state-of-the-art speed. Both meet our three goals from Sec. 1: short keys, reduced multiplication complexity, and compatibility with delayed-operation optimizations and parallelizability. Despite relying on auto-generated code, our designs remain competitive, whereas prior benchmarks used carefully optimized assembly code. Given its higher security due to a larger field, d-2LHash1271 [5] is also an attractive option for a 128-bit prime-field hash. A general finding is that binary fields typically outperform prime fields at similar security levels, largely due to carryless-multiply instructions. Considering NIST's current standardization efforts for a 256-bit block cipher and new AEAD/Accordion modes, use of MHP-NMH$^+$ could lead to performance increases between 5 and 26%, depending on field size and AEAD construction.

Our results point to designs worth further optimization; based on prior hand-optimized work [18, 6, 5, 19], we expect significant additional speedups. Further gains should accrue from vectorized instruction-set extensions that compute several superblocks in parallel. Automating code generation to exploit these features across a wide range of UHF designs remains an important open problem.

Acknowledgments

Part of this work was done while Jérôme Govinden was a Ph.D. candidate at TU Darmstadt, supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Bernstein, D. J. Floating-point arithmetic and message authentication. *Unpublished manuscript* (2004). `https://cr.yp.to/papers.html#hash127`. 9

[2] Bernstein, D. J. The poly1305-AES message-authentication code. In *FSE 2005* (Feb. 2005), H. Gilbert and H. Handschuh, Eds., vol. 3557 of *LNCS*, Springer, Berlin, Heidelberg, pp. 32–49. 9

[3] Bernstein, D. J. Polynomial evaluation and message authentication. *Unpublished manuscript* (2007). `http://cr.yp.to/papers.html#pema`. 2, 9, 18

[4] Bernstein, D. J., and Lange, T. eBACS: ECRYPT benchmarking of cryptographic systems. `https://bench.cr.yp.to`, accessed on 29 January 2024, 2008. 27

[5] Bhattacharyya, S., Nath, K., and Sarkar, P. Polynomial hashing over prime order fields. *Advances in Mathematics of Communications* (2024). 3, 4, 9, 10, 11, 21, 27, 29, 34, 35

[6] Bhattacharyya, S., and Sarkar, P. Improved SIMD implementation of Poly1305. *IET Inf. Secur. 14*, 5 (2020), 521–530. 29

[7] Boesgaard, M., Christensen, T., and Zenner, E. Badger - a fast and provably secure MAC. In *ACNS 2005* (June 2005), J. Ioannidis, A. Keromytis, and M. Yung, Eds., vol. 3531 of *LNCS*, Springer, Berlin, Heidelberg, pp. 176–191. 7

[8] Carter, J. L., and Wegman, M. N. Universal classes of hash functions. *Journal of computer and system sciences 18*, 2 (1979), 143–154. 2

[9] Chakraborty, D., Ghosh, S., and Sarkar, P. A fast single-key two-level universal hash function. *IACR Trans. Symm. Cryptol. 2017*, 1 (2017), 106–128. 3, 4, 9, 10, 11, 21, 24, 25, 27, 29, 35, 43

[10] Degabriele, J. P., Gilcher, J., Govinden, J., and Paterson, K. G. SoK: Efficient design and implementation of polynomial hash functions over prime fields. In *2024 IEEE Symposium on Security and Privacy* (May 2024), IEEE Computer Society Press, pp. 3128–3146. 2, 3, 4, 6, 8, 9, 21, 22, 23, 25, 26, 29, 34, 35

[11] Degabriele, J. P., Gilcher, J., Govinden, J., and Paterson, K. G. Polynomial hashing framework v2.0.3, July 2026. `https://doi.org/10.5281/zenodo.20677844`. 21

[12] DeMillo, R. A., and Lipton, R. J. A probabilistic remark on algebraic program testing. *Inf. Process. Lett. 7*, 4 (1978), 193–195. 9

[13] den Boer, B. A simple and key-economical unconditional authentication scheme. *Journal of Computer Security 2* (1993), 65–71. 2, 9

[14] Drucker, N., and Gueron, S. Software optimization of Rijndael for modern x86-64 platforms. In *ITNG 2022 19th International Conference on Information Technology-New Generations* (Cham, 2022), S. Latifi, Ed., Springer International Publishing, pp. 147–153. 27, 28

[15] Duval, S., and Leurent, G. Lightweight MACs from universal hash functions. In *Smart Card Research and Advanced Applications* (Cham, 2020), S. Belaïd and T. Güneysu, Eds., Springer International Publishing, pp. 195–215. 9

[16] Etzel, M., Patel, S., and Ramzan, Z. Square hash: Fast message authentication via optimized universal hash functions. In *Advances in Cryptology — CRYPTO' 99* (Berlin, Heidelberg, 1999), M. Wiener, Ed., Springer Berlin Heidelberg, pp. 234–251. 3, 9, 11, 12

[17] Gilcher, J. Hashing it out: Advancing universal hash functions with applications to authenticated encryption, 2026. *Doctoral Thesis.* `https://doi.org/10.3929/ethz-c-000800539`. 27

[18] Goll, M., and Gueron, S. Vectorization of Poly1305 message authentication code. In *Proceedings of the 2015 12th International Conference on Information Technology - New Generations* (USA, 2015), ITNG '15, IEEE Computer Society, p. 145–150. 3, 10, 29

[19] Gueron, S. I want to encrypt 2^{64} bytes with AES-GCM using a single key. Real World Crypto Symposium 2024, 2024. 21, 24, 25, 27, 29

[20] Gueron, S., Langley, A., and Lindell, Y. AES-GCM-SIV: Specification and analysis. Cryptology ePrint Archive, Report 2017/168, 2017. 9

[21] Halevi, S., and Krawczyk, H. MMH: Software message authentication in the Gbit/second rates. In *FSE'97* (Jan. 1997), E. Biham, Ed., vol. 1267 of *LNCS*, Springer, Berlin, Heidelberg, pp. 172–189. 2, 3, 9, 11, 12, 13

[22] Hoang, V. T., Krovetz, T., and Rogaway, P. Robust authenticated-encryption AEZ and the problem that it solves. In *EUROCRYPT 2015, Part I* (Apr. 2015), E. Oswald and M. Fischlin, Eds., vol. 9056 of *LNCS*, Springer, Berlin, Heidelberg, pp. 15–44. 2

[23] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley.* 04 1991. 36

[24] Jankowski, K., and Laurent, P. Packed AES-GCM algorithm suitable for AES/PCLMULQDQ instructions. *IEEE Transactions on Computers 60*, 1 (2010), 135–138. 3, 10

[25] Johansson, T., Kabatianskii, G., and Smeets, B. On the relation between A-codes and codes correcting independent errors. In *EUROCRYPT'93* (May 1994), T. Helleseth, Ed., vol. 765 of *LNCS*, Springer, Berlin, Heidelberg, pp. 1–11. 2, 9

[26] Kronecker, L. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik* (1882). 9

[27] Krovetz, T. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, Mar. 2006. 2

[28] Lemire, D., and Kaser, O. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering 6*, 3 (Sept. 2016), 171–185. 9

[29] McGrew, D. A., and Viega, J. The security and performance of the Galois/counter mode (GCM) of operation. In *INDOCRYPT 2004* (Dec. 2004), A. Canteaut and K. Viswanathan, Eds., vol. 3348 of *LNCS*, Springer, Berlin, Heidelberg, pp. 343–355. 9

[30] NIST. PRE-DRAFT Call for Comments: NIST Proposes to Standardize a Wider Variant of AES. NIST Special Publication 800-197, National Institute of Standards and Technology (NIST), 2024. `https://csrc.nist.gov/pubs/sp/800/197/iprd`. 27

[31] NIST. PRE-DRAFT Call for Comments: NIST Launches Development of Cryptographic Accordions. NIST Special Publication 800-197A, National Institute of Standards and Technology (NIST), 2025. `https://csrc.nist.gov/pubs/sp/800/197/a/iprd`. 2, 28

[32] Sarkar, P. A trade-off between collision probability and key size in universal hashing using polynomials. *Designs, Codes and Cryptography 58*, 3 (2011), 271–278. 9

[33] Schwartz, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM) 27*, 4 (1980), 701–717. 9

[34] Stinson, D. R. Universal hashing and authentication codes. In *Annual International Cryptology Conference* (1991), Springer, pp. 74–85. Preliminary version of [35]. 6

[35] Stinson, D. R. Universal hashing and authentication codes. *Designs, Codes and Cryptography 4*, 3 (1994), 369–380. 31

[36] Taylor, R. An integrity check value algorithm for stream ciphers. In *CRYPTO'93* (Aug. 1994), D. R. Stinson, Ed., vol. 773 of *LNCS*, Springer, Berlin, Heidelberg, pp. 40–48. 2, 9

[37] Wegman, M. N., and Carter, L. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences 22* (1981), 265–279. 1, 7

[38] Zippel, R. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation* (1979), Springer, pp. 216–226. 9

A Building VIL Universal Hash Functions Over $\mathbb{F}_q$

We outline techniques that use length encoding for extending fixed-input-length hash functions to handle variable input lengths. These are divided into two approaches. The first applies to any universal hash function producing a finite field element as its tag. The second requires less key material, but applies only to multivariate polynomial hashes that meet the conditions of Thm. 4.

A.1 Universal Hash Functions with Finite Field as Tag Space

Let $H : \mathcal{R} \times [\mathcal{M}]^n \to \mathbb{F}_q$ be ϵ-almost universal for any fixed n. We propose two approaches that differ in terms of efficiency and security trade-offs. Consider the hash functions $H_1 : \left(\mathcal{R} \times [\mathbb{F}_q]^2\right) \times [\mathcal{M}]^{\leq q-1} \to \mathbb{F}_q$ and $H_2 : (\mathcal{R} \times \mathbb{F}_q) \times [\mathcal{M}]^{\leq q-1} \to \mathbb{F}_q$ defined as follows:

$$H_1((r_1, r_2, r_3), M) = (H(r_1, M) + r_2) \cdot ([|M|]_q + r_3)$$
$$H_2((r_1, r_2), M) = (H(r_1, M) \cdot r_2 + [|M|]_q) \cdot r_2.$$

In addition to the complexity of evaluating H, the function H_1 requires two keys and one multiplication, whereas H_2 requires one key but two multiplications. Then H_1 and H_2 are respectively ϵ_1-almost and ϵ_2-almost Δ-universal where $\epsilon_1(M, M') = \epsilon(M, M') + \frac{1}{q}$ and $\epsilon_2(M, M') = \epsilon(M, M') + \frac{2}{q}$.

Proof. Let $z \in \mathbb{F}_q$ and $M, M' \in [\mathcal{M}]^{\leq q-1}$ be distinct messages such that

$$H_1((r_1, r_2, r_3), M) = H_1((r_1, r_2, r_3), M') + z.$$

This leads to the following equation:

$$C_3 \cdot r_3 + C_2 \cdot r_2 + C_1 = 0, \tag{1}$$

where $C_1 = H(r_1, M) \cdot [|M|]_q - H(r_1, M') \cdot [|M'|]_q - z$, $C_2 = [|M|]_q - [|M'|]_q$, and $C_3 = H(r_1, M) - H(r_1, M')$. For eq. (1) to hold two cases are possible. In the first case, the polynomial in r_2 and r_3 is the zero polynomial. In the second case, the polynomial is non-zero, and r_2, r_3 are roots of this polynomial. Given that r_2, r_3 are randomly sampled, the probability of the second case occurring is bounded by $\frac{1}{q}$. The probability of the first case occurring is bounded by the probability that $C_2 = 0$ and $C_3 = 0$, which is bounded by $\epsilon(M, M')$ since H is ϵ-almost universal for messages of equal length.
The proof for H_2 follows the same structure as for H_1. □

A.2 Polynomial Hash

For polynomial hash functions, we propose techniques that require less additional key material than the methods outlined in the previous section. Let $P : [\mathcal{M}]^* \to \mathbb{F}_q[x_1, \ldots, x_v]$ be such that for any $M, M' \in [\mathcal{M}]^*$, $P(M) - P(M')$ is a polynomial of total degree at most $d(M, M')$ and let H be the hash function associated with P. Consider the hash functions $H_3 : [\mathbb{F}_q]^{v+1} \times [\mathcal{M}]^{\leq q-1} \to \mathbb{F}_q$ and $H_4 : [\mathbb{F}_q]^v \times [\mathcal{M}]^{\leq q-1} \to \mathbb{F}_q$ defined as follows:

$$H_3((r_1, \ldots, r_{v+1}), M) = (H((r_1, \ldots, r_v), M) + r_{v+1}) \cdot ([|M|]_q + r_1),$$
$$H_4((r_1, \ldots, r_v), M) = (H((r_1, \ldots, r_v), M) \cdot r_1 + [|M|]_q) \cdot r_1.$$

The construction H_3 requires fewer multiplications than H_4 but needs an additional key, whereas H_4 retains the same number of keys as the original polynomial hash H. If P is injective over $[\mathcal{M}]^n$ for all n, these functions are almost Δ-universal: H_3 is ϵ_3-almost Δ-universal with $\epsilon_3(M, M') = \frac{d(M,M')+1}{q}$, while H_4 is ϵ_4-almost Δ-universal with $\epsilon_4(M, M') = \frac{d(M,M')+2}{q}$.

Proof. Let $z \in \mathbb{F}_q$ and $M, M' \in [\mathcal{M}]^{\leq q-1}$ be distinct messages such that

$$H_3((r_1, \ldots, r_{v+1}), M) = H_3((r_1, \ldots, r_{v+1}), M') + z.$$

Then $(r_1, \ldots, r_{v+1})$ is a root of the polynomial

$$(P(M) - P(M')) \cdot x_1 + P(M) \cdot [|M|]_q - P(M') \cdot [|M'|]_q + ([|M|]_q - [|M'|]_q) \cdot x_{v+1} - z. \tag{2}$$

When M and M' have different lengths, this polynomial is non-zero since the coefficient of x_{v+1} is non-zero. When M and M' are of equal length, the polynomial simplifies to

$$(P(M) - P(M')) \cdot (x_1 + [|M|]_q) - z,$$

which is also non-zero as $P(M) - P(M')$ is a non-zero polynomial by injectivity of P. By applying the Schwartz-Zippel lemma, the probability that $(r_1, \ldots, r_{v+1})$ is a root of the non-zero polynomial in eq. (2) is bounded by $\frac{d(M,M')+1}{q}$.
The proof for H_4 proceeds in a manner analogous to H_3. □

B Proofs from Sec. 4 (Proofs of Injectivity)

In this section, we prove the injectivity of the two polynomials associated to the universal hash functions $\hat{\mathsf{t}}\mathsf{BRW}$ and MHP.

B.1 Proof of Injectivity for $\hat{\mathsf{t}}\mathsf{BRW}$

In the following, we will assume that H^l is a monic polynomial hash of degree at least one. We now prove the injectivity of $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ and that it is a monic polynomial. The proof is done by induction on n.

For the base case considering any messages of length $n \leq B$, $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ is equivalent to H^l, which is injective and monic as it is itself a monic polynomial hash.

For the inductive step, we fix a value $n \geq B$ and assume that $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ is injective and monic for messages of length less or equal to n. Consider now two distinct messages $(M_j)_1^{n+1}$ and $\left(M'_j\right)_1^{n+1}$ of length $n+1 > B$. To prove injectivity, we assume that $\hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^v, (M_j)_1^{n+1}\right)$ is equal to $\hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^v, \left(M'_j\right)_1^{n+1}\right)$. Then, splitting the terms in r_v from the rest, and using that $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ is a monic polynomial for messages of length less or equal to n, we obtain the three following equations,

$$\begin{aligned} M_i &= M'_i, \\ \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, (M_j)_1^{i-1}\right) &= \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, \left(M'_j\right)_1^{i-1}\right), \\ \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, (M_j)_{i+1}^{n+1}\right) &= \hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, \left(M'_j\right)_{i+1}^{n+1}\right), \end{aligned}$$

where $v = \lfloor \log_2(\overline{n}) \rfloor + b + 1$, $i = (B+1) \cdot 2^{\lfloor \log_2(\overline{n}) \rfloor}$ and $\overline{n} = \left\lfloor \frac{n+1}{B+1} \right\rfloor$. The first equation comes from considering the coefficient of the monic term of $\hat{\mathrm{t}}\mathrm{BRW}[H^l]\left((r_j)_1^{v-1}, (M_j)_1^{i-1}\right)$. The second one comes from considering the factors of the key r_v. The third one is the resulting equation, once the two first have been established. As $1 < i < n+1$, from the induction assumption and the three equations, we obtain that $(M_j)_1^{n+1} = \left(M'_j\right)_1^{n+1}$. Thus, $\hat{\mathrm{t}}\mathrm{BRW}[H^l]$ is injective and monic for messages of length less or equal to $n+1$.

B.2 Proof of Injectivity for MHP

In the following, we will assume that H^l is a multivariate polynomial hash. We now prove the injectivity of MHP$[H^l]$. The proof is done by induction on n and v.

For any number of keys $v \geq b+1$ and any messages of length $n \leq B$, MHP$[H^l]$ is equivalent to H^l which is injective as it is itself a polynomial hash.

Let $v = b+1$. For the inductive step, we fix a value $n \geq B$ and assume that MHP$[H^l]$ is injective with v keys for messages of length less or equal to n. Consider now two distinct messages $(M_j)_1^{n+1}$ and $\left(M'_j\right)_1^{n+1}$ of length $n+1 > B$. To prove injectivity, we assume that MHP$[H^l]\left((r_j)_1^v, (M_j)_1^{n+1}\right)$ is equal to MHP$[H^l]\left((r_j)_1^v, \left(M'_j\right)_1^{n+1}\right)$. Then, splitting the terms in r_v from the rest, we obtain the two following equations,

$$\begin{aligned}
\text{MHP}[H^l]\left((r_j)_1^v, (M_j)_1^m\right) &= \text{MHP}[H^l]\left((r_j)_1^v, \left(M'_j\right)_1^m\right), \\
H^l\left((r_j)_1^b, (M_j)_{m+1}^{n+1}\right) &= H^l\left((r_j)_1^b, \left(M'_j\right)_{m+1}^{n+1}\right),
\end{aligned}$$

where $m = B \cdot \left(\left\lceil \frac{n+1}{B} \right\rceil - 1\right) < n+1$. From the induction assumption and the first equation, we have that $(M_j)_1^m = \left(M'_j\right)_1^m$. From the second equation and the fact that H^l is injective, we obtain $(M_j)_{m+1}^{n+1} = \left(M'_j\right)_{m+1}^{n+1}$. Thus, MHP$[H^l]$ is also injective with v keys for messages of length $n+1$.

Consider now fixed values $v \geq b+1$ and $n \geq B$. For the inductive step, we assume that MHP$[H^l]$ is injective for messages of length less or equal to n with $v+1$ keys, and for messages of length less or equal to $n+1$ with v keys. Consider now two distinct messages $(M_j)_1^{n+1}$ and $\left(M'_j\right)_1^{n+1}$ of length $n+1 > B$. To prove injectivity, we assume that MHP$[H^l]\left((r_j)_1^{v+1}, (M_j)_1^{n+1}\right)$ is equal to MHP$[H^l]\left((r_j)_1^{v+1}, \left(M'_j\right)_1^{n+1}\right)$. Then, splitting the terms in r_{v+1} from the rest, we obtain the two following equations,

$$\begin{aligned}
\text{MHP}[H^l]\left((r_j)_1^{v+1}, (M_j)_1^m\right) &= \text{MHP}[H^l]\left((r_j)_1^{v+1}, \left(M'_j\right)_1^m\right), \\
\text{MHP}[H^l]\left((r_j)_1^v, (M_j)_{m+1}^{n+1}\right) &= \text{MHP}[H^l]\left((r_j)_1^v, \left(M'_j\right)_{m+1}^{n+1}\right),
\end{aligned}$$

where $m < n+1$. From the induction assumption and the two equations, we obtain that $(M_j)_1^{n+1} = \left(M'_j\right)_1^{n+1}$. Thus, MHP$[H^l]$ is injective for messages of length less or equal to $n+1$ with $v+1$ keys.

C A Note on d-2LHash

Core contributions of [5] on top of their new design d-2LHash are highly optimized field arithmetic implementations for GF$(2^{130}-5)$ and GF$(2^{127}-1)$ utilizing new instructions on modern Intel-processors that allow for two independent carry chains. This allows instruction level parallelization for a previous sequential bottleneck in prime field arithmetic. Furthermore, the limb arrangement is manually tuned towards these fields and the usage of these new CPU instruction, unlike the field arithmetic generated by the framework from [10].

In Fig. 9 we depict the performance of d-2LHash1271 and d-2LHash1305 from [5]. Note that d-2LHash1163 and d-2LHash1271 allow for similar limb arrangements and should be roughly comparable, though not one-to-one. Overall, the field arithmetic optimizations in [5] lead to a very significant performance increase of 25–35%. In the case of d-2LHash1271 this results in similar performance as our best option MHP-NMH$^+$,

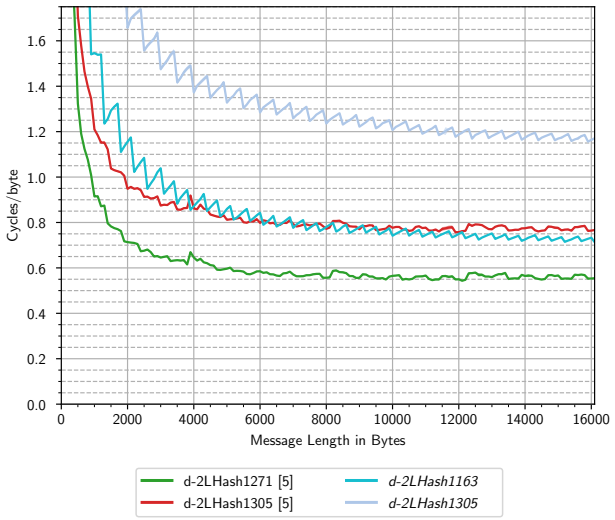


Figure 9: Performance of d-2LHash implementations from [5] compared to d-2LHash implementations from the framework.

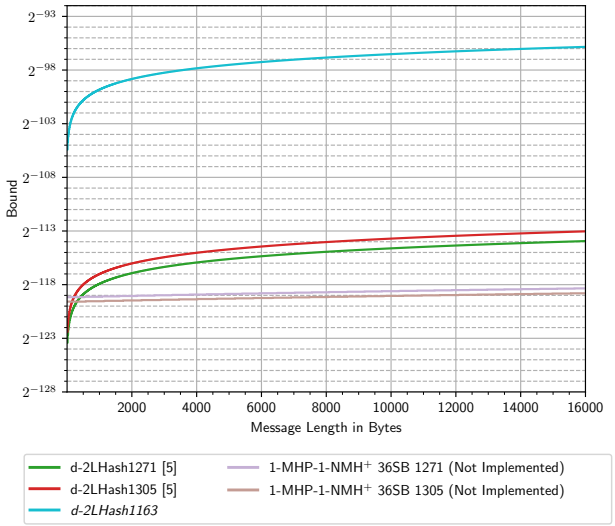


Figure 10: Security levels of d-2LHash compared to MHP-NMH$^+$.

cf. Sec. 5 and Fig. 6a, at higher security levels due to d-2LHash1271's larger field size. When using the same fields and key sizes, however, MHP-NMH$^+$ achieves better security bounds, cf. Fig. 10, and is very likely to be able to benefit from similar field arithmetic optimizations as d-2LHash1271, thus likely being a better alternative.

D Benchmarking Methodology for Hash2L and d-2LHash

For Hash2L we observed that for the original benchmarks in [9], the hash function interface requires that the message and key inputs are already stored in 128-bit SSE registers used by Intel's carryless multiplication instruction. However, the benchmarks in [9] only seem to measure the hash function execution time, excluding any overhead of transforming byte strings into the concrete representation used in the implementation. This is incompatible with the benchmarking framework of [10]. But, as we argue, it is also unrealistic, as the memory operations involved in these transforms can be very costly and have to be done in any real world implementation. Our wrappers – and thus our benchmarks – for Hash2L include these memory operations. This leads to noticeable increase in running time compared to the results originally reported in [9].

For d-2LHash we also observed issues related to memory usage. Specifically, the implementation of [5] requires that the message is stored in a buffer of at least 8 KiB (plus a few bytes depending on the exact variant). Otherwise, the implementation performs out-of-bounds memory accesses leading to stack overflows and segmentation faults. Thus, our wrappers first copy the message into a new buffer of sufficient size. The cost of these copies is again included in our benchmarks. As the memory layout itself is simple and contiguous these copies are rather fast and our measurements are similar to those reported in [5]. As d-2LHash has several parameters that affect performance, and the configuration that worked best on the machines used in [5] might not be the best for our CPU, we benchmark all variants provided in [5] and compare our designs against the variants performing the best. To be consistent with our implementations we include time spent on key precomputation when benchmarking d-2LHash.

We also note that for both Hash2L and d-2LHash, the respective benchmarking scripts have a hard-coded

limit of 8 KiB for the message size. Our benchmarks generally measure for messages up to 16 kB. While the implementations seem to work for messages past 8 KiB, we did not perform any further correctness tests on them.

E Factor Analysis

We apply a factor analysis to our benchmark data to isolate and understand the impact of individual design choices. The methodology is detailed in App. E.1, and the analysis itself, along with its conclusions, is presented in App. E.2.

E.1 Methodology

We use a full factorial design, for our factor analysis, following the methodology of [23], and conduct an ANOVA (analysis of variation) on the resulting model. In such an experimental design our experiments have a fixed number of parameters (called factors), that are set to a number of discrete levels (the number of levels can vary according to the factor). *Full factorial* denotes the fact that all possible combinations of these factors and levels are benchmarked. Our models generally consist of four different factors A, B, C, D, each with a, b, c, d different levels using r samples.[6] We fit our measurements to the following regression model:

$$\begin{aligned} y_{i,j,k,l,n} = \mu &+ \alpha_i + \beta_j + \xi_k + \delta_l + \gamma_{i,j}^{AB} + \gamma_{i,k}^{AC} + \gamma_{i,l}^{AD} + \gamma_{j,k}^{BC} + \gamma_{j,l}^{BD} \\ &+ \gamma_{k,l}^{CD} + \gamma_{i,j,k}^{ABC} + \gamma_{i,j,l}^{ABD} + \gamma_{i,k,l}^{ACD} + \gamma_{j,k,l}^{BCD} + \gamma_{i,j,k,l}^{ABCD} \\ &+ e_{i,j,k,l,n} \end{aligned}$$

with $i \in \{1, \ldots, a\}$, $j \in \{1, \ldots, b\}$, $k \in \{1, \ldots, c\}$, $l \in \{1, \ldots, d\}$ where μ denotes the mean of the measurements, α_i, β_j, ξ_k, and δ_l denotes the effect of factor A at level i, B at level j, C at level k, and D at level l, respectively; $\gamma_{\ldots}^{\ldots}$ denotes the effect of interactions between the factors in the superscript, when these factors are set to the corresponding levels in the subscript; $e_{i,j,k,l,n}$ denotes the error of the model relative to the measurements for the experiment with factors at levels i, j, k, l during the n-th repetition; and finally, $y_{i,j,k,l,n}$ denotes the measurement quantity (e.g. number of cycles) with factors at levels i, j, k, l during the n-th repetition.

To concretely instantiate this model with our experimental results, we use the (decimal) logarithm of the number of cycles measured during our benchmarks as the measurement quantity $y_{i,j,k,l,n}$. The logarithm ensures effects interact multiplicatively in the model and helps ensure the homoscedasticity of the model errors. Lastly, we perform an analysis of variation (ANOVA) on this model to estimate the relative contribution of each factor (and their interactions) to the measured result.

We partition our benchmarking results in several, separate ways to build several models. Firstly, all of our models work on a fixed field and a fixed message length, as varying these two parameters impact the performance in such a dominant way as to not require in-depth analysis. We build all our models for each of the four fields, and message sizes starting from 500 bytes increasing in 500 bytes increments up to and including 16 kB.

E.2 Factor Analysis

For our analysis, we divide our benchmarks into two different classes of models.

1. **Variable-Key-Length (VKL) Polynomials**: For this model, the factors we look at are the higher

[6] Often called repetitions in the general literature

and lower-level polynomial, the superblock size, and the delay parameter with the following factor levels:

Higher-Level Polynomial:	$\hat{\mathsf{t}}$BRW, tMMH, tNMH, (tSQH)
Lower-Level Polynomial:	MMH^+, NMH^+, (SQH^+)
Superblock Size:	4, 8, 12, 16, 20, 24, 28, 32, 36
Delay:	2, 4, 6, 8, 12, 16

2. **Multivariate Horner**: For this model, we remove the higher-level polynomial as a factor by fixing it to be MHP. In its stead, we add the number of keys for MHP as an additional factor, resulting in the following factors and levels.

Lower-Level Polynomial:	1-MMH^+, 1-NMH^+, (1-SQH^+)
Superblock Size:	4, 8, 12, 16, 20, 24, 28, 32, 36
Delay:	2, 4, 6, 8, 12, 16
Number of MHP Keys:	1, 2, 3, 4, 5

Note that the polynomials based on SQH are only used in prime fields.

VKL Polynomials. Figure 11 shows the percentage of variation explained by the different factors for the four different fields at different message sizes for the models including key-generation time and Fig. 12. shows the same for the models excluding key-generation time. We observe that as message length increases the impact of the choice of lower-level polynomial quickly rises, becoming the dominant factor in all fields except the smaller prime field. This is consistent with our expectations as, asymptotically, most of the computation happens at the leaves of the tree construction where the evaluation of the lower-level polynomial takes place. This effect is more pronounced both for binary fields and larger fields and is caused by the different performance characteristics of the hardware instructions used for the field arithmetic, and the need to handle carries for prime fields. The absence of carry bits in binary fields allows us to always fully delay the reduction in the lower-level polynomial. Thus, the delay parameter simply controls the loop unrolling factor and so its relative impact is reduced. This effect is furthermore compounded by the higher contention on the CPU's execution units used for binary field arithmetic when compared to prime field arithmetic. This significantly reduces the impact of loop unrolling, further decreasing the impact of the delay parameter for binary fields. In addition, a larger field size allows for a larger blocksize thus requiring less field operations per message. For prime fields this effect also reduces the impact of using delayed reduction, as reductions generally happen less frequently. On the other hand, smaller fields also increase the effect of good pipeline utilization via loop unrolling, as a single field operation consists of fewer dependent instructions. However, to make use of this at the lower level, the message superblocks have to be large enough to be able to fully exploit the pipelines during the lower level computations. Furthermore, the superblock size has a significant impact for smaller messages due to its effect on the cost of key generation. More specifically, it directly dictates the number of keys required for a message of a fixed size.

Most of the interactions that pass our 1% threshold are interactions involving the superblock size, most notably the two-factor interactions between superblock size and each of the remaining three factors. This is to be expected, since the superblock size ultimately has an effect on the amount of delayed reduction and loop unrolling that can be done at the lower level, as well as the tree depth at the higher level.

Multivariate Horner. The percentage of variation explained by the different factors is shown in Fig. 13 for MHP. We observe similar effects regarding the delay parameter and the impact of the lower-level

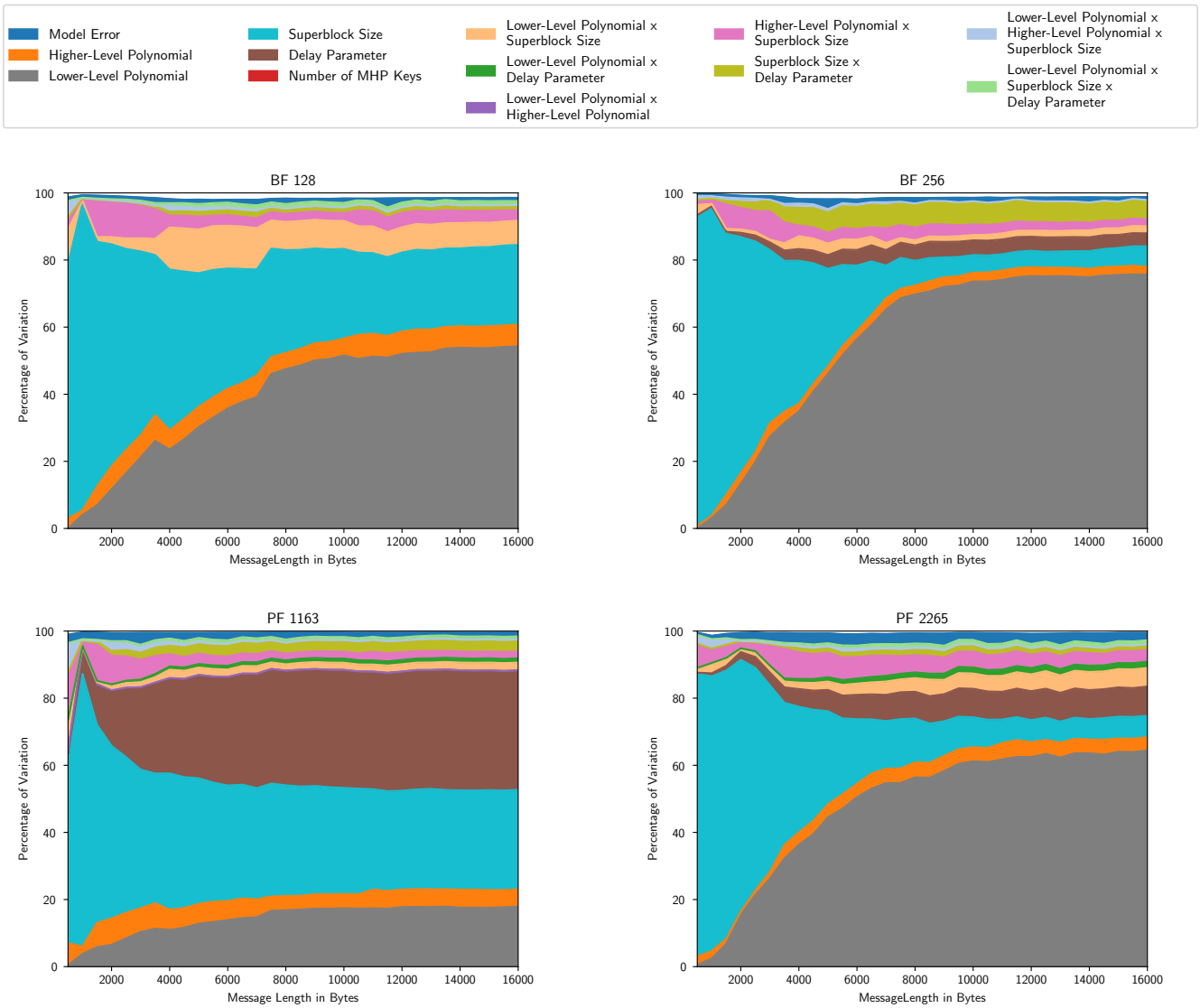


Figure 11: Contribution to variation of factors contributing at least 1% when using VKL polynomials.

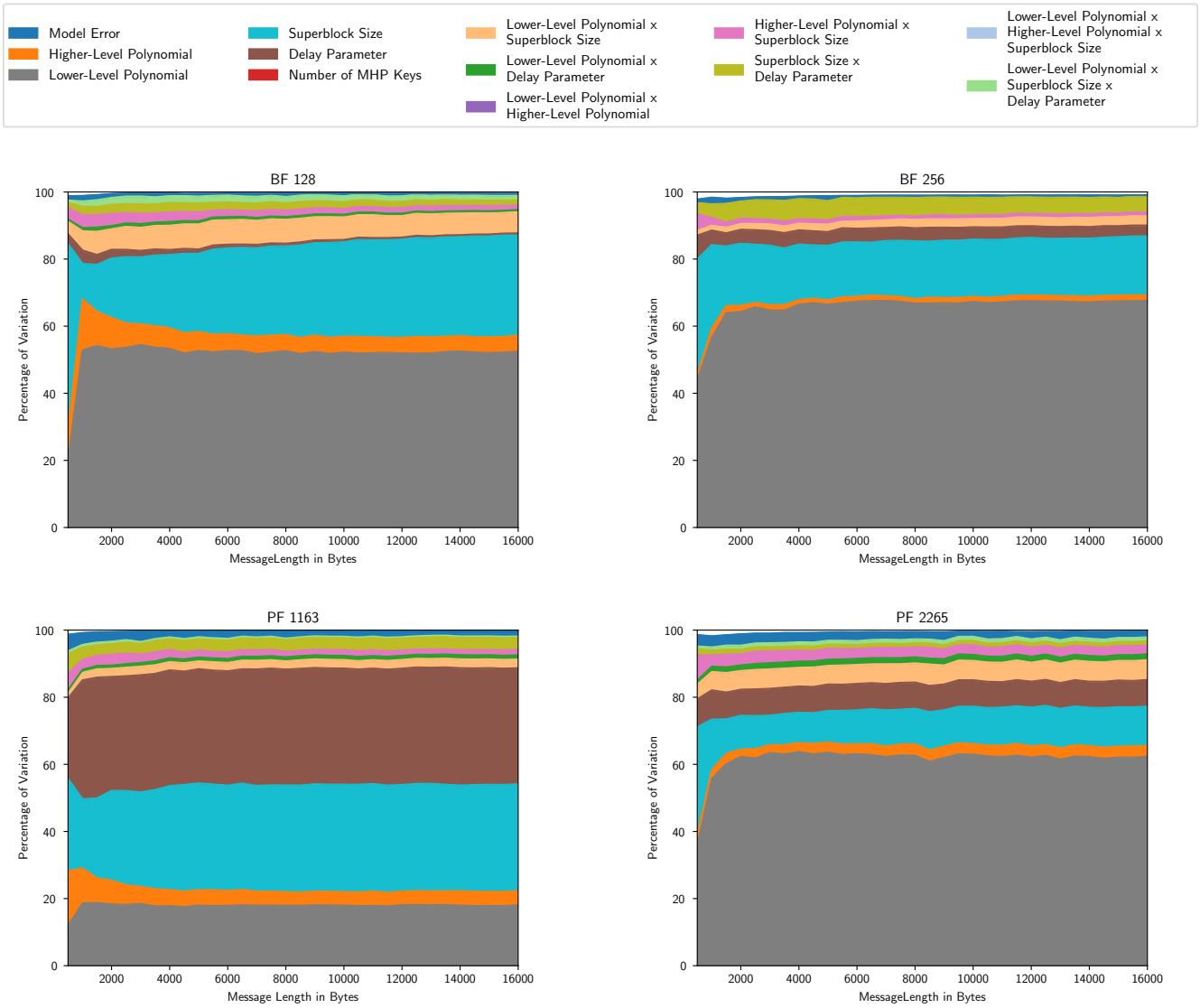


Figure 12: Contribution to variation of factors contributing at least 1% for VKL polynomials excluding key-generation time.

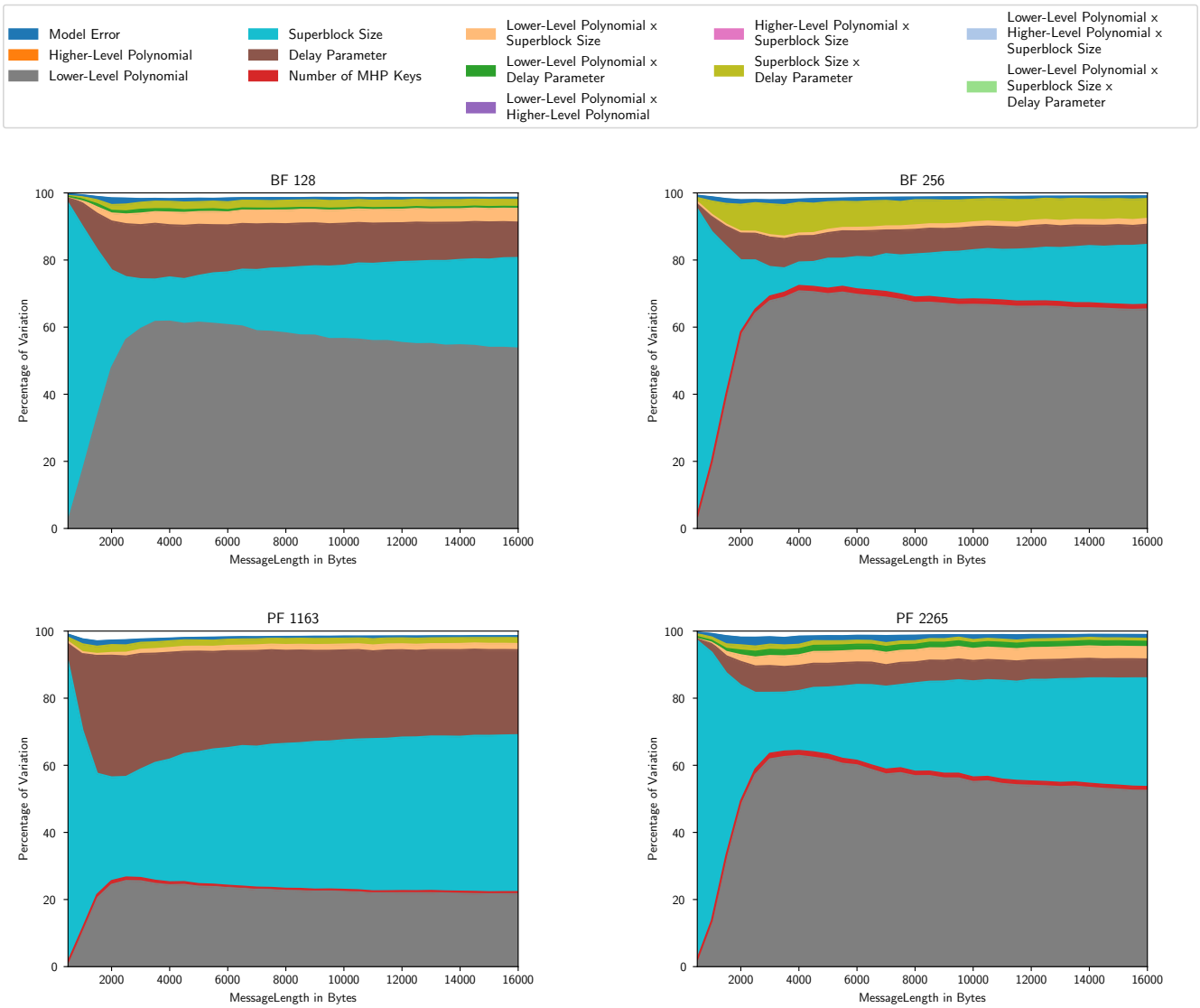


Figure 13: Contribution to variation of factors contributing at least 1% when using MHP.

polynomial as in the previous model. Our main observation is that within the number of keys – and thus variables – for multivariate Horner that we benchmarked, the impact on performance is small compared to other factors. This is as expected, since the only potential impacts of using more keys are cache effects due to handling more key material, which increases the likelihood of register spill, and the generic nature of our implementation resulting in potentially worse instruction cache locality. Given the small relative size of these effects, however, we conclude that the number of keys can be chosen mostly based on security concerns and the eventual combination of the hash function with other primitives (e.g. any key derivation when building an AEAD scheme).

As with the VKL polynomials, the main interactions are two-factor interactions involving the superblock size, as this again controls the amount of delayed reduction and loop unrolling, and general pipeline utilization at the lower level.

F Security Levels

Scheme	Security Bound
Poly1305	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil}{2^{103}}$
Poly1163	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil}{2^{112}}$
Poly2265	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil}{2^{224}}$
Poly2663	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil}{2^{256}}$
$\hat{\text{t}}$BRW-NMH$^+$ 24SB 1163	$\frac{\left\lfloor \log_2\left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil}{25} \right\rfloor\right)\right\rfloor+3}{2^{112}}$
$\hat{\text{t}}$BRW-NMH$^+$ 28SB 2265	$\frac{\left\lfloor \log_2\left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil}{29} \right\rfloor\right)\right\rfloor+3}{2^{224}}$
tMMH-NMH$^+$ 24SB 1163	$\frac{\left\lfloor \log_2\left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil}{24} \right\rceil-1\right)\right\rfloor+3}{2^{112}}$
tMMH-NMH$^+$ 28SB 2265	$\frac{\left\lfloor \log_2\left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil}{28} \right\rceil-1\right)\right\rfloor+3}{2^{224}}$
1-MHP-1-NMH$^+$ 36SB 1163	$\frac{\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil}{36} \right\rceil+37}{2^{112}}$
1-MHP-1-NMH$^+$ 36SB 2265	$\frac{\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil}{36} \right\rceil+37}{2^{224}}$
5-MHP-1-NMH$^+$ 32SB 1163	$\frac{5\cdot\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil}{32} \right\rceil^{1/5}+33}{2^{112}}$
5-MHP-1-NMH$^+$ 36SB 2265	$\frac{5\cdot\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil}{36} \right\rceil^{1/5}+37}{2^{224}}$
d-2LHash1163	$\frac{2\cdot\lceil \max(\lvert M\rvert,\lvert M'\rvert)/112\rceil+1}{2^{107}}$
d-2LHash2265	$\frac{2\cdot\lceil \max(\lvert M\rvert,\lvert M'\rvert)/224\rceil+1}{2^{221}}$

Table 4: Security bounds for the hash functions depicted in Figs. 6 and 7.

Scheme	Security Bound
Classical Polynomial 128	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil+1}{2^{128}}$
Classical Polynomial 256	$\frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil+1}{2^{256}}$
$\hat{\mathsf{t}}$BRW-NMH$^+$ 20SB 128	$\frac{\left\lfloor \log_2\left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil}{21} \right\rfloor\right)\right\rfloor+3}{2^{128}}$
$\hat{\mathsf{t}}$BRW-NMH$^+$ 24SB 256	$\frac{\left\lfloor \log_2\left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil}{25} \right\rfloor\right)\right\rfloor+3}{2^{256}}$
tMMH-NMH$^+$ 24SB 128	$\frac{\left\lfloor \log_2\left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil}{24} \right\rceil-1\right)\right\rfloor+3}{2^{128}}$
tMMH-NMH$^+$ 24SB 256	$\frac{\left\lfloor \log_2\left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil}{24} \right\rceil-1\right)\right\rfloor+3}{2^{256}}$
1-MHP-1-NMH$^+$ 36SB 128	$\frac{\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil}{36} \right\rceil+37}{2^{128}}$
1-MHP-1-NMH$^+$ 24SB 256	$\frac{\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil}{24} \right\rceil+25}{2^{256}}$
5-MHP-1-NMH$^+$ 36SB 128	$\frac{5\cdot\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/128\rceil}{36} \right\rceil^{1/5}+37}{2^{128}}$
5-MHP-1-NMH$^+$ 28SB 256	$\frac{5\cdot\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/256\rceil}{28} \right\rceil^{1/5}+29}{2^{256}}$
Poly-BRW 128[9, Theorem 1]	$\frac{32\cdot\lceil\lceil\lvert M\rvert/128\rceil/31\rceil+1}{2^{128}}$
Poly-BRW 256[9, Theorem 1]	$\frac{32\cdot\lceil\lceil\lvert M\rvert/256\rceil/31\rceil+1}{2^{256}}$
Hash2L 128[9, Theorem 1]	$\frac{32\cdot\lceil\lceil\lvert M\rvert/128\rceil/31\rceil+1}{2^{128}}$
Hash2L 256[9, Theorem 1]	$\frac{32\cdot\lceil\lceil\lvert M\rvert/256\rceil/31\rceil+1}{2^{256}}$

Table 5: Security bounds for the hash functions depicted in Figs. 4 and 5.

Scheme	Security Bound
tMMH-(MMH$^+$/NMH$^+$/SQH$^+$) tSQH-(MMH$^+$/NMH$^+$/SQH$^+$)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_2 \left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{B} \right\rceil - 1 \right) \right\rfloor + 3}{2^\rho}$
tNMH-(MMH$^+$/NMH$^+$/SQH$^+$)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_3 \left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{B} \right\rceil - 1 \right) \right\rfloor + 3}{2^\rho}$
t̂BRW-(MMH$^+$/NMH$^+$/SQH$^+$)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_2 \left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{(B+1)} \right\rfloor \right) \right\rfloor + 3}{2^\rho}$
tMMH-(HKM) tSQH-(HKM)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_2 \left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{B} \right\rceil - 1 \right) \right\rfloor + \frac{B}{2} + 2}{2^\rho}$
tNMH-(HKM)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_3 \left(\left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{B} \right\rceil - 1 \right) \right\rfloor + \frac{B}{2} + 2}{2^\rho}$
t̂BRW-(HKM)	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\left\lfloor \log_2 \left(\left\lfloor \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{(B+1)} \right\rfloor \right) \right\rfloor + \frac{B}{2} + 2}{2^\rho}$
v_1-MHP-v_2-MMH$^+$ v_1-MHP-v_2-NMH$^+$ v_1-MHP-v_2-SQH$^+$	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{v_1 \left\lceil \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{B} \right\rceil^{\frac{1}{v_1}} + v_2 B^{\frac{1}{v_2}} + 1}{2^\rho}$
Poly-BRW	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\lceil \lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil / B \rceil \left(2^{\lfloor \log_2 B \rfloor + 1}\right) + 1}{2^\rho}$
Classical Polynomial	$\left(\left\lfloor \frac{2q-2}{2^\tau} \right\rfloor + 1\right) \cdot \frac{\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil}{2^\rho}$
d-2LHash	$\frac{(1 + 2\lceil \max(\lvert M\rvert,\lvert M'\rvert)/\mu \rceil) \cdot 2^{\lceil \log_2 q \rceil + 1}}{2^{2\rho}}$

Table 6: Security bounds for the constructions considered throughout this paper. ρ: length of a single key in bits, μ: length of a messageblock in bits, τ: length of the tag in bits, B: superblock size in number of blocks, q: number of elements in the field.

G Tables of Benchmarking Results

(a) $\mathsf{GF}(2^{116}-3)$

Higher	Lower	SB	DP	TP
t̂BRW	MMH$^+$	24	16	0.71
t̂BRW	NMH$^+$	24	16	0.58
t̂BRW	SQH$^+$	20	8	0.71
t̂BRW	HKM	20	-	0.85
tMMH	MMH$^+$	24	12	0.72
tMMH	NMH$^+$	24	16	0.60
tMMH	SQH$^+$	28	8	0.74
tMMH	HKM	20	-	0.90
tNMH	MMH$^+$	24	12	0.77
tNMH	NMH$^+$	24	16	0.64
tNMH	SQH$^+$	24	8	0.79
tNMH	HKM	20	-	0.92
tSQH	MMH$^+$	28	16	0.72
tSQH	NMH$^+$	28	16	0.59
tSQH	SQH$^+$	20	8	0.75
tSQH	HKM	20	-	0.90

(b) $\mathsf{GF}(2^{226}-5)$

Higher	Lower	SB	DP	TP
t̂BRW	MMH$^+$	8	8	1.21
t̂BRW	NMH$^+$	28	8	1.00
t̂BRW	SQH$^+$	8	8	1.10
t̂BRW	HKM	20	-	1.10
tMMH	MMH$^+$	12	12	1.36
tMMH	NMH$^+$	28	8	1.02
tMMH	SQH$^+$	28	6	1.18
tMMH	HKM	20	-	1.27
tNMH	MMH$^+$	8	12	1.28
tNMH	NMH$^+$	28	8	1.01
tNMH	SQH$^+$	28	6	1.17
tNMH	HKM	32	-	1.15
tSQH	MMH$^+$	8	16	1.35
tSQH	NMH$^+$	28	6	1.02
tSQH	SQH$^+$	28	6	1.17
tSQH	HKM	20	-	1.12

Table 7: Peak throughput parameters for VKL polynomials over prime fields. SB: Superblock size in number of blocks, DP: Delay Parameter, TP: Throughput in cycles/Byte rounded to two decimal places.

(a) $\mathsf{GF}(2^{128})$

Higher	Lower	SB	DP	TP
t̂BRW	MMH$^+$	24	1	0.53
t̂BRW	NMH$^+$	20	4	0.38
t̂BRW	HKM	8	-	0.79
tMMH	MMH$^+$	24	1	0.55
tMMH	NMH$^+$	24	8	0.40
tMMH	HKM	20	-	0.83
tNMH	MMH$^+$	20	1	0.59
tNMH	NMH$^+$	20	8	0.42
tNMH	HKM	8	-	0.82

(b) $\mathsf{GF}(2^{256})$

Higher	Lower	SB	DP	TP
t̂BRW	MMH$^+$	8	6	0.89
t̂BRW	NMH$^+$	24	2	0.67
t̂BRW	HKM	16	-	0.92
tMMH	MMH$^+$	20	6	0.98
tMMH	NMH$^+$	24	2	0.69
tMMH	HKM	16	-	0.97
tNMH	MMH$^+$	20	6	0.97
tNMH	NMH$^+$	20	2	0.69
tNMH	HKM	32	-	0.95

Table 8: Peak throughput parameters for VKL polynomials over binary fields. SB: Superblock size in number of blocks, DP: Delay Parameter, TP: Throughput in cycles/Byte rounded to two decimal places.

(a) $\mathsf{GF}(2^{116}-3)$

Lower	MHP Keys	SB	DP	TP
1-MMH$^+$	1	36	6	0.77
1-MMH$^+$	2	36	6	0.78
1-MMH$^+$	3	36	6	0.78
1-MMH$^+$	4	32	8	0.78
1-MMH$^+$	5	32	8	0.78
1-NMH$^+$	1	36	12	0.54
1-NMH$^+$	2	36	12	0.54
1-NMH$^+$	3	36	12	0.55
1-NMH$^+$	4	36	12	0.56
1-NMH$^+$	5	36	12	0.56
1-SQH$^+$	1	32	8	0.66
1-SQH$^+$	2	32	8	0.67
1-SQH$^+$	3	32	8	0.69
1-SQH$^+$	4	32	8	0.69
1-SQH$^+$	5	32	8	0.69

(b) $\mathsf{GF}(2^{226}-5)$

Lower	MHP Keys	SB	DP	TP
1-MMH$^+$	1	24	6	1.41
1-MMH$^+$	2	28	4	1.43
1-MMH$^+$	3	28	4	1.44
1-MMH$^+$	4	32	4	1.45
1-MMH$^+$	5	28	4	1.45
1-NMH$^+$	1	36	8	0.93
1-NMH$^+$	2	28	6	0.95
1-NMH$^+$	3	32	8	0.97
1-NMH$^+$	4	36	6	0.97
1-NMH$^+$	5	36	6	0.98
1-SQH$^+$	1	24	6	1.08
1-SQH$^+$	2	28	6	1.11
1-SQH$^+$	3	36	6	1.11
1-SQH$^+$	4	36	6	1.13
1-SQH$^+$	5	36	6	1.13

Table 9: Peak throughput parameters for MHP over prime fields. SB: Superblock size in number of blocks, DP: Delay Parameter, TP: Throughput in cycles/Byte rounded to two decimal places.

(a) $\mathsf{GF}(2^{128})$

Lower	MHP Keys	SB	DP	TP
1-MMH$^+$	1	28	2	0.46
1-MMH$^+$	2	32	4	0.47
1-MMH$^+$	3	32	4	0.47
1-MMH$^+$	4	28	4	0.47
1-MMH$^+$	5	28	2	0.46
1-NMH$^+$	1	36	4	0.29
1-NMH$^+$	2	32	4	0.29
1-NMH$^+$	3	36	4	0.29
1-NMH$^+$	4	36	4	0.30
1-NMH$^+$	5	36	4	0.29

(b) $\mathsf{GF}(2^{256})$

Lower	MHP Keys	SB	DP	TP
1-MMH$^+$	1	32	6	0.85
1-MMH$^+$	2	20	6	0.86
1-MMH$^+$	3	32	6	0.88
1-MMH$^+$	4	20	6	0.88
1-MMH$^+$	5	20	6	0.89
1-NMH$^+$	1	24	2	0.56
1-NMH$^+$	2	24	2	0.57
1-NMH$^+$	3	24	2	0.58
1-NMH$^+$	4	28	2	0.59
1-NMH$^+$	5	28	2	0.59

Table 10: Peak throughput parameters for MHP over binary fields. SB: Superblock size in number of blocks, DP: Delay Parameter, TP: Throughput in cycles/Byte rounded to two decimal places.

H Artifact Appendix

H.1 Abstract

The artifact accompanying this paper contains the source code of our updated polynomial hashing framework, together with the configuration files needed to reproduce the main performance results, described in Sec. 5.5, for the new universal hash functions over binary fields. It also includes the source code and benchmark scripts for the Rijndael256-based AEAD schemes from Sec. 5.6.

We further include docker compose configuration files to easily set up all dependencies. The provided scripts generate, compile, test, and benchmark the relevant implementations, and produce the benchmark data and plots used to validate the paper's main performance claims. Beyond reproducing the reported experiments, the framework is intended to be reusable: researchers can select different field options, benchmark additional parameter choices, and add new polynomial constructions or reference implementations.

Unless otherwise stated all code is licensed under the MIT license. Do note however, that this artifact does redistribute some of its dependencies, which are licensed under their respective original licenses, as noted in the respective files.

H.2 Description & Requirements

The full `README` can be found at the root of the repository. The source code of the framework can be found in `src/`, reference implementations are in `ref/`. The configuration files defining the UHFs we benchmarked are in `configs/multivariate_polynomials/`. The folder `rijndael_aead/` contains source code for the AEAD schemes built using these hash functions. For a full overview of the repository structure, please consult the `README`.

H.2.1 Security, privacy, and ethical concerns

None

H.2.2 How to access

An archived version of this artifact can be accessed via Zenodo at `https://doi.org/10.5281/zenodo.20677844`. This Zenodo record corresponds to the following GitHub repository version v2.0.3: `https://github.com/jangilcher/polynomial_hashing_framework`

H.2.3 Hardware dependencies

This artifact requires a 64-bit x86 CPU with support for carryless multiplication and AES-NI instructions set extensions.

H.2.4 Software dependencies

Requires python3.11, sagemath, gcc and corresponding build tools, libsodium, as well as the python packages specified in `requirements.txt`. We recommend a Linux environment (e.g. Ubuntu 20.04 LTS or newer) for better comparison with our benchmarks. To our knowledge the artifact should also run on Intel based macOS. For easy set-up of all dependencies we recommend using our provided docker compose configurations. For this Docker and Docker Compose are required (tested using Docker version 29.5.2 and Docker Compose v5.1.4).

H.2.5 Benchmarks

The benchmarks in this artifact are mostly self-contained. All dependencies are either part of the artifact or will be automatically set up. For reproducibility we depend on the version HACL* commit `d59b0b08bc07e8f65490c9cfb12d1c3f51f48f61`. This version will be automatically downloaded during setup.

H.3 Set Up

We recommend using a Linux system and configuring the system to run the CPU at a fixed frequency. This can be done in the BIOS as well as by setting specific flags or values in the MSR. As this is CPU (vendor) specific we instead describe a different process that limits the frequency range for dynamic frequency scaling using `cpufrequtils` package, which can usually be found in most Linux distribution's package repositories. On Ubuntu, it can be installed with:

```
sudo apt install cpufrequtils
```

We recommend setting the frequency to the base frequency of the CPU. On our system (Intel Xeon Gold 6258R) this corresponds to 2.7GHz. This can then be achieved using the following commands with root privileges:

```
cores=$(cat /proc/cpuinfo | grep processor | awk '{print $3}')
for i in $cores;
    do
        cpufreq-set -c $i -g performance;
        cpufreq-set -c $i -d 2.7GHz;
        cpufreq-set -c $i -u 2.7GHz;
    done
```

These settings do not persist a reboot.

H.3.1 Installation

We assume Docker and Docker Compose are installed on the system. Instructions to install these can be found at: `https://docs.docker.com/compose/install/`. Download the artifact via `https://doi.org/10.5281/zenodo.20677844` and unzip to a folder of your choice. Open a shell in the root of the unzipped artifact folder and run

```
docker compose build
```

to build the docker container. Run `chmod -R 777 docker_data/` to ensure the docker container has full access to the shared folder. Finally, to start the docker container with an interactive shell, run:

```
docker compose run --rm hashing-framework
```

H.3.2 Basic test

A quick functional test can be performed by running the following command in the docker environment:

```
python3 ./run.py --stepsize=100 --tune --iterations=5 --max_messagesize=500 \
--plot_y_cut=1.8 --no_test ./configs/multivariate_polynomials/config_fig4a.json
```

After the execution finishes, there should now be 52 files within `plots/<timestamp>/png`. Fig. 14 shows an example of how these can look. The plots with `comparison` in their name should contain 9 different graphs.

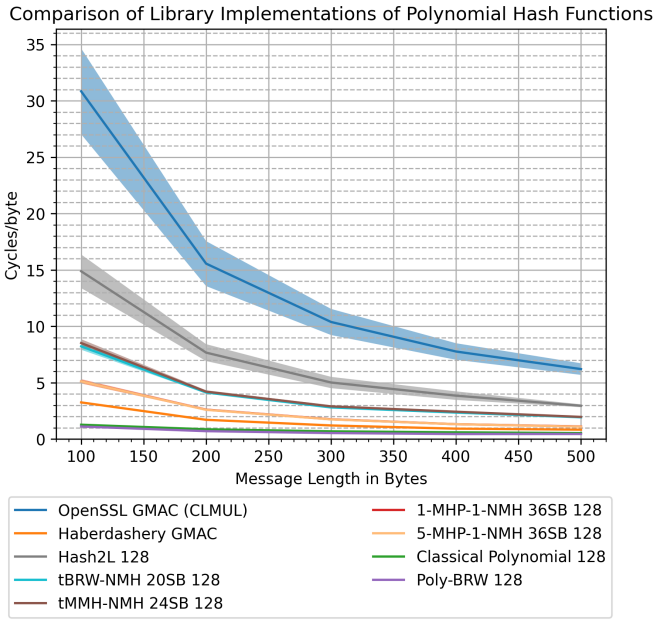


Figure 14: Example Test Plot.

H.4 Evaluation Workflow

H.4.1 Major claims

Note that our artifacts only relate to our performance claims, as such the major claims in this section only refer to performance.

(C1): 1-MHP-1-NMH$^+$ outperforms the state-of-the-art (Hash2L) by up to 25% in the 128-bit case and up to 15% in 256-bit case. This is shown by our benchmarking (E1) and (E2) described in Sec. 5.3, whose results are depicted in Fig. 4 and discussed in Sec. 5.5.

(C2): The performance impact of increasing the number of keys for MHP-NMH$^+$ over binary fields from 1 key to 5 keys is negligible. This is shown by our benchmarking (E1) and (E2) described in Sec. 5.3, whose results are depicted in Fig. 4 and discussed in Sec. 5.5.

(C3): For 128-bit fields the VKL constructions, t̂BRW-NMH$^+$ and tMMH-NMH$^+$, achieve performance matching the state-of-the-art (Hash2L). This is shown by our benchmarking (E1) and (E2) described in Sec. 5.3, whose results are depicted in Fig. 4 and discussed in Sec. 5.5.

(C4): The use of MHP-NMH$^+$ in a 256-bit blocksize Rijndael-based AEAD scheme leads to large performance improvements over using the classical polynomial (about 5% for 128-bit authenticators and 21% for 256-bit authenticators). This is shown by our benchmarking (E3) described in Sec. 5.6, whose results are discussed in Sec. 5.6 and depicted in Fig. 8.

H.4.2 Experiments

(E1): *128-bit Binary Field UHF Benchmarks [10 person-minutes + 0.5–1 compute-hour]:* This experiment benchmarks our best new UHF designs as well as several reference implementations over a 128-bit binary field. This should reproduce the results in Fig. 4a.

Preparation: Ensure the system is set up as described in the sections on installation and set-up. If not yet in the docker environment, run

```
docker compose run --rm hashing-framework
```

as described in the installation section above.
Execution: Run the following command in the shell obtained from starting the docker container:

```
python3 ./run.py --stepsize=100 --tune --iterations=25 --max_messagesize=16000 \
--plot_y_cut=1.8 --no_test ./configs/multivariate_polynomials/config_fig4a.json
```

Results: The raw benchmark data will be stored in:

```
bench/<timestamp>/config_fig4a.json_<id>_results.csv
```

`<id>` is the index of the specific configuration in the configuration file. Note that outside the docker environment the bench folder is located in `docker_data`. The generated plots (PNG and SVG) are saved in `plots/<timestamp>`, with subfolders for the two respective image formats. These folders contain several plots, both only containing the individual hash function and plots comparing all hash functions in the experiment. The primary plot to verify results is:

```
config_fig4a.json_comparison_0_1_2_3_4_5_6_7_8_rate_y_cut_with_keygen
```

The performance of the UHF schemes depicted in this plot should behave similarly to those in Fig. 4a. Please note that the colors automatically chosen by the framework might differ from those in Fig. 4a.

(E2): *256-bit Binary Field UHF Benchmarks [10 person-minutes + 0.5–1 compute-hour]:* This experiment benchmarks our best new UHF designs as well as several reference implementations over a 256-bit binary field. This should reproduce the results in Fig. 4b.
Preparation: Ensure the system is set-up as described in the sections on installation and set-up. If not yet in the docker environment, run

```
docker compose run --rm hashing-framework
```

as described in the installation section above.
Execution: Run the following command in the shell obtained from starting the docker container:

```
python3 ./run.py --stepsize=100 --tune --iterations=25 --max_messagesize=16000 \
--plot_y_cut=1.8 --no_test ./configs/multivariate_polynomials/config_fig4b.json
```

Results: The raw benchmark data will be stored in:

```
bench/<timestamp>/config_fig4b.json_<id>_results.csv
```

`<id>` is the index of the specific configuration in the configuration file. Note that outside the docker environment the bench folder is located in `docker_data`. The generated plots (PNG and SVG) are saved in `plots/<timestamp>`, with subfolders for the two respective image formats. These folders contain several plots, both only containing the individual hash function and plots comparing all hash functions in the experiment. The primary plot to verify results is:

```
config_fig4b.json_comparison_0_1_2_3_4_5_6_rate_y_cut_with_keygen
```

The performance of the depicted UHF schemes should behave similarly to those in Fig. 4b. Please note that the colors automatically chosen by the framework might differ from those in Fig. 4b.

(E3): Rijndael-*Based AEAD Schemes [10 person-minutes + 3–6 compute-hours]:* This experiment benchmarks several 256-bit blocksize Rijndael-based AEAD schemes utilizing the classical polynomial as well as MHP-NMH$^+$ over both 128- and 256-bit binary fields. This demonstrates the performance gains from using MHP-NMH$^+$ over the classical polynomial.

Preparation: Ensure the system is set up as described in the sections on installation and set-up. If not yet in the docker environment, run

```
docker compose run --rm hashing-framework
```

as described in the installation section above. In the shell, navigate to `rijndael_aead/` (i.e. `cd rijndael_aead`), ensure the necessary folder structure is created (`make dir`), and build the various Rijndael implementations via:

```
make MODE=bench FOLDER=bench/ REPETITIONS=25 STEPSIZE=1 ITERATIONS=1000 all
```

On some setups, the benchmarks built via the above command may run for an extremely long time. For a quick replication with noisier results the `REPETITIONS` and `STEPSIZE` can be adjusted. For instance, the following parameters should result in benchmarks that terminate in seconds to minutes:

```
make MODE=bench FOLDER=bench/ REPETITIONS=5 STEPSIZE=100 ITERATIONS=1000 all
```

Finally, create the folder to store the benchmarking results in: `mkdir bench`.
Execution: Execute the benchmarks via the following commands:

```
./bin/rijndael256x6x_no_hash_bench
./bin/rijndael256x_mhp_nmh128x6_bench
./bin/rijndael256x_mhp_nmh256x6_bench
./bin/rijndael256x_poly128x4_bench
./bin/rijndael256x_poly256x4_bench
```

Results: Run `python3 plot.py`. This will create a file, `rijndael_aead.png`, in the same folder. To access the image outside the docker environment copy it into:

```
../plots/
```

The performance of the depicted AEAD schemes should behave similarly to those in Fig. 8.

H.5 Notes on Reusability

This artifact can easily be reused to benchmark the framework generated UHFs over different fields. For this the field specifications in the corresponding configuration files have to be adapted. Further documentation on the syntax can be found in README and documentation provided as part of the artifact.

Other researchers can also add support for new polynomial constructions or further reference implementations. When implemented with the appropriate interface newly added polynomial constructions can then easily be used as either stand-alone polynomials or as one of the levels of new two-level constructions.

H.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926.