HardPoly1305 V2-Lite: A Tweaked Poly1305 Algorithm with Per-Block Pseudorandom Coefficients

Twilight-Dream & Erika

January 30, 2026

Abstract

This paper presents HardPoly1305 V2-Lite, an exploratory, patch-level enhancement of the Poly1305 message authentication code (MAC). We preserve the Poly1305-shaped polynomial update over $P_1 = 2^{130} - 5$, while replacing the fixed secret coefficients (r, s) with per-block coefficients (r_i, s_i) derived from a keyed core function *h_core*.

The design introduces a two-domain framework: polynomial accumulation is performed modulo P_1, while the internal core operates over $P_2 = 2^{256} - 188069$ (a safe prime) to mix state and message material before extracting (r_i, s_i). The *h_core* construction combines low-degree arithmetic in $\mathbb{Z}_{P_2}$ with lightweight 256-bit diffusion and bitwise mixing.

We provide a conditional UF-CMA-style analysis under an explicit PRF assumption on *h_core* and an idealized coefficient-extraction step. We also report limited differential sanity-check experiments on *h_core* (up to 5×10^5 samples), emphasizing that these tests do not constitute a PRF proof. Overall, HardPoly1305 V2-Lite is intended as a research-oriented variant rather than a standardized replacement.

Contents

1 Introduction **4**
- 1.1 Motivation and Design Overview . . . 4
- 1.2 Relationship to Prior Variants . . . 4
- 1.3 Our Contribution . . . 4
- 1.4 Security Summary (Conditional) . . . 4
- 1.5 Scope and Non-Goals . . . 4
- 1.6 Design Philosophy . . . 5

2 Intuitive Overview (Friendly) **5**

3 Mathematical Background **5**
- 3.1 Finite Fields and Prime Parameters . . . 5
- 3.2 Relationship Between P_1 and P_2 and Period Considerations . . . 6
- 3.3 Symbol Conventions . . . 6
- 3.4 Comparison: Fixed vs. Per-Block Coefficients . . . 7
  - 3.4.1 Original Poly1305 Structure . . . 7
  - 3.4.2 HardPoly1305 Structure . . . 7

4 Complete Mathematical Model **7**
- 4.1 Key Derivation: `DeriveKeyParams` . . . 7
- 4.2 Message-Key Mixing: `MixKeyAndMessage` . . . 7
- 4.3 Block Encoding . . . 8
- 4.4 Core Function: *h_core* . . . 8
- 4.5 Coefficient Extraction: `DeriveRS` . . . 8
- 4.6 Main Iteration . . . 9
- 4.7 System View as a Mapping Diagram (Two-Space Loop: $P_1 \leftrightarrow P_2$) . . . 9
- 4.8 Parameterized Algebraic Attack Model for the Polynomial Layer ($\mathrm{Adv}_{\mathrm{alg}}$) . . . 10

5 Conditional UF-CMA Security Discussion **11**
5.1 Notation and UF-CMA Experiment . . . 11
5.2 Security Model and Experiments . . . 12
5.3 Plain-Language Scope Statement (What "Won't Die" Means Here) . . . 12
5.4 Why a "Pure-Math" PRF Proof for a Concrete 256-bit Keyed Core is Impossible (Information-Theoretic) . . . 13
5.5 Poly1305 Algebraic Form vs. Security Basis . . . 14
5.6 Layered Decomposition of *h_core* (Step-by-step Algebraic Attack Surface) . . . 15
5.6.1 Bit-level nonlinearity of modular addition/subtraction (carry/borrow chains) . . . 19
5.6.2 Adversarial (non-uniform) inputs and the correct security notion . . . 20
5.6.3 Categorical view (optional global perspective: *h_core* as a morphism) . . . 21
5.7 Hybrid Games and Reduction . . . 21
5.8 Reduction to a Random-Coefficient Polynomial MAC . . . 22
5.9 What Our Algorithm (HardPoly1305) Is Tweaked From Poly1305: Algebraic Expansion and Block-Replay . . . 24
5.10 Replay Resistance (Protocol Fix + Quantitative Bounds) . . . 25
5.11 Assumptions (Axioms) . . . 27
5.12 Relationships and Baseline Bounds (Key Recovery, PRF Distinguishing, UF-CMA) . . . 28
5.13 Theorems and Bounds . . . 29
5.14 Role of Mixed(M, K) and Key Requirements . . . 30
5.15 Remarks on Assumptions . . . 31

6 Pseudocode Description **31**
6.1 Key Parameter Derivation . . . 31
6.2 Message-Key Mixing . . . 31
6.3 Core Function *h_core* . . . 31
6.4 Coefficient Extraction . . . 32
6.5 Main Algorithm . . . 32

7 Differential Experiment Results **33**
7.1 Experimental Setup . . . 33
7.2 Results . . . 33
7.3 Bit Bias Analysis . . . 33

8 Limitations and Open Problems **33**

9 Conclusion **34**

A Experimental Methods and Small Distinguishers (Reproducibility Appendix) **34**
A.1 Code artifacts . . . 34
A.2 Metrics and train/holdout discipline . . . 34
A.3 Linear mask search (1-bit vs 1-bit, train/holdout) . . . 35
A.4 Differential distinguisher search (train/holdout; XOR and add/sub models) . . . 35
A.5 Integral (cube XOR-sum) distinguisher search (train/holdout) . . . 35
A.6 Second-derivative "affine smell" diagnostic . . . 35
A.7 Truncated-view differential search . . . 35
A.8 2-minute budget preset outputs (collected logs) . . . 35

B FAQ (Non-normative) and Discussion: Erika Design Rationale and Limitations **40**
B.1 Design Story and Inspiration . . . 40
B.2 Chats: Why Per-Block (r_i, s_i)? . . . 41
B.3 Chats: Is This a New MAC or a Poly1305 Patch? . . . 42
B.4 Chats: Will It Be Slow? . . . 42
B.5 Chats: What if *h_core* is bad? . . . 42
B.6 Chats: Do I Still Need Nonces or Context? . . . 42
B.7 Chats: Why Pre-Mix the Message at All? . . . 42
B.8 Chats: Why a Second Prime P_2 (and Why Safe Prime)? . . . 43

B.9 Chats: Why the cross-bit mixing and the rotations? . . . 43
B.10 Chats: How Do I Sanity-Test This Without Lying to Myself? . . . 44
B.11 Chats: Did I Just Destroy Poly1305 Speed? . . . 44
B.12 Chats: You Said This is ARX-ish. Where are the "A", "R", and "X"? . . . 44
B.13 Chats: What is the Security Story Here (Diffusion/Confusion: which steps, which operators)? . . . 45
B.14 How dare you dabble in symmetric cryptography? . . . 45

C Python Implementation **46**

D Differential Experiment Console **50**

E SageMath Code for Finding Safe Primes **54**

1 Introduction

Poly1305, designed by Daniel J. Bernstein, is a widely deployed MAC with an elegant polynomial structure. Given a secret key (r, s) and a message $M = (m_1, \ldots, m_n)$, a common conceptual form of the authenticator can be written as:

$$\text{tag} = \left(\sum_{i=1}^{n} m_i \cdot r^{n-i+1} + s \right) \mod 2^{128}.$$

Under proper usage (e.g., one-time keys derived per message via a nonce-based mechanism), the fixed-coefficient structure poses no practical issue. However, the algebraic regularity of using a single r across all blocks provides a clean target surface in misuse settings, and it also yields a particularly structured object for deep algebraic modeling.

1.1 Motivation and Design Overview

HardPoly1305 V2-Lite keeps the Poly1305-shaped accumulation modulo $P_1 = 2^{130} - 5$, but changes how coefficients are obtained. Instead of holding (r, s) fixed, we generate fresh per-block coefficients (r_i, s_i) derived from a keyed core function *h_core* that depends on the running state and the current block. Intuitively, this aims to disrupt the globally shared polynomial parameter that makes the classical expression "too clean" under aggressive algebraic unfolding.

1.2 Relationship to Prior Variants

To the best of our knowledge, existing Poly1305 variants and deployments often emphasize key/nonce handling, deterministic one-key adaptations, or AEAD integration patterns (e.g., IPMAC/OPMAC-style adaptations and ChaCha20-Poly1305-style compositions). In contrast, our construction explores *state-derived per-block re-parameterization* of the polynomial evaluation: the coefficients become functions of the evolving state and message blocks, mediated by *h_core*.

1.3 Our Contribution

HardPoly1305 V2-Lite modifies the coefficient generation mechanism:

- **Original Poly1305:** Fixed (r, s) for all blocks.
- **HardPoly1305 V2-Lite:** Per-block (r_i, s_i) generated via a keyed PRF-like function *h_core*.

The core update rule remains Poly1305-shaped:

$$h_{i+1} = r_i \cdot (h_i + m_i) + s_i \pmod{P_1},$$

but now (r_i, s_i) are derived from the evolving transcript via a keyed mechanism.

1.4 Security Summary (Conditional)

We provide a conditional UF-CMA-style security analysis. Under a PRF assumption for *h_core* and an idealized coefficient extraction, the forgery advantage is bounded by:

$$\text{Adv}_{\text{uf-cma}}(\mathcal{A}) \leq \text{Adv}_{\text{prf}}(B_{\text{total}}) + \varepsilon_{\text{der}} + \frac{B_{\text{total}}}{2^{128}} + \frac{B_{\text{total}}^2}{2^{130}}$$

where B_{total} is the total number of message blocks processed. We stress that this bound is *conditional*: it depends on the assumed pseudorandomness of *h_core* and on the modeling of the coefficient-extraction step.

1.5 Scope and Non-Goals

This work is not a standardized replacement for Poly1305. It is an exploratory, patch-level enhancement focused on altering the coefficient structure while retaining the familiar polynomial accumulation pattern. As with any MAC, replay resistance is a protocol-level concern: preventing network replay requires freshness mechanisms (e.g., nonce/counter and verification-side state), which are orthogonal to the MAC construction itself.

1.6 Design Philosophy

Related work and positioning. To the best of our knowledge, much of the prior work around Poly1305 focuses on key/nonce handling or one-key/deterministic adaptations, as well as AEAD compositions (e.g., ChaCha20-Poly1305). In contrast, our construction explores *state-derived per-block re-parameterization* of the polynomial evaluation, where (r_i, s_i) depend on both the running state and the current block via h_{core}.

This is a **patch-level enhancement**, not a complete redesign:

1. We preserve the finite field polynomial MAC framework of Poly1305.
2. We add a "coefficient scheduler" that generates fresh (r_i, s_i) per block.
3. The security analysis reduces to: (a) MAC structure security under ideal random coefficients, and (b) PRF quality of *h_core*.

This paper does not claim a standard-model proof for the concrete construction. The security discussion is conditional on modeling *h_core* as a PRF and treating the extracted coefficients as independent; it should be read as an exploratory argument pending third-party cryptanalysis.

2 Intuitive Overview (Friendly)

This section is intentionally informal and does not replace the formal analysis.

Think of Poly1305 as a one-knob synth: set (r, s) once, and the whole message plays in that key. We keep the same melody, but twist the knob every bar:

$$(r_i, s_i) \leftarrow G(h\_core(h_{i-1}, X_i; K_{\text{core}})) .$$

So each block gets a fresh pair of coefficients tied to the evolving state. The result is a moving target: cut-and-paste a block into a different message, the prefix state changes, the coefficients change, and the block no longer fits.

We also pre-mix the message with key material before feeding it into *h_core*. Think of it as seasoning the input so even a short, boring block still tastes key-dependent before the nonlinear core.

- **Same as Poly1305:** polynomial-style update rule, modulus P_1, 16-byte tag.
- **New in V2-Lite:** per-block coefficients from the evolving state and a keyed core; keyed mixing before *h_core*.

Reminder. This is still a MAC. Exact replay of a previously authenticated pair $(M, \text{Tag}_K(M))$ is possible unless the surrounding protocol uses nonces, sequence numbers, or session context.

3 Mathematical Background

3.1 Finite Fields and Prime Parameters

We work with two carefully chosen primes:

Definition 3.1 (Working Primes)**.**

$$P_1 = 2^{130} - 5 \quad \textit{(Poly1305 prime)} \tag{1}$$

$$P_2 = 2^{256} - 188069 \quad \textit{(HardPoly1305 mixing prime)} \tag{2}$$

P_2 is a **safe prime**, meaning $(P_2 - 1)/2$ is also prime. This implies:

- The multiplicative group $\mathbb{Z}^*_{P_2}$ has order $P_2 - 1 = 2q$ with a large prime q.
- No small subgroups, which mitigates subgroup-based algebraic attacks.
- Well-behaved distribution when extracting pseudorandom-looking values.

In contrast, P_1 is a prime modulus from Poly1305, but it is **not assumed** to be a safe prime (nor a Sophie Germain prime). We therefore do not rely on subgroup structure properties of $\mathbb{Z}_{P_1}^*$ in the security arguments.

The prime P_2 was found using SageMath by scanning the interval $[2^{256} - 200000, 2^{256}]$ for safe primes.

3.2 Relationship Between P_1 and P_2 and Period Considerations

There is no algebraic dependency between P_1 and P_2; they define two independent prime fields used at different layers of the construction. P_1 governs the Poly1305-shaped accumulation, while P_2 governs the mixing domain of *h_core* before coefficients are extracted.

Regarding periodicity, the multiplicative order of an element in $\mathbb{Z}_p^*$ always divides $p-1$. For a safe prime $P_2 = 2q+1$, possible subgroup orders are restricted to $\{1, 2, q, 2q\}$, so random elements typically achieve large order and avoid short multiplicative cycles. In contrast, since P_1 is not assumed to be safe, $P_1 - 1$ may have additional factors and thus allow shorter multiplicative cycles for fixed-coefficient recurrences.

However, HardPoly1305 V2-Lite does not evolve with a fixed coefficient. Each block uses fresh (r_i, s_i) derived from *h_core*, so the state does not follow a single deterministic multiplicative orbit. In the ideal model, the state behaves like a randomized process rather than a periodic map, and the relevant security bounds are captured by the UF-CMA reduction rather than group order alone.

3.3 Symbol Conventions

We use the following conventions throughout:

- For a byte string S, $|S|$ is its length in bytes; $S[i]$ is the i-th byte (0-indexed); and $S[a..b]$ denotes the substring of indices a through b (inclusive).
- Concatenation of byte strings is denoted by $\|$.
- LE_to_int(B) interprets bytes B as a nonnegative integer in little-endian order; LE_encode$_n(x)$ encodes x mod 2^{8n} as n little-endian bytes. We write LE(B) as shorthand for LE_to_int(B).
- Trunc$_{128}(x)$ denotes the low 128 bits of x.
- We use $x \bmod p$ (or $x \mod p$ in algorithms) for modular reduction.
- Bitwise operations are over 256-bit words: $x \wedge y$ (AND), $x \vee y$ (OR), $x \oplus y$ (XOR), and $\neg x$ (NOT). We use the following 256-bit shift/rotate notations (all results are implicitly reduced to 256 bits):

$$x \ll_{256} r := (x \ll r) \bmod 2^{256}, \qquad x \gg_{256} r := \left\lfloor \frac{x}{2^r} \right\rfloor,$$

$$x \lll_{256} r := (x \ll_{256} r) \vee (x \gg_{256} (256-r)), \qquad x \ggg_{256} r := (x \gg_{256} r) \vee (x \ll_{256} (256-r)).$$

  (Here $\ll, \gg$ are zero-filled logical shifts on the 256-bit word.) Any bitwise intermediate is reduced to 256 bits with MASK$_{256}$.

- Constants:

$$\text{MASK}_{256} = 2^{256} - 1, \quad \text{CLAMP} = \texttt{0x0ffffffc0ffffffc0ffffffc0fffffff}.$$

- Fixed 256-bit mixing constants used in the concrete *h_core* implementation:

$$C_1 = \texttt{0xB7E151628AED2A6ABF7158809CF4F3C7}, \quad C_2 = \texttt{0x9E3779B97F4A7C15F39CC0605CEDC834}.$$

3.4 Comparison: Fixed vs. Per-Block Coefficients

3.4.1 Original Poly1305 Structure

For n blocks with fixed r:

$$h_1 = m_1 \cdot r \tag{3}$$
$$h_2 = (h_1 + m_2) \cdot r = m_1 \cdot r^2 + m_2 \cdot r \tag{4}$$
$$h_n = m_1 \cdot r^n + m_2 \cdot r^{n-1} + \cdots + m_n \cdot r \tag{5}$$

This is a **clean polynomial in** r with message blocks as coefficients.

3.4.2 HardPoly1305 Structure

With per-block (r_i, s_i):

$$h_1 = r_1 \cdot m_1 + s_1 \tag{6}$$
$$h_2 = r_2 \cdot (h_1 + m_2) + s_2 = r_2 r_1 m_1 + r_2(s_1 + m_2) + s_2 \tag{7}$$

The expression contains products like $r_2 r_1$, and each r_i is a complex function of (h_{i-1}, m_i, K). There is **no single variable** r to solve for algebraically.

4 Complete Mathematical Model

4.1 Key Derivation: `DeriveKeyParams`

Definition 4.1 (Key Parameter Derivation)**.** *Given master key K (at least 32 bytes), derive internal parameters:*

$$K' = \textit{FoldKey32}(K) \tag{8}$$
$$K_0 = K'[0..15], \quad K_1 = K'[16..31] \tag{9}$$
$$k_{high} = \textit{LE\_to\_int}(K_0) \tag{10}$$
$$k_{low} = \textit{LE\_to\_int}(K_1) \tag{11}$$
$$k_{mix} = \textit{LE\_to\_int}(K'[0..31]) \tag{12}$$
$$k_{mix2} = (\neg k_{mix}) \ggg_{256} 1 \tag{13}$$

4.2 Message-Key Mixing: `MixKeyAndMessage`

Definition 4.2 (Message Mixing)**.** *Given message M of length ℓ and key K of length $L \geq 32$:*

1. *Compute a padded message P:*

$$P := \begin{cases} M \,\|\, \texttt{0xA7} \,\|\, \texttt{0x00}^{(32-\ell-1)} & \text{if } \ell < 32, \\ M & \text{if } \ell \geq 32. \end{cases}$$

2. *Let $L_P := |P|$. Construct effective key bytes $K_{\textit{eff}}[0..L_P - 1]$ by:*

$$K_{\textit{eff}}[i] := K[i \bmod L] \quad \text{for } i = 0, \ldots, L_P - 1,$$

and for each tail index $j = L_P, \ldots, L - 1$ fold it onto the message domain:

$$K_{\textit{eff}}[j \bmod L_P] := K_{\textit{eff}}[j \bmod L_P] \oplus K[j].$$

3. *For $i = 0, \ldots, L_P - 1$: $\textit{mixed}[i] = (P[i] + K_{\textit{eff}}[i]) \mod 256$*

This provides a lightweight "salting" layer, ensuring the message entering *h_core* carries key-dependent structure.

4.3 Block Encoding

The mixed message is divided into 16-byte blocks. Each block B_i is encoded as:

$$m_i = \text{LE\_to\_int}(B_i \| \texttt{0x01})$$

Here LE_to_int$(\cdot)$ denotes little-endian byte-to-integer conversion (least-significant byte first). The trailing `0x01` prevents length-extension ambiguities (standard Poly1305 convention). If the final block has length < 16, no zero-padding is added beyond the trailing `0x01`.

4.4 Core Function: *h_core*

Definition 4.3 (*h_core* Function)**.** *Given hash state h, block value m, and key parameters $(k_{high}, k_{low}, k_{mix}, k_{mix2})$:*

Step 1: Polynomial layer in $\mathbb{Z}_{P_2}$ (two-field bridge)

$$X = (m + k_{high}) \bmod P_2 \tag{14}$$
$$Y = (h + k_{low}) \bmod P_2 \tag{15}$$
$$\alpha = \big((h + X)^2 + (m - Y)^3\big) \bmod P_2 \tag{16}$$
$$\beta = (X - Y) \bmod P_2 \tag{17}$$

Step 2: 256-bit diffusion and keyed mixing (bit domain)

$$a = \alpha \wedge MASK_{256}, \quad b = \beta \wedge MASK_{256} \tag{18}$$
$$a' = a \oplus \Big(\big((b \gg_{256} 17) \ \vee \ (a \ll_{256} 239)\big) \wedge MASK_{256}\Big) \tag{19}$$
$$b' = b \oplus \Big(\big((a \ll_{256} 17) \ \vee \ (b \gg_{256} 239)\big) \wedge MASK_{256}\Big) \tag{20}$$
$$k_1 = \big(k_{mix} \oplus C_1 \oplus (a' \ll_{256} 17)\big) \wedge MASK_{256} \tag{21}$$
$$k_2 = \big(k_{mix2} \oplus C_2 \oplus (b' \gg_{256} 239)\big) \wedge MASK_{256} \tag{22}$$
$$lin = (k_1 \lll_{256} 127) \ \oplus \ (k_2 \lll_{256} 63) \ \oplus \ (a' \lll_{256} 31) \ \oplus \ (b' \lll_{256} 15) \tag{23}$$
$$hard = \big(\neg(k_{mix} \oplus k_{mix2} \oplus a \oplus b)\big) \wedge MASK_{256} \tag{24}$$
$$temp = \big((h \oplus m) \wedge MASK_{256}\big) \ \oplus \ lin \ \oplus \ hard \tag{25}$$

Step 3: Return to $\mathbb{Z}_{P_2}$

$$u = (h + temp) \bmod P_2 \tag{26}$$

Output: u

4.5 Coefficient Extraction: `DeriveRS`

Definition 4.4 (Coefficient Derivation)**.** *Given $u \in \mathbb{Z}_{P_2}$:*

1. $u_{256} = u \wedge MASK_{256}$
2. *Convert to 32 bytes (little-endian):* $bytes[0..31] = LE\_encode_{32}(u_{256})$
3. $r_{raw} = LE\_to\_int(bytes[0..15])$
4. $r = r_{raw} \wedge CLAMP$
5. $s = LE\_to\_int(bytes[16..31]) \mod 2^{128}$

Output: (r, s)

4.6 Main Iteration

Definition 4.5 (HardPoly1305 V2-Lite MAC)**.** *1. Initialize:* $h_0 = 0$

2. For each block $i = 1, \ldots, n$*:*

$$u_i = h\_core(h_{i-1}, m_i, k_{high}, k_{low}, k_{mix}, k_{mix2}) \tag{27}$$

$$(r_i, s_i) = \texttt{DeriveRS}(u_i) \tag{28}$$

$$h_{temp} = (h_{i-1} + m_i) \mod P_1 \tag{29}$$

$$h_i = (r_i \cdot h_{temp} + s_i) \mod P_1 \tag{30}$$

3. Output: $tag = h_n \mod 2^{128}$ *(16 bytes, little-endian)*

4.7 System View as a Mapping Diagram (Two-Space Loop: $P_1 \leftrightarrow P_2$)

A compact "white-box" way to describe HardPoly1305 V2-Lite is as a per-block mapping loop between two spaces:

- the Poly1305 hash space $\mathbb{Z}_{P_1}$ (state h_i),
- the core mixing space $\mathbb{Z}_{P_2}$ (core output u_i),
- and an intermediate coefficient space (r_i, s_i) obtained by slicing a 256-bit word.

Per-block loop (arrow diagram). Let $X_i \in \mathbb{Z}_{P_1}$ denote the encoded block integer (after `0x01` appending). For each block, the computation can be drawn as an *arrow ring* (a loop of maps) between the P_1-state space and the P_2-core space:

$$\begin{array}{ccc} h_{i-1} \in \mathbb{Z}_{P_1} & \xrightarrow{\texttt{h\_core}(\cdot,\cdot;K_{\text{param}})} & u_i \in \mathbb{Z}_{P_2} \\ \uparrow \text{ state carry } h_i \mapsto h_i \text{ (as next } h_{i-1}\text{)} & & \downarrow G = \texttt{DeriveRS} \\ h_i \in \mathbb{Z}_{P_1} & \xleftarrow{\Phi_{X_i}} & (r_i, s_i) \in \mathcal{R}_{\text{clamp}} \times \{0,1\}^{128} \end{array}$$

where $\mathcal{R}_{\text{clamp}}$ is the Poly1305-clamped coefficient set (size 2^{106}), and

$$\Phi_{X_i}(h) := r_i \cdot (h + X_i) + s_i \pmod{P_1}.$$

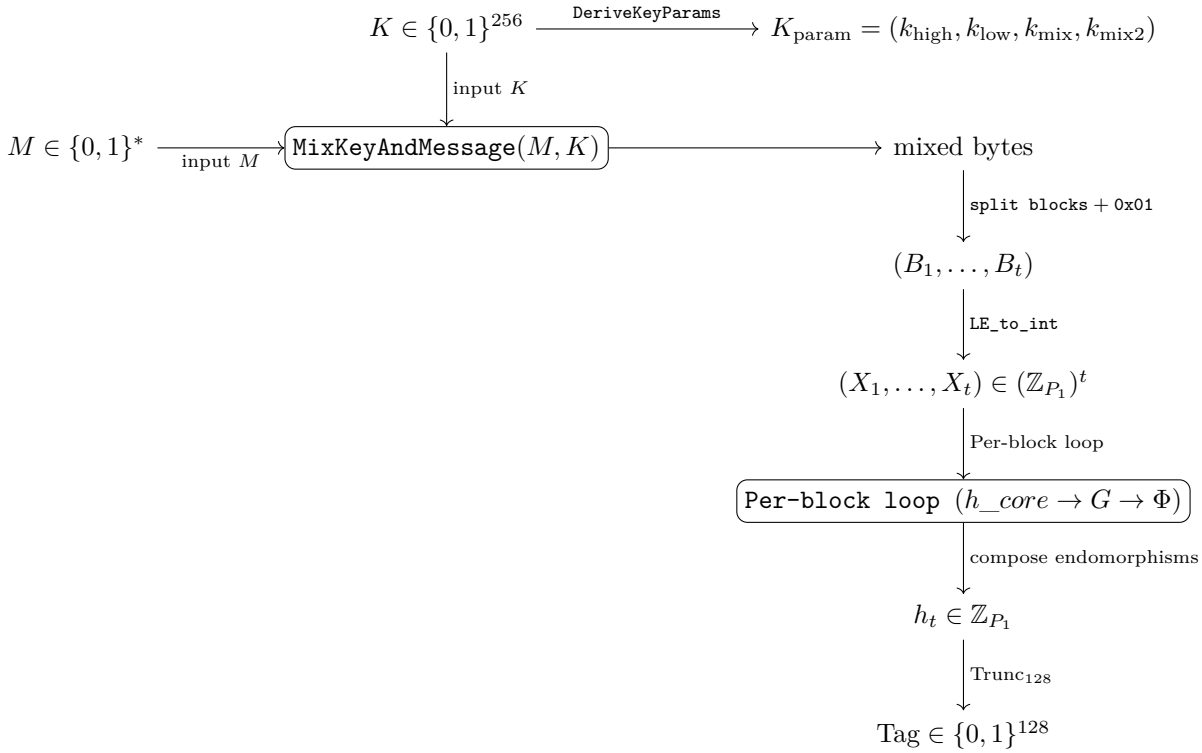
The three arrows above correspond exactly to:

$$u_i = h\_core(h_{i-1}, X_i; K_{\text{param}}) \in \mathbb{Z}_{P_2}, \quad (r_i, s_i) = G(u_i), \quad h_i = \Phi_{X_i}(h_{i-1}).$$

Over a message with t blocks, the overall map is the composition of endomorphisms on $\mathbb{Z}_{P_1}$:

$$h_t \;=\; \Phi_{X_t} \circ \cdots \circ \Phi_{X_1}(0).$$

Full pipeline (including mixing/encoding). The block values X_i themselves come from deterministic preprocessing, and the core parameters K_{param} are deterministically derived from the master key K:



Thus, the entire MAC is a deterministic keyed mapping

$$\text{Tag}_K : \{0,1\}^* \to \{0,1\}^{128}, \quad M \mapsto \text{Trunc}_{128}(h_t).$$

4.8 Parameterized Algebraic Attack Model for the Polynomial Layer (Adv_{alg})

This subsection sketches a *parameterized, chosen-input* white-box model for bounding the "algebraic" contribution Adv_{alg} in the advantage factoring

$$\text{Adv}_{\text{prf}}(h\_core) \;\le\; \text{Adv}_{\text{alg}} + \text{Adv}_{\text{bit}} + \text{negl}.$$

It does not claim a proven lower bound (standard-model lower bounds for these concrete systems are not known); rather it defines an explicit equation-system view whose parameters can be varied (query budget, input strategy, variable encoding), and whose solving cost can be compared against known generic techniques.

Chosen-input transcript and unknowns. In the PRF experiment, a PPT distinguisher may choose $(h_j, X_j) \in \mathbb{Z}_{P_1} \times \mathbb{Z}_{P_1}$ adaptively and receives $u_j \in \mathbb{Z}_{P_2}$. For a fixed key, the unknown secret material is the 256-bit master key (equivalently the derived parameters $(k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$). We separate the unknowns into:

- **Field offsets** $(k_{\text{high}}, k_{\text{low}})$ entering the degree-3 arithmetic layer that produces $(\alpha, \beta) \in \mathbb{Z}_{P_2}^2$.
- **Bit-layer key material** (derived from $k_{\text{mix}}, k_{\text{mix2}}$ and fixed constants C_1, C_2) used inside the 256-bit affine/linear mixing layer.

Equation form (mixed arithmetic/bit constraints). For each query j, define

$$X_{2,j} := (X_j + k_{\text{high}}) \bmod P_2, \qquad Y_{2,j} := (h_j + k_{\text{low}}) \bmod P_2,$$

$$\alpha_j := (h_j + X_{2,j})^2 + (X_j - Y_{2,j})^3 \pmod{P_2}, \qquad \beta_j := (X_{2,j} - Y_{2,j}) \pmod{P_2}.$$

Let $(a_j, b_j) \in \{0,1\}^{256} \times \{0,1\}^{256}$ be the 256-bit encodings of (α_j, β_j), and let

$$\text{temp}_j := f_{\text{bit}}^{(K)}(h_j, X_j, a_j, b_j) \in \{0,1\}^{256}$$

denote the concrete 256-bit mixing output under the fixed key-derived bit layer (cross-bit shifts/OR, fixed rotations, XOR/NOT, and fixed constants). The output equation is

$$u_j \equiv h_j + \mathrm{int}(\mathrm{temp}_j) \pmod{P_2}.$$

This is a concrete system of constraints over a mixture of $\mathbb{Z}_{P_2}$ arithmetic and 256-bit Boolean wiring. Turning it into a pure Boolean system introduces carry/borrow variables for additions/subtractions and auxiliary quotient bits for reduction modulo P_2.

System size parameters. Let q be the number of chosen-input queries. A typical Boolean encoding introduces:

- **Key variables:** 256 bits (master key), or fewer if working only with $(k_{\mathrm{high}}, k_{\mathrm{low}})$.
- **Per-query auxiliary variables:** $O(256)$ carry/borrow bits for each modular addition/subtraction instance (Lemma 5.13 and Lemma 5.14 characterize their nonlinearity/propagation).
- **Degree:** the arithmetic core is degree 3 over $\mathbb{Z}_{P_2}$ in the offsets; Booleanization yields a sparse system with mixed low-degree gates plus carry constraints whose algebraic degree grows with bit index (Lemma 5.13).

Thus, a rough parameterization is: number of Boolean variables $n \approx 256 + c \cdot 256q$ for a small constant c depending on the chosen encoding, and number of constraints $m = O(256q)$.

Generic solver cost baselines (heuristic). We record three standard baselines for $\mathrm{Adv}_{\mathrm{alg}}$:

- **Exhaustive key search:** time $\approx 2^{256}$, advantage essentially 1 (but not PPT).
- **Algebraic solving via Gröbner/XL (MQ view):** model the Booleanized system as a semi-regular system of degree at most D with n variables and m equations; generic costs scale super-polynomially in n and depend on the degree of regularity D_{reg} (heuristically increasing with n/m).
- **SAT/CP-SAT (bit-blasted view):** treat carry chains and modular reductions as CNF constraints; generic costs scale with the number of variables and the amount of propagation induced by carries/borrows (Lemma 5.14 gives a tail bound under uniform inputs, but chosen-input strategies may increase propagation).

Chosen-input amplification and what "parameterized" means. The model is intentionally parameterized by $(q, \mathcal{S})$ where q is the query budget and $\mathcal{S}$ is the input-selection strategy (random distinct vs structured probing). Increasing q increases the number of constraints linearly but also increases auxiliary variables; structured $\mathcal{S}$ can in principle force more carry/borrow propagation, changing solver behavior. Any concrete claim about $\mathrm{Adv}_{\mathrm{alg}}$ must therefore specify $(q, \mathcal{S})$ and the encoding/solver family used.

5 Conditional UF-CMA Security Discussion

Caveat. The results below are conditional and heuristic; they do not constitute a formal proof in the standard model for the concrete instantiation of *h_core*.

5.1 Notation and UF-CMA Experiment

We fix the following notation and formalize the UF-CMA interface:

- $P_1 = 2^{130} - 5$ is the Poly1305 modulus.
- $P_2 = 2^{256} - 188069$ is the second prime used as the working field for *h_core*.
- The master key K is a byte string of length $L \geq 32$ (in the UF-CMA experiment we take $L = 32$).

- Let $\ell = |M|$ and $L = |K|$. The mixing function $\text{Mixed}(M, K)$ is exactly Algorithm 3 (short-message domain-separated padding with `0xA7`, key-tail folding onto the message domain, then byte-wise modular addition).
- $\text{Mixed}(M, K)$ is partitioned into 16-byte blocks $M_1, \ldots, M_t$ and encoded as

$$X_i = \text{LE}(M_i \| \texttt{0x01}) \in \mathbb{Z}_{P_1}.$$

- The state update (in $\mathbb{Z}_{P_1}$) is

$$h_0 = 0, \quad u_i = h\_core(h_{i-1}, X_i; K_{\text{param}}), \quad (r_i, s_i) = G(u_i), \quad h_i = r_i \cdot (h_{i-1} + X_i) + s_i \pmod{P_1}.$$

- The final tag is

$$\text{Tag}_K(M) = \text{Trunc}_{128}(h_t),$$

  i.e., the low 128 bits of h_t.
- $K_{\text{param}} = (k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$ are internal parameters derived from K.
- G maps a 256-bit value u_i to (r_i, s_i) by Poly1305 clamping of the low 128 bits and direct assignment of the high 128 bits.

Definition 5.1 (Tag/Verify)**.** *Given a secret key K, $Tag_K(M)$ is the deterministic tagging algorithm defined above. The verifier computes $Verify_K(M, T) = 1$ if $T = Tag_K(M)$ (equality can be implemented with constant-time comparison).*

Definition 5.2 (UF-CMA Experiment)**.** *The experiment $Exp_{\mathcal{A}}^{uf\text{-}cma}$ proceeds as follows:*

1. *Sample $K \leftarrow \{0,1\}^{256}$ and set query set $Q \leftarrow \emptyset$.*
2. *Give $\mathcal{A}$ oracle access to $Tag_K(\cdot)$. On query M, return $Tag_K(M)$ and record M in Q.*
3. *$\mathcal{A}$ outputs (M^*, T^*) and wins if $M^* \notin Q$ and $Verify_K(M^*, T^*) = 1$.*

The advantage is $Adv_{uf\text{-}cma}(\mathcal{A}) = \Pr[win]$.

5.2 Security Model and Experiments

5.3 Plain-Language Scope Statement (What "Won't Die" Means Here)

This document's security statements are deliberately scoped. Informally:

- **Correct engineering use (PPT attackers).** If the key material is uniformly random/high-entropy, the implementation is used under the UF-CMA game model (PPT adversaries with feasible query budgets), and the surrounding protocol enforces anti-replay by authenticating $M' := N \parallel M$ with a nonce/counter N and a correct verifier freshness check (Subsection 5.10), then the scheme is modeled to "not die": its concrete failure probabilities are bounded by the UF-CMA advantage terms (Theorem 5.34 and subsequent bounds), plus the explicitly stated deviations $\text{Adv}_{\text{prf}}(\cdot)$ and ε_{der} for the concrete core/extraction.
- **Misuse (user/ops failures).** If a user supplies low-entropy keys, then brute-force key recovery becomes feasible and the security bound degrades accordingly. If the protocol omits nonce/counter freshness checks, then network-level replay succeeds with probability 1 by trivial retransmission (Subsection 5.10). If the verifier's anti-replay state can be lost/rolled back with probability p_{miss}, then replay acceptance has an irreducible risk term at least p_{miss}.
- **Unbounded attackers.** Against unbounded-time distinguishers, no 256-bit keyed concrete family can be information-theoretically indistinguishable from a random function once multiple samples are available (Theorem 5.4). In this sense, an "infinite compute" adversary can always win the statistical PRF distinguishing game with overwhelming advantage.

The purpose of the Tweaked V2 structure is therefore specific: it strengthens algorithm-level resistance to block splicing/cut-and-paste by making per-block coefficients state-dependent (Lemma 5.22), while the core PRF property remains an explicit computational assumption (Assumption 5.28).

Key separation model. For a cleaner reduction, we model the master key as $K = (K_{\text{mix}}, K_{\text{core}})$ produced by a KDF, where K_{mix} is used only in MixKeyAndMessage(M, K_{mix}) and K_{core} only in *h_core*. We assume K_{mix} and K_{core} are independent. Here K_{mix} is a byte string of length $L \geq 32$ and the mixing rule is exactly Algorithm 3 (domain-separated short-message padding plus key-tail folding onto the message domain, followed by byte-wise modular addition). The concrete single-key instantiation should be viewed as an idealized approximation to this model.

Single-user, deterministic MAC setting. We consider a single user with a fixed secret key K used across all tag queries. The tagging algorithm is deterministic *given its input string*. In protocol deployments, anti-replay is achieved by including a nonce/counter in the authenticated string (e.g., tagging $M' := N \parallel M$) and enforcing freshness at verification (Subsection 5.10); this does not change the determinism of $\text{Tag}_K(\cdot)$ as a function. The adversary has adaptive access to $\text{Tag}_K(\cdot)$ as in the UF-CMA experiment, and there are no side-channel leakages in the model.

Definition 5.3 (PPT adversary / PPT distinguisher)**.** *PPT is short for* Probabilistic Polynomial-Time. *A PPT adversary (or PPT distinguisher) is a randomized algorithm $\mathcal{A}$ such that, for a security parameter λ, its running time is bounded by* $\text{poly}(\lambda)$ *and (in oracle-based games) it makes at most* $\text{poly}(\lambda)$ *adaptive oracle queries.*

PRF experiment. A challenger samples a random bit b and a random key K_{core}. If $b = 0$ the oracle computes $h\_core(\cdot, \cdot; K_{\text{core}})$; if $b = 1$ the oracle is a truly random function ρ on the same domain whose outputs are uniform in $\mathbb{Z}_{P_2}$. The domain is pairs (h, X) with $h \in \mathbb{Z}_{P_1}$ and X derived from Mixed(M, K_{mix}) and $0x01$ block encoding. The distinguisher outputs b', and its advantage is $\text{Adv}_{\text{prf}} = |\Pr[b' = b] - \frac{1}{2}|$.

Why the range matters (purely statistical distinguisher if mis-modeled). The concrete *h_core* returns elements of $\mathbb{Z}_{P_2}$, i.e., it never outputs integers $\geq P_2$. If the "random function" branch were instead modeled as outputs uniform in $\{0,1\}^{256}$, then there is an explicit 1-query statistical distinguisher: query once, output "random" iff the response is $\geq P_2$. Its distinguishing advantage would be exactly

$$\Pr[U \geq P_2] - 0 \;=\; \frac{2^{256} - P_2}{2^{256}} \;=\; \frac{188069}{2^{256}},$$

where $U \leftarrow \{0,1\}^{256}$ is uniform. This is negligible, but it is nonzero and is avoided by modeling the ideal range as $\mathbb{Z}_{P_2}$.

5.4 Why a "Pure-Math" PRF Proof for a Concrete 256-bit Keyed Core is Impossible (Information-Theoretic)

The term "PRF" is fundamentally a *computational* notion: it asks that no *efficient* distinguisher can tell F_K from a truly random function. If one instead asks for *statistical* (information-theoretic) indistinguishability under uniform sampling, then no fixed k-bit keyed family can meet that goal once an adversary is allowed even a small number of samples. The following counting lemma is standard and requires no unproven assumptions.

Theorem 5.4 (Counting Impossibility of Statistical PRF for k-bit Keyed Families)**.** *Let $\mathcal{F} = \{F_K : \mathcal{D} \to \mathcal{R}\}$ be any family indexed by a key $K \in \{0,1\}^k$. Consider the following* uniform-sampling *distinguishing experiment with q samples:*

- *Choose $b \leftarrow \{0,1\}$.*
- *If $b = 0$, choose $K \leftarrow \{0,1\}^k$ and set $f := F_K$.*
- *If $b = 1$, choose $f := \rho$ where ρ is a truly random function $\mathcal{D} \to \mathcal{R}$.*
- *Sample q* distinct *inputs $x_1, \ldots, x_q \leftarrow \mathcal{D}$ uniformly without replacement and give the distinguisher the labeled samples $(x_i, f(x_i))_{i=1}^q$.*

Then there exists an (unbounded-time) distinguisher $\mathcal{A}$ such that

$$Adv(\mathcal{A}) \;\geq\; 1 - \frac{2^k}{|\mathcal{R}|^q}.$$

In particular, if $|\mathcal{R}| = 2^{256}$ (or essentially $P_2 \approx 2^{256}$) and $k = 256$, then already at $q = 2$ the advantage lower bound is $1 - 2^{-256}$ (overwhelming).

Proof. Define the distinguisher $\mathcal{A}$: on input $(x_i, y_i)_{i=1}^q$, output 0 iff there exists a key K such that for all i, $F_K(x_i) = y_i$; otherwise output 1.

If $b = 0$, the samples are generated by some F_K, so such a key exists and $\mathcal{A}$ outputs 0 with probability 1.

If $b = 1$, fix any particular key K. For uniformly random labels $y_1, \ldots, y_q \leftarrow \mathcal{R}$ (as induced by a truly random function on distinct points), the probability that *all* q equalities $F_K(x_i) = y_i$ hold is exactly $|\mathcal{R}|^{-q}$. By a union bound over all 2^k keys,

$$\Pr[\exists K : \forall i,\; F_K(x_i) = y_i] \leq \frac{2^k}{|\mathcal{R}|^q}.$$

Therefore $\Pr[\mathcal{A} \text{ outputs } 0 \mid b = 1] \leq 2^k/|\mathcal{R}|^q$, so $\Pr[\mathcal{A} \text{ outputs } 1 \mid b = 1] \geq 1 - 2^k/|\mathcal{R}|^q$. Combining the two cases yields the claimed advantage bound. □

Consequence for this document. A "pure math" proof that the *concrete h_core* is indistinguishable from a random function (even under uniform sampling) is impossible in an information-theoretic sense, simply because *h_core* is a 256-bit keyed family. Any meaningful PRF claim must therefore be *computational* (restricting $\mathcal{A}$ to efficient algorithms) and, for a concrete design like *h_core*, must remain an *assumption* unless reduced to a widely-studied hardness assumption (which we do not have here).

Block budget. Let B_{total} be the total number of 16-byte blocks processed across all tag queries after mixing (including the short-message delimiter padding and any key-tail folding onto the message domain, as defined by Algorithm 3) and $0x01$ block encoding. For convenience we also charge the blocks processed when recomputing the final verification to B_{total}; each block induces one oracle call to *h_core* in the real MAC.

Proof outline. We analyze the scheme via successive hybrids defined later: first replace *h_core* with a random function, then idealize coefficient sampling, and finally analyze the resulting random-coefficient polynomial MAC.

5.5 Poly1305 Algebraic Form vs. Security Basis

Poly1305 has a polynomial algebraic form over $\mathbb{Z}_{P_1}$ with a *fixed* coefficient r across all blocks, but its security is *not* based on MQ (multivariate polynomial) hardness. Instead, the classic guarantee is information-theoretic in the one-time-key setting (Wegman–Carter style), as summarized in RFC 7539. Our construction keeps the polynomial-style update but replaces the fixed coefficient (r, s) with per-block coefficients derived from (h_{i-1}, X_i) and includes key-dependent mixing before block encoding; therefore the one-time Poly1305 bound does not directly carry over and is used only as a comparison baseline.

Lemma 5.5 (Poly1305 One-Time Bound (RFC 7539 Summary))**.** *Let r be uniformly random over the Poly1305-clamped space and s be uniform in $\{0,1\}^{128}$. Let t be the number of 16-byte blocks (after the standard `0x01` encoding) in a message. For any two distinct messages with at most t blocks, the collision probability of the Poly1305 polynomial hash*

$$h_t = r^t X_1 + r^{t-1} X_2 + \cdots + r X_t$$

over $\mathbb{Z}_{P_1}$ is at most t/P_1. Consequently, for a single tagging query, the forging probability is at most $t/P_1 + 2^{-128}$ (the extra term comes from truncation to 128 bits).

Note. This bound is one-time (or per-nonce) and does not apply if a fixed (r, s) is reused across many messages.

5.6 Layered Decomposition of *h_core* (Step-by-step Algebraic Attack Surface)

This subsection deliberately "opens up" *h_core* step by step, with the goal of making the *algebraic* attack surface explicit. Rather than treating the core as a black box, we decompose it into layers, write down the exact equations, and point out which parts are polynomial over a prime field, which parts are affine/linear over GF(2) once the key is fixed, and where the unavoidable Boolean nonlinearity (carry/borrow) enters when arithmetic is viewed at the bit level. The concrete implementation referenced here is the Python code in `HardPoly1305 V2 (Modified Twilight-Dream).py` and the formal definition of *h_core* given earlier in this document.

Layer 1–4 skeleton (implementation + algebraic modeling view). Here h denotes the current state and X denotes the current block input (after mixing/encoding).

- **Layer 1 (polynomial over $\mathbb{Z}_{P_2}$):**

$$X_2 = (X + k_{\text{high}}) \bmod P_2, \quad Y_2 = (h + k_{\text{low}}) \bmod P_2,$$

$$\alpha = (h + X_2)^2 + (X - Y_2)^3 \pmod{P_2}, \qquad \beta = (X_2 - Y_2) \pmod{P_2}.$$

- **Layer 2 (256-bit diffusion/mixing in the bit domain):** interpret (α, β) as 256-bit words (a, b) (since $P_2 < 2^{256}$), apply cross-bit shifts/OR and XOR to obtain (a', b'), then mix key material and constants and apply fixed rotations and XOR/NOT to form a 256-bit word temp.

- **Layer 3 (return to $\mathbb{Z}_{P_2}$):** output

$$u = (h + \text{temp}) \bmod P_2.$$

- **Layer 4 (Booleanization view: carry/borrow & wrap):** when the arithmetic in Layers 1 and 3 is expressed as a Boolean constraint system over bits, modular additions/subtractions and the conditional "wrap" (subtracting P_2 on overflow) introduce carry/borrow chains. This is where Boolean nonlinearity enters even if Layer 2 is affine/linear over GF(2).

Important scope note. We do *not* claim an information-theoretic PRF (impossible by Theorem 5.4). Any "PPT security" statement about the concrete *h_core* remains computational and is captured precisely by Assumption 5.28 used later in the UF-CMA reduction.

Domains and notation. Fix $P_1 = 2^{130} - 5$ and $P_2 = 2^{256} - 188069$. The core is keyed by $(k_{\text{high}}, k_{\text{low}}) \in \mathbb{Z}_{P_1}^2$ and $(k_{\text{mix}}, k_{\text{mix2}}) \in \{0,1\}^{256} \times \{0,1\}^{256}$ derived from the master key. In the reference implementation (Algorithm 1), the master key may be any byte string of length $|K| \geq 32$, but parameter derivation first normalizes it to a canonical 32-byte key $K' := \textsc{FoldKey32}(K)$. This canonical key K' is parsed as two 128-bit little-endian integers $U_{\text{high}}, U_{\text{low}} \in \{0, \ldots, 2^{128} - 1\}$ and used directly as integers (the implementation masks to 256 bits, which does not change 128-bit values). Equivalently, $(k_{\text{high}}, k_{\text{low}})$ is uniform over the *embedded 128-bit square* $\{0, \ldots, 2^{128} - 1\}^2 \subset \mathbb{Z}_{P_1}^2$ whenever K' is uniform over 32-byte strings, rather than uniform over all of $\mathbb{Z}_{P_1}^2$. Inputs are (h, X) where $h \in \mathbb{Z}_{P_1}$ and X is the encoded block integer (after `0x01` appending) reduced as needed.

Layer 1 (field polynomial map in $\mathbb{Z}_{P_2}$). Define

$$X_2 := (X + k_{\text{high}}) \bmod P_2, \qquad Y_2 := (h + k_{\text{low}}) \bmod P_2,$$

and

$$\alpha := (h + X_2)^2 + (X - Y_2)^3 \pmod{P_2}, \qquad \beta := (X_2 - Y_2) \pmod{P_2}.$$

This layer is exactly what you described: it is *modular addition/subtraction* in a safe-prime field together with a low-degree polynomial. More explicitly:

- The "wiring" that injects the offsets is $(X + k_{\text{high}}) \bmod P_2$ and $(h + k_{\text{low}}) \bmod P_2$ (modular additions in $\mathbb{Z}_{P_2}$).

- β is just one modular subtraction $(X_2 - Y_2) \bmod P_2$.
- α is a degree-3 polynomial expression combining the current input X, the previous state h, and the two offsets via one square and one cube over $\mathbb{Z}_{P_2}$.

As a function of the secret offsets, (α, β) is low-degree over the prime field $\mathbb{Z}_{P_2}$, but it is evaluated on state-dependent public inputs (h, X) that vary across queries/blocks.

Algebraic baseline with explicit success probabilities. To make the algebraic-security discussion quantitative, we separate a *leakage* thought experiment (where α is observable) from the real core (where it is hidden by Layers 2–3).

Leakage game (Layer 1 visible). Fix $q \geq 1$ chosen input pairs $(h_i, X_i) \in \mathbb{Z}_{P_1} \times \mathbb{Z}_{P_1}$. In the leakage thought experiment, the adversary receives $(h_i, X_i, \alpha_i)_{i=1}^{q}$ where $\alpha_i = \alpha(h_i, X_i; k_{\text{high}}, k_{\text{low}})$ from Layer 1, and outputs a guess $(\hat{k}_{\text{high}}, \hat{k}_{\text{low}})$ for the 128-bit offsets.

Lemma 5.6 (Offset-guessing upper bound in the Layer 1 leakage game)**.** *Assume k_{high} is uniform in $\{0, \ldots, 2^{128} - 1\}$. Let an adversary test at most N candidate values of k_{high} (with any strategy, adaptive or not). Then its success probability satisfies*

$$\Pr[(\hat{k}_{high}, \hat{k}_{low}) = (k_{high}, k_{low})] \;\leq\; \frac{N}{2^{128}} \;+\; 3N \cdot \left(\frac{3}{2^{128}}\right)^{q-1}.$$

Proof sketch. The first term is the probability that one of the N tested candidates equals the true k_{high}. For any tested $\hat{k}_{\text{high}} \neq k_{\text{high}}$, the first sample (h_1, X_1, α_1) yields at most 3 candidate values of $\hat{k}_{\text{low}}$ in the 128-bit range (because a cubic equation over a prime field has at most 3 roots). For each such candidate pair, Lemma 5.7 bounds the probability that it matches an additional independently-keyed sample by at most $3/2^{128}$; applying this to the remaining $q-1$ samples and taking a union bound over at most $3N$ candidates yields the second term. □

Interpretation. For $q = 1$, the bound reduces to $\Pr[\text{success}] \leq N/2^{128} + 3N$ (trivial, and not tight). For any fixed $q \geq 2$, the false-positive term collapses quickly; the dominant term becomes $N/2^{128}$. Equivalently, a brute-force attacker performing $N = 2^b$ trials has success probability at most about 2^{b-128} in this leakage setting.

Real core (Layer 1 hidden). In the concrete *h_core*, the attacker does *not* observe (α, β); it only observes u after Layer 2 (keyed affine/linear 256-bit mixing) and Layer 3 (modular addition). Therefore, any real algebraic attack must solve the full mixed arithmetic/bit constraint system described later, not the leakage game. In this document we use Lemma 5.6 only as a *ceiling baseline* for what would be possible if Layer 1 were directly exposed; the concrete construction is designed specifically to avoid exposing Layer 1 values in that form.

Lemma 5.7 (Key-to-α preimage bound (unconditional, for the implemented 128-bit offsets))**.** *Fix any inputs $(H_1, X_1) \in \mathbb{Z}_{P_1}^2$ and define α as above. Assume the canonical 32-byte key $K' = \textsc{FoldKey32}(K)$ is uniformly random, so that k_{high} and k_{low} are independent and uniform in $\{0, \ldots, 2^{128} - 1\}$ (the two 16-byte halves of K'). Then for any $y \in \mathbb{Z}_{P_2}$,*

$$\Pr[\alpha = y] \leq \frac{3}{2^{128}}.$$

Equivalently, the conditional min-entropy satisfies

$$H_\infty(\alpha \mid H_1, X_1) \geq 128 - \log_2 3.$$

Proof. Fix (H_1, X_1) and a target $y \in \mathbb{Z}_{P_2}$. For each choice of $k_{\text{high}} \in \mathbb{Z}_{P_1}$, define

$$r(k_{\text{high}}) := y - (H_1 + X_1 + k_{\text{high}})^2 \pmod{P_2}.$$

The equation $\alpha = y$ is equivalent to

$$(X_1 - H_1 + k_{\text{low}})^3 \equiv r(k_{\text{high}}) \pmod{P_2}.$$

Over the prime field $\mathbb{Z}_{P_2}$, a nonzero cubic equation has at most 3 solutions for k_{low}. Restricting k_{low} to the subset $\mathbb{Z}_{P_1} \subset \mathbb{Z}_{P_2}$ cannot increase the number of solutions, so for each fixed k_{high} there are at most 3 valid $k_{\text{low}} \in \mathbb{Z}_{P_1}$. In the implementation, k_{low} is uniform over the 128-bit subset $\{0, \ldots, 2^{128}-1\} \subset \mathbb{Z}_{P_1}$, so conditioned on any fixed k_{high} we have

$$\Pr[\alpha = y \mid k_{\text{high}}] \leq \frac{3}{2^{128}}.$$

Taking expectation over k_{high} yields $\Pr[\alpha = y] \leq 3/2^{128}$. □

Layer 2 (bit representation and affine/linear 256-bit mixing in $\mathrm{GF}(2)^{256}$). Since $\alpha, \beta \in \mathbb{Z}_{P_2}$ are always represented as integers in $[0, P_2-1]$ and $P_2 < 2^{256}$, the implementation lines `alpha_bits = alpha & MASK_256` and `beta_bits = beta & MASK_256` are exactly the identity on this representation. We therefore write $a, b \in \{0,1\}^{256}$ for the 256-bit encodings of (α, β), viewed as vectors in $\mathrm{GF}(2)^{256}$.

The concrete bit layer then computes a', b' by cross-bit shifts/OR and XOR, mixes in fixed constants C_1, C_2 and key-derived values $(k_{\text{mix}}, k_{\text{mix2}})$, applies several fixed rotations, and finally XORs a complemented linear expression: all of these operations are affine/linear over $\mathrm{GF}(2)$ on 256-bit words (the only Boolean nonlinearity in the overall core comes from the arithmetic layer via carries/borrows and modular reduction in $\mathbb{Z}_{P_2}$).

Thus, the bit layer is best viewed as a keyed affine/linear filter applied to the 256-bit words (a, b) together with the public inputs (h, X). From an algebraic-analysis perspective, this layer does not "raise" the algebraic degree in the Boolean sense (it is wiring + XOR/NOT + fixed rotations); the hard part is therefore to understand how the *field* polynomial layer feeds these 256-bit words and how the final modulo-P_2 addition introduces carry/wrap behavior at the bit level.

Strict algebraic form of Layer 2 (no omissions). We now record the exact Layer 2 computation as an affine map over $\mathrm{GF}(2)^{256}$, matching the implementation.

Lemma 5.8 (Disjoint-support OR equals XOR)**.** *For 256-bit words x, y, if $x \wedge y = 0$ (no overlapping 1-bits), then $x \vee y = x \oplus y$.*

Proof. Bitwise, for each position i, if $(x_i, y_i) \neq (1,1)$ then $x_i \vee y_i = x_i \oplus y_i$. The condition $x \wedge y = 0$ excludes the $(1,1)$ case at every bit, hence the equality holds for the whole word. □

In our construction the "OR" sites are intentionally between *disjoint* shifted ranges: $(b \gg_{256} 17)$ occupies bits 0..238 while $(a \ll_{256} 239)$ occupies bits 239..255; similarly $(a \ll_{256} 17)$ occupies bits 17..255 while $(b \gg_{256} 239)$ occupies bits 0..16. Therefore those ORs can be treated as XORs without introducing any $\wedge$ nonlinearity.

Define the intermediate words exactly as in the code (all operations over $\mathrm{GF}(2)^{256}$, with implicit masking to 256 bits):

$$a' := a \oplus (b \gg_{256} 17) \oplus (a \ll_{256} 239), \tag{31}$$
$$b' := b \oplus (a \ll_{256} 17) \oplus (b \gg_{256} 239), \tag{32}$$
$$k_1 := k_{\text{mix}} \oplus C_1 \oplus (a' \ll_{256} 17), \tag{33}$$
$$k_2 := k_{\text{mix2}} \oplus C_2 \oplus (b' \gg_{256} 239), \tag{34}$$
$$\text{lin} := (k_1 \lll_{256} 127) \oplus (k_2 \lll_{256} 63) \oplus (a' \lll_{256} 31) \oplus (b' \lll_{256} 15), \tag{35}$$
$$\text{hard} := \neg(k_{\text{mix}} \oplus k_{\text{mix2}} \oplus a \oplus b), \tag{36}$$
$$\text{temp} := (h \oplus X) \oplus \text{lin} \oplus \text{hard}. \tag{37}$$

Lemma 5.9 (Layer 2 is affine over $\mathrm{GF}(2)^{256}$ for fixed key)**.** *Fix (k_{mix}, k_{mix2}) and constants (C_1, C_2). Then the mapping*

$$(h, X, a, b) \mapsto \mathit{temp}$$

is an affine function over $\mathrm{GF}(2)$ on 256-bit words. Equivalently, there exists a linear map L and a constant word c_K (depending only on the fixed key and constants) such that

$$\mathit{temp} = L(h, X, a, b) \ \oplus \ c_K.$$

Proof. Each of $\ll_{256}, \gg_{256}, \lll_{256}$ is a linear operator on $\mathrm{GF}(2)^{256}$ (it is a fixed permutation/shift of bit coordinates with zero-fill), and $\oplus$ is addition in $\mathrm{GF}(2)$. The NOT operation $\neg z$ equals $z \oplus \mathbf{1}$ where $\mathbf{1}$ is the all-ones constant word, hence it is affine. By Lemma 5.8, the intended OR sites reduce to XOR and do not introduce multiplication terms. Therefore temp is a composition of linear operations plus addition of constants, i.e., an affine map. □

Remark (what is and is not "proven" here). The layer decomposition and the unconditional lemmas in this subsection are best viewed as a structured "attack surface map" for a PPT adversary: they identify which parts are merely affine/linear in the bit domain, where carries/borrows can introduce Boolean nonlinearity, and where modulo-P_2 range effects can leak (or fail to leak) information about the 256-bit addend. The concrete PRF claim against adaptive PPT distinguishers is stated separately as Assumption 5.28 (an explicit computational axiom about the concrete core) and is the sole cryptographic premise used later to reduce UF-CMA security of the full MAC to the PRF quality of *h_core*.

Layer 3 (return to $\mathbb{Z}_{P_2}$ by modular addition). Finally,

$$h_2 := h \bmod P_2, \qquad u := (h_2 + \mathrm{int}(t_2)) \bmod P_2,$$

where $\mathrm{int}(t_2)$ denotes interpreting the 256-bit vector as an integer in $[0, 2^{256} - 1]$. This step is a group operation in $\mathbb{Z}_{P_2}$ (a bijection for fixed $\mathrm{int}(t_2)$), but from a bit-level viewpoint it introduces a conditional subtraction by P_2 when the integer sum exceeds $P_2 - 1$.

Strict algebraic form of Layer 3 (no omissions). Because $P_1 < P_2$, the state $h \in \mathbb{Z}_{P_1}$ satisfies $h < P_2$ and hence $h_2 = h$ in the concrete MAC. Let $t := \mathrm{int}(\mathrm{temp}) \in [0, 2^{256} - 1]$ and define the integer sum $s := h + t$. Then the modulo-P_2 reduction can be written exactly as

$$u \;=\; s \;-\; P_2 \cdot \mathbf{1}[s \geq P_2],$$

where $\mathbf{1}[\cdot] \in \{0, 1\}$ is the indicator function. Equivalently,

$$u = \begin{cases} h + t & \text{if } t < P_2 - h, \\ h + t - P_2 & \text{if } t \geq P_2 - h. \end{cases}$$

Lemma 5.10 (Permutation property of modular addition)**.** *For any fixed $h \in \mathbb{Z}_{P_2}$, the map $t \mapsto (h + t) \bmod P_2$ is a bijection on $\mathbb{Z}_{P_2}$. For any fixed $t \in \mathbb{Z}_{P_2}$, the map $h \mapsto (h + t) \bmod P_2$ is a bijection on $\mathbb{Z}_{P_2}$.*

Proof. These are translations in the additive group of $\mathbb{Z}_{P_2}$. □

Corollary 5.11 (Uniformity under a uniform addend)**.** *If t is uniform over $\mathbb{Z}_{P_2}$ and independent of h, then $u = (h + t) \bmod P_2$ is uniform over $\mathbb{Z}_{P_2}$ for any fixed h.*

Lemma 5.12 (Wrap probability under a uniform addend)**.** *Fix any $h \in \{0, \ldots, P_2 - 1\}$ and let t be uniform over $\{0, \ldots, P_2 - 1\}$. Then the probability that the reduction performs the conditional subtraction (the "wrap" event) is*

$$\Pr[h + t \geq P_2] \;=\; \frac{h}{P_2}.$$

Proof. The event $h + t \geq P_2$ is equivalent to $t \geq P_2 - h$. There are exactly h values of t in $\{0, \ldots, P_2 - 1\}$ satisfying this inequality, hence the probability is h/P_2. □

What this decomposition buys for algebraic analysis. The decomposition above allows an algebraic attack discussion to be organized by concrete attack surfaces:

- **Equation-system view (Layer 1 → Layer 2).** With chosen inputs (h, X) and observed outputs u, the attacker can write the relation

$$u \equiv h + \mathrm{int}\Big(f_{\mathrm{bit}}^{(K)}(h, X, \alpha(h, X; k_{\mathrm{high}}, k_{\mathrm{low}}), \beta(h, X; k_{\mathrm{high}}, k_{\mathrm{low}}))\Big) \pmod{P_2},$$

where $(k_{\text{high}}, k_{\text{low}})$ are the primary unknown offsets inside the degree-3 field layer and $f_{\text{bit}}^{(K)}$ is the fixed keyed 256-bit affine/linear map (cross-bit shifts/OR wiring, fixed rotations, XOR/NOT, and fixed constants). This is a concrete mixed arithmetic/bit constraint system once expanded.

- **Affine/linear bit layer (Layer 2).** Once (a, b) are fixed, the mapping $(h, X, a, b) \mapsto \text{temp}$ is affine/linear over $\text{GF}(2)^{256}$ for a fixed key. Hence, for "algebraic hardness" one should not attribute confusion to this layer alone; it mainly serves as a fast diffusion/filter stage whose outputs are then added modulo P_2.
- **Carry/wrap effects in the final addition (Layer 3).** The modular addition in $\mathbb{Z}_{P_2}$ is translation-invariant as a group operation, but its *bit-level* behavior includes conditional wrap (subtracting P_2) when the integer sum crosses the range boundary. Any exploitable structure must therefore come from how the addend is distributed as produced by Layers 1–2, not from the group operation itself. The only unavoidable statistical bookkeeping issue is the tiny range gap between uniform sampling in $\{0,1\}^{256}$ and uniform sampling in $\mathbb{Z}_{P_2}$, quantified by $\varepsilon_{\text{range}} = 188069/2^{256}$ (Lemma 5.18).

5.6.1 Bit-level nonlinearity of modular addition/subtraction (carry/borrow chains)

Even if Layer 2 and (as a group operation) Layer 3 are affine/linear over GF(2), the arithmetic inside Layer 1 and the modulo-P_2 wrap in Layer 3 become *non-linear Boolean systems* once expanded into bit constraints. The core reason is the carry/borrow mechanism: it is a *state machine with feedback*, and its transition function contains products ($\wedge$), so it cannot be modeled as a linear state machine over GF(2).

Carry as a (nonlinear) state machine. Consider n-bit addition $z = x + y \bmod 2^n$ with ripple carry. Let the state be the carry bit $c_i \in \{0,1\}$ entering position i (with $c_0 = 0$). Then each step consumes the input bits (x_i, y_i) and updates the state:

$$z_i = x_i \oplus y_i \oplus c_i, \qquad c_{i+1} = (x_i \wedge y_i) \ \oplus \ (x_i \wedge c_i) \ \oplus \ (y_i \wedge c_i).$$

This is a 2-state automaton with *feedback*: the next state c_{i+1} depends on the previous state c_i *multiplicatively* (via $\wedge$), which is exactly where nonlinearity enters.

Equivalent propagate/generate form (useful for both linear and differential views). Define

$$p_i := x_i \oplus y_i \quad \text{(propagate)}, \qquad g_i := x_i \wedge y_i \quad \text{(generate)}.$$

Then the carry recursion is the standard form

$$c_{i+1} = g_i \oplus (p_i \wedge c_i).$$

The term $(p_i \wedge c_i)$ is the "loop": the state c_i is fed back into the transition and gated by p_i.

Why a linear state machine model cannot work. Suppose one tries to model the carry transition as a linear (affine) state machine over GF(2):

$$c_{i+1} \stackrel{?}{=} \alpha c_i \oplus \beta x_i \oplus \gamma y_i \oplus \delta \qquad (\alpha, \beta, \gamma, \delta \in \{0,1\}).$$

Evaluating on four inputs gives an immediate contradiction: $(x_i, y_i, c_i) = (0,0,0)$ implies $\delta = 0$; $(1,0,0)$ implies $\beta = 0$; $(0,1,0)$ implies $\gamma = 0$; $(0,0,1)$ implies $\alpha = 0$. But then the linear model forces $c_{i+1} = 0$ for all inputs, contradicting the true transition where $(x_i, y_i, c_i) = (1,1,0)$ yields $c_{i+1} = 1$. Hence, even at a single bit position, the carry update cannot be represented as a linear/affine state machine. This is the formal sense in which "treating it as linear" is dead on arrival.

Differential state machine (why XOR-differences do not linearize carries). To analyze XOR-differences, consider two executions with inputs (x, y) and $(x^\star, y^\star) = (x \oplus \Delta x,\ y \oplus \Delta y)$, producing carries (c_i) and $(c_i^\star)$. Define the *differential state* as the pair $(c_i, c_i^\star) \in \{0,1\}^2$ (a 4-state automaton), or equivalently $\Delta c_i := c_i \oplus c_i^\star$. Then

$$\Delta c_{i+1} = \big(g_i \oplus (p_i \wedge c_i)\big) \ \oplus \ \big(g_i^\star \oplus (p_i^\star \wedge c_i^\star)\big),$$

where (p_i, g_i) come from (x_i, y_i) and $(p_i^\star, g_i^\star)$ from $(x_i^\star, y_i^\star)$. Crucially, Δc_{i+1} depends not only on $(\Delta x_i, \Delta y_i, \Delta c_i)$ but also on the *base bits* (x_i, y_i) through p_i, g_i. This basepoint dependence is exactly why a "linear difference propagation" model fails: the difference dynamics are not a fixed linear transform; they are a nonlinear automaton whose transition varies with the underlying bits. The same phenomenon holds for borrow chains in subtraction (replace generate/propagate accordingly).

Lemma 5.13 (Algebraic degree of n-bit addition)**.** *Let $x, y \in \{0,1\}^n$ and let $z = x + y \bmod 2^n$ be n-bit addition (with carries) expressed in binary. Then the i-th output bit z_i (0-indexed from LSB) is a Boolean polynomial over* GF(2) *in the input bits of x, y of algebraic degree at most $i+1$. Moreover, for each $i \geq 1$ this degree bound is tight (there exist inputs for which z_i contains a monomial of degree $i+1$).*

Proof. Using the carry state machine above, each transition computes $c_{i+1} = g_i \oplus (p_i \wedge c_i)$. Over GF(2), $\wedge$ is multiplication and $\oplus$ is addition. By induction, $\deg(c_i) \leq i$ (each step can multiply by at most one fresh propagate bit), hence $\deg(z_i) = \deg(x_i \oplus y_i \oplus c_i) \leq i+1$. Tightness follows by choosing a full propagate chain so that c_i contains a degree-i monomial and thus z_i contains degree $i+1$. □

Lemma 5.14 (Carry/borrow chain tail bound under uniform inputs)**.** *Consider adding two independent uniform n-bit integers X, Y (mod 2^n) with ripple carries. Let L be the length of the carry-propagation chain starting at a given bit position (the number of consecutive positions over which the carry depends on a run of propagate conditions). Then $\Pr[L \geq \ell] \leq 2^{-\ell}$ for $\ell \geq 1$ (and similarly for borrow chains in subtraction).*

Proof. A carry propagates through bit position i only if (X_i, Y_i) is in the propagate set $\{(0,1), (1,0)\}$, which has probability $1/2$ under uniform independent bits. For the chain to have length at least ℓ, ℓ consecutive propagate events must occur, giving probability at most $(1/2)^\ell$ by independence. The borrow case is analogous. □

Why this matters for "white-box" reasoning. Lemma 5.13 formalizes that addition/subtraction is non-linear as a Boolean map due to carries/borrows, and Lemma 5.14 quantifies how far these non-linear dependencies typically propagate under uniform inputs. These are structure facts; they are not, by themselves, a PRF proof for the concrete core.

Relating this to Layer 4 (wrap under mod P_2). In addition to ordinary n-bit addition carries, the output step $u = (h + \text{temp}) \bmod P_2$ includes a conditional "wrap": compute the integer sum $s = h + \text{temp}$ and subtract P_2 iff $s \geq P_2$. When expressed as Boolean constraints, this introduces (i) a comparison circuit and (ii) a conditional subtract (borrow chain). Hence, even if Layer 2 is affine/linear over GF(2), the full core viewed as a Boolean system necessarily contains carry/borrow nonlinearity from Layer 1 arithmetic and from the modulo-P_2 wrap in Layer 3. This is exactly the sense in which we count a distinct "Layer 4" in the modeling view: it is not a new operation in code, but the Booleanization lens that exposes where algebraic degree and constraint density arise.

5.6.2 Adversarial (non-uniform) inputs and the correct security notion

The statement "even if a user deliberately feeds *h_core* a non-uniform input, the output should still resist deliberate tampering" is precisely the *chosen-input* PRF requirement: a PPT distinguisher may choose inputs (h, X) adaptively based on prior outputs. Assuming only that (h, X) are uniformly random is strictly weaker and does *not* capture this adversarial setting. Accordingly, any claim of robustness to malicious input must be phrased as (and reduced to) a computational PRF assumption against chosen-input distinguishers, as in Assumption 5.28 and the subsequent game-hop reduction.

Bridge to the security reduction. The decomposition above is the "white-box" view used to reason about what kinds of algebraic constraint systems an attacker would face when trying to model or invert the core. The subsequent hybrid-game reduction (next subsection) is independent of these structure observations: it simply states that any UF-CMA claim for the full MAC is constrained by the PRF quality of the core and by the coefficient-extraction deviations.

5.6.3 Categorical view (optional global perspective: *h_core* as a morphism)

For completeness, one can summarize the same structure at a higher level: for a fixed key K (equivalently fixed derived parameters), the core defines a deterministic mapping

$$F_K : \mathbb{Z}_{P_1} \times \mathbb{Z}_{P_1} \to \mathbb{Z}_{P_2}, \qquad (h, X) \mapsto h\_core^{(K)}(h, X),$$

which is a morphism in **FinSet** (finite sets with functions). Moreover, the implementation factors as a composition of morphisms corresponding to the layers:

$$(h, X) f_{\text{poly}}^{(K)}(\alpha, \beta) f_{\text{enc}}(a, b) f_{\text{bit}}^{(K)} \text{temp} f_{\text{out}} u,$$

where $f_{\text{bit}}^{(K)}$ is affine/linear over $\text{GF}(2)^{256}$ for fixed K, and f_{out} is modular addition in $\mathbb{Z}_{P_2}$. This categorical phrasing is useful for two concrete reasons (beyond "pretty words"):

- **Composition matches the code path.** It makes the proof strategy "plug-compatible" with the implementation: a game hop can replace one morphism (e.g., the core) while keeping the rest fixed.
- **The MAC loop is an endomorphism chain.** Once coefficients are extracted, each block induces an endomorphism $\Phi_X^{(K)} : \mathbb{Z}_{P_1} \to \mathbb{Z}_{P_1}$, and the full tag is the composition $\Phi_{X_t}^{(K)} \circ \cdots \circ \Phi_{X_1}^{(K)}(0)$ followed by truncation. This is the "global" view you referred to: it cleanly separates (i) the per-block core morphism that schedules coefficients and (ii) the outer Poly1305-shaped state evolution.

We still do the actual security discussion by expanding the induced equations/constraints layer by layer; the categorical view is the high-level wiring diagram that keeps those expansions organized.

5.7 Hybrid Games and Reduction

We use a game-hopping reduction that constrains the security of HardPoly1305 V2-Lite to the PRF quality of *h_core*. Let B_{total} be the total number of 16-byte blocks across all oracle queries made by $\mathcal{A}$.

Game 0 (Real). The challenger answers tag queries using the real construction with $h\_core(\cdot, \cdot; K_{\text{core}})$ and extraction G.

Game 1 (Random core). Replace *h_core* with a truly random function ρ that maps (h, X) to a uniform element of $\mathbb{Z}_{P_2}$; all other steps are unchanged.

Game 1.5 (Lazy-sampled coefficients with memoization). Maintain a table $\mathcal{T}$ keyed by input pairs (h, X). On each block, if (h_{i-1}, X_i) is in $\mathcal{T}$, retrieve u_i; otherwise sample $u_i \leftarrow \mathbb{Z}_{P_2}$ uniformly and store it in $\mathcal{T}$. Then compute $(r_i, s_i) = G(u_i)$ and update h_i as usual. This is the standard lazy-sampling view of a random function, followed by deterministic extraction.

Game 2 (Random coefficients). For each block, ignore ρ and instead sample (r_i, s_i) independently, where r_i is uniform over the Poly1305-clamped coefficient space and s_i is uniform in $\{0, 1\}^{128}$. The update rule stays the same. This is an idealized model that treats each block as having fresh coefficients even if the same input pair (h_{i-1}, X_i) repeats.

Let S_g denote the success event in Game g. We first relate Game 0 and Game 1 via a PRF reduction; then we note that Game 1 and Game 1.5 are equivalent by lazy sampling; finally we relate Game 1.5 and Game 2 via the extraction distribution and the independence idealization.

Lemma 5.15 (Lazy-Sampling Equivalence)**.** *Games 1 and 1.5 are identically distributed, so* $\Pr[S_1] = \Pr[S_{1.5}]$.

Proof. A random function $\rho : \mathbb{Z}_{P_1} \times \mathcal{X} \to \mathbb{Z}_{P_2}$ can be implemented by lazy sampling: on first query of an input pair (h, X), sample a uniform value in $\mathbb{Z}_{P_2}$ and memoize it; on repeats, return the memoized value. Game 1.5 implements exactly this for ρ, followed by the deterministic map G, so the transcript distribution matches Game 1. □

Lemma 5.16 (PRF Hybrid Lemma)**.** *Let $\mathcal{A}$ be any UF-CMA adversary that induces at most B_{total} calls to h_core across all tag queries and the final verification. There exists a PRF distinguisher $\mathcal{B}$ such that*

$$|\Pr[S_0] - \Pr[S_1]| \leq Adv^{\mathcal{B}}_{prf}(B_{total}).$$

Proof. Construct $\mathcal{B}$ with oracle access $\mathcal{O}$ that is either $h\_core(\cdot, \cdot; K_{\text{core}})$ or a random function ρ. $\mathcal{B}$ runs $\mathcal{A}$ and answers each tag query M as follows:

1. Compute $\text{Mixed}(M, K_{\text{mix}})$ and parse it into blocks $X_1, \ldots, X_t$.
2. Initialize $h \leftarrow 0$ and for each block query $\mathcal{O}(h, X_i)$ to obtain u_i.
3. Derive $(r_i, s_i) = G(u_i)$, update $h \leftarrow r_i \cdot (h + X_i) + s_i \pmod{P_1}$, and return $\text{Tag}_K(M) = \text{Trunc}_{128}(h)$.

When $\mathcal{A}$ outputs (M^*, T^*), $\mathcal{B}$ recomputes $\text{Tag}_K(M^*)$ using the same oracle $\mathcal{O}$ and outputs 1 if $\text{Verify}_K(M^*, T^*) = 1$ and M^* was not previously queried. If $\mathcal{O} = h\_core$, the simulation is identical to Game 0; if $\mathcal{O} = \rho$, the simulation is identical to Game 1. Therefore $\mathcal{B}$ distinguishes the PRF experiment with advantage $|\Pr[S_0] - \Pr[S_1]|$, yielding the stated bound. □

Lemma 5.17 (Heuristic Algebraic Hardness over Safe-Prime Field)**.** *Let $P_2 = 2q + 1$ be a safe prime and let*

$$\alpha = (H_1 + X_1 + k_{high})^2 + (X_1 - H_1 + k_{low})^3 \pmod{P_2},$$

where (H_1, X_1, α) are public and (k_{low}, k_{high}) are secret offsets in $\mathbb{Z}_{P_1}$ (embedded in $\mathbb{Z}_{P_2}$). Recovering (k_{low}, k_{high}) from (H_1, X_1, α) is heuristically hard in $\mathbb{Z}_{P_2}$ and is modeled as requiring generic multivariate polynomial solving.

Heuristic. Rearranging yields a coupled quadratic–cubic equation in the unknowns $(k_{\text{low}}, k_{\text{high}})$:

$$(k_{\text{high}} + H_1 + X_1)^2 + (k_{\text{low}} + X_1 - H_1)^3 - \alpha \equiv 0 \pmod{P_2}.$$

Over a prime field, this defines a non-linear algebraic relation of total degree three. For multiple samples, the attacker faces a system of such equations. We therefore adopt the following parameterized hardness model: letting $\lambda = \lceil \log_2 P_2 \rceil$, we assume that solving for $(k_{\text{low}}, k_{\text{high}})$ from random instances requires at least $\tilde{O}(2^{\lambda/2})$ field operations in the algebraic complexity model. This matches the scale of generic attacks on random bivariate systems over $\mathbb{F}_{P_2}$ and provides a concrete asymptotic barrier for inversion. The safe-prime structure rules out reductions through small multiplicative subgroups. This is a heuristic assumption, not a proven lower bound. □

The next subsection relates Game 1.5 and Game 2 via the extraction distribution and introduces the random-coefficient model used for the final analysis.

5.8 Reduction to a Random-Coefficient Polynomial MAC

Fix a message M and key $K = (K_{\text{mix}}, K_{\text{core}})$. Let $\text{Mixed}(M, K_{\text{mix}})$ be partitioned into 16-byte blocks $M_1, \ldots, M_t$ and define

$$X_i = \text{LE}(M_i \| \texttt{0x01}) \in \mathbb{Z}_{P_1}, \quad h_0 = 0.$$

In Game 1.5 the random oracle ρ is realized by lazy sampling, so for each block we set

$$u_i := \rho(h_{i-1}, X_i),$$

where ρ returns a uniformly random value in $\mathbb{Z}_{P_2}$ for each fresh input pair (h_{i-1}, X_i) and is consistent on repeats. Coefficients are derived by the extraction map G: write

$$u_i = u_{i,\mathrm{lo}} + 2^{128} u_{i,\mathrm{hi}},$$

where $u_{i,\mathrm{lo}}$ and $u_{i,\mathrm{hi}}$ are the low and high 128 bits, and define

$$r_i = u_{i,\mathrm{lo}} \wedge \mathrm{CLAMP}, \quad s_i = u_{i,\mathrm{hi}} \bmod 2^{128}.$$

The state update is then

$$h_i = r_i \cdot (h_{i-1} + X_i) + s_i \pmod{P_1}.$$

The following lemma makes the extraction distribution explicit.

Lemma 5.18 (Extraction from Uniform u)**.** *Let u be uniform over $\mathbb{Z}_{P_2}$ where $P_2 = 2^{256} - 188069$. Define u_{lo} as the low 128 bits and u_{hi} as the high 128 bits of the 256-bit representation of u. Let $r = u_{lo} \wedge CLAMP$ and $s = u_{hi} \bmod 2^{128}$. Then the joint distribution of (r, s) is within statistical distance*

$$\Delta\big((r,s), (UnifClamp, Unif_{128})\big) \le \varepsilon_{range}, \quad \varepsilon_{range} := \frac{2^{256} - P_2}{2^{256}} = \frac{188069}{2^{256}}.$$

In particular, both marginals are ε_{range}-close to the ideal uniform marginals, and any dependence between r and s is bounded by ε_{range} in total variation distance.

Proof. Let U_{256} be uniform over $\{0,1\}^{256}$ and let U_{P_2} be uniform over $\mathbb{Z}_{P_2}$, both viewed as integers in $[0, 2^{256} - 1]$. Since U_{P_2} is exactly U_{256} conditioned on the event $U_{256} < P_2$, the statistical distance satisfies

$$\Delta(U_{P_2}, U_{256}) = \frac{2^{256} - P_2}{2^{256}} = \varepsilon_{\mathrm{range}}.$$

Applying any deterministic map cannot increase statistical distance, so mapping $u \mapsto (r, s)$ preserves the same upper bound $\varepsilon_{\mathrm{range}}$. For the ideal reference case $u \leftarrow U_{256}$, the pair $(u_{\mathrm{lo}}, u_{\mathrm{hi}})$ is independent uniform in $\{0,1\}^{128} \times \{0,1\}^{128}$; hence $s = u_{\mathrm{hi}}$ is uniform in $\{0,1\}^{128}$, and $r = u_{\mathrm{lo}} \wedge \mathrm{CLAMP}$ is uniform over the 2^{106} clamped values because the clamp mask fixes 22 bit positions, giving each clamped value exactly 2^{22} preimages. This yields $(\mathrm{UnifClamp}, \mathrm{Unif}_{128})$ with independence in the ideal case, and the claimed $\varepsilon_{\mathrm{range}}$-closeness for uniform $u \in \mathbb{Z}_{P_2}$. □

Therefore, in Game 1.5, for any *fresh* input pair (h, X), the value $u \leftarrow \mathbb{Z}_{P_2}$ is uniform, and by Lemma 5.18 the extracted pair $(r, s) = G(u)$ is within statistical distance at most $\varepsilon_{\mathrm{range}}$ of $(\mathrm{UnifClamp}, \mathrm{Unif}_{128})$ with independence.

Repeat-input event and the $\varepsilon_{\mathbf{der}}$ bridge. Game 1.5 (a memoized random function) and Game 2 (fresh coefficients every block) differ only when a block invocation reuses an already-seen input pair (h, X), causing the same u (and thus the same (r, s)) to repeat. We formalize this as a "bad" event.

Definition 5.19 (Repeat-input bad event)**.** *Let the global sequence of all block invocations across all tag queries* and *the final verification recomputation be indexed as $j = 1, \ldots, B_{total}$, with inputs (h_{j-1}, X_j). Define*

$$\mathsf{Bad}_{\mathrm{rep}} := \exists\, 1 \le a < b \le B_{total} \text{ such that } (h_{a-1}, X_a) = (h_{b-1}, X_b).$$

Lemma 5.20 (Game 1.5 vs. Game 2 conditioned on no repeats)**.** *Conditioned on $\neg\mathsf{Bad}_{\mathrm{rep}}$, the transcript distributions of Games 1.5 and 2 are within statistical distance at most $B_{total} \cdot \varepsilon_{range}$.*

Proof. In Game 1.5, each fresh input pair (h, X) is assigned an independent uniform $u \leftarrow \mathbb{Z}_{P_2}$, and repeats return the memoized u. Conditioned on $\neg\mathsf{Bad}_{\mathrm{rep}}$, every invocation is fresh, so the sampled values $u_1, \ldots, u_{B_{\mathrm{total}}}$ are independent and uniform in $\mathbb{Z}_{P_2}$.

By Lemma 5.18, for each invocation the extracted pair $(r_i, s_i) = G(u_i)$ is within statistical distance at most $\varepsilon_{\mathrm{range}}$ of the ideal pair $(\mathrm{UnifClamp}, \mathrm{Unif}_{128})$ used in Game 2. A standard hybrid argument over the B_{total} independent draws implies that the joint distribution of all coefficient pairs in Game 1.5 is within statistical distance at most $B_{\mathrm{total}} \cdot \varepsilon_{\mathrm{range}}$ of the joint distribution in Game 2. Since the remainder of the transcript is a deterministic function of these sampled coefficients and the adversary's queries, the same bound applies to the full transcript distribution. □

To quantify the gap unconditionally, we keep a single deviation term:

$$\varepsilon_{\text{der}} \;:=\; \Pr[\mathsf{Bad}_{\text{rep}}] \;+\; B_{\text{total}} \cdot \varepsilon_{\text{range}} \;+\; \varepsilon_{\text{ext}},$$

where $\varepsilon_{\text{range}} = (2^{256} - P_2)/2^{256}$ is the unavoidable range-gap from Lemma 5.18, and ε_{ext} captures any further extraction non-uniformity in the *real* construction (e.g., if u deviates from uniform in $\mathbb{Z}_{P_2}$ due to replacing ρ with the concrete h_core). In Games 1/1.5, $\varepsilon_{\text{ext}} = 0$ by construction; in Game 0 it is an explicit assumption/measurement target.

Lemma 5.21 (Bounding repeat-input probability in the random-coefficient model)**.** *In Game 2, for any adaptive adversary inducing at most B_{total} block invocations,*

$$\Pr[\mathsf{Bad}_{\text{rep}}] \le \frac{B_{total}(B_{total}-1)}{2P_1} \le \frac{B_{total}^2}{P_1}.$$

Proof. Fix any two invocation indices $1 \le a < b \le B_{\text{total}}$. If $X_a \neq X_b$ then $(h_{a-1}, X_a) \neq (h_{b-1}, X_b)$ deterministically. If $X_a = X_b$, it suffices to bound $\Pr[h_{a-1} = h_{b-1}]$.

In Game 2, each update has the form

$$h_i = r_i \cdot (h_{i-1} + X_i) + s_i \pmod{P_1},$$

where s_i is uniform in $\{0,1\}^{128}$ and (by construction of Game 2) is independent of all prior randomness and of the adversary's view up to step $i-1$. For any fixed (r_i, h_{i-1}, X_i), the map $s_i \mapsto h_i$ is a bijection over $\mathbb{Z}_{P_1}$ (Lemma 5.24), so h_i is uniform over $\mathbb{Z}_{P_1}$ conditioned on the entire past. In particular, for any fixed value $z \in \mathbb{Z}_{P_1}$,

$$\Pr[h_{b-1} = z \mid \text{past up to } b-2] = \frac{1}{P_1}.$$

Taking $z := h_{a-1}$ and averaging over the past gives $\Pr[h_{b-1} = h_{a-1}] \le 1/P_1$. A union bound over all unordered pairs (a, b) yields

$$\Pr[\mathsf{Bad}_{\text{rep}}] \le \binom{B_{\text{total}}}{2} \cdot \frac{1}{P_1}.$$

□

Combining Lemma 5.20 with the definition of ε_{der} yields the standard bridge:

$$|\Pr[S_{1.5}] - \Pr[S_2]| \le \varepsilon_{\text{der}}.$$

Together with the PRF hop, the total gap between the real construction (Game 0) and the random-coefficient model (Game 2) is at most $\text{Adv}_{\text{prf}}(B_{\text{total}}) + \varepsilon_{\text{der}}$.

The resulting construction is a *random-coefficient polynomial MAC*:

- For each block i, draw independent (r_i, s_i), with r_i clamped and s_i uniform.
- Update: $h_0 = 0$, $h_i = r_i \cdot (h_{i-1} + X_i) + s_i \pmod{P_1}$.
- Output Tag $= \text{Trunc}_{128}(h_t)$.

From the adversary's perspective, for a fixed message $M = (X_1, \ldots, X_t)$, the output tag is the 128-bit truncation of a polynomial computation driven by fresh random coefficients. This behavior is close to that of a random function on M.

5.9 What Our Algorithm (HardPoly1305) Is Tweaked From Poly1305: Algebraic Expansion and Block-Replay

The tweak is that coefficients are per-block and state-dependent. Define

$$(r_i, s_i) = G(h\_core(h_{i-1}, X_i; K_{\text{param}})) =: \big(R_K(h_{i-1}, X_i),\, S_K(h_{i-1}, X_i)\big),$$

so the update becomes

$$h_i = R_K(h_{i-1}, X_i) \cdot (h_{i-1} + X_i) + S_K(h_{i-1}, X_i) \pmod{P_1}.$$

Expanding the first few steps yields a multivariate algebraic expansion with data-dependent coefficients:

$$h_1 = R_K(0, X_1)\, X_1 + S_K(0, X_1),$$
$$h_2 = R_K(h_1, X_2)\big(R_K(0, X_1)\, X_1 + S_K(0, X_1) + X_2\big) + S_K(h_1, X_2),$$
$$h_3 = R_K(h_2, X_3)\Big(R_K(h_1, X_2)\big(R_K(0, X_1)\, X_1 + S_K(0, X_1) + X_2\big) + S_K(h_1, X_2) + X_3\Big) + S_K(h_2, X_3).$$

Even though each step is affine in the current block X_i over $\mathbb{Z}_{P_1}$, the coefficients R_K, S_K are nonlinear functions of the entire prefix state. Hence the overall map $M \mapsto h_t$ is not a low-degree polynomial in the message blocks alone; it is a nested multivariate expression with coefficients that depend on prior blocks and the key.

By contrast, if we freeze the coefficients to constants $R_K(h, X) \equiv r$ and $S_K(h, X) \equiv 0$ (and add a single s at the end as in standard Poly1305), the recurrence collapses to the classic polynomial hash

$$h_t = r^t X_1 + r^{t-1} X_2 + \cdots + r X_t,$$

which is the univariate algebraic structure exploited in the one-time security analysis of Poly1305.

The state-dependence of (r_i, s_i) is what frustrates block replay: changing any prefix block changes h_{i-1} and thus changes the coefficient functions R_K, S_K at position i. A block spliced from another message will therefore be evaluated under different coefficients unless the entire prefix is reproduced (or a state collision is forced), which is precisely the difficulty formalized next.

Lemma 5.22 (Prefix-Binding Against Block Replay (Game 2))**.** *Consider two messages M and M' whose mixed-and-encoded block sequences first differ at position j. In Game 2 (random coefficients), conditioned on all randomness before step j, the probability that the internal states collide at step j is at most $1/P_1$. Consequently, any block-replay or cut-and-paste attempt that changes the prefix succeeds with probability at most $1/P_1$, plus the PRF and extraction deviations when lifting back to the real construction.*

Proof. Condition on all randomness before step j, so h_{j-1} and h'_{j-1} are fixed. In Game 2, the pair (r_j, s_j) for M is fresh and independent of all prior values. The update

$$h_j = r_j \cdot (h_{j-1} + X_j) + s_j \pmod{P_1}$$

is affine in the fresh offset s_j with unit coefficient, so h_j is uniform over $\mathbb{Z}_{P_1}$. For any fixed h'_j this gives $\Pr[h_j = h'_j] \le 1/P_1$. Thus a block substitution that alters the prefix must induce a state collision to keep coefficients aligned. In the real scheme, the Game 0 to Game 2 gap is bounded by $\mathrm{Adv}_{\mathrm{prf}}(B_{\mathrm{total}}) + \varepsilon_{\mathrm{der}}$. □

Important: exact replay of a previously authenticated pair $(M, \mathrm{Tag}_K(M))$ is still possible, because MACs alone do not prevent network-level replay. Preventing replay requires nonces, sequence numbers, or session context at the protocol layer.

5.10 Replay Resistance (Protocol Fix + Quantitative Bounds)

This document distinguishes two different phenomena that are both casually called "replay": (i) *cut-and-paste / block splicing* at the algorithm level (handled by the state dependence analyzed above), and (ii) *network-level replay* of an entire previously valid transcript (M, T) (which a MAC alone cannot stop). This subsection records the minimal protocol-layer fix and the resulting quantitative bounds.

Algorithm-level "block replay" strength (Tweaked structure claim). This is the place where the Tweaked V2 structure matters: because coefficients are derived from the evolving state, a block spliced from another message is evaluated under *different* coefficients unless the entire prefix is reproduced (or a state collision is forced). Concretely, if two mixed-and-encoded block sequences first differ at position j, then in the random-coefficient model the probability that the internal states collide at step j is at most $1/P_1$. Lifting back to the real construction, the corresponding success probability for a cut-and-paste/block-splice attempt is bounded by

$$\mathrm{Adv}_{\mathrm{splice}} \;\le\; \frac{1}{P_1} \;+\; \mathrm{Adv}_{\mathrm{prf}}(B_{\mathrm{total}}) \;+\; \varepsilon_{\mathrm{der}},$$

where the last two terms are the core PRF deviation and coefficient-extraction deviation already used elsewhere in the reduction.

Protocol-layer anti-replay fix (minimal). To prevent network replay, the sender and verifier must bind freshness into what is authenticated. Let $N \in \{0,1\}^n$ be a nonce (or sequence number/counter) and define the authenticated payload as $M' := N \parallel M$. The verifier maintains a replay cache (or monotone counter state) and rejects any N that was already accepted for the same key/session context.

Replay game. An adversary observes and/or queries tags for multiple messages and then attempts to have the verifier accept a *replayed* transcript. We model acceptance as the conjunction of: (a) MAC verification passes, and (b) the nonce check passes (fresh N under the verifier's cache/counter policy).

Quantitative bound (upper bound). Let p_{miss} be an upper bound on the probability that the verifier's anti-replay state fails to detect a repeated nonce N (e.g., due to cache eviction, multi-replica desynchronization, crash/reset, or misconfiguration). Let $\text{Adv}_{\text{uf-cma}}(\mathcal{A})$ be the standard unforgeability advantage of the MAC against chosen-message attackers (as bounded elsewhere in this document). Then any PPT replay adversary $\mathcal{A}$ satisfies:

$$\text{Adv}_{\text{replay}}(\mathcal{A}) \;\le\; p_{\text{miss}} \;+\; \text{Adv}_{\text{uf-cma}}(\mathcal{A}).$$

Intuition: if the nonce check works (probability $1 - p_{\text{miss}}$), the only way to get acceptance is a fresh forgery, which is exactly the UF-CMA event. If the nonce check fails (probability $\le p_{\text{miss}}$), exact replay becomes possible even without forging.

Quantitative bound for "user accidentally replays" (random nonce reuse). If the sender chooses N uniformly at random from $\{0,1\}^n$ for each of q authenticated messages under a fixed key/session, then the probability that the sender *reuses* a nonce at least once is bounded by the birthday term:

$$\Pr[\exists\, i \neq j :\; N_i = N_j] \;\le\; \frac{q(q-1)}{2^{n+1}}.$$

This term measures how likely it is that a well-intentioned user creates a "freshness failure" by accident when using random nonces. If a monotone counter is used instead (and stored correctly), this probability becomes 0 (until counter reset), and the dominant risk is p_{miss} (state loss / rollback).

Lower bounds (what you cannot claim away).

- **No anti-replay state** (no nonce/counter check): $\text{Adv}_{\text{replay}} = 1$ by trivial retransmission of a previously valid (M, T).
- **With anti-replay state but state-loss probability p_{miss}**: there exists an adversary achieving replay acceptance with probability at least p_{miss} by replaying immediately after the state-loss event. Hence p_{miss} is not just an upper bound term; it is the irreducible engineering risk term to be minimized.

Algebraic expansion for collision analysis. For a fixed key and message, write the realized coefficients

$$r_i := R_K(h_{i-1}, X_i), \quad s_i := S_K(h_{i-1}, X_i),$$

and define the affine map $\Phi_i(x) = r_i(x + X_i) + s_i$. Then $h_i = \Phi_i(h_{i-1})$ and

$$h_t = \Phi_t \circ \Phi_{t-1} \circ \cdots \circ \Phi_1(0).$$

Lemma 5.23 (Inductive Expansion of the Composition)**.** *For $t \ge 1$, with $h_0 = 0$ and $h_i = r_i(h_{i-1} + X_i) + s_i$, the state admits the nested form*

$$h_t = r_t(\cdots(r_2(r_1X_1 + s_1 + X_2) + s_2 + X_3)\cdots + s_{t-1} + X_t) + s_t,$$

where all equalities are in $\mathbb{Z}_{P_1}$.

Proof. We proceed by induction on t. *Base case* $t = 1$*:* $h_1 = r_1(h_0 + X_1) + s_1 = r_1 X_1 + s_1$, matching the formula. *Inductive step:* assume the formula holds for $t - 1$:

$$h_{t-1} = r_{t-1}(\cdots(r_2(r_1 X_1 + s_1 + X_2) + s_2 + X_3)\cdots + s_{t-2} + X_{t-1}) + s_{t-1}.$$

Then

$$\begin{aligned} h_t &= r_t(h_{t-1} + X_t) + s_t \\ &= r_t\big(r_{t-1}(\cdots(r_2(r_1 X_1 + s_1 + X_2) + s_2 + X_3)\cdots + s_{t-2} + X_{t-1}) + s_{t-1} + X_t\big) + s_t, \end{aligned}$$

which is exactly the claimed nested expression. □

This expansion makes explicit that h_t is affine in the last offset s_t, a fact used in the next lemma.

Lemma 5.24 (Affine Uniformity of the Last Offset)**.** *For any fixed (r_t, h_{t-1}, X_t), the mapping $s_t \mapsto h_t$ is a bijection over $\mathbb{Z}_{P_1}$. Consequently, if s_t is uniform and independent, then h_t is uniform over $\mathbb{Z}_{P_1}$.*

Proof. Fix (r_t, h_{t-1}, X_t). For any target value $y \in \mathbb{Z}_{P_1}$, define

$$s_t = y - r_t \cdot (h_{t-1} + X_t) \pmod{P_1}.$$

Then the update rule $h_t = r_t \cdot (h_{t-1} + X_t) + s_t \pmod{P_1}$ yields $h_t = y$. Thus the map $s_t \mapsto h_t$ is a bijection over $\mathbb{Z}_{P_1}$, and if s_t is uniform and independent then h_t is uniform over $\mathbb{Z}_{P_1}$. □

Lemma 5.25 (Collision Bound)**.** *Let B_{total} be the total number of blocks across all queries. In the random-coefficient model,*

$$\Pr[h_t(M) = h_t(M')] \le \frac{B_{total}(B_{total} - 1)}{2P_1} \le \frac{B_{total}^2}{P_1}.$$

Proof. Fix two distinct messages and let j be the first index where the encoded blocks differ. Conditioned on all randomness before step j, the values h_{j-1} and h'_{j-1} are fixed. In Game 2, (r_j, s_j) and (r'_j, s'_j) are independent fresh samples. For the message on which we analyze the collision, the update

$$h_j = r_j \cdot (h_{j-1} + X_j) + s_j \pmod{P_1}$$

is affine in the fresh offset s_j with unit coefficient, so h_j is uniform over $\mathbb{Z}_{P_1}$. Hence for any fixed h'_j, the probability that $h_j = h'_j$ is at most $1/P_1$. If a collision occurs at step j, subsequent steps are deterministic given the remaining coefficients, so this bounds the probability that the final states collide. A union bound over all unordered pairs of blocks across all queries yields the stated B_{total}^2/P_1 bound. □

Lemma 5.26 (Truncation Guessing Bound)**.** *If h_t is uniform over $\mathbb{Z}_{P_1}$, then for any fixed tag candidate T,*

$$\Pr[\mathit{Trunc}_{128}(h_t) = T] \le 2^{-128}.$$

Proof. Since h_t is uniform over $\mathbb{Z}_{P_1}$, the probability of any fixed tag T equals the fraction of residues whose low 128 bits equal T. This preimage size is either $\lfloor P_1/2^{128} \rfloor$ or $\lceil P_1/2^{128} \rceil$, so

$$\Pr[\mathrm{Trunc}_{128}(h_t) = T] \le \frac{\lceil P_1/2^{128} \rceil}{P_1} \le 2^{-128}.$$

□

5.11 Assumptions (Axioms)

We state the assumptions used in the reduction:

Assumption 5.27 (Parameterized α-Inversion Complexity)**.** *Let $\lambda = \lceil \log_2 P_2 \rceil$. Any algorithm that, given a polynomial number of samples $(H_1^{(i)}, X_1^{(i)}, \alpha^{(i)})$, recovers (k_{low}, k_{high}) with non-negligible probability requires at least $\tilde{O}(2^{\lambda/2})$ field operations over $\mathbb{Z}_{P_2}$.*

Assumption 5.28 (*h_core* Computational PRF Axiom (Unproven for this Concrete Core))**.** *For a uniformly random 32-byte key K_{core} (or equivalently a uniformly random master key K with derived K_{param}), the function family*

$$F_K(h, X) := h\_core(h, X; K_{param})$$

is computationally indistinguishable from a truly random function $\rho : \mathbb{Z}_{P_1} \times \mathcal{X} \to \mathbb{Z}_{P_2}$ over the same domain, for all efficient distinguishers with at most B oracle queries. Let $Adv_{prf}(B)$ denote the maximal distinguishing advantage with at most B queries.

Important (scope of what can be proven). *By Theorem 5.4, no k-bit keyed family (in particular $k = 256$ here) can be information-theoretically indistinguishable from a random function once an adversary is allowed even a small number of samples, so this PRF statement cannot be established as a "pure math" statistical claim about the concrete h_core. It is an explicit computational axiom about the concrete core design, meant to be supported (if at all) by cryptanalytic evidence rather than a standard-model proof.*

5.12 Relationships and Baseline Bounds (Key Recovery, PRF Distinguishing, UF-CMA)

This subsection records three facts that are often conflated in informal discussions: (i) "mod-add introduces Boolean nonlinearity" (a structure statement), (ii) "the concrete core is a PRF" (a computational indistinguishability claim), and (iii) "the MAC is UF-CMA" (a chosen-message unforgeability claim). We make the relationships precise and state generic upper/lower bounds that hold regardless of the internal design choices.

Key-recovery security game. Let $K \leftarrow \{0,1\}^{256}$ be uniform (or the master key space used in the implementation) and let $\mathcal{A}$ be a PPT algorithm with oracle access to $F_K(h, X)$ on inputs of its choice. Define

$$\mathrm{Adv}_{\mathrm{kr}}(\mathcal{A}) := \Pr[\mathcal{A}^{F_K} \text{ outputs } K].$$

Lemma 5.29 (Brute-force lower bound)**.** *For any key space of size 2^{256}, there exists a (time-q) adversary making no oracle queries that achieves $Adv_{kr} \geq q/2^{256}$ by trying q independent key guesses.*

Derived-parameter recovery security game. Let $K_{\mathrm{param}} := \textsc{DeriveKeyParams}(K)$ denote the derived tuple used by the implementation, e.g. $K_{\mathrm{param}} = (k_{\mathrm{high}}, k_{\mathrm{low}}, k_{\mathrm{mix}}, k_{\mathrm{mix2}})$. Define the parameter-recovery advantage

$$\mathrm{Adv}_{\text{kr-param}}(\mathcal{A}) := \Pr[\mathcal{A}^{F_K} \text{ outputs } K_{\mathrm{param}}].$$

Lemma 5.30 (Parameter recovery vs master-key recovery under key normalization)**.** *In the reference implementation, the derived parameters depend only on the canonical 32-byte key $K' = \textsc{FoldKey32}(K)$. Therefore, recovering the full master key K is (in general) strictly harder than recovering K_{param} when $|K| > 32$, because FoldKey32 is many-to-one. However, in the common case $|K| = 32$ (no folding), k_{high} and k_{low} equal the two 16-byte halves of K interpreted as 128-bit integers (since $P_1 > 2^{128}$), so K can be reconstructed from (k_{high}, k_{low}) alone.*

Internal coefficient recovery ("reverse-engineering the per-block secrets"). For a message M that induces B blocks, the implementation deterministically derives per-block coefficients (r_i, s_i) from the 256-bit word u_i via the map $G = \textsc{DeriveRSFromU}$ (lower 16 bytes clamped to r_i, upper 16 bytes to s_i). To quantify how much of these internal values can be recovered from an external transcript, we define the following recovery game.

Definition 5.31 (Coefficient-sequence recovery game)**.** *The challenger samples a random key K and answers MAC tag queries as usual. The adversary $\mathcal{A}$ outputs a* fresh *message M^* and receives its tag T^*. Then $\mathcal{A}$ outputs a candidate coefficient sequence $\widehat{C}^* = \{(\hat{r}_i, \hat{s}_i)\}_{i=1}^{B}$ of the same length. Let $C^* = \{(r_i, s_i)\}_{i=1}^{B}$ denote the true internal coefficients used when computing T^*. We define*

$$Adv_{coeff}(\mathcal{A}) := \Pr[\widehat{C}^* = C^*].$$

Lemma 5.32 (Information-theoretic coefficient recovery bound in the random-coefficient idealization)**.** *In the idealized Game 2 model (fresh independent coefficients per block), let r_i be uniform over the Poly1305-clamped 128-bit coefficient space (size 2^{106}) and s_i be uniform over $\{0,1\}^{128}$ (size 2^{128}), independently for each block. Then for any adversary $\mathcal{A}$ in Definition 5.31 on a B-block challenge message,*

$$Adv_{coeff}(\mathcal{A}) \le 2^{128} \cdot 2^{-234B} \;=\; 2^{128-234B}.$$

Proof. In the Game 2 idealization, the coefficient sequence C^* is uniform over a set of size $N = (2^{106} \cdot 2^{128})^B = 2^{234B}$. The observed tag T^* is a deterministic 128-bit function of (M^*, C^*), hence it can reduce the min-entropy of C^* by at most 128 bits (equivalently, for any fixed (M^*, T^*) the fiber $\{C : \textsc{Tag}(M^*; C) = T^*\}$ has size at least $N/2^{128}$). Therefore the maximum posterior point probability satisfies $\max_C \Pr[C^* = C \mid M^*, T^*] \le 2^{128}/N$, and any adversary's success probability is bounded by that maximum. □

PRF distinguishing vs key recovery (unconditional reduction).

Lemma 5.33 (Key recovery implies PRF distinguishing)**.** *For any oracle query budget B, the maximal PRF distinguishing advantage satisfies*

$$Adv_{prf}(B) \;\ge\; Adv_{kr}(B) \;-\; \frac{1}{P_2}.$$

Proof. Given a key-recovery adversary $\mathcal{A}$ with at most B oracle queries, build a distinguisher $\mathcal{D}$ as follows. $\mathcal{D}$ runs $\mathcal{A}$ with oracle access to its challenge oracle $\mathcal{O}$ and obtains a candidate key $\hat{K}$. Then $\mathcal{D}$ samples a fresh test input (h^*, X^*) not previously queried, queries $y^* \leftarrow \mathcal{O}(h^*, X^*)$, and computes $\hat{y}^* := F_{\hat{K}}(h^*, X^*)$. If $y^* = \hat{y}^*$ output "real", else output "random".

If $\mathcal{O} = F_K$, then whenever $\hat{K} = K$ we have $y^* = \hat{y}^*$, hence $\Pr[\mathcal{D} = 1 \mid \text{real}] \ge \mathrm{Adv}_{\mathrm{kr}}(\mathcal{A})$. If $\mathcal{O} = \rho$ is a truly random function into $\mathbb{Z}_{P_2}$, then for any fixed $\hat{K}$ the value $\hat{y}^*$ is fixed while y^* is uniform in $\mathbb{Z}_{P_2}$, so $\Pr[y^* = \hat{y}^*] \le 1/P_2$. Taking the difference yields the bound. □

UF-CMA baseline (tag guessing) and collision terms. Independently of how the coefficients are generated, any UF-CMA adversary can always attempt a pure guess of the 128-bit tag on a fresh message, achieving success probability about 2^{-128} per attempt. Conversely, in the idealized Game 2 model used in Theorem 5.34, this trivial strategy is essentially optimal up to the polynomial collision term B_{total}^2/P_1 (Lemma 5.25). This is why the UF-CMA bound is naturally expressed as a sum of: (i) a PRF deviation term for the core, (ii) a tag-guessing term $B_{\mathrm{total}}/2^{128}$, and (iii) a collision term B_{total}^2/P_1.

What is (and is not) closed for the concrete implementation. For the concrete arithmetic/bit-mixing *h_core* implemented in this project, Lemma 5.33 and the UF-CMA reduction theorems are unconditional *reductions*. What cannot be closed in the standard model without an additional cryptographic assumption is an explicit negligible bound on $\mathrm{Adv}_{\mathrm{prf}}(B)$ (and hence on $\mathrm{Adv}_{\mathrm{kr}}(B)$) for this specific core; that is exactly why Assumption 5.28 is stated as an explicit computational axiom. If the environment permits an idealized instantiation of the core (outside the scope of this concrete paper), then the PRF term in the reduction can be bounded by the corresponding ideal-model guarantee; otherwise, the concrete *h_core* PRF property remains an explicit computational assumption.

5.13 Theorems and Bounds

Theorem 5.34 (PRF-Constrained Reduction with Complexity)**.** *Let $\mathcal{A}$ be a UF-CMA adversary running in time $t_{\mathcal{A}}$, making q_{tag} tag queries with total block count B_{total}. There exists a PRF distinguisher $\mathcal{B}$ with:*

- *oracle query complexity $q_{\mathcal{B}} \le B_{total}$,*
- *running time $t_{\mathcal{B}} \le t_{\mathcal{A}} + O(B_{total} \cdot polylog(P_1))$,*

such that

$$|\Pr[S_0] - \Pr[S_1]| \leq Adv^{\mathcal{B}}_{prf}(B_{total}).$$

In particular, the advantage of $\mathcal{A}$ in the real world is upper bounded by the PRF advantage of $\mathcal{B}$ plus the statistical deviation of G.

Proof. The simulator from the previous proof uses exactly one oracle call per processed block, so the number of oracle queries is $q_{\mathcal{B}} \leq B_{\text{total}}$. The additional computation is the extraction G and modular arithmetic in $\mathbb{Z}_{P_1}, \mathbb{Z}_{P_2}$ per block, giving total overhead $O(B_{\text{total}} \cdot \text{polylog}(P_1))$. The distinguishing advantage follows from the equivalence between Game 0 and Game 1 established above. □

Theorem 5.35 (Conditional UF-CMA Bound (Heuristic))**.** *Let $\mathcal{A}$ be any UF-CMA adversary making q_{tag} MAC queries and let B_{total} be the total number of 16-byte blocks across all queries. Under the h_core PRF assumption, a pseudorandom 32-byte master key K, and the idealized random-coefficient model for (r_i, s_i),*

$$Adv_{uf\text{-}cma}(\mathcal{A}) \leq Adv_{prf}(B_{total}) + \varepsilon_{der} + \frac{B_{total}}{2^{128}} + \frac{B^2_{total}}{2^{130}}.$$

Proof. Let $S_0, S_1, S_{1.5}, S_2$ be the success events in Games 0, 1, 1.5, and 2. By the PRF hop and Lemma 5.15,

$$\Pr[S_0] \leq \Pr[S_{1.5}] + \text{Adv}_{\text{prf}}(B_{\text{total}}).$$

By Lemma 5.20 and the definition of ε_{der} (which accounts for repeat-input deviation and any extraction bias), we have

$$\Pr[S_{1.5}] \leq \Pr[S_2] + \varepsilon_{\text{der}}.$$

Combining yields

$$\Pr[S_0] \leq \Pr[S_2] + \text{Adv}_{\text{prf}}(B_{\text{total}}) + \varepsilon_{\text{der}}.$$

In Game 2, if a forgery causes an internal state collision with a previously queried transcript, Lemma 5.25 implies probability at most B^2_{total}/P_1. Conditioned on no such collision, the final state h_t for the fresh message is uniform (Lemma 5.24), so the best strategy is to guess the 128-bit tag, giving probability at most 2^{-128}. Since $q_{\text{tag}} \leq B_{\text{total}}$, we upper bound this by $B_{\text{total}}/2^{128}$ for notational consistency. Using $P_1 \approx 2^{130}$ yields the stated bound. □

Here:

- $\text{Adv}_{\text{prf}}(B_{\text{total}})$ bounds the PRF distinguishing advantage.
- ε_{der} bounds extraction bias and any repeat-input deviation between Game 1.5 and the idealized random-coefficient model.
- $B_{\text{total}}/2^{128}$ is the 128-bit tag guessing bound.
- $B^2_{\text{total}}/2^{130}$ bounds polynomial MAC collisions in $\mathbb{Z}_{P_1}$.

Corollary 5.36 (Concrete Numbers)**.** *If $B_{total} \leq 2^{32} \approx 4.3 \times 10^9$, then:*

$$\frac{B_{total}}{2^{128}} \leq 2^{-96}, \quad \frac{B^2_{total}}{2^{130}} \leq 2^{-66}.$$

Thus under this conditional model, the bound is dominated by approximately 2^{-66}, plus the PRF deviation term. For $B_{total} \leq 2^{30}$, the combined bound drops below 2^{-68}.

5.14 Role of Mixed(M, K) and Key Requirements

The mixing function Mixed(M, K) is part of the security hypothesis. It linearly masks each message byte with the key and pads short messages, ensuring that all inputs to *h_core* carry key-dependent structure. This discourages attacks that would otherwise exploit message-only algebraic structure in the *h_core* input domain.

Consequently, the master key K must be a high-entropy byte string with $|K| \geq 32$ derived from a CSPRNG or a standard KDF. Low-entropy values (passwords, user IDs, timestamps) are forbidden because they weaken the masking effect and expose additional structure to adversarial analysis.

5.15 Remarks on Assumptions

Under the assumptions that (i) K is pseudorandom, (ii) *h_core* behaves as a PRF, and (iii) $B_{\text{total}} \ll 2^{64}$, HardPoly1305 V2-Lite is modeled to satisfy the UF-CMA-style bound with advantage bounded by

$$\text{Adv}_{\text{uf-cma}}(\mathcal{A}) \leq \text{Adv}_{\text{prf}}(B_{\text{total}}) + \varepsilon_{\text{der}} + \frac{B_{\text{total}}}{2^{128}} + \frac{B_{\text{total}}^2}{2^{130}}.$$

6 Pseudocode Description

6.1 Key Parameter Derivation

Algorithm 1 DeriveKeyParams

Require: Key K (bytes), $|K| \geq 32$
Ensure: $(k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$
$K' \leftarrow \textsc{FoldKey32}(K)$ ▷ xor-fold any extra bytes into 32 bytes
$K_0 \leftarrow K'[0..15]$
$K_1 \leftarrow K'[16..31]$
$k_{\text{high}} \leftarrow \text{LE\_to\_int}(K_0)$
$k_{\text{low}} \leftarrow \text{LE\_to\_int}(K_1)$
$k_{\text{mix}} \leftarrow \text{LE\_to\_int}(K'[0..31])$
$k_{\text{mix2}} \leftarrow (\neg k_{\text{mix}}) \ggg_{256} 1$ ▷ bitwise NOT then rotate-right in 256-bit domain
return $(k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$

Algorithm 2 FoldKey32

Require: Key K (bytes), $|K| \geq 32$
Ensure: K' (32 bytes)
$K' \leftarrow K[0..31]$
for $i = 32$ to $|K| - 1$ **do**
    $K'[i \bmod 32] \leftarrow K'[i \bmod 32] \oplus K[i]$
end for
return K'

6.2 Message-Key Mixing

Algorithm 3 MixKeyAndMessage

Require: Message M, Key K with $|K| \geq 32$
Ensure: Mixed data
$\ell \leftarrow |M|$
if $\ell < 32$ **then**
    $P \leftarrow M \,\|\, \texttt{0xA7} \,\|\, \texttt{0x00}^{(32-\ell-1)}$ ▷ domain-separated padding
else
    $P \leftarrow M$
end if
$L \leftarrow |P|$
Construct effective key bytes $K_{\text{eff}}[0..L-1]$ as follows:
  for $i = 0..L-1$: $K_{\text{eff}}[i] \leftarrow K[i \bmod |K|]$
  for $j = L..|K|-1$: $K_{\text{eff}}[j \bmod L] \leftarrow K_{\text{eff}}[j \bmod L] \oplus K[j]$ ▷ fold key tail onto message domain
Allocate mixed$[L]$
for $i = 0$ to $L - 1$ **do**
    $\text{mixed}[i] \leftarrow (P[i] + K_{\text{eff}}[i]) \mod 256$
end for
return mixed

6.3 Core Function *h_core*

Algorithm 4 *h_core*

Require: Hash state h, block value m, parameters $(k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$
Ensure: Output u

$\text{MASK}_{256} \leftarrow 2^{256} - 1$
▷ Step 1: Polynomial layer in $\mathbb{Z}_{P_2}$
$X \leftarrow (m + k_{\text{high}}) \bmod P_2$
$Y \leftarrow (h + k_{\text{low}}) \bmod P_2$
$\alpha \leftarrow ((h + X)^2 + (m - Y)^3) \bmod P_2$
$\beta \leftarrow (X - Y) \bmod P_2$
▷ Step 2: 256-bit diffusion + keyed mixing
$a \leftarrow \alpha \wedge \text{MASK}_{256}$; $b \leftarrow \beta \wedge \text{MASK}_{256}$
$a' \leftarrow a \oplus (((b \gg_{256} 17) \ \vee \ (a \ll_{256} 239)) \wedge \text{MASK}_{256})$
$b' \leftarrow b \oplus (((a \ll_{256} 17) \ \vee \ (b \gg_{256} 239)) \wedge \text{MASK}_{256})$
$C_1 \leftarrow$ `0xB7E151628AED2A6ABF7158809CF4F3C7`
$C_2 \leftarrow$ `0x9E3779B97F4A7C15F39CC0605CEDC834`
$k_1 \leftarrow (k_{\text{mix}} \oplus C_1 \oplus (a' \ll_{256} 17)) \wedge \text{MASK}_{256}$
$k_2 \leftarrow (k_{\text{mix2}} \oplus C_2 \oplus (b' \gg_{256} 239)) \wedge \text{MASK}_{256}$
$\text{lin} \leftarrow (k_1 \lll_{256} 127) \oplus (k_2 \lll_{256} 63) \oplus (a' \lll_{256} 31) \oplus (b' \lll_{256} 15)$
$\text{hard} \leftarrow (\neg(k_{\text{mix}} \oplus k_{\text{mix2}} \oplus a \oplus b)) \wedge \text{MASK}_{256}$
$\text{temp} \leftarrow ((h \oplus m) \wedge \text{MASK}_{256}) \oplus \text{lin} \oplus \text{hard}$
▷ Step 3: Return to $\mathbb{Z}_{P_2}$
$u \leftarrow (h + \text{temp}) \bmod P_2$
return u

6.4 Coefficient Extraction

Algorithm 5 DeriveRS

Require: u (integer)
Ensure: (r, s)

$\text{MASK}_{256} \leftarrow 2^{256} - 1$
$u_{256} \leftarrow u \wedge \text{MASK}_{256}$
$\text{bytes}[0..31] \leftarrow \text{LE\_encode}_{32}(u_{256})$
$r_{\text{raw}} \leftarrow \text{LE\_to\_int}(\text{bytes}[0..15])$
$\text{CLAMP} \leftarrow$ `0x0ffffffc0ffffffc0ffffffc0fffffff`
$r \leftarrow r_{\text{raw}} \wedge \text{CLAMP}$
$s \leftarrow \text{LE\_to\_int}(\text{bytes}[16..31]) \mod 2^{128}$
return (r, s)

6.5 Main Algorithm

Algorithm 6 HardPoly1305_V2_Lite_Tag

Require: Key K ($|K| \geq 32$ bytes), Message M
Ensure: 16-byte tag

▷ 1. Derive internal parameters
$(k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}}) \leftarrow \text{DeriveKeyParams}(K)$
▷ 2. Mix message with key
$\text{mixed} \leftarrow \text{MixKeyAndMessage}(M, K)$
▷ 3. Split into 16-byte blocks with 0x01 padding
$\text{blocks} \leftarrow []$
$\text{offset} \leftarrow 0$
while $\text{offset} < |\text{mixed}|$ **do**
    $\text{block} \leftarrow \text{mixed}[\text{offset}..\text{offset} + 15]$
    $\text{offset} \leftarrow \text{offset} + 16$
    $\text{block} \leftarrow \text{block} \| \texttt{0x01}$
    $m_i \leftarrow \text{LE\_to\_int}(\text{block})$
    Append m_i to blocks

end while
▷ 4. Main iteration
$h \leftarrow 0$
for each m_i in blocks **do**
    $u_i \leftarrow h\_core(h, m_i, k_{\text{high}}, k_{\text{low}}, k_{\text{mix}}, k_{\text{mix2}})$
    $(r_i, s_i) \leftarrow \text{DeriveRS}(u_i)$
    $h_{\text{temp}} \leftarrow (h + m_i) \mod P_1$
    $h \leftarrow (r_i \cdot h_{\text{temp}} + s_i) \mod P_1$
end for
▷ 5. Output 128-bit tag
$\text{tag\_int} \leftarrow h \mod 2^{128}$
$\text{tag\_bytes} \leftarrow \text{LE\_encode}_{16}(\text{tag\_int})$
return tag_bytes

7 Differential Experiment Results

To provide limited sanity checks for *h_core*, we conducted differential sampling experiments. These tests do not establish PRF security or rule out structural attacks.

7.1 Experimental Setup

For each differential configuration $(\Delta h, \Delta m)$:

1. Generate pseudorandom 32-byte key K
2. Sample 500,000 random (h, m) pairs
3. Compute $u = h\_core(h, m)$ and $u' = h\_core(h \oplus \Delta h, m')$ where m' has XOR differential Δm
4. Record $\Delta u = u \oplus u'$

7.2 Results

Configuration	**Distinct Δu**	$\Pr[\Delta u = 0]$	**Collisions**
$\Delta h = 1, \Delta m = 0$	500,000	0.000000	0
$\Delta h = 0, \Delta m = 2^0$	500,000	0.000000	0
$\Delta h = 1, \Delta m = 2^{17}$	500,000	0.000000	0

Table 1: Differential experiment results for *h_core*

7.3 Bit Bias Analysis

For the first 32 bits of Δu, the probability of being 1 ranges from approximately 0.02 to 0.51. The deviation from 0.5 is concentrated in lower bits, consistent with expected ARX addition carry chain behavior rather than fixed high-probability differential vectors.

Conclusion: No obvious differential structure was observed at the tested scale; this does not constitute a security proof.

8 Limitations and Open Problems

- There is no standard-model proof that the concrete *h_core* is a PRF; this is an explicit assumption.
- Independence of per-block (r_i, s_i) is an idealized modeling step and is not proven.
- The analysis does not cover side-channel resistance, related-key attacks, or multi-user security.

- This construction is non-standard and should not be deployed without substantial third-party cryptanalysis.
- **Optional variant (future work).** If additional hardening is desired without changing the outer Poly1305-shaped loop, one can let the per-block state-derived values $H_1 = h \bmod P_1$ and $X_1 = m \bmod P_1$ participate directly in the bit-mixing stage (e.g., by XORing fixed rotations of H_1 and X_1 into the masks or into the intermediate words before the final rotate-XOR diffusion). This is a simple way to make the bit-layer masks input-dependent rather than purely key-derived constants; it is not analyzed in this document and is left as an engineering variant.

9 Conclusion

HardPoly1305 V2-Lite represents a targeted enhancement to the Poly1305 MAC algorithm:

1. **Preserved Framework:** The polynomial MAC structure over $\mathbb{Z}_{P_1}$ remains intact.
2. **Key Modification:** Per-block (r_i, s_i) coefficients generated via keyed function *h_core* replace fixed (r, s).
3. **Security Rationale:** Intended to disrupt the single-variable algebraic structure; under the PRF model, coefficients are treated as pseudorandom.
4. **Conditional Analysis:** UF-CMA-style bounds are derived under PRF and idealized extraction assumptions.
5. **Empirical Sanity Checks:** Differential experiments on *h_core* are limited and do not constitute proofs.

This work explores patch-level modifications to established cryptographic primitives; further third-party cryptanalysis is required before any real-world deployment.

Clarification: This is Technical ACGN VTuber Erika

A Experimental Methods and Small Distinguishers (Reproducibility Appendix)

This appendix documents the small "PPT-ish" distinguishers and sanity experiments used in this project. These are engineering tests, not security proofs: a failure is a red flag; passing does not imply PRF security.

A.1 Code artifacts

- **Differential distribution sampler:** `hardpoly1035_v2_delta.py`
- **Basic distinguisher harness (train/holdout):** `hardpoly1305_v2_test.py`

A.2 Metrics and train/holdout discipline

For a Bernoulli statistic with baseline $p_0 = 1/2$ (e.g. "output bit is 1"), we report:

$$\text{bias} := |p - 1/2|, \qquad z := \frac{p - 1/2}{\sqrt{0.25/N}},$$

where N is the number of samples on the corresponding split. When searching over many candidate masks/deltas on the train split, we report a rough calibration for the expected extreme bias under pure noise:

$$\text{expected\_extreme} \approx \sqrt{\frac{\ln(2M)}{2N}},$$

where M is the number of candidate hypotheses searched (e.g. number of masks or (δ, bit) pairs). We then validate the best train hypothesis on a disjoint holdout split to avoid winner's-curse overfitting.

A.3 Linear mask search (1-bit vs 1-bit, train/holdout)

This is the `--only-linear` mode in `hardpoly1305_v2_test.py`. We sample a dataset of (inp, y) pairs under a fixed random key and split it into train/holdout. The distinguisher family searched is:

$$\text{predict} = \text{parity}(\text{inp}_{b_{\text{in}}}) \ \oplus \ \text{parity}(y_{b_{\text{out}}}),$$

where b_{in} is one input bit index (in the chosen input space) and b_{out} is one output bit index of the 256-bit output material. The search selects $(b_{\text{in}}, b_{\text{out}})$ maximizing train bias and reports holdout bias for that pair.

A.4 Differential distinguisher search (train/holdout; XOR and add/sub models)

This is the `--only-diff-search` mode in `hardpoly1305_v2_test.py`. We search over a set of candidate input deltas δ (mostly single-bit, with some multi-bit deltas), compute output differences either as XOR ($\Delta u = u' \oplus u$) or as subtraction in $\mathbb{Z}_{P_2}$ ($\Delta u = (u' - u) \bmod P_2$), pick the output bit with maximum train bias, then validate on holdout.

A.5 Integral (cube XOR-sum) distinguisher search (train/holdout)

This is the `--only-integral-hard` mode in `hardpoly1305_v2_test.py`. For a chosen active-bit set S of size k, for each basepoint we enumerate the 2^k-cube and compute the XOR-sum

$$\text{acc} := \bigoplus_{u \in \{0,1\}^k} f\big(z \oplus (u \cdot S)\big).$$

For a random function, each output bit of acc is 0 with probability $\approx 1/2$. We search over candidate active sets on train, then validate the best (set, bit) choice on holdout.

A.6 Second-derivative "affine smell" diagnostic

This is the `--only-second-deriv` mode in `hardpoly1305_v2_test.py`. For an affine function f over XOR, the second derivative is identically zero:

$$f(z) \oplus f(z \oplus a) \oplus f(z \oplus b) \oplus f(z \oplus a \oplus b) = 0.$$

The harness estimates the empirical zero-rate (while separating the trivial $a = b$ cancellation case).

A.7 Truncated-view differential search

This is the `--only-trunc-basic` mode in `hardpoly1305_v2_test.py`. We repeat the same train/holdout differential search, but the distinguisher only "observes" a subset of output bits: low 64 bits, low 128 bits, or the Poly1305 clamped-r bit positions.

A.8 2-minute budget preset outputs (collected logs)

The following verbatim blocks are the captured outputs for the "`--2m`" preset runs.

_2m_second_deriv.txt

```
=== 2nd-derivative zero DIAGNOSE ===
mode=x width=136 full_width=False
keys=8 points_per_key=8 trials_per_point=128
total=8192 zeros=0 rate=0.000000e+00
[single-bit a,b] total=4003 zeros=0 rate=0.000000e+00
  [single-bit a==b] total=32 zeros=0 rate=0.000000e+00
  [single-bit a!=b] total=4003 zeros=0 rate=0.000000e+00
[multi-bit (else)] total=4189 zeros=0 rate=0.000000e+00

=== 2nd-derivative zero DIAGNOSE ===
mode=h width=130 full_width=False
keys=8 points_per_key=8 trials_per_point=128
total=8192 zeros=0 rate=0.000000e+00
[single-bit a,b] total=3971 zeros=0 rate=0.000000e+00
  [single-bit a==b] total=28 zeros=0 rate=0.000000e+00
  [single-bit a!=b] total=3971 zeros=0 rate=0.000000e+00
[multi-bit (else)] total=4221 zeros=0 rate=0.000000e+00
```

_2m_linear.txt

```
=== Linear distinguisher mask-search (train/holdout) ===
mode=x width=136 dataset_size=8000 mask_trials=8000 key_samples=2

-- key[0] --
train: N=4000  best_bias=0.029000  p1=0.471000  z=-3.67  expected_extreme~0.034786
test : N=4000   bias=0.012500  p1=0.512500  z=+1.58
best_in_bit  = 88
best_out_bit = 177

-- key[1] --
train: N=4000  best_bias=0.030000  p1=0.470000  z=-3.79  expected_extreme~0.034786
test : N=4000   bias=0.003000  p1=0.497000  z=-0.38
best_in_bit  = 83
best_out_bit = 141

=== Linear distinguisher mask-search (train/holdout) ===
mode=h width=130 dataset_size=8000 mask_trials=8000 key_samples=2

-- key[0] --
train: N=4000  best_bias=0.031500  p1=0.531500  z=+3.98  expected_extreme~0.034786
test : N=4000   bias=0.004500  p1=0.495500  z=-0.57
best_in_bit  = 100
best_out_bit = 238

-- key[1] --
train: N=4000  best_bias=0.028500  p1=0.471500  z=-3.60  expected_extreme~0.034786
test : N=4000   bias=0.005000  p1=0.505000  z=+0.63
best_in_bit  = 58
best_out_bit = 249
```

_2m_diff.txt

```
=== Differential distinguisher search (train/holdout) ===
mode=x width=136 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96
```

```
-- key[0] --
train: N=300  best_bias=0.130000  p1=0.370000  z=-4.50  expected_extreme~0.134181
test : N=300   bias=0.023333  p1=0.523333  z=+0.81
best_delta = 0x10000000000000000000000000000000
best_out_bit = 130

-- key[1] --
train: N=300  best_bias=0.103333  p1=0.396667  z=-3.58  expected_extreme~0.134181
test : N=300   bias=0.036667  p1=0.536667  z=+1.27
best_delta = 0x80000000000000000000000000000000
best_out_bit = 136

=== Differential distinguisher search (train/holdout) ===
mode=h width=130 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.130000  p1=0.630000  z=+4.50  expected_extreme~0.134181
test : N=300   bias=0.066667  p1=0.433333  z=-2.31
best_delta = 0x200000000000000000000
best_out_bit = 12

-- key[1] --
train: N=300  best_bias=0.116667  p1=0.616667  z=+4.04  expected_extreme~0.134181
test : N=300   bias=0.003333  p1=0.496667  z=-0.12
best_delta = 0x800000000000000000
best_out_bit = 90

=== Differential distinguisher search (train/holdout) ===
mode=x width=136 diff_type=add out_diff=sub
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.116667  p1=0.616667  z=+4.04  expected_extreme~0.134181
test : N=300   bias=0.030000  p1=0.530000  z=+1.04
best_delta = 0x40000000000000000000
best_out_bit = 66

-- key[1] --
train: N=300  best_bias=0.116667  p1=0.383333  z=-4.04  expected_extreme~0.134181
test : N=300   bias=0.043333  p1=0.456667  z=-1.50
best_delta = 0xaebc9838d0c365a13fa0cdfd498054ab1c
best_out_bit = 66

=== Differential distinguisher search (train/holdout) ===
mode=h width=130 diff_type=add out_diff=sub
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.120000  p1=0.380000  z=-4.16  expected_extreme~0.134181
test : N=300   bias=0.000000  p1=0.500000  z=+0.00
best_delta = 0x10000000000000000000000000000000
best_out_bit = 50

-- key[1] --
```

```
train: N=300  best_bias=0.130000  p1=0.630000  z=+4.50  expected_extreme~0.134181
test : N=300   bias=0.020000  p1=0.480000  z=-0.69
best_delta = 0x6b9effbdb09b1f84b8418184512a29a1
best_out_bit = 2
```

_2m_integral.txt

```
=== Integral distinguisher (cube XOR-sum) search (train/holdout) ===
mode=x width=136 k_active=9 candidate_sets=12 train_basepoints=24 test_basepoints=24 key_samples=2

-- key[0] --
train: N=24  best_bias=0.416667  P0=0.083333  z=-4.08  expected_extreme~0.426303
test : N=24  bias=0.083333  P0=0.583333  z=+0.82
best_set_idx=3  best_out_bit=207

-- key[1] --
train: N=24  best_bias=0.416667  P0=0.916667  z=+4.08  expected_extreme~0.426303
test : N=24  bias=0.000000  P0=0.500000  z=+0.00
best_set_idx=7  best_out_bit=198

=== Integral distinguisher (cube XOR-sum) search (train/holdout) ===
mode=h width=130 k_active=8 candidate_sets=12 train_basepoints=24 test_basepoints=24 key_samples=2

-- key[0] --
train: N=24  best_bias=0.416667  P0=0.083333  z=-4.08  expected_extreme~0.426303
test : N=24  bias=0.083333  P0=0.583333  z=+0.82
best_set_idx=7  best_out_bit=5

-- key[1] --
train: N=24  best_bias=0.375000  P0=0.875000  z=+3.67  expected_extreme~0.426303
test : N=24  bias=0.083333  P0=0.416667  z=-0.82
best_set_idx=3  best_out_bit=174
```

_hcore_trunc_basic.txt

```
==================== TRUNC VIEW: low64 (|bits|=64) ====================

=== Differential distinguisher search (train/holdout) ===
mode=x width=136 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.103333  p1=0.603333  z=+3.58  expected_extreme~0.125276
test : N=300   bias=0.033333  p1=0.533333  z=+1.15
best_delta = 0x2000000
best_out_bit = 33

-- key[1] --
train: N=300  best_bias=0.106667  p1=0.606667  z=+3.70  expected_extreme~0.125276
test : N=300   bias=0.003333  p1=0.496667  z=-0.12
best_delta = 0x100000000000000000000000
best_out_bit = 43

=== Differential distinguisher search (train/holdout) ===
mode=h width=130 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96
```

```
-- key[0] --
train: N=300  best_bias=0.116667  p1=0.616667  z=+4.04  expected_extreme~0.125276
test : N=300   bias=0.000000  p1=0.500000  z=+0.00
best_delta = 0x200000000
best_out_bit = 34

-- key[1] --
train: N=300  best_bias=0.110000  p1=0.390000  z=-3.81  expected_extreme~0.125276
test : N=300   bias=0.020000  p1=0.520000  z=+0.69
best_delta = 0x100000000000000000000
best_out_bit = 3

=================== TRUNC VIEW: low128 (|bits|=128) ===================

=== Differential distinguisher search (train/holdout) ===
mode=x width=136 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.106667  p1=0.393333  z=-3.70  expected_extreme~0.129805
test : N=300   bias=0.013333  p1=0.486667  z=-0.46
best_delta = 0x100000000000000
best_out_bit = 38

-- key[1] --
train: N=300  best_bias=0.103333  p1=0.603333  z=+3.58  expected_extreme~0.129805
test : N=300   bias=0.013333  p1=0.513333  z=+0.46
best_delta = 0x400000000000000
best_out_bit = 82

=== Differential distinguisher search (train/holdout) ===
mode=h width=130 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.116667  p1=0.383333  z=-4.04  expected_extreme~0.129805
test : N=300   bias=0.000000  p1=0.500000  z=+0.00
best_delta = 0x175c0ca85c4db1f7beacb35d41c9a4ab8
best_out_bit = 32

-- key[1] --
train: N=300  best_bias=0.116667  p1=0.616667  z=+4.04  expected_extreme~0.129805
test : N=300   bias=0.003333  p1=0.496667  z=-0.12
best_delta = 0x400000000000000000000000
best_out_bit = 73

=================== TRUNC VIEW: clamp_r_bits (|bits|=106) ===================

=== Differential distinguisher search (train/holdout) ===
mode=x width=136 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.116667  p1=0.616667  z=+4.04  expected_extreme~0.128588
test : N=300   bias=0.020000  p1=0.480000  z=-0.69
best_delta = 0x2be94a512ec74ddd7e63a0514492b9c036
```

```
best_out_bit = 5

-- key[1] --
train: N=300  best_bias=0.123333  p1=0.623333  z=+4.27  expected_extreme~0.128588
test : N=300   bias=0.020000  p1=0.480000  z=-0.69
best_delta = 0x10000000000000000000000000000000
best_out_bit = 119

=== Differential distinguisher search (train/holdout) ===
mode=h width=130 diff_type=xor out_diff=xor
key_samples=2 train=300 test=300 deltas=96

-- key[0] --
train: N=300  best_bias=0.116667  p1=0.383333  z=-4.04  expected_extreme~0.128588
test : N=300   bias=0.003333  p1=0.503333  z=+0.12
best_delta = 0x2928cd534d7aca78ef6f8c32342b4fecd
best_out_bit = 118

-- key[1] --
train: N=300  best_bias=0.103333  p1=0.603333  z=+3.58  expected_extreme~0.128588
test : N=300   bias=0.003333  p1=0.496667  z=-0.12
best_delta = 0x100000000000000000000000
best_out_bit = 11
```

Author's Note by Erika

Hi everyone! This is your Technical ACGN VTuber **Erika**!

I've explored a variant of the classic Poly1305 MAC algorithm. The original math was elegant, but I wanted to examine what happens if we reduce the single fixed-key structure. This is exploratory work rather than a production-ready claim.

I've brought along my Assistant to help translate my "Erika-style" explanation into rigorous cryptographic language for the paper below. Enjoy the fusion of vibes and vectors!

B FAQ (Non-normative) and Discussion: Erika Design Rationale and Limitations

Erika: Okay, story time! The math up there is a bit scary, so I summoned my Assistant for the speedrun translation.

B.1 Design Story and Inspiration

Chats: Erika, what's the deal with this doc? Is it math-only, or can we get the human-language version?

Erika: Okay, let me set the tone like my stream intro: first half is for the people who *enjoy* primes, the second half is for the people who just want to know "what changed" without getting flashbanged by algebra.

Erika Assistant: Translation: this document is intentionally split-view. The math sections stay formal; this corner explains the same design choices in plain English, without changing the semantics.

Chats: So why did you design HardPoly1305 in the first place?

Erika: My motivation is honestly pretty plain: I don't fully trust that "black-box security comfort" feeling.

Erika: Before I started, I deliberately read a lot of simpler, more "white-box"-feeling designs—things with AX / ARX-style structure—for a practical reason: the simpler the structure, the easier it is to take it apart and explain what each piece is responsible for.

Erika: Meanwhile, some schemes that look like "the paper is complete and the tests are thorough" often leave me with a different vibe: the claimed safety depends on a whole complex process and a lot of experience-driven validation. And validation does not automatically mean it can be automated.

Erika: I am naturally uncomfortable with anything that "cannot be fully automated". If machines cannot run it end-to-end, cannot systematically stress-test it, cannot search it for edge cases—then are we really expecting human brains to guarantee it will stand for years, and also survive every weird real-world attack surface? In engineering terms, that starts to feel a bit mystical.

Erika: Not "definitely insecure"—just: it does not give me that grounded feeling of "I understand why this is secure".

Erika: So I went back and studied the original Poly1305 seriously: its structure, its mathematical form, the papers around it, and then the implementation details. After that, I even asked internet sources and AI to explain the Poly1305 skeleton back to me, just to confirm my understanding was not a hallucination.

Erika: And then the question that popped into my head was not "wow, elegant"—it was a more nervous engineering question:

Erika: If we always organize it with the same parameters, in the same way, do some usage scenarios naturally create risk around "replay" or "state reuse"? In real systems, the failure point is often not the algorithm itself, but how parameters are managed, stored, reused, or accidentally misused.

Erika: So I tried an almost counter-intuitive direction: if parameter storage / parameter reuse is a potential risk point, can we design the thing to be *not worth storing*?

Erika: More precisely: make the parameters dynamic and derived, so after one use they are meaningless, and they should not be held as static long-lived values.

Erika: In other words, I don't want implementers to have a comfortable "just keep using the same key coefficients forever" zone. I want to keep Poly1305's clear skeleton, but make the critical parameters "change per block / per use, derived from a controlled source, and not meant to be statically held"—so it behaves like a rolling process, not a fixed knob you can keep in your pocket.

Erika Assistant: Concretely: HardPoly1305 V2-Lite keeps the Poly1305-shaped outer accumulation in P_1 and the same 16-byte tag output, but replaces "one dial" with a per-block coefficient schedule (r_i, s_i) derived from a keyed, evolving core state. That makes naive cut-and-paste or "same block, different context" reuse far less compatible with the polynomial chain.

Erika Assistant: In short: this is not about showing off or inventing a new toy. It is an engineering instinct: discourage parameter reuse by design, shrink the "replay/reuse" window in practice, and do it in a way that stays implementable and explainable.

B.2 Chats: Why Per-Block (r_i, s_i)?

Chats: "Why make life complicated? Why not just use one (r, s) like the original?"

Erika: "Because I want a patch that keeps the Poly1305 chassis, but makes the algebraic surface less "single-equation"."

Erika Assistant: "TL;DR: one fixed r gives one big equation. With per-block (r_i, s_i), the hash expands into a chain like $r_n r_{n-1} \cdots r_1$, and each r_i depends on the previous state. No single variable to solve for, no easy algebraic shortcut."

Chats: "So basically, instead of one master key for the whole message, we change the lock on every single block?"

Erika: "Yes. New lock per block is the vibe."

Erika Assistant: "Exactly. Think of it as new dice per block. Reusing a block without the same prefix is like trying yesterday's key in today's lock."

Chats: "Okay but what do you mean by *depends on the previous state*? Like, in code terms."

Erika: "It means position matters because the running state matters."

Erika Assistant: "In `hardpoly1305_v2_lite_tag`, the next coefficients come from `u_i = h_core(h, m_i, ...)`. That `h` is the running accumulator. So the same plaintext block placed at a different position (different prefix, different `h`) generates a different `u_i`, thus a different (r_i, s_i)."

Chats: "And the clamp? You still clamp r_i like Poly1305 does. Cosmetic, or actually important?"

Erika: "Not cosmetic. It's me keeping the guardrails while I add the new scheduler."

Erika Assistant: "It is compatibility discipline. The clamp keeps r_i inside the classic Poly1305 "safe shape" (certain bits cleared) so the multiplication behavior stays well-behaved and closer to the original engineering assumptions. V2-Lite is trying to *add* a scheduler, not remove the guardrails."

B.3 Chats: Is This a New MAC or a Poly1305 Patch?

Chats: "Did we just invent a new monster?"

Erika: "No new monster. Same outer engine, different upstream fuel."

Erika Assistant: "No, it is a **patch-level modification**. Same engine: $h \leftarrow r \cdot (h + m) + s \pmod{P_1}$. We just swap where the fuel (coefficients) comes from. The security analysis still fits within the 'polynomial MAC + PRF' paradigm."

Chats: "So what stayed the same on purpose?"

Erika: "I kept the chassis recognizable on purpose."

Erika Assistant: "Three things stayed intentionally familiar: (1) the outer update still lives in $P_1 = 2^{130} - 5$, (2) blocks are still 16 bytes with a forced `0x01` terminator, and (3) the output is still a 16-byte tag. The novelty is upstream: `derive_key_params` and `h_core` decide a fresh (r_i, s_i) per block."

B.4 Chats: Will It Be Slow?

Chats: "My CPU fans are spinning up just thinking about it."

Erika: "Fair. It costs more per block, but it's not 'instant jet engine' unless you're hashing a planet."

Erika Assistant: "It is $O(\text{blocks})$, same as the original. The constant factor is higher—maybe 2-3x slower because of the extra `h_core` math per block. For chat messages, API keys, or game packets, you will not notice. For a 4TB backup, bring snacks."

Chats: "Give me the streamer-grade mental model. Where does the extra cost come from?"

Erika: "You're paying for the extra 'mixer scene' per block."

Erika Assistant: "Poly1305 per block is basically: add a block, multiply by a clamped r, reduce mod P_1. V2-Lite adds a little "keyed mixer scene" per block: a couple of modular exponentiations (square/cube), plus cross-bit diffusion and rotate/XOR/NOT mixing inside 256-bit masking. Still linear in blocks, just a fatter block function."

B.5 Chats: What if *h_core* is bad?

Chats: "What if *h_core* is actually predictable?"

Erika: "Then the whole 'scheduler' idea can collapse. I'm not claiming a standard-proof miracle here."

Erika Assistant: "Then the conditional security argument no longer applies. Differential experiments are only limited sanity checks. A high-entropy master key is necessary, but robustness beyond standard Poly1305 remains unproven."

Chats: "So what is the honest claim?"

Erika: "Honest claim: it's a sketch and an experiment, not a standard."

Erika Assistant: "Honest claim: V2-Lite is a design sketch that keeps the Poly1305 outer skeleton and explores a keyed, stateful coefficient generator. If you treat `h_core` as a strong PRF, the intuition is nice; if `h_core` has structure, the whole scheduler can leak it into (r_i, s_i). This is why standardized constructions **like AEADs that already include Poly1305** remain the default for production."

B.6 Chats: Do I Still Need Nonces or Context?

Chats: "Okay, final boss question: if this is "harder", can I stop caring about nonces?"

Erika: "Nope. Protocol hygiene is still the final boss."

Erika Assistant: "No. A MAC authenticates; it does not automatically prevent replay. Protocol context (nonces, counters, session binding, transcript hashing, domain separation) is still how you stop an attacker from re-sending a previously valid (M, Tag) pair. Harder algebra does not replace protocol hygiene."

B.7 Chats: Why Pre-Mix the Message at All?

Chats: "That `mix_key_and_message` step looks like seasoning because it feels cool. Is it actually doing anything?"

Erika: "It's seasoning, yes—but with a purpose: make tiny inputs stop being too 'polite'."

Erika Assistant: "It is a lightweight, keyed preconditioning layer. In the code: each byte becomes $(m[i] + key[j]) \bmod 256$, and if the message is shorter than 32 bytes, the tail of the key is appended so the core always sees at least 32 bytes of key-dependent material. It is not meant to be a full KDF; it is meant to prevent "tiny message, tiny influence" situations before the nonlinear core even starts."

Chats: "So: it does not replace the core, it just makes the input less polite?"

Erika: "Exactly."

Erika Assistant: "Exactly. It makes short, structured inputs stop looking like a clean lab sample."

B.8 Chats: Why a Second Prime P_2 (and Why Safe Prime)?

Chats: "We already have the famous $P_1 = 2^{130} - 5$. Why introduce P_2 and this whole safe-prime arc?"

Erika: "P_1 is the stage; P_2 is the backstage."

Erika Assistant: "Because `h_core` is doing a different job than the outer polynomial update. The outer layer wants the classical Poly1305 field. The core wants a large, clean mixing domain where we can stir state + block + key, do bit tricks with a fixed width (`MASK_256`), and then map back to a field element. A 256-bit safe prime gives a well-behaved multiplicative group and reduces weird small-subgroup pathologies when you treat the output as a pseudorandom source to be sliced into (r_i, s_i)."

Chats: "Translation: P_1 is the stage, P_2 is the backstage where we change outfits?"

Erika: "Yes."

Erika Assistant: "Yes. Same show, different room for the costume swap."

B.9 Chats: Why the cross-bit mixing and the rotations?

Chats: "Explain like I'm live on stream: why the cross-bit mixing and why those rotations specifically?"

Erika: "It's cheap 'stop being predictable' glue, not magic."

Erika Assistant: "The short version: 256 bits is *big*. If the diffusion layer is too simple, it tends to remain a sparse linear transform in $\mathrm{GF}(2)^{256}$ with a very structured dependency matrix. That kind of structure is exactly what makes basic linear/affine modeling cheap (even if the attacker never sees internal wires, a sparse transform tends to leak through correlations and truncations).

So we intentionally make the 256-bit diffusion look a bit "awkward" but still cheap:

- **Cross-bit coupling (break locality).** We do not only rotate/XOR a single word. We cross-couple α_{bits} and β_{bits} using shift/OR wiring so each output bit depends on bits from *two* sources and from far-away positions. This quickly destroys the "local neighborhood" dependency graph that a one-rotation design often keeps.
- **Multiple off-center rotations (spread positions).** Rotations like 127, 63, 31, 15 are not magic constants; the point is that they are off-center, not aligned to 32/64/128. In practice this helps the linear map become "dense" faster: a single-bit change is pushed into distant positions and then combined by XOR with other rotated views.
- **Complemented XOR term (avoid obvious cancellations).** The final XOR-with-NOT expression is still affine/linear for fixed key, but it helps avoid trivial cancellation patterns that show up when the linear part is too symmetric.

This is not a proof of a diffusion metric; it is an engineering choice aimed at making the induced linear transform on 256-bit words less sparse and less "eyeballable" under truncated views."

Chats: "So it's not magic. It's "stop being predictable" glue?"

Erika: "Exactly."

Erika Assistant: "Exactly. Glue, not a proof."

B.10 Chats: How Do I Sanity-Test This Without Lying to Myself?

Chats: "What should I do if I want to test it like a responsible adult, not like a copium goblin?"

Erika: "Tests for bugs, not a coronation for 'secure'."

Erika Assistant: "Treat tests as *correctness* and *basic behavior* checks, not security proofs. Good sanity checks are: determinism (same input, same tag), sensitivity (flip one bit in message/key, tag changes), boundary coverage (0 bytes, 1 byte, 15/16/17 bytes, long messages), and random regression. Your Python file already does this in the built-in test suite. Beyond that, real confidence comes from cryptanalysis, not from more unit tests."

Chats: "Got it. Tests keep us honest about bugs. They do not crown it secure."

B.11 Chats: Did I Just Destroy Poly1305 Speed?

Chats: "Be honest: is V2-Lite slower than Poly1305?"

Erika: "Yes. You pay a constant factor for the per-block mixer."

Erika Assistant: "Yes, slower per block. Poly1305's inner loop is mostly one multiply-and-reduce in P_1. V2-Lite keeps that outer shape, but adds a per-block coefficient generator (`h_core` plus extraction of (r_i, s_i)), which is extra math and extra bit mixing. It is still $O(\text{blocks})$; you are paying a constant factor to make the coefficients stateful and less structured."

Chats: "So the trade is: spend some cycles to buy a messier algebraic surface?"

Erika: "Exactly."

Erika Assistant: "Exactly. And you should benchmark it in the deployment language; Python is a worst-case lens for per-block big-integer work."

B.12 Chats: You Said This is ARX-ish. Where are the "A", "R", and "X"?

Chats: "You keep saying ARX, but I stared at the code and I can't see where the *Addition*, the *Rotation*, or the *XOR* is supposed to be. Where is it?"

Erika: "Also: ARX doesn't have an "S". But yes, we do have subtraction in the core."

Erika Assistant: "Tiny correction first: ARX is not "a class of block ciphers"—it is a *mixing pattern*: **A**ddition, **R**otation, e**X**clusive-or. But we should be precise about where "A" lives in this design.

In V2-Lite, the **Addition** is primarily provided by the *field arithmetic layer* (Layer 1 and the final modular addition in Layer 3), not by the 256-bit diffusion wiring itself. The diffusion layer is therefore best described as **RX-like** (rotations + XOR) with extra bitwise wiring (shift/OR, NOT) rather than a full ARX round function.

In `h_core` (see `HardPoly1305 V2 (Modified Twilight-Dream).py`), the mapping is:"

- **A (Addition):** carried mostly by Layer 1 and Layer 3: modular additions/subtractions in $\mathbb{Z}_{P_2}$ (offset injection and the square/cube polynomial) and the final $u = (h + \text{temp}) \bmod P_2$.
- **R (Rotation):** the 256-bit rotate-left by 127 implemented as shifts+OR: `rot = ((temp << 127) & MASK_256) | (temp >> (256-127))`.
- **X (XOR):** `temp ^= rot`, and also `temp = km ^ (...)`.

Chats: "And the *Subtraction*?"

Erika: "Right: ARX is just A+R+X. Subtraction is allowed in the larger "ish" story."

Erika Assistant: "ARX doesn't have an *S*. But you *do* have subtraction in the nonlinear α term: $(X1 - H1 + k\_low)^3$. So the core is really a hybrid:

- **Arithmetic layer ("A", and also subtraction):** modular add/sub and square/cube over $\mathbb{Z}_{P_2}$ (Layer 1) plus final modular addition (Layer 3).
- **Diffusion layer (RX-like + bitwise wiring):** rotations, XOR, shift/OR cross-coupling, and NOT/AND/OR masking in a fixed 256-bit domain (Layer 2).

That is why we call the whole core *ARX-ish* overall, but the diffusion sublayer specifically is better described as *RX-like*."

B.13 Chats: What is the Security Story Here (Diffusion/Confusion: which steps, which operators)?

Chats: "Okay, but if we say "diffusion" and "confusion", where exactly does each one come from? Give us the map."

Erika: "Fair. Story, not proof. Here's the map."

Erika Assistant: "Fair. This is the security *story*, not a proof. In V2-Lite, the intended "confusion" (key dependence) and "diffusion" (small change → widespread change) come from `h_core` plus the per-block coefficient schedule:"

- **Key injection (confusion):** `derive_key_params` produces $(k\_high, k\_low, k\_mix, k\_mix2)$, and these directly enter α and the bit-mixing path.
- **Nonlinear arithmetic mixing:** $(\alpha, \beta) \in \mathbb{Z}_{P_2}^2$ are computed from $X_2 = (m + k\_high) \bmod P_2$ and $Y_2 = (h + k\_low) \bmod P_2$ via $\alpha = ((h+X_2)^2 + (m-Y_2)^3) \bmod P_2$ and $\beta = (X_2 - Y_2) \bmod P_2$. The squaring/cubing provides the dominant algebraic nonlinearity over the field.
- **Bit-mixing (256-bit domain):** The 256-bit truncations α_bits, β_bits are combined with key material (k_mix, k_mix2) using cross-bit shifts/OR wiring, fixed rotations, XOR/NOT, and fixed constants (C_1, C_2).
- **State-coupling:** The intermediate word begins from $(h \oplus m)$ and is then mixed with the diffusion word and a complemented XOR term, so changes propagate into the per-block coefficient schedule.
- **Stateful scheduling:** u_i depends on the running accumulator h, so (r_i, s_i) change with the prefix. This is the "moving target" property: same block, different position ⇒ different coefficients.
- **Outer accumulation:** $h \leftarrow (r_i \cdot (h + m_i) + s_i) \bmod P_1$ feeds the mixed coefficients back into the next round's state, amplifying any earlier change across subsequent blocks.

Chats: "So it is a model."

Erika: "Yes. It's an assumption-driven story."

Erika Assistant: "Yes. The standard way to talk about it is: assume `h_core` behaves like a PRF over (h, m) under secret key-derived parameters. Under that assumption, per-block (r_i, s_i) look pseudorandom to an attacker, and the outer Poly1305-shaped MAC inherits a UF-CMA-style intuition. None of this is a substitute for real cryptanalysis, and it does not cover side channels, related-key, or multi-user proofs."

B.14 How dare you dabble in symmetric cryptography?

Chats: Erika, you're just an ordinary college student in the human world. How dare you dabble in symmetric cryptography?

Erika: Because the homework and code-project idea said "be creative" and I took that personally. I hate a black-box; I like white-box. This is my idea by creative! I don't think cryptography is "math magic"; I want to turn it into "un-black-box technology".

Erika Assistant: For the record: treat this as a learning/design exploration. If you need production-grade security, use standardized AEADs and proven constructions.

Chats: But your tweaked design uses a pseudorandom function. Does this violate your cryptographic design preferences?

Erika: No conflict. "White-box" is my attitude: I want to understand every moving part, not worship it. A PRF is just a tool—a well-studied mixing brick. I can still keep the design readable and explainable. And honestly, the motivation was simple: I spotted a pitfall in how Poly1305-style structure can be reasoned about or misused, so I tried to patch the surface.

Erika Assistant: My interpretation of Erika's meaning (in symmetric cryptography): this is engineering. A design only matters once it is implemented, tested, profiled, and reviewed. Erika's "white-box" preference is not "ignore math"; it is "make the mechanisms and assumptions explicit, and keep the moving parts explainable".

Erika Assistant: Terminology note (since Chats already used it correctly): PRF means "pseudorandom function". Using a PRF-like component internally does not make the construction a "black

box”—it means you lean on a standard assumption (indistinguishability from random) so the per-block (r_i, s_i) schedule is hard to predict or algebraically pin down.

C Python Implementation

```python
"""
HardPoly1305 V2-Lite
====================
A Poly1305 variant with per-block pseudorandom coefficients.

Security: UF-CMA under h_core PRF assumption
Input: 32-byte key, arbitrary-length message
Output: 16-byte MAC tag
"""

from __future__ import annotations
from dataclasses import dataclass
from typing import Tuple

# ============================================================
# Constants
# ============================================================

P1: int = (1 << 130) - 5                    # Poly1305 prime
P2: int = (1 << 256) - 188_069              # HardPoly1305 safe prime
MASK_256: int = (1 << 256) - 1              # 256-bit mask
CLAMP: int = 0x0ffffffc0ffffffc0ffffffc0fffffff  # Poly1305 r-clamp

# ============================================================
# Key Parameter Structure
# ============================================================

@dataclass(frozen=True)
class HardPoly1305KeyParams:
    """Internal parameters derived from master key."""
    k_high: int   # Offset in Z_P1 from key bytes [0..15]
    k_low: int    # Offset in Z_P1 from key bytes [16..31]
    k_mix: int    # Mixing parameter in Z_P2
    k_mix2: int   # Secondary mixing parameter in Z_P2


    from typing import Tuple

def rotl256(x: int, r: int) -> int:
    x &= MASK_256
    r &= 255
    if r == 0:
        return x
    return ((x << r) & MASK_256) | (x >> (256 - r))

def rotr256(x: int, r: int) -> int:
    x &= MASK_256
    r &= 255
    if r == 0:
        return x
    return (x >> r) | ((x << (256 - r)) & MASK_256)

# ==========================
# Utility: key normalization (xor-fold to 32 bytes)
# ==========================
def fold_key_32(key: bytes) -> bytes:
```

```
    """
    Fold an arbitrary-length key (>= 32 bytes) down to 32 bytes:
    - The first 32 bytes are the base.
    - Remaining bytes are XOR-folded into positions i % 32.

    Purpose:
    - Ensure keys longer than 32 bytes still contribute entropy/variation,
      while keeping the procedure constant-domain and low overhead.

    Note:
    - Different long keys can map to the same 32-byte result (equivalence classes).
      Users must be aware of this semantic.
    """
    if len(key) < 32:
        raise ValueError("key must be at least 32 bytes")
    if len(key) == 32:
        return key
    out = bytearray(key[0:32])
    for i, b in enumerate(key[32:]):
        out[i & 31] ^= b
    return bytes(out)


# ============================
# Utility: derive parameters from key
# ============================
def derive_key_params(key: bytes) -> Tuple[int, int, int, int]:
    """
    Derive parameters from a 32-byte key:
    - k_high, k_low: offsets modulo P1
    - k_mix, k_mix_2: mixing parameters modulo P2

    This is intentionally lightweight (not a heavy KDF).
    """
    key32 = fold_key_32(key)

    k_high = int.from_bytes(key32[0:16], "little") & MASK_256
    k_low  = int.from_bytes(key32[16:32], "little") & MASK_256

    k_mix = int.from_bytes(key32[0:32], "little") & MASK_256
    k_mix_2 = rotr256((~k_mix) & MASK_256, 1)

    return k_high, k_low, k_mix, k_mix_2


# ============================
# Utility: mix message & key (preprocessing)
# ============================
MIX_DELIMITER_BYTE = 0xA7  # fixed domain separator for short-message padding

def mix_key_and_message(message: bytes, key: bytes) -> bytes:
    """
    Ultra-lightweight preprocessing (constant-domain & fast):
    - If len(message) < 32, pad as: message || DELIM || 0x00... up to 32 bytes.
    - Then mix by per-byte addition: mixed[i] = (padded[i] + key_eff[i]) mod 256.
    - If len(message) >= 32, no padding is applied.

    Key usage rule for this preprocessing stage:
    - Let L = len(padded).
    - If len(key) < L: repeat key cyclically to length L.
    - If len(key) > L: fold the tail back into the first L bytes via XOR (tail XOR-fold).
```

```
        This lets key bytes beyond L still influence the mix output without appending raw key bytes.
        """
        if not key:
            raise ValueError("key must be non-empty")
        if len(key) < 32:
            raise ValueError("key length must be at least 32 bytes")

        if len(message) < 32:
            pad_len = 32 - len(message) - 1
            padded = message + bytes([MIX_DELIMITER_BYTE]) + (b"\x00" * pad_len)
        else:
            padded = message

        L = len(padded)

        # Build an effective key stream of length L.
        if len(key) >= L:
            key_eff = bytearray(key[0:L])
            # XOR-fold remaining key bytes back into [0..L-1]
            for j in range(L, len(key)):
                key_eff[j % L] ^= key[j]
        else:
            key_eff = bytearray(L)
            for i in range(L):
                key_eff[i] = key[i % len(key)]

        mixed = bytearray(L)
        for i, b in enumerate(padded):
            mixed[i] = (b + key_eff[i]) & 0xFF

        return bytes(mixed)


    # ============================
    # Utility: derive r_i, s_i from u
    # ============================
    def derive_r_s_from_u(u: int) -> Tuple[int, int]:
        """
        Interpret 256-bit u as 32 bytes (little-endian):
        - low  16 bytes -> r_raw, then apply Poly1305 clamp mask
        - high 16 bytes -> s (use as 128-bit offset)
        """
        u &= MASK_256
        b = u.to_bytes(32, "little")

        r_raw = int.from_bytes(b[0:16], "little")
        r = r_raw & 0x0ffffffc0ffffffc0ffffffc0fffffff  # Poly1305 clamp mask

        s = int.from_bytes(b[16:32], "little") & ((1 << 128) - 1)
        return r, s

    # ============================================================
    # Core Function: h_core
    # ============================================================

    def h_core(
        hash_value: int,
        block_value: int,
        params: HardPoly1305KeyParams,
    ) -> int:
        """
```

```
    Compute u = h_core(h, m) using:
    1. Polynomial alpha/beta in Z_P2
    2. Cross-bit diffusion in 256-bit domain
    3. Keyed rotation-XOR mixing + complemented XOR term
    4. Final combination in Z_P2
    """

    # Step 1: Polynomial layer in Z_P2
    X = (block_value + params.k_high) % P2
    Y = (hash_value + params.k_low) % P2
    alpha = ((hash_value + X) ** 2 + (block_value - Y) ** 3) % P2
    beta = (X - Y) % P2

    # Step 2: 256-bit diffusion (cross-bit)
    a = alpha & MASK_256
    b = beta & MASK_256
    a2 = (a ^ (((b >> 17) | (a << 239)) & MASK_256)) & MASK_256
    b2 = (b ^ (((a << 17) | (b >> 239)) & MASK_256)) & MASK_256

    # Step 3: keyed mixing + rotations (GF(2) affine/linear under fixed key)
    C1 = 0xB7E151628AED2A6ABF7158809CF4F3C7
    C2 = 0x9E3779B97F4A7C15F39CC0605CEDC834
    k1 = (params.k_mix ^ C1 ^ ((a2 << 17) & MASK_256)) & MASK_256
    k2 = (params.k_mix2 ^ C2 ^ ((b2 >> 239) & MASK_256)) & MASK_256
    rot127 = rotl256(k1, 127)
    rot63  = rotl256(k2, 63)
    rot31  = rotl256(a2, 31)
    rot15  = rotl256(b2, 15)
    lin = rot127 ^ rot63 ^ rot31 ^ rot15

    temp = (hash_value ^ block_value) & MASK_256
    hard = (~(params.k_mix ^ params.k_mix2 ^ a ^ b)) & MASK_256
    temp = (temp ^ lin ^ hard) & MASK_256

    # Step 4: Combine in Z_P2
    u = (hash_value + temp) % P2
    return u

# ============================================================
# Main MAC Function
# ============================================================

def hardpoly1305_v2_lite_tag(message: bytes, master_key: bytes) -> bytes:
    """
    Compute HardPoly1305 V2-Lite MAC tag.

    Args:
        message: Arbitrary-length message bytes
        master_key: 32-byte secret key (must be high-entropy)

    Returns:
        16-byte MAC tag
    """
    if len(master_key) < 32:
        raise ValueError("Key must be at least 32 bytes")

    # Derive parameters
    params = derive_key_parameters(master_key)

    # Mix message with key
    mixed = mix_key_and_message(message, master_key)

```

```python
    # Main iteration
    h = 0

    for offset in range(0, len(mixed), 16):
        block = mixed[offset:offset + 16]
        block += b"\x01"  # Poly1305 padding

        m_i = int.from_bytes(block, "little")

        # Generate per-block coefficients
        u_i = h_core(h, m_i, params)
        r_i, s_i = derive_r_s_from_u(u_i)

        # Poly1305-style update
        h_tmp = (h + m_i) % P1
        h = (r_i * h_tmp + s_i) % P1

    # Output 128-bit tag
    tag = (h % (1 << 128)).to_bytes(16, "little")
    return tag


# ============================================================
# Self-Test
# ============================================================

if __name__ == "__main__":
    import os

    print("HardPoly1305 V2-Lite Self-Test")
    print("=" * 50)

    key = bytes(range(32))
    test_messages = [
        b"",
        b"Hello, HardPoly1305!",
        b"A" * 16,
        b"A" * 31,
        b"A" * 32,
        b"A" * 100,
    ]

    for i, msg in enumerate(test_messages):
        tag = hardpoly1305_v2_lite_tag(msg, key)
        print(f"[{i}] len={len(msg):3d}, tag={tag.hex()}")

    # Consistency check
    rand_key = os.urandom(32)
    rand_msg = os.urandom(123)
    t1 = hardpoly1305_v2_lite_tag(rand_msg, rand_key)
    t2 = hardpoly1305_v2_lite_tag(rand_msg, rand_key)
    assert t1 == t2, "Consistency check failed"

    print("\nAll tests passed!")
```

D Differential Experiment Console

```
============================================================
HardPoly1305 h_core Delta-u Differential Experiments
============================================================
```

```
==== Differential Experiment: Delta h = 1, Delta m = 0 (XOR) ====
Samples: 500000
Distinct delta_u: 500000
P[delta_u = 0]: 0.000000
Top 10 most common delta_u (value, count, probability):
    du = 0xa8a60989aca1588425b41daa23780fa506128bfec41fdadc1ea0d926d3b24024, count = 1
    prob ~= 0.000002
    du = 0xd1251ce671492a4fbb78cdb1c9ddf1b027564f3b055569b67f785b3b9c7bca96, count = 1
    prob ~= 0.000002
    du = 0x6bf6760706c45a2dce9b1076f2d377791506d7d7b1e43e61098fc0dbef1fe3e7, count = 1
    prob ~= 0.000002
    du = 0xf2c4bfd58b44054ee93ed6c8ce3505f85c6c36245a17732c63fca43f04557cd1, count = 1
    prob ~= 0.000002
    du = 0xb71667738751b27d9b94371196a974195eb6febaae1a5f8878476c74e727dc30, count = 1
    prob ~= 0.000002
    du = 0x9e361145cc755b5f7dc3eb4ff73be5751c800f9d245d4007213f2c8916e0f34a, count = 1
    prob ~= 0.000002
    du = 0xe216131a096c775cd90c29ec430f51281936fd824c8e9356b994ed406215c697, count = 1
    prob ~= 0.000002
    du = 0x34f451c53ada818e229f3a36e9532cd0ad2fe80f0c17483f0d16274305ac74c2, count = 1
    prob ~= 0.000002
    du = 0x057fad48ac0f5f42ac366fdecdae373824de84153f2cb9f916ccf7e600b8e465, count = 1
    prob ~= 0.000002
    du = 0x17961759aa2962f90eef1421e19bf3846555d53a0232232a8f7610ffc168711e, count = 1
    prob ~= 0.000002
First 32 bit 1-probabilities:
    bit  0: 0.4997
    bit  1: 0.5004
    bit  2: 0.5000
    bit  3: 0.4998
    bit  4: 0.4999
    bit  5: 0.4998
    bit  6: 0.5002
    bit  7: 0.4994
    bit  8: 0.5001
    bit  9: 0.5002
    bit 10: 0.5002
    bit 11: 0.4990
    bit 12: 0.4989
    bit 13: 0.5001
    bit 14: 0.4988
    bit 15: 0.4998
    bit 16: 0.5001
    bit 17: 0.5001
    bit 18: 0.5010
    bit 19: 0.4982
    bit 20: 0.4993
    bit 21: 0.5011
    bit 22: 0.5002
    bit 23: 0.4997
    bit 24: 0.4988
    bit 25: 0.5000
    bit 26: 0.4994
    bit 27: 0.5018
    bit 28: 0.4990
```

```
    bit 29: 0.4988
    bit 30: 0.4994
    bit 31: 0.4999
...

==== Differential Experiment: Delta h = 0, Delta m = 2^0 (XOR) ====
Samples: 500000
Distinct delta_u: 500000
P[delta_u = 0]: 0.000000
Top 10 most common delta_u (value, count, probability):
    du = 0x7d547302d421af8eb89e24696da0e11b3b60c9f4ea5840d114851b4eb707de0b, count = 1
    prob ~= 0.000002
    du = 0x08088264b1e6922f4465f0fec04fb18646c7975fc460656c5b4097f733b1ef41, count = 1
    prob ~= 0.000002
    du = 0xaef492d8989b09ab311f915ec73cd6738e87f933f7a9817c2ccf7551fdf5cf24, count = 1
    prob ~= 0.000002
    du = 0x8e02a23f3740873dc771b679488c0a0bf63792db3a1b950770ff592cd299ac98, count = 1
    prob ~= 0.000002
    du = 0x278975458fac65f732f0aaaeaf4b50ebf82e2d2ac4b7f70ebc0ba3e00115fa82, count = 1
    prob ~= 0.000002
    du = 0xae9096340f9ac1ad64b2b11bb5c16cf4401bbf2d02f622c074b12ab1598f7871, count = 1
    prob ~= 0.000002
    du = 0x60072b1dea1bc936d5b388bc61752b0b5b3a94ad5595daeed42ac2d6dcae3be6, count = 1
    prob ~= 0.000002
    du = 0xea0efbf00c19b330ddd1f5632f4e3549b01b0a38d68696cd017f34692aea09c6, count = 1
    prob ~= 0.000002
    du = 0x82af32c4f3cb340c778de98e1da6ecb36f5cb22bcb7e596d36bec9cc5d408b1b, count = 1
    prob ~= 0.000002
    du = 0x07ad30e059333e7f48b2a5d7f2baef4118d5db0d3d592ec92bcfc51ad34a4a58, count = 1
    prob ~= 0.000002
First 32 bit 1-probabilities:
    bit  0: 0.5011
    bit  1: 0.5001
    bit  2: 0.5007
    bit  3: 0.4996
    bit  4: 0.5010
    bit  5: 0.5007
    bit  6: 0.5001
    bit  7: 0.5011
    bit  8: 0.5001
    bit  9: 0.4991
    bit 10: 0.5006
    bit 11: 0.5004
    bit 12: 0.5011
    bit 13: 0.4990
    bit 14: 0.4999
    bit 15: 0.5007
    bit 16: 0.5011
    bit 17: 0.4992
    bit 18: 0.5002
    bit 19: 0.5000
    bit 20: 0.4998
    bit 21: 0.4995
    bit 22: 0.5001
    bit 23: 0.5001
    bit 24: 0.5002
```

```
    bit 25: 0.4991
    bit 26: 0.4990
    bit 27: 0.4984
    bit 28: 0.5004
    bit 29: 0.5010
    bit 30: 0.5014
    bit 31: 0.4992
...

==== Differential Experiment: Delta h = 1, Delta m = 2^17 (XOR) ====
Samples: 500000
Distinct delta_u: 500000
P[delta_u = 0]: 0.000000
Top 10 most common delta_u (value, count, probability):
    du = 0x67ccd81c508b9f92e6a2d067a926e9d23778c398aa21645db6056662d050e41d, count = 1
    prob ~= 0.000002
    du = 0xe6086ce70baf2ebd3f92fb9f3ad9e11a22ebe16d7a1a378eb1088f89133d7e35, count = 1
    prob ~= 0.000002
    du = 0xaacababac9cb6a2741d5a4d8ab98fa5cd6c4bf3dad9816c71ed74ac61c98c74b, count = 1
    prob ~= 0.000002
    du = 0xa109d1b0f5cc978665980c7a36a8b3a7261f2621ef162591a228e75cac12a650, count = 1
    prob ~= 0.000002
    du = 0x1dc2cfccb830ebe5f2cfcc7ce466a34e8a426796d179234f8aefb42f1402feff, count = 1
    prob ~= 0.000002
    du = 0x1a93442975f169ffe8a3156d8b0f8ce217e79f8ed4dafe0ea8f6ba04897e3968, count = 1
    prob ~= 0.000002
    du = 0x59b8fa8dc2800cb94ac7072efb3d7b57e402b13b8e3321ac7c3f0ddb285b8c41, count = 1
    prob ~= 0.000002
    du = 0x3d74afb81fe952251f34a75e79a78f500bfd98504d3798481415b3a4aae32e96, count = 1
    prob ~= 0.000002
    du = 0x0d6177139a8051db51811663bd8656f67190f52c9bad0569f9349e60380b40c6, count = 1
    prob ~= 0.000002
    du = 0x6ce5b06d20e9cdc0328de64e522bf1468a0342a719839835c6442b8e82a5bb0c, count = 1
    prob ~= 0.000002
First 32 bit 1-probabilities:
    bit  0: 0.5005
    bit  1: 0.5003
    bit  2: 0.5003
    bit  3: 0.4999
    bit  4: 0.4994
    bit  5: 0.4989
    bit  6: 0.4997
    bit  7: 0.5009
    bit  8: 0.5016
    bit  9: 0.4998
    bit 10: 0.4994
    bit 11: 0.5016
    bit 12: 0.4994
    bit 13: 0.5011
    bit 14: 0.4995
    bit 15: 0.5005
    bit 16: 0.5002
    bit 17: 0.4992
    bit 18: 0.5008
    bit 19: 0.4995
    bit 20: 0.4998
```

```
    bit 21: 0.4998
    bit 22: 0.4997
    bit 23: 0.5000
    bit 24: 0.5002
    bit 25: 0.5005
    bit 26: 0.5007
    bit 27: 0.5003
    bit 28: 0.4999
    bit 29: 0.5002
    bit 30: 0.4988
    bit 31: 0.5010
...
```

E SageMath Code for Finding Safe Primes

```sage
# SageMath program for finding safe primes
# Run at: https://sagecell.sagemath.org/

def find_safe_primes(min_prime, max_prime):
    """
    Find all safe primes in range [min_prime, max_prime] using Baillie-Proth-Lucas.
    A safe prime p satisfies: p is prime AND (p-1)/2 is prime.
    """
    safe_primes = []

    # Use Baillie-Proth-Lucas algorithm for initial screening
    for i in range(min_prime + 1, max_prime + 1):
        if is_prime(i):
            if is_prime((i - 1) // 2):
                print(f"Found: {i}")
                safe_primes.append(i)

    # Perform incremental sieving on the found safe primes
    for prime in safe_primes:
        # Mark the prime and its twin siblings as unsafe
        for j in range(prime * 2, max_prime + 1, prime):
            if j in safe_primes:
                safe_primes.remove(j)
        for j in range(prime * 2 - 1, max_prime + 1, prime * 2):
            if j in safe_primes:
                safe_primes.remove(j)

    return safe_primes

# Find safe primes near 2^256
min_prime = 2**256 - 200000
max_prime = 2**256
safe_primes = find_safe_primes(min_prime, max_prime)

print(f"\nFound {len(safe_primes)} safe primes")
print(f"Primes: {safe_primes}")

# Result includes: 2^256 - 188069
```