A Survey of Non-Cryptographic Hash Functions

As Implemented in SMHasher3

Thomas Dybdahl Ahle

February 3, 2026

Abstract

This document provides a comprehensive survey of non-cryptographic hash functions as implemented in the SMHasher3 testing framework [Woj22]. We present each algorithm using clear pseudocode and mathematical notation, analyze their mixing properties, and discuss their relative strengths and weaknesses.

Contents

1 Introduction

Non-cryptographic hash functions are fundamental building blocks in computer science, used extensively in hash tables, checksums, and probabilistic data structures. This survey documents hash functions implemented in SMHasher3, presenting algorithms with clear pseudocode and analyzing their mixing properties.

1.1 Performance Overview

Table 1 shows performance measurements for the hash functions surveyed in this paper, from the SMHasher3 test suite, sorted by bulk throughput. Hashes marked with ✗ do not pass the full test battery; the number in parentheses is the count of tests passed out of 250.

Hash	Bits	B/cycle (bulk)	Cycles (1–32 B)	Pass
AquaHash*	64	15.99	34.31	✗ (195)
XXH3*	64	12.95	30.69	✗ (223)
MeowHash	128	12.43	61.51	✓
rapidhash	64	8.82	30.41	✓
CLHash*	64	7.38	48.19	✗ (64)
MuseAir	64	7.19	28.45	✓
wyhash*	64	7.11	29.31	✗ (235)
HalftimeHash*	128	6.92	95.58	✗ (202)
komihash	64	6.24	32.62	✓
ChibiHash	64	6.15	34.53	✓
UMASH*	64	6.09	41.74	✗ (122)
aHash	64	4.52	29.14	✓
SpookyHash	32	4.40	60.20	✓
Polymur	64	4.02	42.57	✓
xxHash64*	64	4.00	52.61	✗ (241)
khashv	64	3.20	56.17	✓
a5hash	64	2.63	26.91	✓
FarmHash	128	2.61	69.61	✓
xxHash32*	32	2.00	44.38	✗ (167)
Rainbow	64	1.28	61.63	✓
prvhash	64	0.94	51.07	✓
XMSX	32	0.67	42.56	✓
poly-mersenne	32	0.50	69.79	✓
HalfSipHash	32	0.34	81.75	✓
SipHash-2-4	64	0.32	152.26	✓

Table 1: Performance sorted by bulk throughput (bytes/cycle). *Does not pass the full SMHasher3 test battery; see individual sections for details.

Two performance regimes. Hash performance differs dramatically between short and long keys:

- **Short keys (1–32 bytes):** Latency-bound. Specialized code paths avoid loop overhead. The best hashes achieve ∼26–31 cycles total.
- **Bulk throughput:** Bandwidth-bound. Parallel lanes and wide reads dominate. The best MUM-based hashes achieve 7–9 bytes/cycle; AES-based hashes (MeowHash) reach ∼12 bytes/cycle.

Hash	Output	Block	Lanes	Primitive	Keying	Provable width
a5hash	64	16	1	MUM	seed	N/A
a5hash-128	128	32/64	2/4	MUM	seed	N/A
MuseAir	64	96	6	MUM	seed	N/A
wyhash	64	48	3	MUM	seed+secret	N/A
rapidhash	64	112	7	MUM	seed+secret	N/A
rapidhashMicro	64	80	5	MUM	seed+secret	N/A
rapidhashNano	64	48	3	MUM	seed+secret	N/A
xxHash32	32	16	4	ARX	seed	N/A
xxHash64	64	32	4	ARX	seed	N/A
XXH3-64	64	64	8	MUM	seed+secret	N/A
SipHash-2-4	64	8	1	ARX	128-bit key	N/A
komihash	64	64	4	MUM	seed	N/A
ChibiHash	64	32	4	mult+rot	seed	N/A
CLHash	64	1024	8 (ILP)	CLMUL	random key	$\approx 64 - \log_2 n$
UMASH	64/128	256	4 (ILP)	CLMUL+poly	random key	≈ 61
Polymur	64	49	1	poly($2^{61}-1$)	derived key	≈ 61
AquaHash	128	64	4	AES	128-bit seed	N/A
MeowHash	128	256	8	AES	128-bit seed	N/A
HalftimeHash	64	var	var	32-bit mul	random key	N/A
SpookyHash	128	96	12	ARX	seed	N/A
FarmHash	64	64	7	ARX+MUM	seed	N/A
Rainbow	64	16	4	mult+rot	seed	N/A
prvhash	64	8	1	PRNG	seed	N/A
XMSX	32	4	1	mult+shift	seed	N/A
khashv	64	16	4	S-box+ARX	seed	N/A
aHash	64	16	2	AES	random key	N/A
HalfSipHash	32	4	1	ARX	64-bit key	N/A
poly-mersenne	32	8	1	poly($2^{61}-1$)	random key	≈ 61

Table 2: Structural comparison of hash functions. "Lanes" indicates parallel accumulators; "(ILP)" denotes instruction-level parallelism without explicit state lanes. "Keying" shows the security model: seed-only hashes are vulnerable to seed-independent attacks; secret/key-based hashes resist them if the key is hidden. "Provable width" is the effective bits $w = -\log_2(\epsilon)$ for a published ϵ-almost-universal collision bound under a random secret key (often length-dependent, e.g. $\epsilon \approx n/2^{64}$).

2 Comparison

Design trade-offs:

- **MUM-based** (a5hash, MuseAir, wyhash, rapidhash, XXH3, komihash): Fast on modern CPUs with efficient 64×64→128 multiply. Excellent throughput for long keys.
- **ARX-based** (xxHash32/64, SipHash, SpookyHash, FarmHash): Portable, constant-time friendly. SipHash sacrifices speed for cryptographic security margin. SpookyHash uses 12 lanes for high throughput.
- **Parallel lanes**: More lanes = higher throughput but more state to maintain. rapidhash (7 lanes) vs wyhash (3 lanes) trades complexity for speed. MuseAir uses 6 lanes with chained dependencies.
- **Polynomial** (Polymur, UMASH): Theoretically grounded with provable collision bounds. UMASH combines CLMUL compression with polynomial finalization.
- **AES-based** (AquaHash, MeowHash, aHash): Highest throughput on supporting hardware via AES-NI instructions. aHash is the default in Rust's HashMap.

- **Keying**: Hashes with secret keys (SipHash, CLHash, UMASH) resist HashDoS attacks if the key is kept hidden. Seed-only hashes can be attacked with seed-independent collisions.

3 Preliminaries

3.1 Notation

Symbol	Meaning
$\oplus$	Bitwise XOR
$\cdot$	Multiplication modulo 2^w on uw words (keep the low w bits) unless noted
$\textsc{rot}_k(x)$	Left rotation of x by k bits
$\gg$	Logical right shift
$\ll$	Left shift
$\textsc{Lo}(x)$	Low 64 bits of 128-bit value x
$\textsc{Hi}(x)$	High 64 bits of 128-bit value x
$\textsc{Lo32}(x)$	Low 32 bits of 64-bit value x
$\textsc{Hi32}(x)$	High 32 bits of 64-bit value x
p	Pointer to input bytes
n	Input length in bytes
seed	User-provided seed (may be secret for keyed hashes)
k_i	Secret key words (e.g. SipHash)
s_i	Algorithm-specific fixed constants / secret tables
v_i	Evolving internal state words (accumulators/lanes)
$\textsc{Read}_w(p, i)$	Read w-bit word at byte offset i from p
$p[i]$	Byte at offset i (equivalent to $\textsc{Read}_8(p, i)$)
m_i	The i-th block of input (block size B varies by hash)
x_j	The j-th word within current block (64-bit unless noted)
uw	Unsigned w-bit word type (e.g. u32, u64)

Terminology. A **block** is the fixed-size unit processed by the main loop (e.g., 48 bytes for wyhash, 112 bytes for rapidhash). The term **stripe** is XXH3-specific, denoting a 64-byte segment processed across 8 parallel lanes. We use “block” consistently; some hash authors use “chunk” synonymously.

3.2 Primitive Categories

Modern hash functions are built from a small set of mixing primitives:

MUM (Multiply-Mix) Wide multiplication followed by XOR folding:

$$\textsc{Mum}(a, b) = \textsc{Lo}(a \times b) \oplus \textsc{Hi}(a \times b) \tag{1}$$

Fast on modern CPUs with efficient 64×64→128 multiply units. Used by wyhash, rapidhash, XXH3, komihash, t1ha.

ARX (Add-Rotate-XOR) Combines addition, rotation, and XOR in patterns like:

$$h \leftarrow \textsc{rot}_r(h + x) \oplus y \quad \text{or} \quad h \leftarrow \textsc{rot}_r(h \oplus x) + y \tag{2}$$

Portable and constant-time friendly. Used by MurmurHash, xxHash32/64, SipHash, SpookyHash, lookup3.

CLMUL (Carry-less Multiply) Polynomial multiplication in GF(2^n):

$$a \otimes b = a \cdot b \mod p(x) \tag{3}$$

where $\cdot$ is polynomial multiplication (XOR for addition, AND for coefficient multiplication). Theoretically grounded with provable collision bounds. Used by CLHash, UMASH. Requires PCLMULQDQ instruction.

Polynomial over GF(p) Evaluates polynomial over a prime field:

$$h \leftarrow (h \cdot k + m_i) \mod p \quad \text{typically } p = 2^{61} - 1 \tag{4}$$

Strong theoretical guarantees from algebra. Used by UMASH, Polymur.

AES-NI Uses AES encryption rounds for mixing:

$$\text{AESENC}(\mathit{state}, \mathit{key}) = \text{MixColumns}(\text{ShiftRows}(\text{SubBytes}(\mathit{state}))) \oplus \mathit{key} \tag{5}$$

Excellent diffusion (full avalanche) in a single instruction. Used by AquaHash, MeowHash.

Simple Multiply Basic multiply-accumulate:

$$h \leftarrow (h \oplus b) \cdot p \quad \text{(FNV-1a)} \qquad h \leftarrow h \cdot 33 + b \quad \text{(DJB2)} \tag{6}$$

Minimal code size but weaker mixing. One byte per iteration.

Most high-performance hashes use MUM or ARX. MUM-based hashes are generally faster on x86-64; ARX-based hashes are more portable. CLMUL and AES-based hashes offer the highest throughput on supported hardware.

3.3 Common Hash Structure

Most non-cryptographic hash functions follow a similar high-level structure:

Initialization Set up state variables (accumulators) from seed and/or constants. Multiple accumulators enable parallel processing.

Main Loop Process input in fixed-size blocks. Each block updates the state using the hash's core mixing primitive. Multiple *parallel lanes*—independent accumulators updated simultaneously—allow modern CPUs to exploit instruction-level parallelism.

Tail Handling Process remaining bytes that don't fill a complete block. Often uses different logic than the main loop (overlapping reads, byte-by-byte, etc.).

Lane Collapse Combine multiple parallel lanes into a single value, typically via XOR trees, addition chains, or mixing functions.

Finalization Apply an avalanche finalizer (xorshift and multiplication) to ensure all input bits affect all output bits (avalanche property).

The block size and number of lanes vary by hash: wyhash uses 48-byte blocks with 3 lanes, rapidhash uses 112-byte blocks with 7 lanes, and MuseAir uses 96-byte blocks with 6 lanes.

3.4 Finalizers and Mixers

Most fast non-cryptographic hashes end with a short *avalanche* stage that destroys residual structure from the main loop. These finalizers are usually built from two permutations on w-bit words:

- **Xorshift**: $\textsc{Xorshift}_r(x) = x \oplus (x \gg r)$ is invertible for $0 < r < w$.
- **Multiply by an odd constant**: $x \leftarrow x \cdot c$ is invertible mod 2^w when c is odd.

The common template is:

$$
\begin{aligned}
h &\leftarrow \textsc{Xorshift}_{r_1}(h) && (7)\\
h &\leftarrow h \cdot c_1 && (8)\\
h &\leftarrow \textsc{Xorshift}_{r_2}(h) && (9)\\
h &\leftarrow h \cdot c_2 && (10)\\
h &\leftarrow \textsc{Xorshift}_{r_3}(h) && (11)
\end{aligned}
$$

Different hashes choose (r_i) and (c_i) empirically (e.g. XXH32_AVALANCHE, XXH64_AVALANCHE, the XXH3 finalizer, and Polymur's mx3 finalizer).

Common sub-steps. It is convenient to name two ubiquitous building blocks:

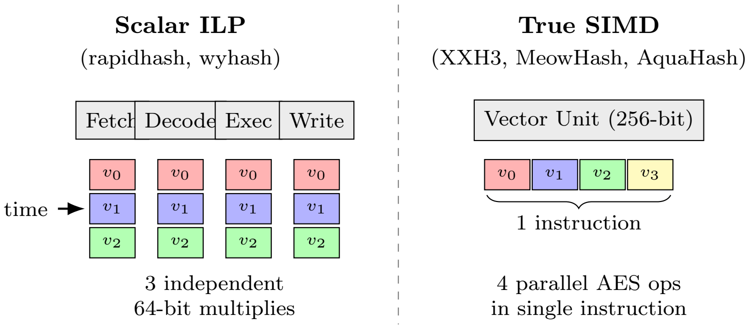
$$
\begin{aligned}
\mathrm{XMS}_{r,c}(x) &= (x \oplus (x \gg r)) \cdot c \qquad \text{(xorshift-then-multiply)} && (12)\\
\textsc{ShiftMix}(x) &= x \oplus (x \gg 47) && (13)
\end{aligned}
$$

Many "Murmur-style" combiners (e.g. FarmHash's Hash128to64 and HashLen16) use ShiftMix between multiplications; xxHash uses XMS with two different constants.

Pseudocode conventions. Main loop processing is shown in full detail. Tail handling (which often involves complex case analysis for 1–15 remaining bytes) is sometimes summarized as "handle remaining bytes" to keep pseudocode readable. The core mixing primitives are always fully specified. Comments use `%` for section boundaries (Bulk/Tail/Finalization) where helpful.

Scalar ILP vs. SIMD lanes. Hash "lanes" come in two flavors:

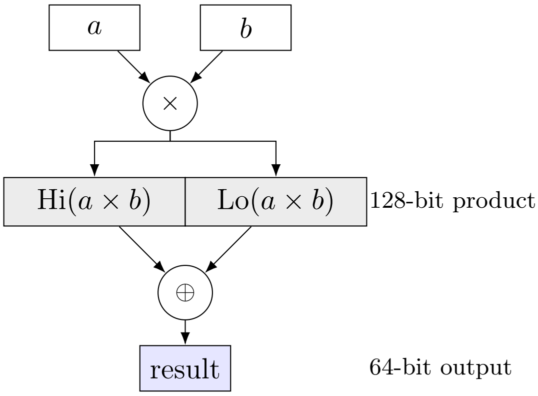
- **Scalar ILP lanes** (wyhash, rapidhash, SpookyHash): Independent scalar accumulators that exploit the CPU's out-of-order execution engine. The 64×64→128-bit multiply used by MUM *has no SIMD instruction*—SSE/AVX only provide 32×32→64. These hashes use 3–7 independent dependency chains to keep the scalar multiply unit fed. Lane counts like 3 or 7 are tuned for register pressure and multiply latency, not SIMD widths.
- **True SIMD lanes** (XXH3, AquaHash, MeowHash, HalftimeHash): Vector operations using SSE/AVX/NEON intrinsics with operations that *do* vectorize: AES-NI instructions (`aesenc`/`aesdec`) or 32×32→64 multiplies (`_mm_mul_epu32`). XXH3 uses 32×32 multiplies for its long-key path (explicit AVX2/SSE2/NEON implementations), but falls back to scalar 64×64 MUM for short/medium keys where SIMD overhead isn't worth it.



4 Round Functions and Combiners

This section details specific mixing functions used by hash families. For the core primitives (MUM, ARX, etc.), see Section 3.2.

4.1 The MUM Primitive



We define three variants used throughout this survey:

$\textsc{Mum}(a,\, b)$ **Standard (folded):** Returns $\textsc{Lo}(a \times b) \oplus \textsc{Hi}(a \times b)$. Used by wyhash (`_wymix`), rapidhash (`rapid_mix`), XXH3.

$\textsc{Mum}^*(a, b)$ **Protected (folded):** XORs the original inputs back into the product before folding, preventing degenerate cases (e.g. a=0) from collapsing state:

$$\textsc{Mum}^*(a, b) = a \oplus b \oplus \textsc{Lo}(a \times b) \oplus \textsc{Hi}(a \times b) \tag{14}$$

Used by wyhash-strict, rapidhash-protected, MuseAir.

$\textsc{Mum}_{128}(a, b)$ **Full product:** Returns the pair $(\textsc{Lo}(a \times b),\, \textsc{Hi}(a \times b))$ without folding. Used in finalizations where both halves update separate state variables.

$\textsc{Mum}^*_{128}(a, b)$ **Protected full product:** Returns the pair $(a \oplus \textsc{Lo}(a \times b),\, b \oplus \textsc{Hi}(a \times b))$. Folding this pair with XOR gives $\textsc{Mum}^*(a, b)$.

4.2 Round Functions

Many hashes use a *round function* that mixes input into an accumulator. The xxHash round functions are defined in Section 7.1.1.

4.3 Xorshift

The xorshift primitive XORs a value with a shifted copy of itself:

$$\textsc{Xorshift}_r(v) = v \oplus (v \gg r) \tag{15}$$

Permutation proof. $\textsc{Xorshift}_r$ is a bijection (permutation) for any $0 < r < w$ where w is the word size. To see this, note that the top r bits of v are unchanged, and each subsequent bit can be recovered by XORing with the already-recovered bit r positions above it. Equivalently, applying $\textsc{Xorshift}_r$ a total of $\lceil w/r \rceil$ times yields the identity.

4.4 ShiftMix

A specific xorshift used by CityHash and FarmHash:

$$\textsc{ShiftMix}(v) = \textsc{Xorshift}_{47}(v) = v \oplus (v \gg 47) \tag{16}$$

This diffuses high bits into the low 17 bits of a 64-bit value.

4.5 HashLen16 (CityHash)

CityHash's core 128-to-64 bit reducer, inspired by MurmurHash:

$$\begin{aligned}
a &\leftarrow (u \oplus v) \cdot \mathit{mul} && (17)\\
a &\leftarrow \textsc{ShiftMix}(a) && (18)\\
b &\leftarrow (v \oplus a) \cdot \mathit{mul} && (19)\\
b &\leftarrow \textsc{ShiftMix}(b) && (20)\\
\textsc{HashLen16}(u, v, \mathit{mul}) &\leftarrow b \cdot \mathit{mul} && (21)
\end{aligned}$$

where mul is an odd 64-bit multiplier (often a fixed constant, or $k_2 + 2n$ for length-dependent mixing). When the multiplier is the fixed default, we may omit it and write $\textsc{HashLen16}(u, v)$. This resembles $\textsc{Mum}$ but applies the shifts between multiplications rather than using 128-bit product folding.

4.6 Finalization Mixers

All finalization mixers follow a common pattern: alternating xorshift and multiply operations. The general form is:

$$\begin{aligned}
h &\leftarrow \textsc{Xorshift}_{r_1}(h) && (22)\\
h &\leftarrow h \cdot c_1 && (23)\\
h &\leftarrow \textsc{Xorshift}_{r_2}(h) && (24)\\
h &\leftarrow h \cdot c_2 && (25)\\
\textsc{fmix}(h) &\leftarrow \textsc{Xorshift}_{r_3}(h) && (26)
\end{aligned}$$

Different hashes choose different shift amounts (r_1, r_2, r_3) and multiplier constants (c_1, c_2) based on avalanche analysis. Typical 32-bit shifts: $(16, 13, 16)$. Typical 64-bit shifts: $(33, 33, 33)$ or $(33, 29, 32)$.

This is a bijection: each $\textsc{Xorshift}_r$ is a permutation (see Section 4.3), and multiplication by an odd constant is a permutation mod 2^w.

5 The wyhash/rapidhash Family

This family uses the MUM primitive as its core operation:

$$\textsc{Mum}(a,\, b) = \textsc{Lo}(a \times b) \oplus \textsc{Hi}(a \times b) \tag{27}$$

5.1 Secrets

Both families use 64-bit “secret” constants $s_0, s_1, \ldots$ for mixing. wyhash uses 4 secrets; rapidhash uses 8 (the first 4 shared with wyhash).

5.2 wyhash (64-bit)

Source: `hashes/wyhash.cpp` [Yi19].

SMHasher3 test results. wyhash fails 15 of 250 tests: Permutation tests with single-bit keys at all block sizes (4–16 bytes, both LE and BE), and SeedZeroes at 1280 and 8448 bytes. The Permutation failures reflect insufficient diffusion when only one bit of each input word is set.

The main loop uses 3 scalar lanes (not 4 or 8) because Mum relies on 64×64→128-bit scalar multiplication, which doesn’t vectorize in SSE/AVX. The 3 independent dependency chains exploit ILP while keeping register pressure manageable.

```
$v_0 \leftarrow seed \oplus \textsc{Mum}(seed \oplus s_0,\, s_1)$
for $i \in \{1,2\}$ do
    $v_i \leftarrow v_0$
end for
for each block $(x_0,\ldots,x_5) \in \texttt{u64}^6$ do          ▷ Main loop: 3 parallel lanes
    for $i \in \{0,1,2\}$ do
        $v_i \leftarrow \textsc{Mum}(x_{2i} \oplus s_{i+1},\, x_{2i+1} \oplus v_i)$
    end for
end for
$v_0 \leftarrow v_0 \oplus v_1 \oplus v_2$          ▷ Collapse lanes
Process remaining bytes; read last 16 bytes as $t_0, t_1$
$t_0 \leftarrow t_0 \oplus s_1, \quad t_1 \leftarrow t_1 \oplus v_0$
$(t_0, t_1) \leftarrow (\textsc{Lo}(t_0 \times t_1), \textsc{Hi}(t_0 \times t_1))$
return $\textsc{Mum}(t_0 \oplus s_0 \oplus n,\, t_1 \oplus s_1)$
```

5.3 rapidhash

Source: `hashes/rapidhash.cpp`, version 3 [De 24].

By Nicolas De Carli, based on wyhash. Uses 7 scalar ILP lanes (112-byte blocks), trading higher register pressure for more parallelism on wide superscalar CPUs. Uses s_2 for seed initialization and 8 secrets (s_0–s_7).

The “protected” variant XORs the original inputs back into the MUM result:

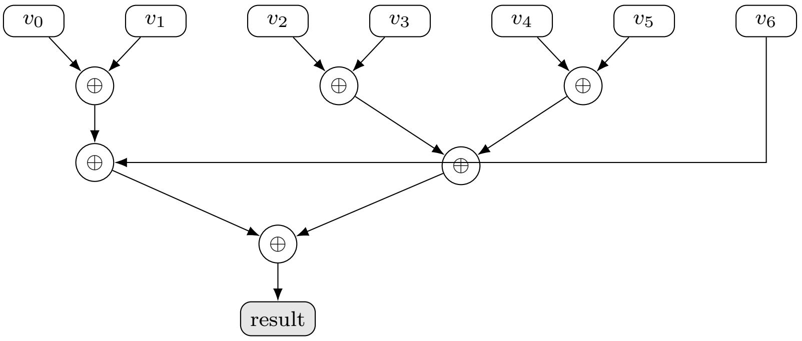
$$\textsc{Mum}^*_{128}(a, b) = (a \oplus \textsc{Lo}(a \times b),\ b \oplus \textsc{Hi}(a \times b)) \tag{28}$$

This prevents zero-multiplication weaknesses at a small performance cost.

```
$v_0 \leftarrow seed \oplus \textsc{Mum}(seed \oplus s_2,\, s_1)$          ▷ Note: $s_2$ not $s_0$
for $i \in \{1,\ldots,6\}$ do
    $v_i \leftarrow v_0$
end for
for each block $(x_0,\ldots,x_{13}) \in \texttt{u64}^{14}$ do          ▷ Main loop: 7 parallel lanes
    for $i \in \{0,\ldots,6\}$ do
```

```
        $v_i \leftarrow \textsc{Mum}(x_{2i} \oplus s_i,\ x_{2i+1} \oplus v_i)$
    **end for**
**end for**
$v_0 \leftarrow v_0 \oplus v_1$; $v_2 \leftarrow v_2 \oplus v_3$; $v_4 \leftarrow v_4 \oplus v_5$        ▷ Collapse lanes
$v_0 \leftarrow v_0 \oplus v_6$; $v_2 \leftarrow v_2 \oplus v_4$; $v_0 \leftarrow v_0 \oplus v_2$
Process remaining bytes; read last 16 bytes as $t_0, t_1$ with $t_0 \leftarrow t_0 \oplus n$
$t_0 \leftarrow t_0 \oplus s_1$,    $t_1 \leftarrow t_1 \oplus v_0$
$(t_0, t_1) \leftarrow (\textsc{Lo}(t_0 \times t_1), \textsc{Hi}(t_0 \times t_1))$
**return** $\textsc{Mum}(t_0 \oplus s_7,\ t_1 \oplus s_1 \oplus n)$        ▷ Note: $s_7$ and extra $n$
```

XOR-tree lane combination:



Tail (17–112 bytes). Nested conditionals process 16 bytes at a time with alternating secrets:

Bytes remaining	Offset	Secret
> 16	0, 8	s_2
> 32	16, 24	s_2
> 48	32, 40	s_1
> 64	48, 56	s_1
> 80	64, 72	s_2
> 96	80, 88	s_1

Finalization. Read last 16 bytes as (t_0, t_1) with $t_0 \leftarrow t_0 \oplus n$, then apply $t_0 \leftarrow t_0 \oplus s_1$, $t_1 \leftarrow t_1 \oplus v_0$, and $(t_0, t_1) \leftarrow (\textsc{Lo}(t_0 \times t_1), \textsc{Hi}(t_0 \times t_1))$. Return $\textsc{Mum}(t_0 \oplus s_7,\ t_1 \oplus s_1 \oplus n)$. Differs from wyhash: length is XORed into a before MUM (not the final result), uses s_7 instead of s_0, and includes length twice in finalization.

5.4 rapidhash Variants

Variant	Chunk size	Lanes	Secrets used
rapidhash	112 bytes	7	s_0–s_7
rapidhashMicro	80 bytes	5	s_0–s_4, s_7
rapidhashNano	48 bytes	3	s_0–s_2, s_7

Rust port. The `rust-rapidhash` implementation uses a different seed initialization: secrets are regenerated from the user seed using the protected MUM variant, ensuring non-zero quality in specific bit ranges. This makes hashes incompatible between the C and Rust versions for the same seed.

6 The a5hash/MuseAir Family

These hashes also use the MUM primitive but with different architectural choices than wyhash/rapidhash.

6.1 a5hash

Source: `hashes/a5hash.cpp`, version 5.21 [Van25].

A MUM-based hash by Aleksey Vaneev with a single MUM lane in the main loop (simpler than wyhash's 3 lanes). Uses mantissa bits of π for initial state and checkerboard masks $c_0 =$ `0x5555...`, $c_1 =$ `0xAAAA...` that ensure different bit positions are treated differently. Addition of masks after each round prevents zero-multiplication weaknesses. Available in 32-bit, 64-bit, and 128-bit variants.

```
$v_0 \gets \mathtt{0x243F6A8885A308D3} \oplus n$, $v_1 \gets \mathtt{0x452821E638D01377} \oplus n$    ▷ From $\pi$
$(v_0, v_1) \gets \textsc{Mum}_{128}(v_1 \oplus (\mathit{seed} \wedge c_1), v_0 \oplus (\mathit{seed} \wedge c_0))$
if $n > 16$ then
    $c_0 \gets c_0 \oplus v_0$; $c_1 \gets c_1 \oplus v_1$
    for each block $(x_0, x_1) \in \mathtt{u64}^2$ excluding final 16 bytes do    ▷ Main loop
        $(v_0, v_1) \gets \textsc{Mum}_{128}(x_0 \oplus v_0, x_1 \oplus v_1)$
        $v_0 \gets v_0 + c_0$; $v_1 \gets v_1 + c_1$
    end for
end if
Handle tail (0–16 bytes) with overlapping reads
$(v_0, v_1) \gets \textsc{Mum}_{128}(v_0, v_1)$
$(v_0, v_1) \gets \textsc{Mum}_{128}(v_0 \oplus c_0, v_1)$
return $v_0 \oplus v_1$
```

Here $\textsc{Mum}_{128}(a, b)$ returns both halves of the 128-bit product: $(\textsc{Lo}(a \times b), \textsc{Hi}(a \times b))$.

6.2 MuseAir

Source: `hashes/museair.cpp`, version 0.3 [K-A24].

A MUM-based hash by K–Aethiax with 6-lane parallelism and chained dependencies between lanes. Uses 7 constants from Ai(0) mantissa ($s_0 =$ `0x5ae31e589c56e17a`, ...). The MUMIX primitive either XORs MUM output into state (standard) or replaces it (bfast). Cross-lane MUM pattern in the epilogue ensures full mixing. Available in standard, bfast (faster, slightly weaker), and 128-bit variants.

MUMIX.

```
function MUMIX$(v_p, v_q, x_p, x_q)$    ▷ State words $(v_p, v_q)$, input words $(x_p, x_q)$
    $a \gets v_p \oplus x_p$;    $b \gets v_q \oplus x_q$
    $(\ell, h) \gets \textsc{Mum}_{128}(a, b)$
    $(v_p, v_q) \gets (a \oplus \ell, b \oplus h)$    ▷ Standard variant
                                                      ▷ bfast variant: $(v_p, v_q) \gets (\ell, h)$
end function
```

Long Keys.

```
Initialize 6 state variables from seed:
   $v_0 \gets s_0 + \mathit{seed}$, $v_1 \gets s_1 - \mathit{seed}$, $v_2 \gets s_2 \oplus \mathit{seed}$
   $v_3 \gets s_3 + \mathit{seed}$, $v_4 \gets s_4 - \mathit{seed}$, $v_5 \gets s_5 \oplus \mathit{seed}$
$t \gets s_6$    ▷ Initial chain value
for each block $(x_0, \ldots, x_{11}) \in \mathtt{u64}^{12}$ do    ▷ Main loop: 6 parallel lanes
    for $i \in \{0, \ldots, 5\}$ do
```

```
        $v_i \leftarrow v_i \oplus x_{2i}$
        $v_{(i+1) \bmod 6} \leftarrow v_{(i+1) \bmod 6} \oplus x_{2i+1}$
        $(lo, hi) \leftarrow \textsc{Mum}_{128}(v_i, v_{(i+1) \bmod 6})$
        $v_i \leftarrow v_i + (t \oplus hi)$        ▷ Chain: previous lo XOR current hi
        $t \leftarrow lo$
    end for
end for
$v_0 \leftarrow v_0 + t$        ▷ Fold in last chain value
Process remaining 48, 32, 16 bytes with MUMIX
Apply MUMIX to the last 16 bytes (tail summarized)
$t_0 \leftarrow v_0 - v_1$, $t_1 \leftarrow v_2 - v_3$, $t_2 \leftarrow v_4 - v_5$        ▷ Finalization: collapse lanes
$t_0 \leftarrow \mathrm{ROT}_{n \bmod 64}(t_0)$, $t_1 \leftarrow \mathrm{ROT}_{64-(n \bmod 64)}(t_1)$, $t_2 \leftarrow t_2 \oplus n$        ▷ $\mathrm{ROT}_{64-r}()$ is right-rotate by $r$
$(t'_0, t'_1) \leftarrow \textsc{Mum}_{128}(t_0, t_1)$, $(t'_2, t'_3) \leftarrow \textsc{Mum}_{128}(t_1, t_2)$, $(t'_4, t'_5) \leftarrow \textsc{Mum}_{128}(t_2, t_0)$
$t_0 \leftarrow t'_0 \oplus t'_5$, $t_1 \leftarrow t'_2 \oplus t'_1$, $t_2 \leftarrow t'_4 \oplus t'_3$        ▷ Cross lo/hi
$(u_0, u_1) \leftarrow \textsc{Mum}_{128}(t_0, t_1)$, $(u_2, u_3) \leftarrow \textsc{Mum}_{128}(t_1, t_2)$, $(u_4, u_5) \leftarrow \textsc{Mum}_{128}(t_2, t_0)$
return $(u_0 \oplus u_5) + (u_2 \oplus u_1) + (u_4 \oplus u_3)$        ▷ 64-bit
```

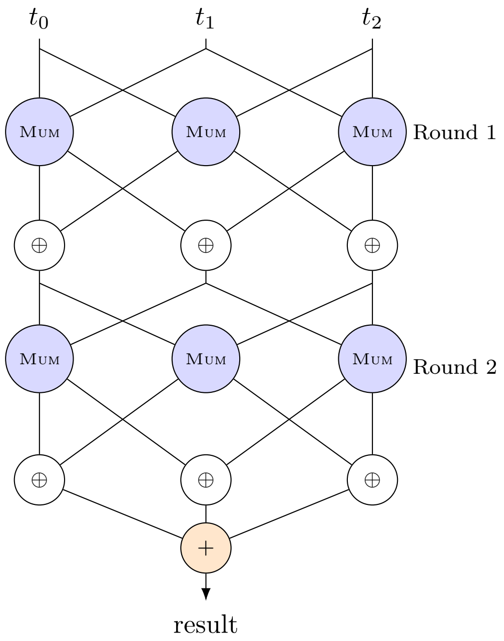


Figure 1: MuseAir finalizer. Each round performs three $\textsc{Mum}_{128}$ operations on all pairs of its three inputs, then cross-XORs the lo/hi halves. After two rounds the three values are summed for the 64-bit result.

7 xxHash Family

Source: `hashes/xxhash.cpp` [Col12].

xxHash uses prime constants $s_1, \ldots, s_5$ selected by empirical testing for avalanche properties. The primes have roughly half their bits set and avoid degenerate bit patterns. For xxHash64:

$s_1 =$ `0x9E3779B185EBCA87` $s_2 =$ `0xC2B2AE3D27D4EB4F`

$s_3 =$ `0x165667B19E3779F9` $s_4 =$ `0x85EBCA77C2B2AE63`

$s_5 =$ `0x27D4EB2F165667C5`

The 32-bit versions use corresponding 32-bit primes.

7.1 Round Functions

7.1.1 Multiply-Rotate Round

The xxHash round function mixes input into an accumulator:

$$\textsc{Round}(v, x) = \textsc{rot}_r(v + x \cdot s_2) \cdot s_1 \tag{29}$$

where $r = 13$ for 32-bit, $r = 31$ for 64-bit. This is an ARX variant using multiplication instead of XOR for the final mix.

7.1.2 Merge Round (xxHash64)

After the main loop, xxHash64 merges each accumulator into the hash:

$$\textsc{Merge}(h, v) = (h \oplus \textsc{Round}(0, v)) \cdot s_1 + s_4 \tag{30}$$

This "rounds" the accumulator value, XORs it into the hash, then applies another multiply-add for diffusion.

SMHasher3 test results. None of the xxHash variants pass the full SMHasher3 test battery. xxHash32 fails 83 of 250 tests, including numerous Sparse, Permutation, and seed-sensitivity tests (SeedBlockLen, SeedBlockOffset, SeedBIC, SeedBitflip), reflecting the limited mixing of its 32-bit multiply-rotate round. xxHash64 fares much better, failing only 9 tests, all seed-related (SeedBlockLen, SeedBlockOffset, SeedBIC). XXH3 (64-bit) fails 27 of 250 tests: BIC, Sparse, PerlinNoise, and Bitflip tests expose weaknesses in the non-seeded path, while SeedZeroes produces 571 full 64-bit collisions (expected 0) on keys up to 1280 bytes with low-weight seeds—a consequence of XXH3's seeding mechanism, which adds ±*seed* to the secret and thus maps *seed* $= 0$ and any seed that leaves all relevant secret words unchanged to identical output.

7.2 xxHash32

```
$v_0 \leftarrow seed + s_1 + s_2$
$v_1 \leftarrow seed + s_2$
$v_2 \leftarrow seed$
$v_3 \leftarrow seed - s_1$
if $n \geq 16$ then
    for each block $(x_0, x_1, x_2, x_3) \in \mathsf{u32}^4$ do        ▷ Main loop: 4 parallel lanes
        for $i \in \{0, \ldots, 3\}$ do
            $v_i \leftarrow \textsc{Round}(v_i, x_i)$
        end for
    end for
    $v_4 \leftarrow \textsc{rot}_1(v_0) + \textsc{rot}_7(v_1) + \textsc{rot}_{12}(v_2) + \textsc{rot}_{18}(v_3)$
else
    $v_4 \leftarrow seed + s_5$
end if
$v_4 \leftarrow v_4 + n$
for each remaining $x_i \in \mathsf{u32}$ do
    $v_4 \leftarrow \textsc{rot}_{17}(v_4 + x_i \cdot s_3) \cdot s_4$
end for
for each remaining $x_i \in \mathsf{u8}$ do
    $v_4 \leftarrow \textsc{rot}_{11}(v_4 + x_i \cdot s_5) \cdot s_1$
end for
return $\text{XXH32\_}\textsc{avalanche}(v_4)$
```

XXH32_avalanche: xxHash32 finalizer (an XMS cascade):

$$\mathrm{XXH32\_}\textsc{avalanche}(h) = \textsc{Xorshift}_{16}\Big(\mathrm{XMS}_{13,s_3^{32}}\Big(\mathrm{XMS}_{15,s_2^{32}}(h)\Big)\Big) \tag{31}$$

7.3 xxHash64

Same structure as xxHash32 with 64-bit words, $r = 31$, and 32-byte blocks.

```
$v_0 \leftarrow \mathit{seed} + s_1 + s_2$
$v_1 \leftarrow \mathit{seed} + s_2$
$v_2 \leftarrow \mathit{seed}$
$v_3 \leftarrow \mathit{seed} - s_1$
if $n \geq 32$ then
    for each block $(x_0, x_1, x_2, x_3) \in \mathsf{u64}^4$ do        ▷ Main loop: 4 parallel lanes
        for $i \in \{0, \ldots, 3\}$ do
            $v_i \leftarrow \textsc{Round}(v_i, x_i)$
        end for
    end for
    $v_4 \leftarrow \textsc{rot}_1(v_0) + \textsc{rot}_7(v_1) + \textsc{rot}_{12}(v_2) + \textsc{rot}_{18}(v_3)$
    for $i \in \{0, \ldots, 3\}$ do
        $v_4 \leftarrow \textsc{Merge}(v_4, v_i)$
    end for
else
    $v_4 \leftarrow \mathit{seed} + s_5$
end if
$v_4 \leftarrow v_4 + n$
for each remaining $x_i \in \mathsf{u64}$ do
    $v_4 \leftarrow \textsc{rot}_{27}(v_4 \oplus \textsc{Round}(0, x_i)) \cdot s_1 + s_4$
end for
for each remaining $x_i \in \mathsf{u32}$ do
    $v_4 \leftarrow \textsc{rot}_{23}(v_4 \oplus (x_i \cdot s_1)) \cdot s_2 + s_3$
end for
for each remaining $x_i \in \mathsf{u8}$ do
    $v_4 \leftarrow \textsc{rot}_{11}(v_4 \oplus (x_i \cdot s_5)) \cdot s_1$
end for
return $\mathrm{XXH64\_}\textsc{avalanche}(v_4)$
```

XXH64_avalanche: xxHash64 finalizer:

$$\mathrm{XXH64\_}\textsc{avalanche}(h) = \textsc{Xorshift}_{32}(\mathrm{XMS}_{29,s_3}(\mathrm{XMS}_{33,s_2}(h))) \tag{32}$$

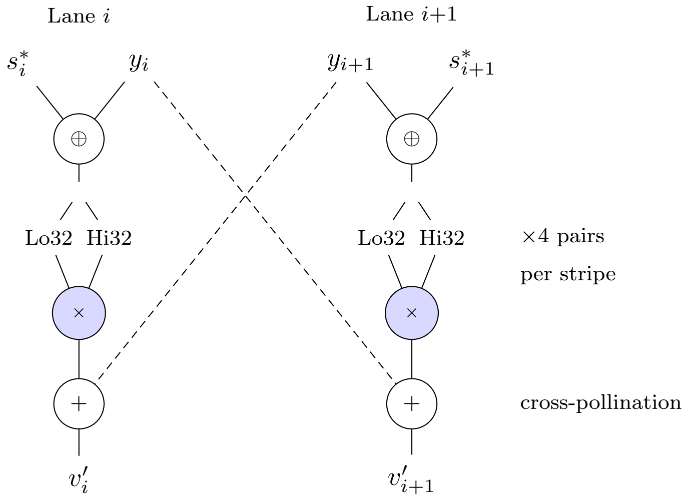
7.4 XXH3

Source: `hashes/xxhash.cpp`.

XXH3 (2019) uses a 192-byte secret S (pseudorandom bytes from FARSH; seeded variant adds ±*seed* to alternating 8-byte blocks). The 32×32 multiply has native SIMD support (unlike 64×64). Original input is added to accumulators to prevent multiply-by-zero weakness. Six distinct code paths optimize for different input sizes; short keys ($n \leq 240$) use specialized scalar paths with 64×64 MUM. We summarize those special cases and focus on the long-key path.

Long Keys ($n > 240$). Uses 8 parallel 64-bit accumulators with UMAC-inspired structure.

Accumulate (per 64-byte stripe). One stripe feeds 8 input words into 4 lane pairs. Within each pair, the two lanes cross-pollinate: the raw input of each lane is added to the *other* lane's accumulator, hardening against multiply-by-zero.



Key insight: The 32×32→64 multiply (not 64×64!) enables efficient SIMD: SSE2/AVX2 have `_mm_mul_epu32` for this operation. The cross-lane addition of original input (dashed lines) hardens against multiply-by-zero.

```
Initialize $(v_0, \ldots, v_7) \leftarrow (s_3^{32}, s_1^{64}, s_2^{64}, s_3^{64}, s_4^{64}, s_2^{32}, s_5^{64}, s_1^{32})$
Let $s_0^{\text{end}}, \ldots, s_7^{\text{end}}$ be eight 64-bit secret words from the end of $S^\star$
Let $s_0^{\text{m}}, \ldots, s_7^{\text{m}}$ be eight 64-bit secret words from $S^\star$ at byte offset 11
for each block $(x_i)_{i=0}^{127} \in \texttt{u64}^{128}$ do                ▷ 1024 bytes = 16 stripes
    for $j \in \{0, \ldots, 15\}$ do                                          ▷ Stripe $j$ (64 bytes)
        $(y_0, \ldots, y_7) \leftarrow (x_{8j}, \ldots, x_{8j+7})$
        Let $(s_0^*, \ldots, s_7^*) \in \texttt{u64}^8$ be the secret words for stripe $j$
        for $i \in \{0, \ldots, 7\}$ do
            $t \leftarrow y_i \oplus s_i^*$                      ▷ 32×32 multiply of two halves (not MUM)
            $v_i \leftarrow v_i + (\textsc{Lo32}(t) \cdot \textsc{Hi32}(t))$
            $v_{i\oplus 1} \leftarrow v_{i\oplus 1} + y_i$                         ▷ Cross-pollination: raw input
        end for
    end for
    for $i \in \{0, \ldots, 7\}$ do                                           ▷ Scramble accumulators
        $v_i \leftarrow (v_i \oplus (v_i \gg 47) \oplus s_i^{\text{end}}) \cdot s_1^{32}$
    end for
end for
Handle remaining bytes (short/medium paths summarized)
$v \leftarrow n \cdot s_1^{64}$                                                ▷ Merge accumulators
for $i \in \{0, 2, 4, 6\}$ do
    $v \leftarrow v + \textsc{Mum}(v_i \oplus s_i^{\text{m}}, v_{i+1} \oplus s_{i+1}^{\text{m}})$
end for
$v \leftarrow v \oplus (v \gg 37)$                         ▷ Finalization (xorshift/multiply avalanche)
$v \leftarrow v \cdot \texttt{0x165667919E3779F9}$
return $v \oplus (v \gg 32)$
```

8 SipHash Family

Source: `hashes/siphash.cpp` [AB12].

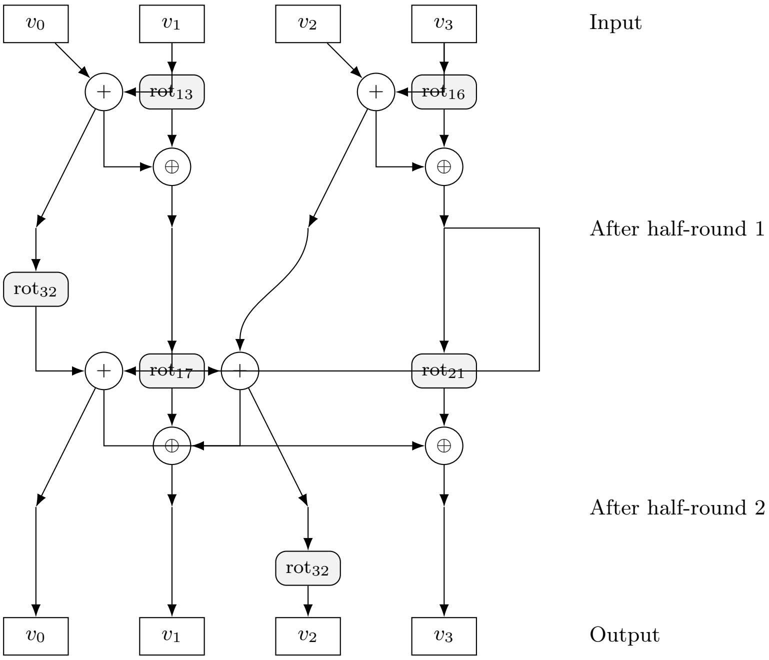
SipHash is a cryptographically-informed hash function designed to resist hash-flooding DoS attacks. Requires a 128-bit secret key. Uses an ARX construction (no multiplications) with a keyed permutation and Feistel-like cross-lane mixing. Sequential by design—cannot be parallelized across blocks.

8.1 Constants

Initialization vectors s_0, s_1, s_2, s_3 are fixed 64-bit constants (they encode the ASCII string "somepseudorandomlygeneratedbytes").

8.2 SipRound

The core permutation operates on four 64-bit state words (v_0, v_1, v_2, v_3). It consists of two half-rounds with a cross-over in between:



Each half-round: $v_i \leftarrow v_i + v_{i+1}$, then $v_{i+1} \leftarrow \text{ROT}_r(v_{i+1}) \oplus v_i$. After the first half-round, $v_0 \leftarrow \text{ROT}_{32}(v_0)$ and the next half-round re-pairs the lanes to mix (v_0, v_3) and (v_2, v_1). The second half-round uses rotations 17 and 21, followed by $v_2 \leftarrow \text{ROT}_{32}(v_2)$.

The complete algebraic form:

$$v_0 \leftarrow v_0 + v_1 \qquad v_2 \leftarrow v_2 + v_3 \tag{33}$$

$$v_1 \leftarrow \text{ROT}_{13}(v_1) \qquad v_3 \leftarrow \text{ROT}_{16}(v_3) \tag{34}$$

$$v_1 \leftarrow v_1 \oplus v_0 \qquad v_3 \leftarrow v_3 \oplus v_2 \tag{35}$$

$$v_0 \leftarrow \text{ROT}_{32}(v_0) \tag{36}$$

$$v_2 \leftarrow v_2 + v_1 \qquad v_0 \leftarrow v_0 + v_3 \tag{37}$$

$$v_1 \leftarrow \text{ROT}_{17}(v_1) \qquad v_3 \leftarrow \text{ROT}_{21}(v_3) \tag{38}$$

$$v_1 \leftarrow v_1 \oplus v_2 \qquad v_3 \leftarrow v_3 \oplus v_0 \tag{39}$$

$$v_2 \leftarrow \text{ROT}_{32}(v_2) \tag{40}$$

8.3 SipHash-c-d

Parameters c and d specify compression and finalization rounds.

```
$v_0 \leftarrow k_0 \oplus s_0$
$v_1 \leftarrow k_1 \oplus s_1$
$v_2 \leftarrow k_0 \oplus s_2$
$v_3 \leftarrow k_1 \oplus s_3$
for each block $x \in$ u64 do
    $v_3 \leftarrow v_3 \oplus x$
    for $i \leftarrow 1$ to $c$ do                          ▷ Compression rounds
        SipRound
    end for
    $v_0 \leftarrow v_0 \oplus x$
end for
$x \leftarrow$ padded final block with length byte
$v_3 \leftarrow v_3 \oplus x$
for $i \leftarrow 1$ to $c$ do                              ▷ Compression rounds
    SipRound
end for
$v_0 \leftarrow v_0 \oplus x$
$v_2 \leftarrow v_2 \oplus$ 0xff
for $i \leftarrow 1$ to $d$ do                              ▷ Finalization rounds
    SipRound
end for
return $v_0 \oplus v_1 \oplus v_2 \oplus v_3$
```

Common variants: SipHash-2-4 (c=2, d=4): recommended. SipHash-1-3 (c=1, d=3): faster, lower security margin.

8.4 HalfSipHash

32-bit variant for resource-constrained environments (used in Linux kernel).

SipRound (32-bit): Different rotation constants: $(5, 8, 7, 13, 16)$ instead of $(13, 16, 17, 21, 32)$.

9 Other Notable Hashes

9.1 ARX-Based Hashes

These hashes use Add-Rotate-XOR primitives without multiplication in their main loops, relying on superscalar execution for parallelism.

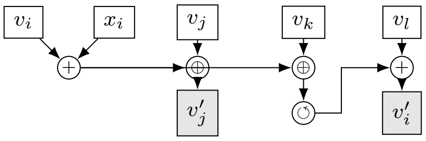
9.1.1 SpookyHash

Source: `hashes/spookyhash.cpp` [Jen12].

By Bob Jenkins (creator of lookup3). A 128-bit ARX hash using 12 parallel 64-bit lanes with 96-byte blocks. Version 2 is most commonly used.

Mix Function. The Mix function updates 12 state variables with 12 input words simultaneously. Each line combines: word addition, cross-lane XOR, self-XOR, rotation, and accumulation into adjacent lane. Rotations are: 11, 32, 43, 31, 17, 28, 39, 57, 55, 54, 22, 46.

Pattern for each lane i:



Each of the 12 lanes follows this pattern with different rotation amounts and cross-lane indices j, k, l.

Exact long-key round functions (SpookyHash v2):

```
function Mix($x_0, \dots, x_{11}, v_0, \dots, v_{11}$)          ▷ 96 bytes into 12 lanes
    Let $(r_0, \dots, r_{11}) \leftarrow (11, 32, 43, 31, 17, 28, 39, 57, 55, 54, 22, 46)$
    for $i \in \{0, \dots, 11\}$ do                              ▷ All indices mod 12
        $v_i \leftarrow v_i + x_i$
        $v_{(i+2) \bmod 12} \leftarrow v_{(i+2) \bmod 12} \oplus v_{(i+10) \bmod 12}$
        $v_{(i+11) \bmod 12} \leftarrow v_{(i+11) \bmod 12} \oplus v_i$
        $v_i \leftarrow \mathrm{ROT}_{r_i}(v_i)$
        $v_{(i+11) \bmod 12} \leftarrow v_{(i+11) \bmod 12} + v_{(i+1) \bmod 12}$
    end for
end function
function EndPartial($v_0, \dots, v_{11}$)                          ▷ Finalizer step (run 3 times)
    Let $(r_0, \dots, r_{11}) \leftarrow (44, 15, 34, 21, 38, 33, 10, 13, 38, 53, 42, 54)$
    for $i \in \{0, \dots, 11\}$ do                              ▷ All indices mod 12
        $v_{(i+11) \bmod 12} \leftarrow v_{(i+11) \bmod 12} + v_{(i+1) \bmod 12}$
        $v_{(i+2) \bmod 12} \leftarrow v_{(i+2) \bmod 12} \oplus v_{(i+11) \bmod 12}$
        $v_{(i+1) \bmod 12} \leftarrow \mathrm{ROT}_{r_i}(v_{(i+1) \bmod 12})$
    end for
end function
```

SpookyHash (Long Path).

```
$v_0, v_3, v_6, v_9 \leftarrow seed_1$
$v_1, v_4, v_7, v_{10} \leftarrow seed_2$
$v_2, v_5, v_8, v_{11} \leftarrow$ 0xdeadbeefdeadbeef
for each block $(x_0, \dots, x_{11}) \in$ u64$^{12}$ do             ▷ Main loop: 96-byte blocks
    Mix($x_0, \dots, x_{11}, v_0, \dots, v_{11}$)
end for
Let $(x_0, \dots, x_{11}) \in$ u64$^{12}$ be the final padded block (zeros + length byte)
for $i \in \{0, \dots, 11\}$ do
    $v_i \leftarrow v_i + x_i$
end for
for $r \in \{1, 2, 3\}$ do
    EndPartial($v_0, \dots, v_{11}$)
end for
return $(v_0, v_1)$                                              ▷ 128-bit output
```

SpookyHash v1 difference. SpookyHash v1 applies one extra Mix to the final padded block before the three EndPartial rounds.

9.1.2 FarmHash

Source: `hashes/farmhash.cpp`, version 1.1 [Pik14].

By Geoff Pike at Google, successor to CityHash. Uses Murmur3-derived constants and offers multiple implementation variants (na, uo, xo) optimized for different input sizes.

Constants. From MurmurHash3:

$$s_0 = \texttt{0xc3a5c85c97cb3127} \qquad s_1 = \texttt{0xb492b66fbe98f273}$$
$$s_2 = \texttt{0x9ae16a3b2f90404f} \qquad s_3 = \texttt{0x9ddfea08eb382d69}$$

HashLen16.

```
function HashLen16($u, v, m$)                                  ▷ 128-to-64 combiner
    $t_0 \leftarrow$ ShiftMix($(u \oplus v) \cdot m$)
    $t_1 \leftarrow$ ShiftMix($(v \oplus t_0) \cdot m$)
    return $t_1 \cdot m$
end function
```

WeakHashLen32WithSeeds.

```
function WeakHashLen32WithSeeds($x_0, x_1, x_2, x_3, a, b$)    ▷ 128-bit for 32 bytes
    $a \leftarrow a + x_0$
    $b \leftarrow \mathrm{ROT}_{43}(b + a + x_3)$                    ▷ ROTR$_{21}$
    $t \leftarrow a$; $a \leftarrow a + x_1 + x_2$
    $b \leftarrow b + \mathrm{ROT}_{20}(a)$                          ▷ ROTR$_{44}$
    return $(a + x_3, b + t)$
end function
```

FarmHash64 (Long Path). Internal state: 7 words $(v_0, \ldots, v_6)$.

```
$v_0 \leftarrow 81$, $v_1 \leftarrow 81 \cdot s_1 + 113$
$v_2 \leftarrow$ ShiftMix($v_1 \cdot s_2$) $\cdot s_2$
$(v_3, v_4), (v_5, v_6) \leftarrow (0, 0), (0, 0)$
for each block $(x_0, \ldots, x_7) \in$ u64$^8$ do                ▷ 64-byte blocks
    On first block: $v_0 \leftarrow v_0 \cdot s_2 + x_0$, then continue below
    $v_0 \leftarrow \mathrm{ROT}_{37}(v_0 + v_1 + v_3 + x_1) \cdot s_1$
    $v_1 \leftarrow \mathrm{ROT}_{42}(v_1 + v_4 + x_6) \cdot s_1$
    $v_0 \leftarrow v_0 \oplus v_6$; $v_1 \leftarrow v_1 + v_3 + x_5$
    $v_2 \leftarrow \mathrm{ROT}_{33}(v_2 + v_5) \cdot s_1$
    $(v_3, v_4) \leftarrow$ WeakHashLen32WithSeeds($x_0, x_1, x_2, x_3, v_4 \cdot s_1, v_0 + v_5$)
    $(v_5, v_6) \leftarrow$ WeakHashLen32WithSeeds($x_4, x_5, x_6, x_7, v_2 + v_6, v_1 + x_2$)
    swap $v_2, v_0$
end for
Process final 64 bytes with length-dependent multiplier $m$
return HashLen16($v_3 + v_5, v_4 + v_6, m$)
```

9.1.3 Rainbow

Source: `hashes/rainbow.cpp`, version 3.7.1 [Str23].

By Cris Stringfellow (DOSYAGO). A stream-based hash with 256-bit internal state using multiply-rotate primitives.

Constants. Eight prime constants $s_0, \ldots, s_7$ with good avalanche properties ($s_0 = 2^{64} - 59$, etc.).

```
function mixA($v_0, v_1, v_2, v_3$)                            ▷ Full state mixing
    $v_0 \leftarrow \mathrm{ROT}_{23}(v_0 \cdot s_0) \cdot s_1$
    $v_1 \leftarrow \textsc{rot}_{29}((v_1 \oplus v_0) \cdot s_2) \cdot s_3$
    $v_2 \leftarrow \textsc{rot}_{31}(v_2 \cdot s_4) \cdot s_5$
    $v_3 \leftarrow \textsc{rot}_{37}((v_3 \oplus v_2) \cdot s_6) \cdot s_7$
end function
function $\textsc{mix}\mathrm{B}(v_0, v_1, v_2, v_3, \mathit{seed})$        ▷ Lighter; operates on $v_1, v_2$ only
    $v_1 \leftarrow \textsc{rot}_{23}(v_1 \cdot s_6) \cdot s_7$
    $v_2 \leftarrow \textsc{rot}_{23}((v_2 \oplus v_1 + \mathit{seed}) \cdot s_2) \cdot s_3$
    swap $v_1, v_2$
end function

$(v_0, v_1, v_2, v_3) \leftarrow (\mathit{seed} + n + 1, \mathit{seed} + n + 2, \mathit{seed} + n + 3, \mathit{seed} + n + 5)$
for each block $(x_0, x_1) \in \mathtt{u64}^2$ do        ▷ 16-byte blocks
    $v_0 \leftarrow v_0 - x_0$; $v_1 \leftarrow v_1 + x_0$
    $v_2 \leftarrow v_2 + x_1$; $v_3 \leftarrow v_3 - x_1$
    if block index is even then
        $\textsc{mix}\mathrm{A}(v_0, v_1, v_2, v_3)$
    else
        $\textsc{mix}\mathrm{B}(v_0, v_1, v_2, v_3, \mathit{seed})$
        $(v_0, v_1, v_2, v_3) \leftarrow (v_3, v_0, v_1, v_2)$        ▷ Rotate state right
    end if
end for
Handle remaining bytes
$\textsc{mix}\mathrm{A}$; $\textsc{mix}\mathrm{B}$; $\textsc{mix}\mathrm{A}$
return $-(v_2 + v_3)$
```

9.2 PRNG-Style Hashes

These hashes have structure resembling pseudo-random number generators, often using LCG-like state updates.

9.2.1 komihash

Source: `hashes/komihash.cpp`, version 5.27 [Van21].

A MUM-based hash with PRNG-inspired structure. Uses constants derived from the mantissa bits of π.

Core Primitive. The hash uses a modified MUM that accumulates the high part:

$$\textsc{kh\_m}128(m_1, m_2, \mathit{lo}, \mathit{hi}) : \mathit{lo} \leftarrow \textsc{Lo}(m_1 \times m_2), \quad \mathit{hi} \leftarrow \mathit{hi} + \textsc{Hi}(m_1 \times m_2) \tag{41}$$

Equivalently, a single "lane round" can be written as:

$$\textsc{kh\_round}(u, v, h) : \quad (t_{\mathrm{lo}}, t_{\mathrm{hi}}) \leftarrow \textsc{Mum}_{128}(u, v) \tag{42}$$

$$\text{return } (t_{\mathrm{lo}}, h + t_{\mathrm{hi}}) \tag{43}$$

where in the main loop we use $u \leftarrow v_p \oplus x_p$ and $v \leftarrow v_q \oplus x_q$.

komihash (Long Path). Constants $s_1, \ldots, s_8$ are derived from π.

```
$v_1 \leftarrow s_1 \oplus (\mathit{seed} \wedge \mathtt{0x5555...})$        ▷ Checkerboard seed split
$v_5 \leftarrow s_5 \oplus (\mathit{seed} \wedge \mathtt{0xAAAA...})$
$(v_1, v_5) \leftarrow \textsc{kh\_round}(v_1, v_5, v_5)$        ▷ Initial round for diffusion
$v_i \leftarrow s_i \oplus v_1$ for $i \in \{2, 3, 4\}$;    $v_i \leftarrow s_i \oplus v_5$ for $i \in \{6, 7, 8\}$
```

```
for each block $(x_0, \ldots, x_7) \in \texttt{u64}^8$ do                ▷ 4 parallel lanes
    for $i \in \{1, \ldots, 4\}$ do
        $(v_i, v_{i+4}) \leftarrow$ KH_ROUND$(v_i \oplus x_{i-1},\ v_{i+4} \oplus x_{i+3},\ v_{i+4})$
    end for
    $v_4 \leftarrow v_4 \oplus v_7$; $v_1 \leftarrow v_1 \oplus v_8$; $v_3 \leftarrow v_3 \oplus v_6$; $v_2 \leftarrow v_2 \oplus v_5$        ▷ Cross-lane
end for
for $i \in \{6, \ldots, 8\}$ do                ▷ Collapse
    $v_5 \leftarrow v_5 \oplus v_i$
end for
for $i \in \{2, \ldots, 4\}$ do
    $v_1 \leftarrow v_1 \oplus v_i$
end for
Handle remaining bytes (up to 2 more 16-byte rounds via KH_M128)
$v_1 \leftarrow v_1 \oplus v_5$                ▷ Final mixing
Apply one more KH_ROUND, then $v_1 \leftarrow v_1 \oplus v_5$
return $v_1$
```

9.2.2 prvhash

Source: `hashes/prvhash.cpp`, version 4.3.4 [Van20].

By Aleksey Vaneev. A PRNG-style hash that generates pseudo-random values during hashing. Uses checkerboard constants like a5hash.

Core Function.

```
function PRVHASH_CORE64$(v_0, v_1, v_2)$
    $v_0 \leftarrow v_0 \cdot (v_1 \cdot 2 + 1)$
    $t \leftarrow (v_0 \gg 32) \mid (v_0 \ll 32)$                ▷ Swap halves
    $v_2 \leftarrow v_2 + t + \texttt{0xAAAAAAAAAAAAAAAA}$
    $v_1 \leftarrow v_1 + v_0 + \texttt{0x5555555555555555}$
    $v_0 \leftarrow v_0 \oplus v_2$
    return $v_1 \oplus t$
end function
```

prvhash64.

```
Initialize $(v_0, v_1, v_2)$ from pre-computed constants
$v_0 \leftarrow v_0 \oplus seed$; $v_1 \leftarrow v_1 \oplus seed$
for each word $x \in \texttt{u64}$ do
    PRVHASH_CORE64$(v_0, v_1, v_2)$
    $v_0 \leftarrow v_0 \oplus x$; $v_1 \leftarrow v_1 \oplus x$
end for
Apply 2 more core rounds
return PRVHASH_CORE64 output
```

9.3 Compact and Embedded Hashes

These hashes prioritize small code size and simplicity, suitable for resource-constrained environments.

9.3.1 ChibiHash

Source: `hashes/chibihash.cpp`, version 2 [NRK24].

A compact, high-quality 64-bit hash by NRK with small code footprint for embedded use. Uses a single 64-bit constant (digits of e) and truncated 64×64→64 multiplies (not MUM). Cross-lane rotation feeding prevents lane independence.

```
$s_0 \leftarrow$ 0x2B7E151628AED2A7                                  ▷ Digits of $e$
$t \leftarrow \mathrm{ROT}_{15}(seed - s_0) + \mathrm{ROT}_{47}(seed - s_0)$
$v_0, v_1, v_2, v_3 \leftarrow [seed,\ seed + s_0,\ t,\ t + s_0^2 \oplus s_0]$
for each block $(x_0, \ldots, x_3) \in$ u64$^4$ do                  ▷ 32-byte blocks
    for $i \in \{0, \ldots, 3\}$ do
        $v_i \leftarrow (x_i + v_i) \cdot s_0$
        $v_{(i+1) \bmod 4} \leftarrow v_{(i+1) \bmod 4} + \mathrm{ROT}_{27}(x_i)$
    end for
end for
Handle remaining bytes
$v_0 \leftarrow v_0 + (\mathrm{ROT}_{31}(v_2 \cdot s_0) \oplus (v_2 \gg 31))$
$v_1 \leftarrow v_1 + (\mathrm{ROT}_{31}(v_3 \cdot s_0) \oplus (v_3 \gg 31))$
$v_0 \leftarrow v_0 \cdot s_0$; $v_0 \leftarrow v_0 \oplus (v_0 \gg 31)$
$v_1 \leftarrow v_1 + v_0$
$t \leftarrow n \cdot s_0$; $t \leftarrow t \oplus \mathrm{ROT}_{29}(t)$; $t \leftarrow t + seed$; $t \leftarrow t \oplus v_1$
$t \leftarrow t \oplus \mathrm{ROT}_{15}(t) \oplus \mathrm{ROT}_{42}(t)$; $t \leftarrow t \cdot s_0$
return $t \oplus \mathrm{ROT}_{13}(t) \oplus \mathrm{ROT}_{31}(t)$
```

9.3.2 XMSX

Source: `hashes/xmsx.cpp`.

By Dmitrii Lebed. A minimal 32-bit hash for microcontrollers using XOR-Multiply-Shift-XOR rounds. Uses 32×32→64 multiply (widely available in HW). Faster than software CRC32 on 32-bit CPUs.

Round Function.

$$\mathrm{XMSX32\_ROUND}(v, x) = ((v \oplus x) \cdot s_0) \oplus (((v \oplus x) \cdot s_0) \gg 32) \tag{44}$$

where $s_0 =$ `0xcdb32970830fcaa1`.

XMSX32 Algorithm.

```
$v \leftarrow (seed \ll 32) \mid seed$
$v \leftarrow \mathrm{XMSX32\_ROUND}(v, n)$
for each word $x \in$ u32 do
    $v \leftarrow \mathrm{XMSX32\_ROUND}(v, x)$
end for
return $\mathrm{XMSX32\_ROUND}(v, v \gg 47)$
```

10 CLMUL and Polynomial Hashes

These hashes use carry-less multiplication (CLMUL) or polynomial arithmetic over finite fields to achieve theoretically grounded collision bounds.

Carry-less vs. standard multiplication. In standard multiplication, partial products are added with carry propagation. In carry-less multiplication (polynomial multiplication over GF(2)), partial products are XORed without carries:

Standard (with carry)

$$\begin{array}{rl} \times\ 1\ 0\ 1\ 1 & \\ 1\ 1\ 0\ 1 & \\ \hline & \text{partial products} \\ & \text{added with carry} \\ \hline = 143\ (8\ \text{bits}) & \end{array}$$

Carry-less (XOR)

$$\begin{array}{rl} \otimes\ 1\ 0\ 1\ 1 & \\ 1\ 1\ 0\ 1 & \\ \hline & \text{partial products} \\ & \text{XORed (no carry)} \\ \hline = 127\ (7\ \text{bits}) & \end{array}$$

The PCLMULQDQ instruction performs 64×64→128 carry-less multiplication in a single cycle on modern CPUs.

Pseudocode primitives.

$$\textsc{clmul}(a, b) = a \otimes b \quad \text{(128-bit carry-less product of two 64-bit inputs)} \tag{45}$$

Both map to a single `PCLMULQDQ` instruction with ∼1 cycle throughput. The symbol $\otimes$ denotes polynomial multiplication in GF(2)[x] (XOR replaces addition, AND replaces multiplication of coefficients).

10.1 CLHash

Source: `hashes/clhash.cpp` [LK16].

SMHasher3 test results. CLHash fails 186 of 250 tests, the worst result of any hash in this survey. Failures span nearly every category: Avalanche, BIC, Sparse, Permutation, Text, TwoBytes, PerlinNoise, Bitflip, and extensive seed-related tests. Many seed-test failures are expected: CLHash is a keyed hash designed for a full random key, and SMHasher3's seed-sensitivity tests probe a regime outside its design model. The non-seed failures (Avalanche, Sparse, Text) indicate that the CLMUL-based compression without a strong finalizer leaves detectable bias in the output.

A theoretically-grounded hash by Daniel Lemire using carry-less multiplication (CLMUL instruction, x86-64 Haswell+). Based on almost-universal hashing over GF(2^{127}) with collision bound $\epsilon \approx n/2^{64}$. Uses 133 random 64-bit keys derived from seed.

10.1.1 Mathematical Foundation

CLHash computes a polynomial hash over GF(2^{127}) with irreducible polynomial $p(x) = x^{127} + x + 1$. (Irreducibility over GF(2^{128}) ensures the field structure.) The core operation is carry-less multiplication:

$$a \otimes b = a \cdot b \mod p(x) \tag{46}$$

where $\cdot$ denotes polynomial multiplication in GF(2)[x] (XOR instead of addition, AND instead of multiplication for coefficients).

10.1.2 Key Primitives

Lazy mod $2^{127} + 2 + 1$: Given 254-bit product $(A_{\text{hi}}, A_{\text{lo}})$:

$$A_{\text{lo}} \oplus (A_{\text{hi}} \ll 1) \oplus (A_{\text{hi}} \ll 2) \tag{47}$$

128-to-64 reduction: Uses the irreducible polynomial $p(x) = x^{64} + x^4 + x^3 + x + 1$ (notation: $(64, 4, 3, 1, 0)$). The reduction is performed in two steps:

Step 1: CLMUL reduction. Let $C = x^4 + x^3 + x + 1$ (the low-degree terms of $p(x)$). Since $x^{64} \equiv C \pmod{p(x)}$, we have $A_{\text{hi}} \cdot x^{64} \equiv A_{\text{hi}} \cdot C$:

$$Q_2 \leftarrow \textsc{clmul}(A_{\text{hi}}, C) \tag{48}$$

This produces a result up to $64 + 4 = 68$ bits (since A_{hi} is 64 bits and C is degree 4).

Step 2: Table lookup for final bits. The CLMUL result Q_2 may have bits in positions 64–67. These are reduced using a precomputed 16-entry table where entry i contains $i \cdot C \bmod 2^{64}$:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$i \cdot C$	0	27	54	45	108	119	90	65	216	195	238	245	180	175	130	153

The high 4 bits of Q_2 (bits 64–67) index into this table via `pshufb`:

$$Q_3 \leftarrow \textit{table}[(Q_2 \gg 64) \wedge \texttt{0xF}] \tag{49}$$

Final combination:

$$\textsc{precompReduction64}(A) = A_{\text{lo}} \oplus Q_2 \oplus Q_3 \tag{50}$$

The table lookup replaces what would otherwise require another CLMUL for the overflow bits, saving one instruction in the critical path.

10.1.3 CLHash Algorithm

```
Generate 133 random 64-bit key words $s_0, \ldots, s_{132}$ from seed using xorshift128+
$s_{\text{poly}} \leftarrow (s_{128} \mid s_{129} \ll 64) \wedge (\texttt{0x3FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF})$   ▷ 128-bit; top 2 bits cleared for GF($2^{127}$)
$s_{\text{final}} \leftarrow (s_{130}, s_{131}) \in \texttt{u128}$;   $s_{\text{len}} \leftarrow s_{132} \in \texttt{u64}$
$v \leftarrow 0$ (128-bit)
for each block $(x_i)_{i=0}^{127} \in \texttt{u64}^{128}$ do   ▷ 1024 bytes; Horner: $v \leftarrow v \otimes s_{\text{poly}} \oplus h_i$
    if not first block then
        $v \leftarrow v \otimes s_{\text{poly}}$
    end if
    for $i \in \{0, \ldots, 63\}$ do   ▷ 64 word pairs
        $v \leftarrow v \oplus \textsc{clmul}((x_{2i} \oplus s_{2i}), (x_{2i+1} \oplus s_{2i+1}))$
    end for
end for
for remaining word pairs $(x_{2i}, x_{2i+1})$ do   ▷ Tail: same CLMUL pattern
    $v \leftarrow v \oplus \textsc{clmul}((x_{2i} \oplus s_{2i}), (x_{2i+1} \oplus s_{2i+1}))$
end for
$t \leftarrow v \oplus s_{\text{final}}$;   $v \leftarrow \textsc{clmul}(\textsc{Lo}(t), \textsc{Hi}(t))$
$v \leftarrow v \oplus \textsc{clmul}(s_{\text{len}}, n)$   ▷ Mix in length
return $\textsc{precompReduction64}(v)$   ▷ bitmix variant applies MurmurHash3 $\textsc{fmix64}$ after
```

10.2 UMASH

Source: `hashes/umash.cpp` [Khu20b].

SMHasher3 test results. UMASH-128 fails 128 of 250 tests. Like CLHash, UMASH is a keyed hash expecting a full random key (generated via Salsa20), and most failures are in seed-related tests (Seed, SeedSparse, SeedAvalanche, SeedBIC, SeedBitflip, SeedBlockLen, SeedBlockOffset). Non-seed failures include BIC, some Sparse configurations, Permutation (single-bit keys), TwoBytes, PerlinNoise, and Bitflip—reflecting the inherent linearity of CLMUL-based compression.

UMASH by Paul Khuong combines CLMUL-based compression with polynomial hashing over GF($2^{61}-1$). It produces two independent hash functions for fingerprinting (128-bit output).

10.2.1 Mathematical Foundation

UMASH works modulo the Mersenne prime $p = 2^{61} - 1$. Key operations:

Modular addition (fast):

$$\textsc{add\_mod\_fast}(x, y) = \begin{cases} x + y + 8 & \text{if } x + y \text{ overflows} \\ x + y & \text{otherwise} \end{cases} \tag{51}$$

Modular multiplication:

$$\textsc{mul\_mod\_fast}(m, x) = \textsc{add\_mod\_fast}(\textsc{Lo}(m \times x), 8 \cdot \textsc{Hi}(m \times x)) \tag{52}$$

Horner double update:

$$\begin{aligned} \textsc{horner\_double\_update}(v, m_0, m_1, x, y) = {} & \textsc{mul\_mod\_fast}(m_0, v + x) \\ & + \textsc{mul\_mod\_fast}(m_1, y) \end{aligned} \tag{53}$$

10.2.2 OH Compression

The "OH" (One-pass Hash) compresses 256-byte blocks using CLMUL:

```
Interpret the 256-byte input as $(x_i)_{i=0}^{31} \in \mathsf{u64}^{32}$
$v \leftarrow 0$ (128-bit)
for $i \in \{0, \ldots, 14\}$ do                                   ▷ 15 CLMUL pairs
    $v \leftarrow v \oplus \textsc{clmul}(x_{2i} \oplus s_{2i}, x_{2i+1} \oplus s_{2i+1})$
end for
ENH finalization:
$(t_0, t_1) \leftarrow (x_{30} + s_{30},\ x_{31} + s_{31})$                ▷ Final ENH chunk
$t_2 \leftarrow \textsc{Hi}(t_0 \times t_1) + \mathit{seed}$; $t_3 \leftarrow \textsc{Lo}(t_0 \times t_1)$
return $(v_{\text{lo}} \oplus t_3,\ v_{\text{hi}} \oplus t_2 \oplus t_3)$
```

10.2.3 UMASH Algorithm

We focus on the main (long-key) structure. Short-key special casing and exact tail handling are summarized.

```
Generate 34 random 64-bit key words $s_0, \ldots, s_{33}$ from seed via Salsa20
Prepare polynomial multipliers $f, f^2$ in GF($2^{61} - 1$)
$v \leftarrow 0$
for each block $(x_i)_{i=0}^{31} \in \mathsf{u64}^{32}$ do                  ▷ Main loop (256 bytes)
    $(t_0, t_1) \leftarrow \text{OH\_}\textsc{compress}((x_i)_{i=0}^{31})$
    $v \leftarrow \textsc{horner\_double\_update}(v, f^2, f, t_0, t_1)$
end for
Process remaining bytes with variable-length OH (tail summarized)
return $(v \oplus \textsc{rot}_8(v)) \oplus \textsc{rot}_{33}(v)$                     ▷ Finalization
```

10.2.4 Fingerprinting (128-bit)

For fingerprinting, UMASH computes two independent hashes using:

- Different polynomial multipliers (f_0, f_0^2) and (f_1, f_1^2)
- A "twisted" version using an LRC (longitudinal redundancy check) XOR [Khu20a]
- Both hashes use the same OH keys but different polynomial accumulators

10.3 Polymur

Source: `hashes/polymur.cpp` [Pet23].

Polymur by Orson Peters is a polynomial hash over $\mathrm{GF}(2^{61}-1)$. Uses 56-bit chunks to avoid reduction within chunk processing. The polynomial multiplier s_0 is chosen as a generator of the multiplicative group (via $s_0 = 37^e$ for random odd e coprime to $2^{61}-2$), with $s_0^7 < 2^{60} - 2^{56}$ ensuring efficient reduction.

```
Keys: $(s_0, s_1) \leftarrow$ secret parameters derived from seed          ▷ $s_0$ multiplier, $s_1$ tweak
$v \leftarrow 0$
**if** $n \leq 7$ **then**
    **return** $s_1 + (s_0 + x_0)(s_0^2 + n) \mod p$
**else if** $n \geq 50$ **then**
    **while** $n \geq 50$ **do**                    ▷ Main loop: 7×56-bit chunks per iteration
        Let $(x_0, \ldots, x_6)$ be the next 7 words (each 56 bits, zero-extended)
        $t_0 \leftarrow (s_0 + x_0)(s_0^6 + x_1)$
        $t_1 \leftarrow (s_0^2 + x_2)(s_0^5 + x_3)$
        $t_2 \leftarrow (s_0^3 + x_4)(s_0^4 + x_5)$
        $t_3 \leftarrow (v + x_6) \cdot s_0^7$
        $v \leftarrow (t_0 + t_1 + t_2 + t_3) \mod p$
    **end while**
    Finalize with $v \cdot s_0^{14}$
**end if**
Handle tail (8–49 bytes) with overlapping reads
$s_2 \leftarrow$ 0xe9846af9b1a615d                                          ▷ mx3 finalizer
$v \leftarrow \textsc{Xorshift}_{32}(v) \cdot s_2$
$v \leftarrow \textsc{Xorshift}_{32}(v) \cdot s_2$
$v \leftarrow \textsc{Xorshift}_{28}(v)$
**return** $v + s_1$
```

10.4 poly-mersenne

Source: `hashes/poly_mersenne.cpp` [AKT20].

A simple polynomial hash over the Mersenne prime $p = 2^{61} - 1$. Outputs 32 bits. Keys from SplitMix64: multiplier $s_0 < 2^{60}$ and random values $s_1, \ldots, s_{K+1}$ for optional K-independence.

Core primitive. The multiply-accumulate uses *lazy Mersenne reduction*: since $2^{61} \equiv 1 \pmod{p}$, a 128-bit product can be reduced by folding the bits above position 61 back into the low part with a shift and add—no division required.

```
**function** MULT_COMBINE61$(v, m, x)$                    ▷ $(v \cdot m + x) \bmod (2^{61} - 1)$
    $(lo, hi) \leftarrow v \times m$                              ▷ Full 128-bit product
    $lo \leftarrow lo + x$                                        ▷ Accumulate input word
    $t \leftarrow (hi \ll 3) \mid (lo \gg 61)$                    ▷ Bits $\geq 2^{61}$: fold back
    $lo \leftarrow lo \wedge (2^{61} - 1)$
```

```
        return $lo + t$                                ▷ Lazy: may exceed $p$ by at most 1
    end function
```

The final result is fully reduced ($v \leftarrow v - p$ if $v \geq p$) once, after all rounds.

```
v ← n                                                  ▷ Initialize with length
for each word $x \in$ u32 do
    $v \leftarrow$ MULT_COMBINE61$(v, s_0, x)$
end for
if $K > 0$ then                                        ▷ Optional $K$-independence transform
    $t \leftarrow v$; $v \leftarrow s_1$
    for $i \in \{2, \ldots, K+1\}$ do: $v \leftarrow$ MULT_COMBINE61$(v, t, s_i)$
    end for
end if
return $v \bmod 2^{32}$                                ▷ After final reduction if $v \geq p$
```

Variants with higher K provide K-wise independence (collision bound $(n/2^{61})^{K+1}$):

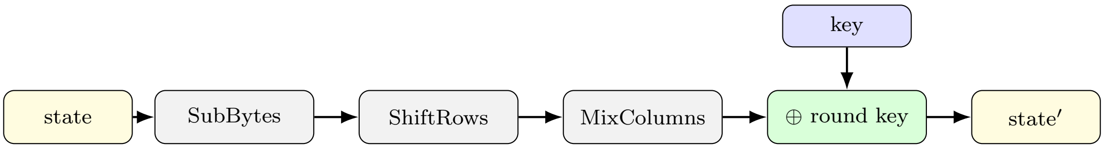
Variant	K	Collision bound
poly-mersenne-deg0	0	$n/2^{61}$
poly-mersenne-deg1	1	$(n/2^{61})^2$
poly-mersenne-deg2	2	$(n/2^{61})^3$
poly-mersenne-deg3	3	$(n/2^{61})^4$
poly-mersenne-deg4	4	$(n/2^{61})^5$

Higher-degree variants provide K-wise independence at the cost of additional multiplications, improving theoretical collision bounds.

11 AES-Based Hashes

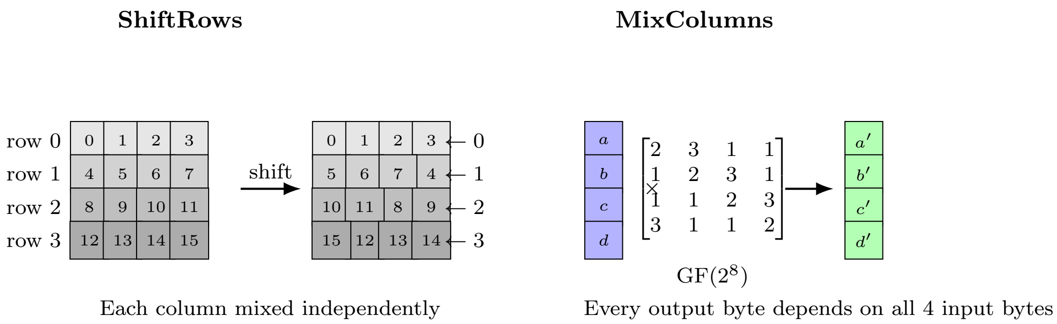
These hashes leverage the AES-NI instructions (`aesenc`, `aesdec`) for fast mixing. AES provides excellent diffusion in a single instruction.

AES round structure. Each `aesenc` instruction applies three transformations to a 128-bit state, then XORs a round key:



`aesenc(state, key)`

State transformations. The AES state is a 4×4 matrix of bytes. ShiftRows and MixColumns provide diffusion across the entire state:



ShiftRows

MixColumns

Each column mixed independently

Every output byte depends on all 4 input bytes

SubBytes: Non-linear S-box substitution (each byte independently). **ShiftRows**: Rotates each row by different amounts (0, 1, 2, 3 positions left). **MixColumns**: Multiplies each 4-byte column by a fixed matrix in GF(2^8). After one round, each output bit depends on multiple input bits—full avalanche requires ∼2 rounds.

Pseudocode primitives. We use AESENC and AESDEC as 128-bit → 128-bit operations throughout:

$$\textsc{aesenc}(v, k) = \textsc{MixColumns}(\textsc{ShiftRows}(\textsc{SubBytes}(v))) \oplus k \tag{54}$$

$$\textsc{aesdec}(v, k) = \textsc{InvMixColumns}(\textsc{InvShiftRows}(\textsc{InvSubBytes}(v))) \oplus k \tag{55}$$

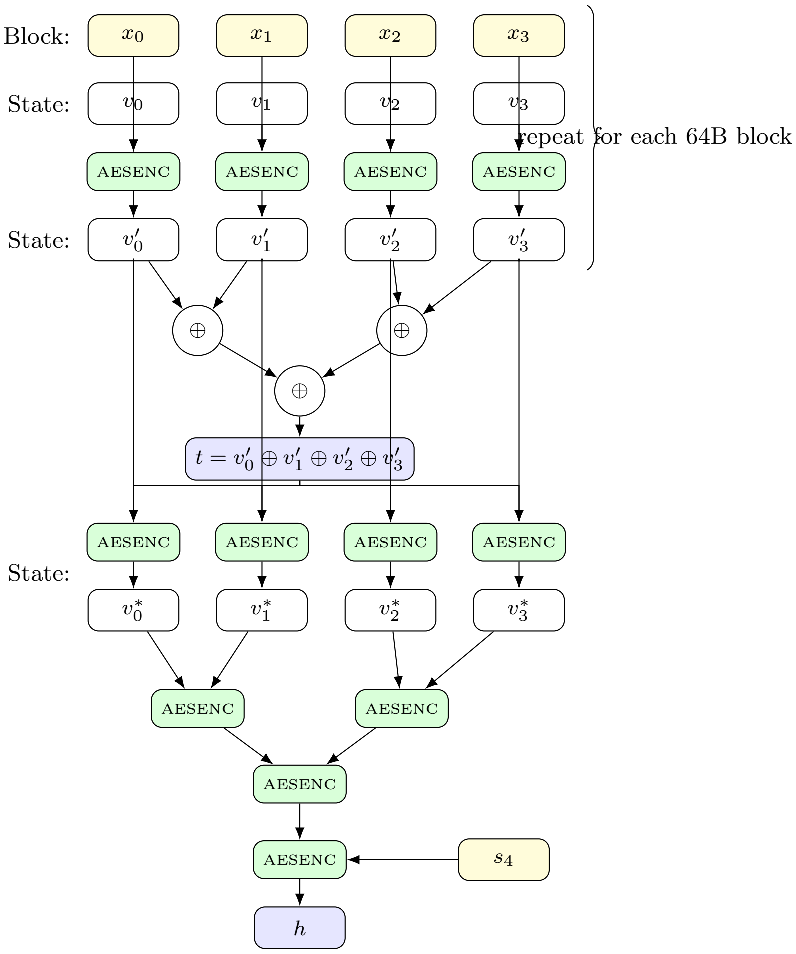
Each maps to a single x86 instruction (AESENC / AESDEC) with ∼1 cycle throughput and ∼4 cycle latency on modern CPUs.

11.1 AquaHash

Source: `hashes/aquahash.cpp` [Rog19].

SMHasher3 test results. AquaHash fails 55 of 250 tests. The main weaknesses are in Sparse tests (18 configurations, including long-key variants up to 2048 bytes), Permutation (single-bit keys), and several Text patterns. Some seed-related failures (SeedSparse, SeedBlockLen, Seed) also appear.

AquaHash by J. Andrew Rogers uses `aesenc` for mixing with 4 parallel 128-bit lanes (512-bit state). We denote the fixed 128-bit constants used by the reference implementation as $s_0, \ldots, s_4$; s_4 is used in the final AES round.



Short keys ($n < 64$): Uses 2 parallel lanes with AESENC, combining via AESENC(v_0, v_1) before a 3-round finalization.

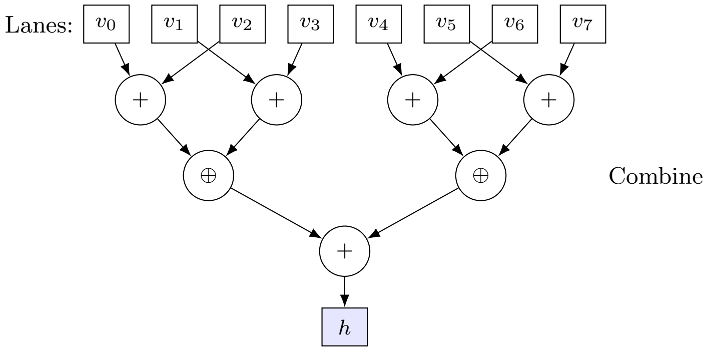
Long keys ($n \geq 64$).

```
(v_0, v_1, v_2, v_3) ← (seed ⊕ s_0, seed ⊕ s_1, seed ⊕ s_2, seed ⊕ s_3)
for each block (x_0, x_1, x_2, x_3) ∈ u128^4 do                          ▷ 64 bytes
    for i ∈ {0, ..., 3} do
        v_i ← AESENC(v_i, x_i)
    end for
end for
Process remaining 32, 16, 8, 4, 2, 1 bytes
t ← v_0 ⊕ v_1 ⊕ v_2 ⊕ v_3
for i ∈ {0, ..., 3} do
    v_i ← AESENC(v_i, t)
end for
v ← AESENC(AESENC(v_0, v_1), AESENC(v_2, v_3))
return AESENC(v, s_4)
```

11.2 MeowHash

Source: `hashes/meowhash.cpp`, version 0.5/calico [Mur18].

MeowHash by Casey Muratori (Molly Rocket) uses 8 parallel 128-bit lanes (1024-bit state) with `aesdec` for mixing. Processes 256-byte blocks matching cache line boundaries. High throughput by saturating AES execution units; for inputs > 256KB, explicit prefetch improves performance.



Default seed: 128 bytes from digits of π. All additions below are 64-bit lane-wise (`paddq`).

```
function MEOW_MIX($r_1, r_2, r_3, r_4, r_5, x_a, x_b$)          ▷ 32 bytes into 5 lanes
    $r_1 \leftarrow$ AESDEC($r_1, r_2$)          ▷ Source uses 4 overlapping 128-bit reads;
    $r_3 \leftarrow r_3 + x_a$;    $r_2 \leftarrow r_2 \oplus x_b$          ▷ simplified to 2 blocks here
    $r_2 \leftarrow$ AESDEC($r_2, r_4$)
    $r_5 \leftarrow r_5 + x_a$;    $r_4 \leftarrow r_4 \oplus x_b$
end function
function MEOW_SHUFFLE($r_1, r_2, r_3, r_4, r_5, r_6$)          ▷ Cross-lane mixing (no input)
    $r_1 \leftarrow$ AESDEC($r_1, r_4$)
    $r_2 \leftarrow r_2 + r_5$;    $r_4 \leftarrow r_4 \oplus r_6$
    $r_4 \leftarrow$ AESDEC($r_4, r_2$)
    $r_5 \leftarrow r_5 + r_6$
end function

Initialize 8 lanes ($v_0, \ldots, v_7$) from 128-byte seed
for each block $(x_i)_{i=0}^{15} \in$ u128$^{16}$ do          ▷ Main loop (256 bytes)
    for $i \in \{0, \ldots, 7\}$ do          ▷ 32 bytes per iteration; touches 5 lanes
        MEOW_MIX($v_i, v_{(i+4) \bmod 8}, v_{(i+6) \bmod 8}, v_{(i+1) \bmod 8}, v_{(i+2) \bmod 8}, x_{2i}, x_{2i+1}$)
    end for
end for
Load residual < 32 bytes with careful alignment handling
Mix residual and length into lanes
Process remaining 32-byte blocks (0–7)
MixDown: Apply 12 MEOW_SHUFFLE operations
Combine: $v_0 \leftarrow ((v_0 + v_2) \oplus (v_1 + v_3)) + ((v_4 + v_6) \oplus (v_5 + v_7))$
return $v_0$
```

11.3 aHash

Source: `hashes/rust-ahash.cpp` [Kai18].

By Tom Kaitchuck. Default hash in Rust's `HashMap`. Two variants: AES-based (2 parallel paths using `aesdec` + shuffle-add) and fallback (portable MUM-based, similar to wyhash). Uses variable-amount final rotation for additional mixing.

AES Variant. Uses `aesdec` with a special shuffle permutation (selected by automated search): Shuffle is a fixed byte permutation of a 128-bit value (as in the implementation).

```
$(v_0, v_1) \leftarrow$ state from random seed (derived from $\pi$)
$t \leftarrow v_0 \oplus v_1$          ▷ Save initial combined state
```

```
for each block $x \in$ u128 do
    $v_0 \leftarrow \textsc{aesdec}(v_0, x)$
    $v_1 \leftarrow \textsc{Shuffle}(v_1) + x$
end for
$v \leftarrow \textsc{aesenc}(v_1, v_0)$
return $\textsc{aesdec}(\textsc{aesdec}(v, t), v)$
```

Fallback Variant. MUM-based, using Knuth's LCG multiplier $s_0 = 6364136223846793005$:

```
Initialize $v, s_1, s_2, s_3$ from random state
$v \leftarrow \textsc{Mum}(v \oplus n,\ s_0)$
$v \leftarrow (v + n) \cdot s_0$
for each block $(x_0, x_1) \in$ u64$^2$ do                      ▷ 16 bytes
    $t \leftarrow \textsc{Mum}(x_0 \oplus s_2,\ x_1 \oplus s_3)$
    $v \leftarrow (v + s_1) \oplus t$
    $v \leftarrow \textsc{rot}_{23}(v)$
end for
return $\textsc{rot}_{v \bmod 64}(\textsc{Mum}(v,\ s_1))$
```

12 SIMD-Scalable Hashes

12.1 HalftimeHash

Source: `hashes/halftimehash.cpp` [App21].

SMHasher3 test results. HalftimeHash-128 fails 48 of 250 tests. The dominant failure category is Sparse (27 configurations including long-key variants), followed by Permutation (single-bit and low-bit keys) and SeedZeroes. As a keyed hash requiring ∼50 KB of random state, the SeedZeroes failures reflect SMHasher3's seed model rather than a design flaw.

HalftimeHash by Jim Apple is a SIMD-scalable hash that efficiently uses any vector width (64-bit scalar to 512-bit AVX-512). Based on erasure coding theory for provable mixing properties. Uses 32-bit multiplies (`_mm_mul_epu32`), a tree structure for long inputs, and requires ∼50KB of random state.

The core mixing uses $\textsc{Times}(a, b) = (a \wedge \texttt{0xFFFFFFFF}) \times (b \wedge \texttt{0xFFFFFFFF})$ within 64-bit lanes. The Mix function computes $v + \textsc{Times}(t, t \gg 32)$ where $t = s +_{32} x$ (independent 32-bit lane addition, no carry between halves).

Erasure coding (like RAID) achieves strong mixing: input blocks are encoded with redundancy (e.g., Encode2: 6 → 7 blocks), then reduced using weighted sums. For long inputs, a tree structure with fanout 8 combines block groups via depth-first traversal. Style variants select block size: Style64 (scalar), Style128 (SSE2/NEON), Style256 (AVX2), Style512 (AVX-512). Output is finalized using byte-level tabulation: $\mathit{result} = \textsc{TabulateBytes}(n) \oplus \bigoplus_j \textsc{TabulateBytes}(h_j)$, where (h_j) are the final 64-bit output words for the chosen variant.

```
function $\textsc{Mix}(v, x, s)$                                ▷ Core mixing per block
    $t \leftarrow s +_{32} x$                                   ▷ 32-bit lane addition
    return $v + \textsc{Times}(t, t \gg 32)$       ▷ Times: 32×32 low-half multiply
end function
function $\textsc{Combine}(u_0, \ldots, u_{d'-1})$     ▷ Fixed linear reducer (variant-specific)
    Compute $(y_0, \ldots, y_{L-1})$ as weighted sums (dot products) of the $u_i$
    return $(y_0, \ldots, y_{L-1})$
end function
function $\textsc{BaseLayer}$(input block group, $s$)
    Encode $d$ input blocks into $d'$ blocks via erasure code (XOR-based)
    Hash each encoded block into one value $u_i$ using repeated Mix (details elided)
    return Combine($u_0, \ldots, u_{d'-1}$)
end function
function UpperLayer($(u_0^{(0)}, \ldots, u_{L-1}^{(0)}), \ldots, (u_0^{(7)}, \ldots, u_{L-1}^{(7)}), s$)
    for $i \in \{0, \ldots, L-1\}$ do
        $y_i \leftarrow u_i^{(0)}$
        for $j \in \{1, \ldots, 7\}$ do
            $y_i \leftarrow$ Mix($y_i, u_i^{(j)}, s_{i,j}$)
        end for
    end for
    return $(y_0, \ldots, y_{L-1})$
end function
function TabulateBytes($x, T$)                                  ▷ $T$ is an 8×256 table
    $r \leftarrow 0$
    for $i \in \{0, \ldots, 7\}$ do
        $r \leftarrow r \oplus T[i][byte_i(x)]$
    end for
    return $r$
end function

Let $L \in \{2, 3, 4, 5\}$ be the number of output words (variant-specific)
$stack[0 \ldots 8] \leftarrow$ empty            ▷ Tree with fanout 8; each entry is a tuple in $\mathsf{uw}^L$
for each block group in input do
    $(y_0, \ldots, y_{L-1}) \leftarrow$ BaseLayer(block group, $s$)
    Push $(y_0, \ldots, y_{L-1})$ onto $stack[0]$
    $\ell \leftarrow 0$
    while $|stack[\ell]| = 8$ do                                   ▷ Collapse full levels
        $(y_0, \ldots, y_{L-1}) \leftarrow$ UpperLayer($stack[\ell], s$)
        Clear $stack[\ell]$; push $(y_0, \ldots, y_{L-1})$ onto $stack[\ell + 1]$
        $\ell \leftarrow \ell + 1$
    end while
end for
Initialize accumulators $a_0, \ldots, a_{L-1} \leftarrow 0$
Let $T^{(0)}, \ldots, T^{(L)}$ be 8×256 tables derived from entropy ($T^{(0)}$ for length, $T^{(i+1)}$ for $h_i$)
Mix remaining stack tuples and remaining input blocks into $(a_0, \ldots, a_{L-1})$ using Mix (tail
  summarized)
Reduce each accumulator block by summing its 64-bit lanes: $h_i \leftarrow \sum$ lanes($a_i$)
$r \leftarrow$ TabulateBytes($n, T^{(0)}$)
for $i \in \{0, \ldots, L-1\}$ do
    $r \leftarrow r \oplus$ TabulateBytes($h_i, T^{(i+1)}$)
end for
return $r$
```

12.2 khashv

Source: `hashes/khashv.cpp` [Can23].

By Keith-Cancel. A vectorizable hash using S-box byte substitution and SIMD-friendly shuffles that map to `pshufb` (SSSE3+). Uses no multiplication—only add, XOR, rotate, and table lookup. Endian-independent with compact 128-bit state (four 32-bit words, initialized from SHA-256 hashes of bytes 1–4).

Non-linear mixing uses a nibble-decomposed S-box (similar to AES SubBytes), and SHUFFLE applies fixed byte permutation $(7, 14, 9, 0, 12, 15, 13, 8, 5, 11, 6, 3, 4, 2, 10, 1)$.

Primitives. Two 16-byte lookup tables S_1, S_2 define the S-box:

```
function SBOX($x$)                                    ▷ 128-bit → 128-bit, applied per byte
    for each byte $b$ in $x$ do
        $b \leftarrow S_1[b \wedge \texttt{0xF}] \oplus S_2[b \gg 4]$          ▷ Low/high nibble lookup + XOR
    end for
    return $x$
end function
function ROTR5($v$)                                   ▷ Rotate 16-byte state right by 5 bytes
    return bytes of $v$ permuted: $b_i \leftarrow b_{(i+5) \bmod 16}$
end function
function SHUFFLE($v$)                                 ▷ Fixed byte permutation (pshufb-style)
    return bytes of $v$ permuted: $b_i \leftarrow b_{\pi(i)}$ where $\pi = (7, 14, 9, 0, 12, 15, 13, 8, 5, 11, 6, 3, 4, 2, 10, 1)$
end function
function MIX_WORDS($v$)                               ▷ Finalization: operates on 32-bit words
    $v \leftarrow v \oplus (v \gg_{32} 3)$                      ▷ Shift each 32-bit word right by 3, XOR back
    for $r \in$ (5, 7, 11, 17) do
        $t \leftarrow$ ROTR5$(v) + v$
        $v \leftarrow v \oplus$ ROT$_r(t)$                      ▷ Rotate each 32-bit word right by $r$
    end for
    return $v$
end function
```

khashv Algorithm.

```
$v \leftarrow seed$                                   ▷ 128-bit state: four 32-bit words $(v_0, v_1, v_2, v_3)$
$v_0 \leftarrow v_0 \oplus (n \bmod 2^{32})$;   $v_1 \leftarrow v_1 \oplus (n \gg 32)$
for each block $x \in$ u128 do
    $t \leftarrow$ SBOX$(x)$
    $v \leftarrow v \oplus (t \cdot 8193)$                          ▷ $8193 = 2^{13} + 1$
    $v \leftarrow$ ROTR5$(v) + t$
    $v \leftarrow v +$ SHUFFLE$(v)$
end for
$v \leftarrow$ MIX_WORDS$(v)$
return $v_3$ (32-bit) or $(v_0, v_1)$ (64-bit)
```

A Folded Multiply $(\mathrm{lo} \oplus \mathrm{hi})$

Let $m = 2^n$. For each $x \in \{0, 1, \ldots, m-1\}$, define

$$\mathrm{op}(x, y) \;=\; \left\lfloor \frac{xy}{m} \right\rfloor \;\oplus\; (xy \bmod m),$$

and fiber sizes

$$N_x(a) \;=\; \#\{y \in [0, m-1] : \mathrm{op}(x, y) = a\}.$$

The collision count is

$$C(x) \;=\; \sum_{a=0}^{m-1} N_x(a)^2, \qquad \mathbb{E}[C] \;=\; \frac{1}{m} \sum_{x=0}^{m-1} C(x).$$

To bound $\mathbb{E}[C]$ we separate the two spike outputs $a = 0$ and $a = m - 1$ (the only ones amenable to exact analysis) and treat the rest as a residual term:

$$C(x) = N_x(0)^2 + N_x(m-1)^2 + C^*(x), \qquad C^*(x) = \sum_{a \notin \{0,m-1\}} N_x(a)^2.$$

Below are sharp, self-contained bounds on the **spike contribution**, plus a rigorous explanation of why "repeating multipliers" (like 0101...) cannot accumulate enough mass to change $\mathbb{E}[C]$ at the $m \log\log n$ scale.

A.1 Exact formulas for the two spike fibers

Lemma 1 (Exact spike counts)**.** *For $1 \le x \le m-1$,*

$$N_x(0) = \gcd(x,\, m+1), \qquad N_x(m-1) = \gcd(x,\, m-1).$$

Proof. $\operatorname{op}(x,y) = 0$ means "high = low," i.e. $q = r$ in $xy = mq + r$. Then

$$xy = mq + q = (m+1)q,$$

so $(m+1) \mid xy$. Let $d = \gcd(x, m+1)$. Then $(m+1) \mid xy$ is equivalent to $\frac{m+1}{d} \mid y$, because $\gcd(x/d, (m+1)/d) = 1$. Since $x < m$, we have $d < m+1$, hence $(m+1)/d \ge 2$, and the multiples of $(m+1)/d$ in $[0, m-1]$ are exactly

$$0,\ \frac{m+1}{d},\ 2\frac{m+1}{d},\ \ldots,\ (d-1)\frac{m+1}{d},$$

whose last term is $(m+1) - (m+1)/d \le m-1$. So there are exactly d solutions y, i.e. $N_x(0) = d$.

$\operatorname{op}(x,y) = m-1$ means $q \oplus r = m-1$, which forces $r = (m-1) - q$. Then

$$xy = mq + r = mq + (m-1-q) = (m-1)(q+1),$$

so $(m-1) \mid xy$. Let $d = \gcd(x, m-1)$. As above, $(m-1) \mid xy$ is equivalent to $\frac{m-1}{d} \mid y$, and the multiples in $[0, m-1]$ are

$$0,\ \frac{m-1}{d},\ 2\frac{m-1}{d},\ \ldots,\ (d-1)\frac{m-1}{d} = m - 1 - \frac{m-1}{d} < m,$$

giving exactly d solutions. So $N_x(m-1) = d$. □

For $x = 0$, $\operatorname{op}(0,y) = 0$ for all y, so $N_0(0) = m$ and $N_0(m-1) = 0$. This single x contributes only $O(m)$ to $\mathbb{E}[C]$, so it never affects asymptotics.

A.2 Turning the spike identities into $\Theta(m \log\log n)$

We need $\mathbb{E}[N_x(0)^2]$ and $\mathbb{E}[N_x(m-1)^2]$. These are essentially averages of $\gcd(\cdot, N)^2$.

Lemma 2 (Sum of squared gcds)**.** *For any $N \ge 1$,*

$$\sum_{k=1}^{N} \gcd(k,N)^2 \;=\; N^2 \sum_{d \mid N} \frac{\varphi(d)}{d^2},$$

where φ is Euler's totient.

Proof. Group k by $g = \gcd(k,N)$. Writing $k = gk'$ with $\gcd(k', N/g) = 1$, there are $\varphi(N/g)$ such k. Hence

$$\sum_{k=1}^{N} \gcd(k,N)^2 = \sum_{g \mid N} g^2\, \varphi(N/g) = \sum_{d \mid N} \Big(\frac{N}{d}\Big)^2 \varphi(d) = N^2 \sum_{d \mid N} \frac{\varphi(d)}{d^2}. \qquad \square$$

Define

$$F(N) := \sum_{d|N} \frac{\varphi(d)}{d^2}.$$

Lemma 3 ($F(N)$ versus $\sigma(N)/N$)**.** *Let $\sigma(N) = \sum_{d|N} d$. Then for all N,*

$$\frac{6}{\pi^2} \cdot \frac{\sigma(N)}{N} \;\le\; F(N) \;\le\; \frac{\sigma(N)}{N}.$$

Proof. Write $N = \prod p^{e_p}$. Both F and $\sigma(\cdot)/\cdot$ factor over prime powers:

$$\frac{\sigma(N)}{N} = \prod_{p^e \| N} \Big(1 + \frac{1}{p} + \cdots + \frac{1}{p^e}\Big),$$

and one can compute

$$F(N) = \prod_{p^e \| N} \Big(1 + \frac{1}{p} - \frac{1}{p^{e+1}}\Big).$$

For each prime power,

$$1 + \frac{1}{p} - \frac{1}{p^{e+1}} \;\le\; 1 + \frac{1}{p} + \cdots + \frac{1}{p^e},$$

and also

$$\begin{aligned} 1 + \frac{1}{p} - \frac{1}{p^{e+1}} &= \Big(1 + \frac{1}{p} + \cdots + \frac{1}{p^e}\Big) - \Big(\frac{1}{p^2} + \cdots + \frac{1}{p^{e+1}}\Big) \\ &\ge \Big(1 - \frac{1}{p^2}\Big)\Big(1 + \frac{1}{p} + \cdots + \frac{1}{p^e}\Big). \end{aligned}$$

Multiplying over primes gives

$$F(N) \;\ge\; \Big(\prod_{p|N} \big(1 - \tfrac{1}{p^2}\big)\Big)\frac{\sigma(N)}{N} \;\ge\; \Big(\prod_{p} \big(1 - \tfrac{1}{p^2}\big)\Big)\frac{\sigma(N)}{N} \;=\; \frac{6}{\pi^2}\,\frac{\sigma(N)}{N}.$$

□

Consequence for spike second moments.

- For $a = m-1$: here $N = m-1$ and x runs through $1, 2, \ldots, m-1$ exactly, so

$$\frac{1}{m}\sum_{x=1}^{m-1} N_x(m-1)^2 = \frac{1}{m}\sum_{x=1}^{m-1} \gcd(x, m-1)^2 = \frac{(m-1)^2}{m}\, F(m-1) = \Theta\big(\sigma(m-1)\big).$$

- For $a = 0$: $N = m+1$ but x only runs up to $m-1 = N-2$. Dropping the final two terms only changes the sum by $O(N^2)$, which becomes $O(m)$ after dividing by m. So

$$\mathbb{E}[N_x(0)^2] = \Theta\big(\sigma(m+1)\big).$$

Thus the **spike part of** $\mathbb{E}[C]$ is

$$\mathbb{E}\big[N_x(0)^2 + N_x(m-1)^2\big] = \Theta\big(\sigma(2^n+1) + \sigma(2^n-1)\big).$$

Upper bound $O(m \log\log n)$ for the spike part. A theorem of Erdős [Erd71] gives

$$\sigma(2^n - 1) \;\le\; c\,(2^n - 1)\log\log n \quad \text{for large } n,$$

for an absolute constant c.

To also bound $\sigma(2^n + 1)$, observe that

$$2^{2n} - 1 = (2^n - 1)(2^n + 1), \qquad \gcd(2^n - 1,\, 2^n + 1) = 1,$$

so $\sigma(2^{2n} - 1) = \sigma(2^n - 1)\,\sigma(2^n + 1)$. Applying the same bound at exponent $2n$ gives

$$\sigma(2^{2n} - 1) \le c'(2^{2n} - 1)\log\log(2n),$$

and since $\sigma(2^n - 1) \ge 2^n - 1$, we get

$$\sigma(2^n + 1) \le \frac{\sigma(2^{2n} - 1)}{\sigma(2^n - 1)} = O(2^n \log\log n).$$

Therefore, unconditionally,

$$\mathbb{E}\big[N_x(0)^2 + N_x(m-1)^2\big] = O(m \log\log n).$$

Lower bound $\Omega(m \log\log n)$ for infinitely many n. Take $n = t!$ and $m = 2^n$. For every odd prime $p \le t$, we have $p - 1 \mid t!$, so by Fermat's little theorem $2^{t!} \equiv 1 \pmod p$, hence $p \mid (2^{t!} - 1)$.

Thus $2^{t!} - 1$ is divisible by $\prod_{p \le t} p$, so

$$\frac{\sigma(2^{t!} - 1)}{2^{t!} - 1} \;\ge\; \prod_{p \le t}\Big(1 + \frac{1}{p}\Big).$$

Now

$$\prod_{p \le t}\Big(1 + \frac{1}{p}\Big) = \prod_{p \le t}\Big(1 - \frac{1}{p^2}\Big) \cdot \prod_{p \le t}\Big(1 - \frac{1}{p}\Big)^{-1}.$$

Mertens' theorem says $\prod_{p \le t}(1 - \frac{1}{p}) \sim e^{-\gamma}/\log t$, i.e. the inverse grows like $e^{\gamma}\log t$. Also $\prod_{p \le t}(1 - \frac{1}{p^2})$ stays bounded below by a positive constant (it converges to $6/\pi^2$). Hence

$$\prod_{p \le t}\Big(1 + \frac{1}{p}\Big) \;\ge\; c\,\log t \quad \text{for large } t,$$

so

$$\sigma(2^{t!} - 1) \;\ge\; c\,(2^{t!} - 1)\log t.$$

Since $n = t!$, we have $\log\log n = \log\log(t!) = \Theta(\log t)$. Therefore

$$\sigma(2^n - 1) \;\ge\; c'\,2^n \log\log n \quad \text{for infinitely many } n,$$

and since $\mathbb{E}[C] \ge \mathbb{E}[N_x(m-1)^2] = \Theta(\sigma(2^n - 1))$, we get

$$\mathbb{E}[C] = \Omega(m \log\log n) \quad \text{infinitely often.}$$

Summary so far (rigorous).

$$\mathbb{E}[C] \;=\; \mathbb{E}[C^*] \;+\; O(m \log\log n),$$

and also $\mathbb{E}[C] \ge \Omega(m \log\log n)$ along the subsequence $n = t!$. So the entire problem of a tight upper bound is now concentrated in $\mathbb{E}[C^*]$.

A.3 Non-spike outputs: do repeating multipliers matter?

Outside $\{0, m-1\}$, some x's have noticeably larger fibers $N_x(a)$ (e.g. $x = 0101\ldots$, which is $x = (m-1)/3$ when n is even). The key point for the expectation is: **those multipliers form a tiny set**, so they cannot accumulate enough mass to change $\mathbb{E}[C]$ at the $m \log\log n$ scale.

A. Any fixed-period family contributes only $O(m)$. Fix a period p. There are at most 2^p n-bit strings with period p (choose the length-p template; the rest repeats). For each x,

$$C(x) = \sum_a N_x(a)^2 \ \le\ \Big(\sum_a N_x(a)\Big)^2 = m^2.$$

So the total contribution of all period-p multipliers to the expectation is

$$\frac{1}{m} \sum_{\substack{x \text{ has} \\ \text{period } p}} C(x) \ \le\ \frac{1}{m} \cdot 2^p \cdot m^2 \ =\ 2^p\, m.$$

For constant p (like $p = 2$ for $0101\ldots$), this is $O(m)$, which is negligible compared to the spike term $m \log\log n$ once n is large.

B. The whole divisor family $x \mid (m-1)$ stays at $O(m \log\log n)$. This is a more relevant "big structured class," because periodic patterns like $0101\ldots$ are typically divisors of $m-1$ (or closely related).

For any x, a trivial but useful bound is

$$C(x) \le mx,$$

since for $1 \le x \le m-1$ we have $q = \lfloor xy/m \rfloor \in \{0, 1, \ldots, x-1\}$, and for a fixed output $a = q \oplus r$ each choice of q forces $r = a \oplus q$ and hence at most one y solving $xy = mq + r$. Thus $N_x(a) \le x$, so $C(x) = \sum_a N_x(a)^2 \le (\max_a N_x(a)) \sum_a N_x(a) \le x \cdot m$. so for the divisor family $x \mid (m-1)$,

$$\frac{1}{m} \sum_{x \mid (m-1)} C(x) \ \le\ \frac{1}{m} \sum_{x \mid (m-1)} mx \ =\ \sigma(m-1) \ =\ O(m \log\log n),$$

using Erdős' bound on $\sigma(2^n - 1)$ [Erd71].

So **even if every single divisor of** $m-1$ had maximally bad non-spike behavior (it does not), its *total* effect on $\mathbb{E}[C]$ cannot exceed the same $O(m \log\log n)$ scale that the spike already forces.

A.4 What remains for bounding $\mathbb{E}[C]$

The sharp part of the expectation is now cleanly isolated:

- The **only provably growing term** is the spike contribution, and it is

  $$\Theta\big(\sigma(2^n - 1) + \sigma(2^n + 1)\big) = O(m \log\log n),$$

  with a matching $\Omega(m \log\log n)$ lower bound infinitely often.

- Repeating multipliers and similar structured classes are provably too sparse to change the leading scale of $\mathbb{E}[C]$.

To obtain a full $\mathbb{E}[C] = O(m \log\log n)$ upper bound, the missing ingredient is:

Show $\mathbb{E}[C^*] = O(m)$ (or at least $O(m \log\log n)$).

Empirically $\mathbb{E}[C^*]/m$ appears bounded (it hovers around a small constant for all small n), but proving it seems to require a genuine "mixing" statement: roughly, that for typical x, the non-spike outputs behave like a near-random mapping, so $\sum_{a \ne 0, m-1} N_x(a)^2$ stays $\Theta(m)$.

A.5 Empirics for folded multiply

The operation $\mathrm{op}(x, y) = \lfloor xy/m \rfloor \oplus (xy \bmod m)$ can also be viewed, for fixed x, as a map $f_x : \{0, \dots, m-1\} \to \{0, \dots, m-1\}$. We write $m(x) = |\mathrm{Im}(f_x)|$ for its image size.

Image size versus collisions. If a function f on an m-element domain has image size M, then (writing $C^{\mathrm{pairs}} = \#\{(a, b) : 0 \le a < b < m,\ f(a) = f(b)\}$) one has

$$C^{\mathrm{pairs}} \ge \frac{1}{2}\left(\frac{m^2}{M} - m\right).$$

In terms of the ordered-pair statistic used above, $C(x) = \sum_a N_x(a)^2 = m + 2C^{\mathrm{pairs}}$.

Variants compared. We compare four closely related fold constructions (all returning a `un` value):

$$\texttt{nmul\_xor} : \ \mathrm{lo}(xa) \oplus \mathrm{hi}(xa), \quad \texttt{nmul\_add} : \ (\mathrm{lo}(xa) + \mathrm{hi}(xa)) \bmod 2^n,$$

where xa is ordinary integer multiplication, and

$$\texttt{cmul\_xor} : \ \mathrm{lo}(x \otimes a) \oplus \mathrm{hi}(x \otimes a), \quad \texttt{cmul\_add} : \ (\mathrm{lo}(x \otimes a) + \mathrm{hi}(x \otimes a)) \bmod 2^n,$$

where $x \otimes a$ is carryless (polynomial) multiplication over $\mathbb{F}_2$.

Field baselines. In a true field, multiplication by $x \neq 0$ is a permutation, hence has no collisions. Therefore, for any field of size N,

$$\mathbb{E}_x[C_x] = \frac{1}{N}\binom{N}{2} = \frac{N-1}{2}, \qquad \mathbb{E}_x[C_x]/N \approx \frac{1}{2}.$$

In the plots below we include two baselines: `gf2n_field` (a field of size 2^n, for any irreducible polynomial of degree n) and `prime_field` (multiplication modulo the largest prime $p \le 2^n$).

Carryless `cmul_xor` admits an exact collision formula. For carryless multiplication, the map $a \mapsto \mathrm{lo}(x \otimes a) \oplus \mathrm{hi}(x \otimes a)$ is multiplication by x in the ring $\mathbb{F}_2[t]/(t^n + 1)$. Writing $g = \gcd(x, t^n + 1)$ and $d = \deg(g)$, one has $|\ker f_x| = 2^d$, $m(x) = 2^{n-d}$, and

$$C_x = 2^{n-1}(2^d - 1).$$

This explains the strong n-dependence seen for `cmul_xor`.

Empirical results. Figure 2 shows $\mathbb{E}[C_x]$ and its normalized version $\mathbb{E}[C_x]/2^n$ for all variants. For `nmul_xor` we observe an approximately constant factor $\mathbb{E}[C_x]/2^n \approx 1.5$–$2.2$ for $n \le 15$ (exact enumeration); for $n > 15$ we estimate using Monte Carlo sampling over multipliers x.

Achievable (δ, ε) frontier. Define the image-size threshold $M(\delta) = 2^{(1-\delta)n}$ and the "bad multiplier" event $m(x) \le M(\delta)$. Let $p_n(\delta) = \Pr_x[m(x) \le M(\delta)]$. The plot in Figure 3 shows $p_n(\delta)$ using the more stable x-axis $\Delta = \delta n$ (bits below n), which aligns curves across n.

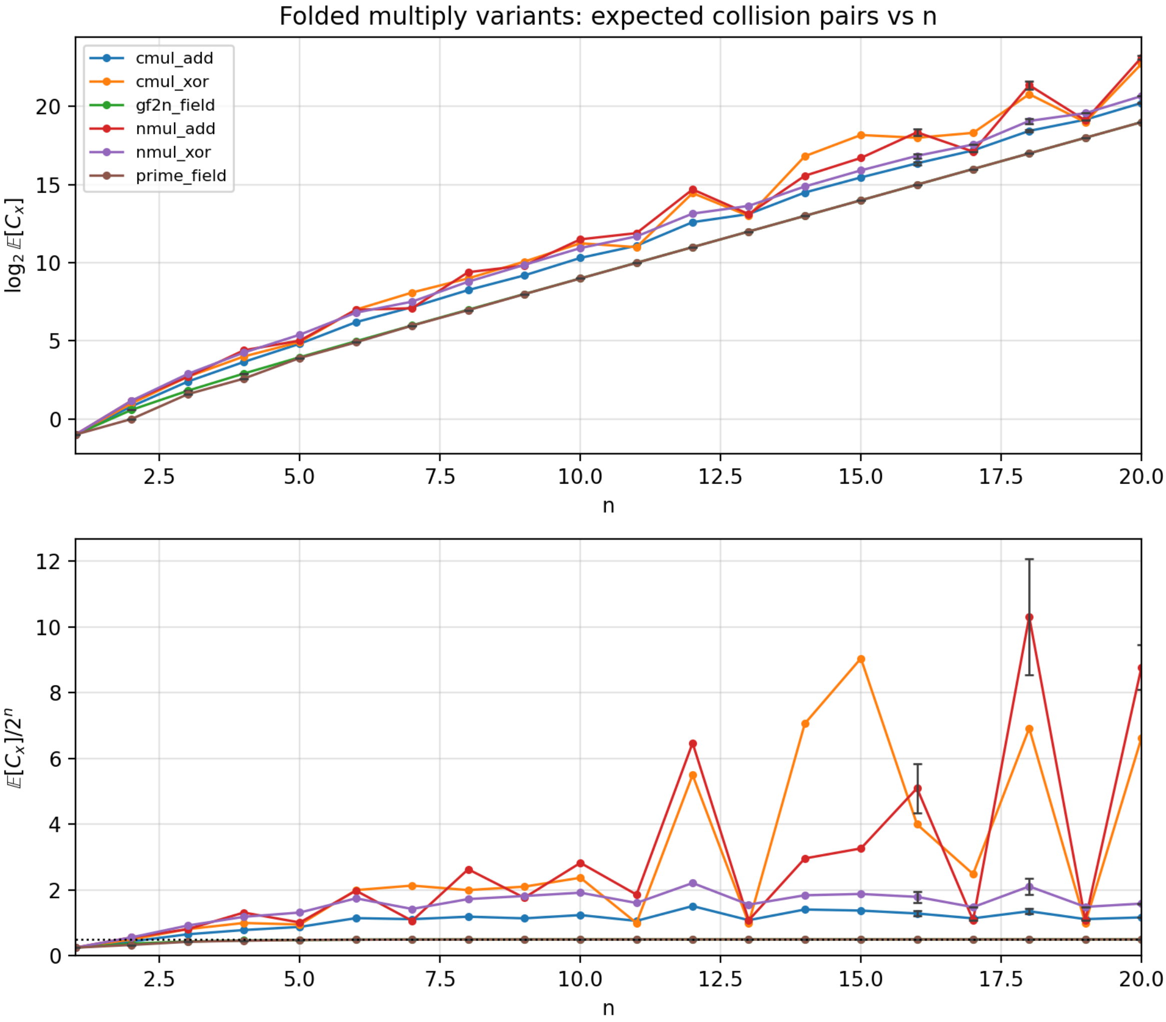


Figure 2: Expected collision pairs $\mathbb{E}[C_x]$ (top) and normalized constant $\mathbb{E}[C_x]/2^n$ (bottom) for folded multiply variants.

References

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *Progress in Cryptology – INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012.

[AKT20] Thomas Dybdahl Ahle, Jakob Tejs Baek Knudsen, and Mikkel Thorup. The power of hashing with mersenne primes. *arXiv preprint arXiv:2008.08654*, 2020.

[App21] Jim Apple. HalftimeHash: Modern hashing without 64-bit multipliers or finite fields. In *Proceedings of the 17th International Symposium on Algorithms and Data Structures (WADS 2021)*, Lecture Notes in Computer Science. Springer, 2021.

[Can23] Keith Cancel. k-hashv – a vectorized hash function. GitHub repository, 2023.

[Col12] Yann Collet. xxHash – extremely fast non-cryptographic hash algorithm. GitHub repository, 2012.

[De 24] Nicolas De Carli. rapidhash – very fast, high quality, platform-independent hashing. GitHub repository, 2024.

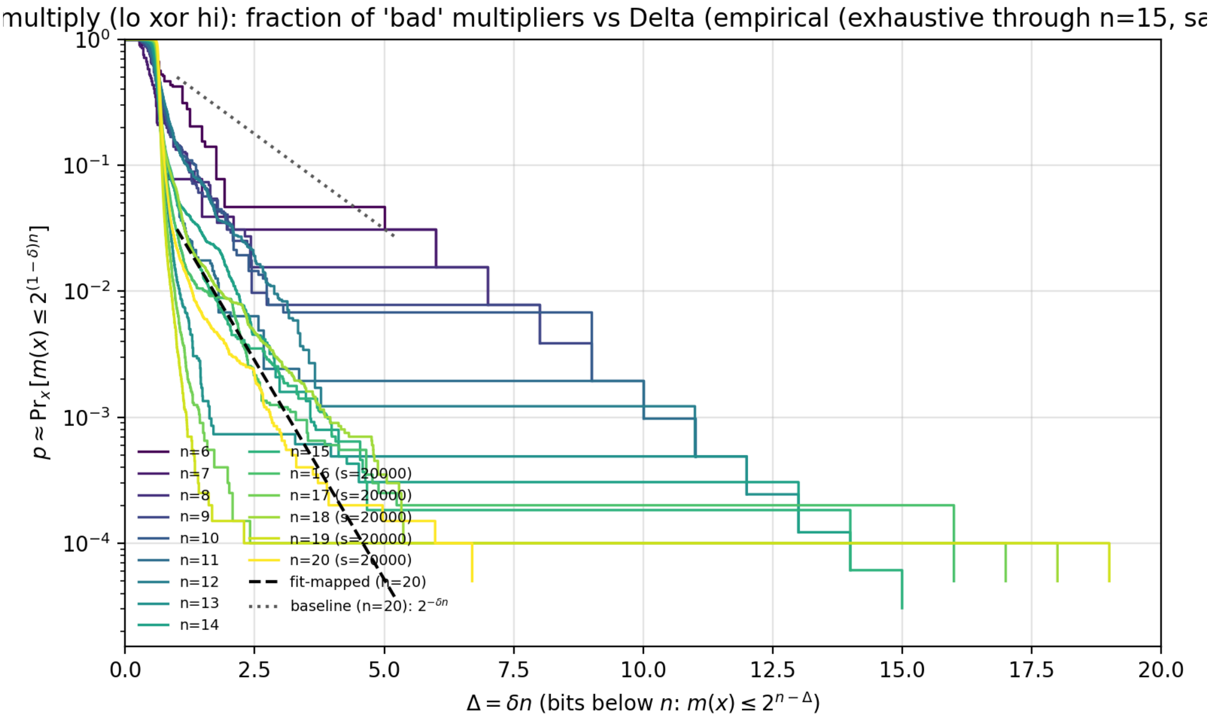


Figure 3: Fraction of multipliers with small image: $p_n(\delta) = \mathrm{Pr}_x[m(x) \leq 2^{n-\Delta}]$, plotted against $\Delta = \delta n$ on a log-y scale (exhaustive for $n \leq 15$, sampled after).

[Erd71] Paul Erdős. On the sum $\sum_{d|2^n-1} d^{-1}$. *Israel Journal of Mathematics*, 9:43–48, 1971.

[Jen12] Bob Jenkins. SpookyHash: A 128-bit noncryptographic hash. Author's website, 2012.

[K-A24] K-Aethiax. MuseAir – a portable hashing algorithm. GitHub repository, 2024.

[Kai18] Tom Kaitchuck. aHash – a non-cryptographic hashing algorithm using AES. GitHub repository, 2018.

[Khu20a] Paul Khuong. Nearly double the PH bits with one more CLMUL. Blog post, 2020.

[Khu20b] Paul Khuong. UMASH: Fast enough, almost universal fingerprinting. Blog post, 2020.

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

[Mur18] Casey Muratori. Meow hash. Project page, 2018.

[NRK24] NRK. ChibiHash: Small, fast 64-bit hash function. Blog post, 2024.

[Pet23] Orson Peters. PolymurHash – a universal hash function. GitHub repository, 2023.

[Pik14] Geoff Pike. Introducing FarmHash. Google Open Source Blog, 2014.

[Rog19] J. Andrew Rogers. AquaHash: Fast hashing with AES intrinsics. Blog post, 2019.

[Str23] Cris Stringfellow. Rain hashes: Rainbow, Rainstorm and more. GitHub repository, 2023.

[Van20] Aleksey Vaneev. PRVHASH – pseudo-random-value hash. GitHub repository, 2020.

[Van21] Aleksey Vaneev. komihash – very fast, high-quality hash function. GitHub repository, 2021.

[Van25] Aleksey Vaneev. a5hash – ultra fast, high-quality hash functions. GitHub repository, 2025.

[Woj22] Frank J. T. Wojcik. SMHasher3 – hash function quality and speed test suite. GitLab repository, 2022.

[Yi19] Wang Yi. wyhash – fast portable non-cryptographic hashing. GitHub repository, 2019.