# What's Wrong with Poly1305?

## Improving Poly1305 through a Systematic Exploration of Design Aspects of Polynomial Hash Functions


Jean Paul Degabriele **Jan Gilcher** **Jérôme Govinden** Kenneth G. Paterson


RWC 2024

Technology Innovation Institute

ETH zürich

TECHNISCHE UNIVERSITÄT DARMSTADT

# Outline

# Δ-Universal Hash in Practice

- **Definition:** Given $z \in \mathcal{T}$ and $M \neq M' \in \mathcal{M}$,

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

- **Various practical applications:**
  - Data Structures: hash tables [CW79].
  - Message Authentication Codes: UMAC, Badger, Poly1305-AES, GMAC [ISO/IEC 9797-3].
  - AEAD: AES-GCM, ChaCha20-Poly1305 [RFC 8446].

# The Adoption of ChaCha20-Poly1305 (ChaChaPoly)

- 2005,08 -Poly1305 and ChaCha20 designed separately by Bernstein.
- 2013 -First ChaChaPoly IETF draft, supported in chrome and OpenSSH.
- 2015 -ChaChaPoly specified for IETF protocols in [RFC 7539].
- 2016 -ChaChaPoly proposed standard for **TLS** in [RFC 7905].
  -Default choice in OpenSSH and WireGuard.
- 2019 -Default choice in OTRv4 and the Bitcoin Lightning Network.

# The Adoption of ChaCha20-Poly1305 (ChaChaPoly)

2005,08 -Poly1305 and ChaCha20 designed separately by Bernstein.

2013 -First ChaChaPoly IETF draft, supported in chrome and OpenSSH.

2015 -ChaChaPoly specified for IETF protocols in [RFC 7539].

2016 -ChaChaPoly proposed standard for **TLS** in [RFC 7905].
-Default choice in OpenSSH and WireGuard.

2019 -Default choice in OTRv4 and the Bitcoin Lightning Network.

**Key Points:**

- Good performance across all architectures without needing specific hardware support.
- Alternative and backup AEAD scheme to AES-GCM.
- Fast adoption even with the predominance of AES-GCM.
- Conservative and simple design, focused on performance with standard AEAD security.

# Poly1305 [Ber05]

For $M = M_1 \| \cdots \| M_n$,

$$\text{Poly1305}(r, M) = (c_1 x^n + c_2 x^{n-1} + \cdots + c_n x^1 \mod 2^{130} - 5) \mod 2^{128},$$

where $c_i = M_i \| 1$ and $x = \text{clamp}(r, 22)$.

**Limitations:**

- Clamping introduced for fast implementations using FPUs (Floating-Point Units).
  → Almost all implementations of Poly1305 use integer ALUs (Arithmetic Logic Units).
  → Provides only ≈103 bits of security with a 128-bit key and tag.
- Tailored for 32-bit architectures.
- Limited security of ChaChaPoly in the multi-user setting due to Poly1305 [DGGP21].

# Poly1305 [Ber05]

For $M = M_1 \| \cdots \| M_n$,

$$\text{Poly1305}(r, M) = (c_1 x^n + c_2 x^{n-1} + \cdots + c_n x^1 \mod 2^{130}-5) \mod 2^{128},$$

where $c_i = M_i \| 1$ and $x = \text{clamp}(r, 22)$.

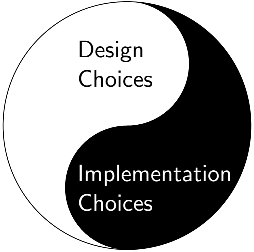
**Given today's advancements and applications, would we still converge to this same design?**

# Systematization of Knowledge (SoK)

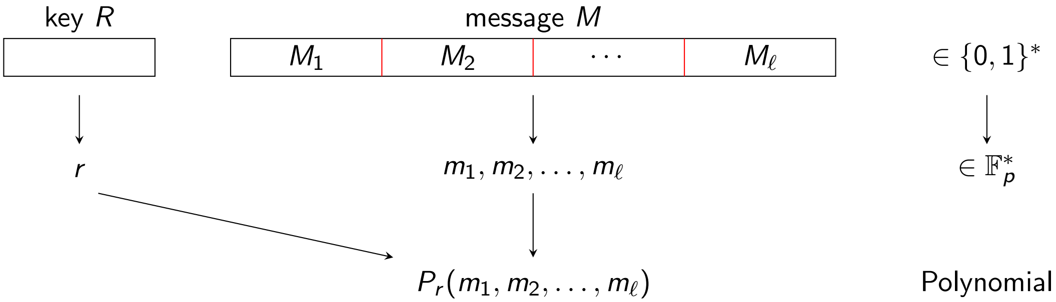
**Current Standpoint:**

- Broad design space.
- Multiple interactions between available choices.
- Knowledge spreads across research papers, cryptographic libraries, and developers' blogs.

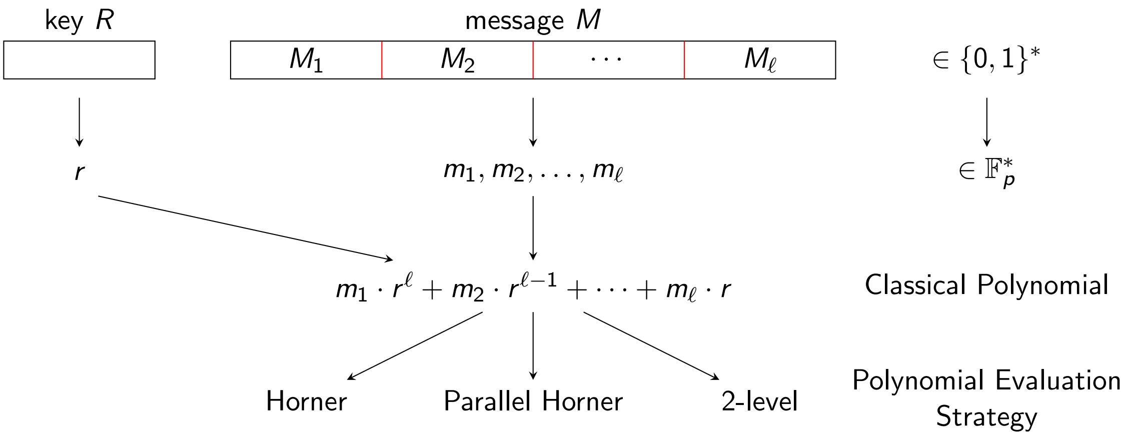
**Our Exposition [DGGP24]:**



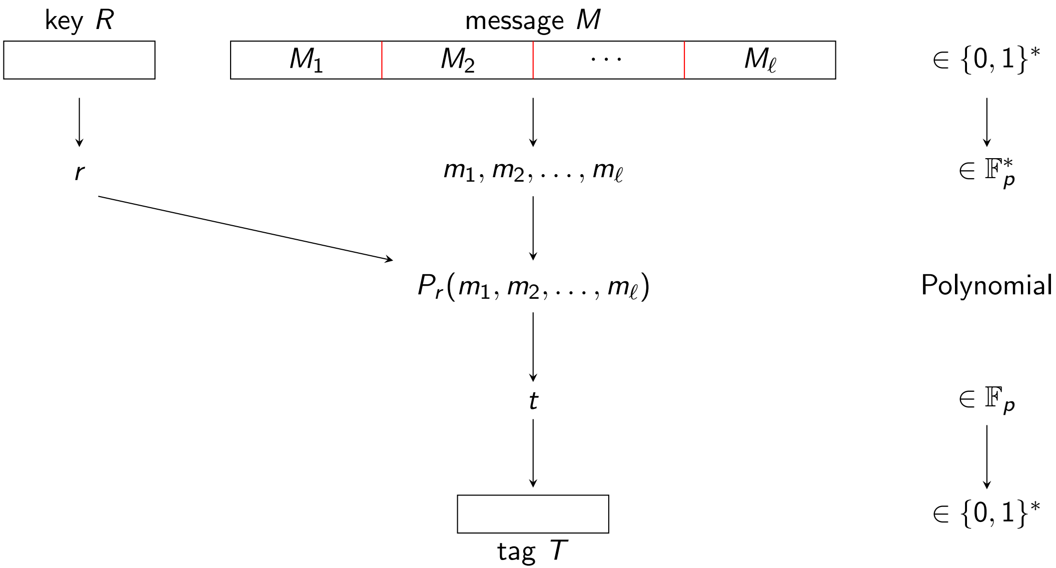
# Brief Description of the Design Space



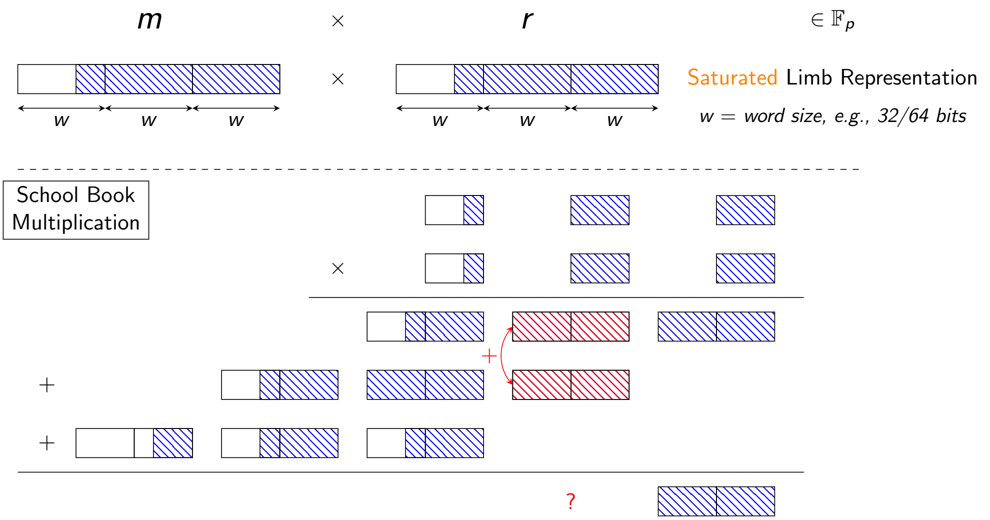
# Brief Description of the Design Space



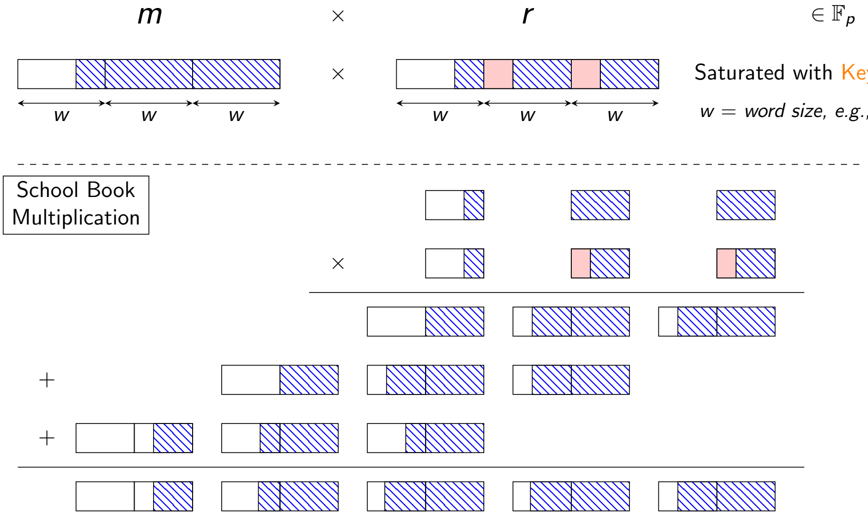
# Brief Description of the Design Space



# Field Multiplication (Saturated Limb Representation)



# Field Multiplication (Saturated Limb Representation with Key-Clamping)

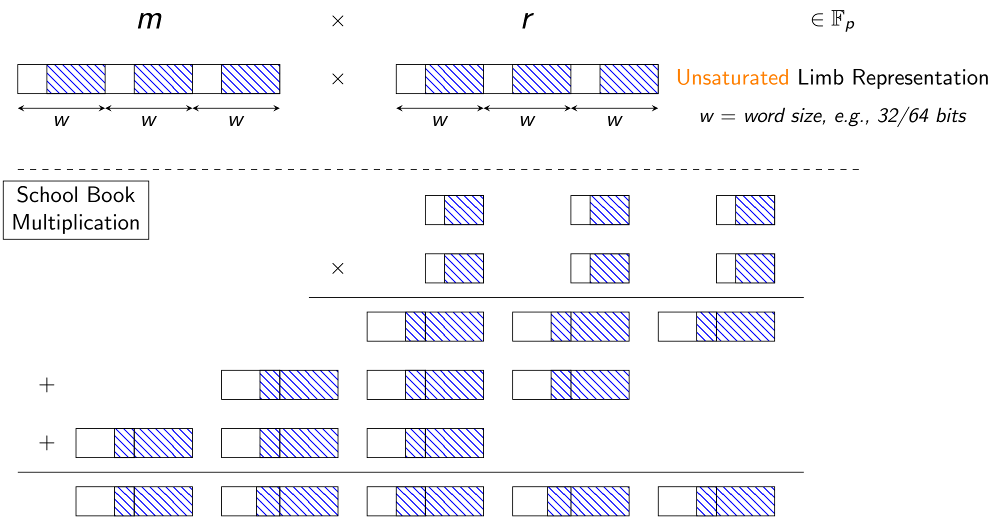



Saturated with Key-Clamping

$w$ = word size, e.g., 32/64 bits

**Limitation:** Not exploitable using parallel Horner and 2-level evaluation algorithms.

# Field Multiplication (Unsaturated Limb Representation)




Exploitable using parallel Horner and 2-level evaluation algorithms.

# Huge Design Space – What Now?

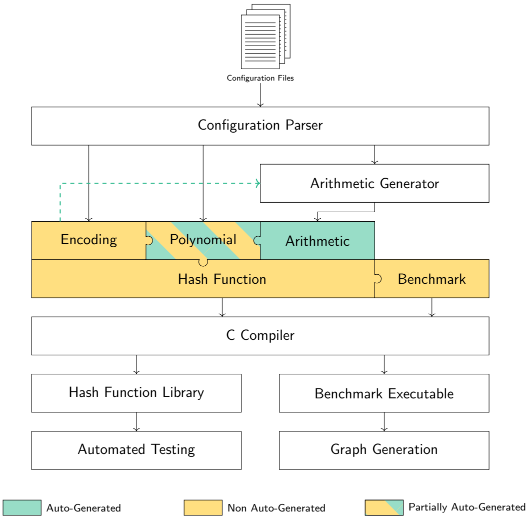
**Problem:**

- How do we pick a concrete design from this huge space?
- We want to be able to understand and test different combinations.
- Different choices make sense for different hardware.

**Solution:**

- Modularize!
  - We use our systematization to define modular *configurations*.
- Generic Implementations and Auto-Generation!
  - Write generic implementations, setting specific parameters at compile time.
  - However, fully generic code can lead to bad performance.
  - Where this is likely to occur we automatically generate efficient implementations.

# Modular Benchmarking Framework



# So, What **is** Wrong with Poly1305?

- Choice of prime is not ideal for 64-bit implementations.
  - Requires a unbalanced representation.
  - This requires 2 additional bits for the modular reduction, wasting 3% of limb space.
- There is a lot of unused space in the limbs, wasting cycles.
  - **32-bit:** 26-bit limbs leave 12% of the limbs unused.
  - **64-bit:** Mixed 44-/42-bit limbs leave up to 23% of the limbs unused.
- Clamping sacrifices 22 bits of security to enable FPU implementations.
  - Also wastes space in the key limbs (17%).
  - Sensible at the time. Now, not so much.

**openssl poly1305-x86.pl**

> *[B]esides SSE2 there are floating-point and AVX options; FP is deemed unnecessary, because pre-SSE2 processor are too old to care about, while it's not the fastest option on SSE2-capable ones;*

# Goals for New Designs

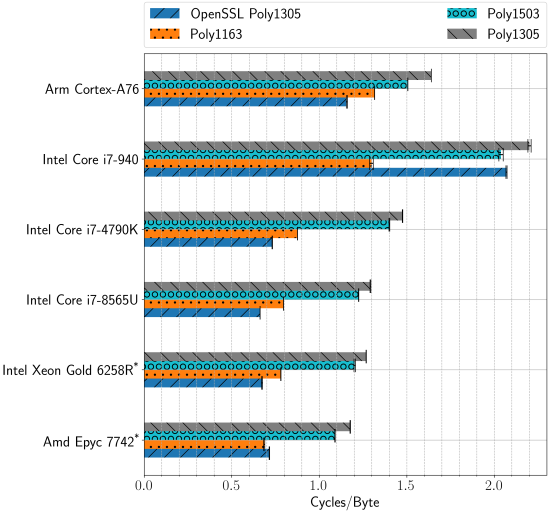
- More efficient than Poly1305 (i.e., better runtime-security tradeoff).
- Keep things simple, familiar to developers.
- Allow various optimization strategies to tune implementations to different hardware.
- But without tailoring the design towards a specific implementation.
  - Don't design for FPUs!

# New Designs

- No clamping to support FPU implementations as these are not worth the security loss.
- Stick with Classical Polynomial over $\mathbb{F}_p$. Pack limbs as full as we can.
- Designs allow: Delayed reduction, 2-level polynomial evaluation, exploiting CPU parallelism.

| **Options:** | Fewer limbs to increase performance | Same number of limbs and increase security |
|---|---|---|
| Prime for fast reduction: | $p_1 = 2^{116} - 3$ | $p_2 = 2^{150} - 3$ |
| Bits per limb (32/64): | 29/58 | 30/50 |
| Security Level: | ≈107 bits | ≈137 bits |
| Resulting Hash function: | **Poly1163** | **Poly1503** |

# Benchmarking




*Turbo Boost/Core Adjusted

**Results:**

- Our modular implementations achieve **high performance without vectorization or hand-optimization.**
- Poly1163 performance makes it **suitable as drop-in replacement for Poly1305.**

**Our Expectations for Vectorization:**

- Poly1163: Significantly outperforms Poly1305 at the same security level.
- Poly1503: Replacement for Poly1305 with 34 bits of extra security $(103 \rightarrow 137)$ at similar performance.

# Where to Find More Details

**SoK on Polynomial Hash:**

https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00132

**Code of Polynomial Hash Framework:**

https://github.com/jangilcher/polynomial_hashing_framework

# References I

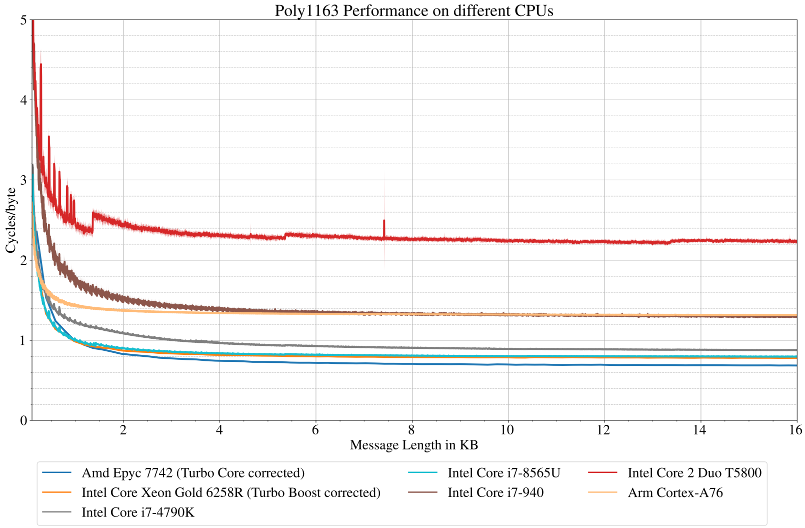
Daniel J. Bernstein.
The poly1305-AES message-authentication code.
In Henri Gilbert and Helena Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, Heidelberg, February 2005.

J Lawrence Carter and Mark N Wegman.
Universal classes of hash functions.
*Journal of computer and system sciences*, 18(2):143–154, 1979.

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

Jean Paul Degabriele, Jan Gilcher, Jérôme Govinden, and Kenneth G. Paterson.
Sok: Efficient design and implementation of polynomial hash functions over prime fields.
In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 131–131, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

# Benchmarks: Poly1163



# Benchmarks: Poly1503