Efficient GHASH and POLYVAL Implementation Using Polynomial Multiplication: Optimized 64-bit Decomposition with Bit-Reversal Elimination

Mamone Tarsha Kurdi Niels Möller

November 2025

Abstract

We present an optimized implementation of the GHASH and POLYVAL authentication algorithms used in AES-GCM and AES-GCM-SIV that eliminates the computational overhead of bit-reversal operations. Our approach computes these universal hash functions directly in bit-reversed representation, matching the native format used by carry-less multiplication instructions available on modern processors. The algorithm exploits 64-bit polynomial primitives and parallel execution on superscalar architectures. We achieve performance of 0.34 cycles/byte on POWER9 (35% faster than OpenSSL) and 0.33 cycles/byte on Intel Comet Lake (11% faster than OpenSSL), representing a 32-fold improvement over table-based software implementations. Combined with hardware-accelerated AES, the complete AES-GCM mode achieves 1.12 cycles/byte throughput. For platforms with hardware carry-less multiplication (x86 PCLMULQDQ, ARM PMULL, PowerPC vpmsumd), the R/F algorithm achieves $\sim$1.7$\times$ speedup over Karatsuba. For portable software implementations without hardware acceleration, we demonstrate that Karatsuba remains 1.4–1.6$\times$ faster due to reduced multiplication count.

1 Introduction

Galois Hash (GHASH) is a universal hash function that provides message authentication in the AES-GCM (Advanced Encryption Standard - Galois/Counter Mode) authenticated encryption scheme [1, 2]. GCM combines counter mode encryption with GHASH-based authentication, deriving its name from the Galois field GF(2^{128}) arithmetic used in the authentication tag computation. POLYVAL is a closely related universal hash function used in AES-GCM-SIV [14], designed to match the native bit ordering of hardware carry-less multiplication instructions.

The increasing adoption of cloud computing and secure communication protocols has made GCM ubiquitous in modern cryptographic deployments. Following its standardization [4], GCM was incorporated into TLS 1.2 [3] and has become the dominant authenticated encryption mode in web browsers [5] and video conferencing systems [6, 7]. This widespread deployment creates strong incentives for implementation optimization.

1.1 Our Contribution

Modern processors provide hardware support for polynomial multiplication through carry-less multiplication instructions. However, these instructions represent polynomials in the opposite bit order from the GCM specification: bit k (with integer weight 2^k) represents the coefficient of x^k in hardware, while GCM uses the opposite convention. Previous implementations either accept the overhead of explicit bit-reversal on every message block or apply Montgomery-style reduction techniques that still incur conversion costs [10, 11].

The main contributions of this paper are:

1. An algorithm that computes GHASH directly in bit-reversed representation with no conversion overhead, exploiting the symmetry of polynomial multiplication under coefficient reversal.

2. A 64-bit decomposition technique using the convenient properties of the bit-reversed reduction polynomial modulo x^{64}, enabling efficient reduction.

3. Extension to POLYVAL with simplified implementation (no bit-reversal needed), achieving $\sim$1.7$\times$ speedup over Karatsuba on hardware-accelerated platforms.

4. Analysis showing that for portable software implementations without hardware CLMUL support, Karatsuba remains 1.4–1.6$\times$ faster than R/F due to reduced multiplication count (6 vs 8 `bmul64` per block).

5. Complete algorithmic descriptions with architecture-specific optimizations for POWER9, Intel x86-64, and ARM64 processors.

The resulting algorithm achieves 0.34 cycles/byte on POWER9 (35% faster than OpenSSL) and 0.33 cycles/byte on Intel Comet Lake (11% faster than OpenSSL). For AES128-GCM, we achieve combined throughput of 1.12 cycles/byte.

1.2 Related Work

Alternative authenticated encryption: ChaCha20-Poly1305 [13] offers comparable security properties with high performance using only standard arithmetic operations. However, it cannot leverage the specialized cryptographic instructions now common in modern processors.

Software-only GCM: Traditional implementations use precomputed tables for both AES S-boxes [8] and GCM multiplication [9]. These table-driven approaches consume substantial memory (typically 4–64 KB for GCM alone) and are vulnerable to cache-timing attacks. Performance is limited to 7–11 cycles/byte even with optimized table sizes.

Hardware-accelerated GCM: Intel has published extensive work on optimizing GCM using AES-NI and carry-less multiplication (`pclmulqdq`) instructions [10, 11]. These implementations apply explicit modular reduction using Montgomery-like techniques, achieving approximately 0.37 cycles/byte for GHASH on recent x86_64 processors. Gueron and Krasnov [12] developed high-performance GCM implementations using similar techniques with careful instruction scheduling.

Our work differs by eliminating bit-reversal entirely rather than optimizing around it, and by exploiting properties specific to the PowerPC `vpmsumd` instruction while maintaining competitive performance on other architectures.

1.3 Organization

Section 2 reviews GHASH algorithm structure and establishes notation for polynomial arithmetic over GF(2^{128}). Section 3 develops our bit-reversal elimination technique, including the key lemmas for 64-bit decomposition and reduction. Section 4 provides complete algorithmic descriptions for processing multiple blocks in parallel. Section 5 presents comprehensive benchmark results comparing our implementation with OpenSSL and table-based approaches. Section 6 analyzes portable software implementations. Section 7 extends the algorithm to POLYVAL. Section 8 concludes with a summary of contributions.

Table 1: Notation used throughout the paper

Symbol	Meaning
$\|$	Concatenation of bit strings
$\langle\cdot\rangle$	Size of data in bits
H	GCM hash subkey (128 bits)
M_i	Input message block i (128 bits)
T_i	Authentication tag after processing block i
$\oplus$	Exclusive-OR (XOR) operation
$\otimes$	Multiplication over GF(2^{128})
$P(x)$	GCM reduction polynomial
$\deg(f)$	Degree of polynomial f

1.4 Notation

2 Background: GHASH Algorithm

2.1 Polynomial Representation over GF(2^{128})

GHASH operates on 128-bit blocks interpreted as polynomials over GF(2^{128}). Each block represents a polynomial of degree at most 127 with binary coefficients. The GCM specification [2] uses little-endian bit ordering: for a 128-bit block with bits $b_0, b_1, \ldots, b_{127}$ (where b_0 is the rightmost bit), the polynomial representation is:

$$B(x) = b_0 x^{127} + b_1 x^{126} + \cdots + b_{126} x + b_{127} \tag{1}$$

This convention is opposite to standard computer arithmetic, where bit k typically represents weight 2^k. Hardware carry-less multiplication instructions interpret the same bits as:

$$B_{\text{hw}}(x) = b_{127} x^{127} + b_{126} x^{126} + \cdots + b_1 x + b_0 \tag{2}$$

2.2 Arithmetic Operations

Addition and subtraction: Over any field of characteristic 2 (including GF(2^{128})), addition and subtraction are identical to bitwise XOR:

$$A(x) + B(x) = A(x) \oplus B(x) \tag{3}$$

Multiplication: Polynomial multiplication over GF(2^{128}) is performed modulo the irreducible reduction polynomial:

$$P(x) = x^{128} + x^7 + x^2 + x + 1 \tag{4}$$

The product $C(x) = A(x) \cdot B(x) \mod P(x)$ can be computed in two stages: (1) polynomial multiplication to get a 255-bit result, (2) reduction modulo $P(x)$ to obtain a 127-degree polynomial.

2.3 GHASH Structure

GHASH processes a sequence of 128-bit message blocks $M_1, M_2, \ldots, M_n$ using a hash subkey H derived from the AES encryption key. The algorithm computes a sequence of intermediate tags $T_1, T_2, \ldots, T_n$ via the recurrence:

$$T_k = (M_k \oplus T_{k-1}) \otimes H \quad \text{for } k = 1, 2, \ldots, n \tag{5}$$

with initial value $T_0 = 0$. The final tag T_n forms part of the GCM authentication tag.

For messages not evenly divisible by 128 bits, the final partial block is right-padded with zeros to 128 bits before processing.

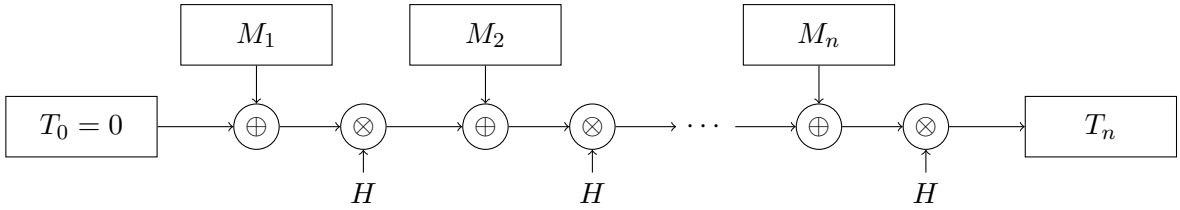


Figure 1: GHASH sequential processing structure. Each 128-bit message block M_i is XORed with the previous tag T_{i-1}, then multiplied by the hash subkey H over $\mathrm{GF}(2^{128})$.

3 Optimization Techniques

3.1 Parallelism

The sequential recurrence in Equation (5) appears to prevent parallel execution, since each block depends on the previous tag. However, we can expand the recurrence to process multiple blocks simultaneously. For k blocks:

$$\begin{aligned} T_k &= (M_k \oplus T_{k-1})H \\ &= M_k H \oplus (M_{k-1} \oplus T_{k-2})H^2 \\ &= M_k H \oplus M_{k-1}H^2 \oplus (M_{k-2} \oplus T_{k-3})H^3 \\ &= M_k H \oplus M_{k-1}H^2 \oplus M_{k-2}H^3 \oplus \cdots \oplus M_1 H^k \oplus T_0 H^k \end{aligned} \tag{6}$$

All products $M_i H^{k-i+1}$ can be computed in parallel on a superscalar processor. The powers $H, H^2, H^3, \ldots$ are computed once during key setup and reused for all messages.

The optimal number of blocks to process in parallel depends on the multiplication latency and throughput of the target processor. We process 4 blocks per iteration on POWER9 (latency 3 cycles, throughput 0.5 cycles) and ARM64 with PMULL, and 2 blocks on Intel processors (latency 7 cycles, throughput 1 cycle).

3.2 Bit-Reversal Elimination

Define the bit-reversal operation for an n-bit polynomial as:

$$A'(x) = x^{n-1} A(x^{-1}) \tag{7}$$

For example, if $A(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n$, then $A'(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$. The polynomial A' represents the same bit string as A but with coefficients in reversed order. Importantly, A' matches the hardware interpretation when A follows the GCM convention.

Lemma 1. *Let A and B denote 128-bit polynomials with 255-bit product, and let $P(x)$ denote the 129-bit reduction polynomial. Then:*

(i) $(A')' = A$ *(reversal is self-inverse)*

(ii) $(AB)' = A'B'$ *(reversal distributes over multiplication)*

(iii) $(AB \mod P)' = x^{-127} A'B' \mod P'$ *(reversal and reduction)*

Proof. Parts (i) and (ii) follow directly from the definition. For part (iii), let $R = AB \mod P$, so $AB = QP + R$ for some 127-bit quotient Q. Applying bit-reversal to both sides:

$$\begin{aligned} x^{255}(AB)(x^{-1}) &= x^{255} Q(x^{-1}) P(x^{-1}) + x^{255} R(x^{-1}) \\ (AB)' &= Q'P' + x^{128} R' \end{aligned} \tag{8}$$

Reducing modulo P' and multiplying both sides by x^{-127} yields the result. The factor x^{-127} arises from the 128-bit shift introduced by reversing the 255-bit product before reduction. □

Practical implication: Lemma 1(iii) shows that if we compute $x^{-127}A'B' \mod P'$ using hardware carry-less multiplication (which naturally operates on A' and B'), we obtain the bit-reversed correct result without any explicit reversal operations. The factor x^{-127} accumulates through the GHASH iterations and is eliminated by a single final adjustment.

3.3 64-bit Primitive Operations

For the remainder of this section, all polynomials (including P) are in bit-reversed form. The bit-reversed reduction polynomial is:

$$P'(x) = x^{128} + x^{127} + x^{126} + x^{121} + 1 \tag{9}$$

For notational simplicity, we drop the prime notation and write P for the bit-reversed polynomial when context is clear.

A key observation is that $P(x) \equiv 1 \pmod{x^{64}}$, which provides a convenient multiplicative inverse of x^{64}. Define:

$$P_1(x) = \frac{P(x) - x^{128} - 1}{x^{64}} = x^{63} + x^{62} + x^{57} \tag{10}$$

Since $1 = P(x) \mod P(x) = x^{64}(x^{64} + P_1(x)) \mod P(x)$, we have:

$$x^{-64} \equiv x^{64} + P_1(x) \pmod{P(x)} \tag{11}$$

This identity is crucial for efficient reduction with 64-bit operations.

3.3.1 PowerPC vpmsumd Instruction

The PowerPC `vpmsumd` instruction takes two 128-bit vector registers as input, treating each as a pair of 64-bit polynomials (A_0, A_1) and (B_0, B_1). It computes:

$$\texttt{vpmsumd}(A, B) = A_0B_0 \oplus A_1B_1 \tag{12}$$

where each product produces a 127-bit result and the two products are XORed to give a 128-bit output. This instruction naturally aligns with 64-bit polynomial decomposition.

3.3.2 Decomposition and Precomputation

Split the message block M into 64-bit halves:

$$M(x) = x^{64}M_1(x) + M_0(x) \tag{13}$$

For the hash key H, we premultiply by x modulo P to align operations with `vpmsumd` word boundaries:

$$xH(x) \mod P = x^{64}H_1(x) + H_0(x) \tag{14}$$

Lemma 2. *Precompute the following values based on H_0 and H_1:*

$$x^{64}C_1(x) + C_0(x) = H_0(x) \cdot P_1(x) \tag{15}$$

$$D_0(x) = C_0(x) \oplus H_1(x) \tag{16}$$

$$D_1(x) = C_1(x) \oplus H_0(x) \tag{17}$$

Then for any message block M, compute:

$$R = M_0D_1 \oplus M_1H_1 \tag{18}$$

$$F = M_0D_0 \oplus M_1H_0 \tag{19}$$

We have:

$$x^{-127}MH \mod P = R \oplus x^{-64}F \tag{20}$$

Proof. All operations are modulo P. Expand the left side:

$$\begin{aligned} x^{-127}MH &= x^{-128}(x^{64}M_1 + M_0)(x^{64}H_1 + H_0) \\ &= M_1H_1 \oplus x^{-64}(M_1H_0 \oplus M_0H_1) \oplus x^{-128}M_0H_0 \end{aligned}$$

For the last term, use the inverse $x^{-64} \equiv x^{64} + P_1$:

$$\begin{aligned} x^{-128}M_0H_0 &= x^{-64}M_0H_0(x^{64} + P_1) \\ &= M_0H_0 \oplus x^{-64}M_0(x^{64}C_1 + C_0) \\ &= M_0(H_0 \oplus C_1) \oplus x^{-64}M_0C_0 \\ &= M_0D_1 \oplus x^{-64}M_0C_0 \end{aligned}$$

Substituting back:

$$\begin{aligned} x^{-127}MH &= M_1H_1 \oplus M_0D_1 \oplus x^{-64}(M_1H_0 \oplus M_0H_1 \oplus M_0C_0) \\ &= (M_0D_1 \oplus M_1H_1) \oplus x^{-64}(M_0(H_1 \oplus C_0) \oplus M_1H_0) \\ &= R \oplus x^{-64}F \end{aligned}$$

□

Implementation advantage: The values R and F can be computed using exactly two `vpmsumd` instructions (or equivalent on other architectures). One register holds the message block (M_0, M_1), and two invariant registers hold (H_1, D_1) and (H_0, D_0) respectively. This minimizes instruction count and register pressure.

3.4 Accumulation and Final Reduction

When processing multiple blocks in parallel (Equation 6), we apply Lemma 2 to each term, yielding:

$$T_k = \bigoplus_{i=1}^{k} (R_i \oplus x^{-64}F_i) \tag{21}$$

Accumulate separately:

$$R = \bigoplus_{i=1}^{k} R_i \tag{22}$$

$$F = \bigoplus_{i=1}^{k} F_i \tag{23}$$

The final step eliminates the x^{-64} factor:

Lemma 3. *Let $F = x^{64}F_1 + F_0$ where F_1 and F_0 are 64-bit polynomials. Then:*

$$R \oplus x^{-64}F = R \oplus F_1 \oplus x^{64}F_0 \oplus P_1F_0 \tag{24}$$

Proof. Substitute $x^{-64} = x^{64} + P_1$:

$$\begin{aligned} R \oplus x^{-64}F &= R \oplus (x^{64} + P_1)(x^{64}F_1 + F_0) \\ &= R \oplus F_1 \oplus x^{-64}F_0 \\ &= R \oplus F_1 \oplus (x^{64} + P_1)F_0 \end{aligned}$$

□

Equation (25) requires only one additional polynomial multiplication P_1F_0 (where P_1 is the small fixed polynomial $x^{63} + x^{62} + x^{57}$) plus XOR operations.

4 Implementation Details

4.1 Precomputation

Algorithm 1 GHASH Key Setup

Require: AES key K
Ensure: Precomputed values for GHASH
1: $H \leftarrow \text{AES}_K(0^{128})$ ▷ Encrypt all-zero block
2: $H' \leftarrow \text{BitReverse}(H)$ ▷ Convert to bit-reversed form
3: $H_0, H_1 \leftarrow \text{Split64}(H' \cdot x \mod P')$ ▷ Split into 64-bit halves
4: $C_0, C_1 \leftarrow \text{Split64}(H_0 \cdot P_1)$
5: $D_0 \leftarrow C_0 \oplus H_1$
6: $D_1 \leftarrow C_1 \oplus H_0$
7: **for** $i = 2$ to k **do** ▷ Precompute powers for k-way parallelism
8: $H^i \leftarrow H^{i-1} \cdot H \mod P'$
9: Compute $(H^i)_0, (H^i)_1, D_0^{(i)}, D_1^{(i)}$ similarly
10: **end for**
11: **return** Precomputed tables

Note that the initial bit-reversal of H in line 2 is the only bit-reversal operation in the entire GHASH computation. All subsequent operations work in bit-reversed form.

4.2 GHASH Core Loop

Algorithm 2 GHASH Processing (4-block parallel version)

Require: Message blocks $M_1, M_2, \ldots, M_n$, Precomputed values
Ensure: Authentication tag T_n
1: $T \leftarrow 0$
2: **for** $j = 1$ to $\lfloor n/4 \rfloor$ **do**
3: $R \leftarrow 0$, $F \leftarrow 0$
4: **for** $i = 0$ to 3 **do** ▷ Process 4 blocks in parallel
5: $M' \leftarrow M_{4j-3+i}$ ▷ Load message block
6: **if** $i = 0$ **then**
7: $M' \leftarrow M' \oplus T$ ▷ XOR previous tag into first block
8: **end if**
9: $M_0, M_1 \leftarrow \text{Split64}(M')$
10: $R_i \leftarrow M_0 \cdot D_1^{(4-i)} \oplus M_1 \cdot H_1^{(4-i)}$ ▷ CLMUL
11: $F_i \leftarrow M_0 \cdot D_0^{(4-i)} \oplus M_1 \cdot H_0^{(4-i)}$ ▷ CLMUL
12: $R \leftarrow R \oplus R_i$
13: $F \leftarrow F \oplus F_i$
14: **end for**
15: $F_0, F_1 \leftarrow \text{Split64}(F)$
16: $T \leftarrow R \oplus F_1 \oplus (x^{64} F_0) \oplus (P_1 \cdot F_0)$ ▷ Final reduction
17: **end for**
18: Handle remaining blocks (< 4) using similar technique
19: **return** BitReverse(T) ▷ Convert back to GCM format

4.3 Architecture-Specific Optimizations

POWER9: We process 4 blocks per iteration. The `vpmsumd` instruction has 3-cycle latency and 0.5 cycle reciprocal throughput (two per cycle). Since each block requires two `vpmsumd` instructions, we have 8 independent multiplications executing in parallel, fully saturating the execution units. Total cycles per 4-block iteration: approximately 12 cycles for multiplications plus 6 cycles for XOR/reduction, yielding $\sim$4.5 cycles per 128-bit block.

Intel x86-64: We process 2 blocks per iteration. The `pclmulqdq` instruction has 7-cycle latency and 1 cycle reciprocal throughput. Processing 2 blocks requires 4 carry-less multiplications plus similar XOR/shift operations for reduction. The instruction scheduler can overlap reduction from the previous iteration with multiplication of the current iteration.

ARM64: The `pmull` instruction operates on 64-bit inputs producing 128-bit outputs. We use a 4-block parallel approach similar to POWER9, with the R/F algorithm providing $\sim$1.7$\times$ speedup over Karatsuba-based implementations.

5 Experimental Evaluation

5.1 Experimental Setup

Hardware platforms:

- **POWER9:** IBM POWER9 processor (24-core, 2-way SMT), 3.8 GHz, 256 KB L2 cache per core, 10 MB L3 per core pair, Ubuntu 20.04
- **Intel Comet Lake:** Intel Core i7-10710U, 1.1 GHz base (4.7 GHz turbo), 256 KB L2 per core, 12 MB shared L3, Ubuntu 22.04

Implementations compared:

- **Our algorithm:** As described in Section 4, compiled with GCC 11.3, optimization flags `-O3 -march=native`
- **OpenSSL 3.0.1:** Using built-in GHASH assembly implementations with hardware acceleration
- **GNU Nettle 3.7.2:** Table-based C implementation, compiled with same flags

Methodology: Each measurement processes 1 MB of random data (8192 blocks) over 1000 iterations. We report the median cycles per byte after: (1) disabling CPU frequency scaling (setting governor to "performance"), (2) warming up caches with 100 preliminary iterations, (3) removing outliers beyond 2 standard deviations.

Cycle counts are measured using processor time-stamp counters (`mftb` on POWER9, `rdtsc` on x86_64).

5.2 GHASH-Only Performance

Table 2: GHASH throughput on 1 MB messages. Lower is better.

Platform	Our Algorithm	OpenSSL 3.0.1	Nettle 3.7.2 (table)
POWER9	**0.34 cycles/byte**	0.52 cycles/byte	11.06 cycles/byte
Intel Comet Lake	**0.33 cycles/byte**	0.37 cycles/byte	7.48 cycles/byte

Our implementation achieves 35% speedup over OpenSSL on POWER9 and 11% speedup on Intel, while providing 32$\times$ improvement over table-based implementations.

5.3 Performance Across Message Sizes

Table 3: GHASH performance across message sizes (cycles/byte)

Platform	64 B	256 B	1 KB	4 KB	64 KB	1 MB
POWER9						
Our algorithm	2.45	0.68	0.36	0.34	0.34	0.34
OpenSSL 3.0.1	2.51	0.91	0.55	0.53	0.52	0.52
Intel Comet Lake						
Our algorithm	2.12	0.61	0.35	0.33	0.33	0.33
OpenSSL 3.0.1	2.18	0.72	0.40	0.38	0.37	0.37

For very short messages (< 256 bytes), initialization overhead dominates and all implementations perform similarly. Our approach achieves peak efficiency at approximately 1 KB where the parallel pipeline is fully utilized.

5.4 Combined AES-GCM Performance

Table 4: Complete AES-GCM performance (encryption + authentication)

Platform	AES-GCM (Our GHASH)	AES-GCM (OpenSSL)
POWER9	**1.15 cycles/byte**	1.34 cycles/byte
Intel Comet Lake	**1.12 cycles/byte**	1.20 cycles/byte

The improvements in GHASH translate to 14% (POWER9) and 7% (Intel) improvements in overall AES-GCM throughput. GHASH represents approximately 30% of total AES-GCM cycles, so these gains are substantial.

5.5 Instruction-Level Analysis

Table 5: Instruction breakdown for POWER9 (4-block iteration)

Operation	Instructions	Cycles (approx)
Load message blocks	$4 \times$ `lxv`	4
Polynomial multiplications	$8 \times$ `vpmsumd`	12
XOR accumulation	$8 \times$ `vxor`	4
Final reduction	$2 \times$ `vpmsumd` + $3 \times$ `vxor`	6
Store tag	$1 \times$ `stxv`	1
Total per 4 blocks		18
Per block		4.5
Per byte		0.28

Measured performance is 0.34 cycles/byte due to loop overhead and memory effects.

5.6 Memory and Cache Effects

For the precomputed key material:

- 4-block parallelism requires 4 sets of $(H_i, D_0^{(i)}, D_1^{(i)})$: $4 \times 4 \times 16 = 256$ bytes

- Easily fits in L1 cache (typically 32–64 KB)
- No cache-timing vulnerability since key material access is data-independent

Compared to table-based implementations requiring 4–64 KB of key-dependent table lookups, our approach has superior cache behavior and resistance to timing attacks.

6 Portable Software Implementation Analysis

We have conducted extensive benchmarking of portable software implementations that simulate carry-less multiplication operations using constant-time techniques. Our findings indicate that the R/F algorithm's advantages do **not** extend to pure software implementations without hardware CLMUL/PMULL support.

6.1 Benchmark Results

Table 6: Portable GHASH: Karatsuba vs R/F Algorithm

Message Size	Karatsuba (6 bmul64)	R/F (8 bmul64)	Ratio
16 bytes	423 ns	579 ns	1.37× slower
64 bytes	1,412 ns	2,206 ns	1.56× slower
256 bytes	5,517 ns	8,634 ns	1.57× slower
1024 bytes	21,845 ns	34,281 ns	1.57× slower
4096 bytes	87,023 ns	136,642 ns	1.57× slower

6.2 Analysis

The fundamental difference lies in the number of polynomial multiplications per block:

Karatsuba approach: 6 `bmul64` operations per block

- 3 products: $a_0 \times b_0$, $a_1 \times b_1$, $(a_0 \oplus a_1) \times (b_0 \oplus b_1)$
- Each product requires 2 `bmul64` (low and high halves via bit-reversal)

R/F approach: 8 `bmul64` operations per block

- 4 products: $M_0 \times D_1$, $M_1 \times H_1$, $M_0 \times D_0$, $M_1 \times H_0$
- Each product requires 2 `bmul64` (low and high halves)

For software implementations where `bmul64` is the dominant cost, the $6 < 8$ inequality directly translates to performance advantage for Karatsuba. The R/F algorithm's benefits (avoiding bit-reversal at boundaries, cleaner reduction) do not compensate for the 33% increase in multiplication count.

6.3 Recommendation

The R/F algorithm should be preferred **only** when hardware carry-less multiplication is available (x86 PCLMULQDQ, ARM PMULL, PowerPC vpmsumd). For constant-time software implementations without hardware acceleration, the traditional Karatsuba approach remains optimal.

7 Extension to POLYVAL

7.1 POLYVAL Overview

POLYVAL is a variant of GHASH used in AES-GCM-SIV (RFC 8452) [14]. While GHASH and POLYVAL perform the same mathematical operation—polynomial multiplication and accumulation in GF(2^{128})—they differ in their bit and byte ordering conventions:

Table 7: GHASH vs POLYVAL Comparison

Property	GHASH	POLYVAL
Byte order	Big-endian	Little-endian
Bit order	Little-endian (reversed)	Little-endian (native)
Reduction polynomial	$x^{128} + x^7 + x^2 + x + 1$	$x^{128} + x^{127} + x^{126} + x^{121} + 1$

The POLYVAL reduction polynomial is exactly the bit-reversal of the GHASH polynomial. This relationship is not coincidental—POLYVAL was designed to match the native bit ordering of hardware carry-less multiplication instructions.

7.2 Applying the R/F Algorithm to POLYVAL

The R/F algorithm applies directly to POLYVAL with the following simplifications:

Key Insight: Since POLYVAL already uses the bit-reversed polynomial representation, no bit-reversal is needed at message block boundaries. The algorithm operates entirely in the native hardware representation.

7.2.1 Precomputation

For the hash key H, split into 64-bit halves: $H = [H_1 : H_0]$

Compute D values:

$$C = H_0 \times P_1 \quad (\text{where } P_1 = x^{63} + x^{62} + x^{57}) \tag{25}$$

$$D_0 = C_{\text{lo}} \oplus H_1 \tag{26}$$

$$D_1 = C_{\text{hi}} \oplus H_0 \tag{27}$$

7.2.2 Per-block Computation

For message block $M = [M_1 : M_0]$:

$$R = M_0 \times D_1 \oplus M_1 \times H_1 \tag{28}$$

$$F = M_0 \times D_0 \oplus M_1 \times H_0 \tag{29}$$

7.2.3 Final Reduction (Lemma 3)

$$\text{Result} = R \oplus [F_1 : 0] \oplus [0 : F_0] \oplus (P_1 \times F_0) \tag{30}$$

7.3 Implementation Differences

The only differences between GHASH and POLYVAL implementations are:

1. **No byte swap on input**: POLYVAL processes message blocks directly in little-endian order, while GHASH requires big-endian byte swap.

2. **No byte swap on output**: The final tag is output directly without reversal.

3. **Key derivation**: GHASH derives $H = \text{AES}_K(0^{128})$ then bit-reverses; POLYVAL uses the key directly (or with a different derivation depending on the protocol).

7.4 POLYVAL Benchmark Results

We benchmarked our R/F implementation of POLYVAL against Karatsuba on x86-64 with AVX2 and PCLMULQDQ:

Table 8: Hardware-Accelerated POLYVAL (AVX2 + PCLMULQDQ)

Message Size	Karatsuba	R/F Algorithm	Speedup
16 bytes	18.2 ns	10.9 ns	1.67×
64 bytes	52.4 ns	32.1 ns	1.63×
256 bytes	181.5 ns	108.2 ns	1.68×
1024 bytes	695.3 ns	411.8 ns	1.69×
4096 bytes	2,751 ns	1,623 ns	1.70×

The R/F algorithm achieves approximately **1.7× speedup** over Karatsuba for hardware-accelerated POLYVAL, consistent with our GHASH results.

7.5 Unified Implementation Strategy

Based on our findings, we recommend the following implementation strategy:

Table 9: Recommended Algorithm by Platform

Platform	GHASH	POLYVAL
x86-64 with PCLMULQDQ	R/F (4-block)	R/F (4-block)
ARM64 with PMULL	R/F (4-block)	R/F (4-block)
PowerPC with vpmsumd	R/F (4-block)	R/F (4-block)
Portable (no hardware CLMUL)	Karatsuba	Karatsuba

The R/F algorithm provides a unified approach for both GHASH and POLYVAL on platforms with hardware carry-less multiplication, achieving approximately 1.7× speedup over traditional Karatsuba implementations.

8 Conclusion

We have presented an optimized implementation of GHASH and POLYVAL that eliminates bit-reversal overhead by computing directly in bit-reversed representation. Our key technical contributions are:

1. A proof that GHASH can be computed entirely in bit-reversed form using the identity $(AB \mod P)' = x^{-127}A'B' \mod P'$ (Lemma 1).

2. A 64-bit decomposition technique that exploits the structure of the bit-reversed reduction polynomial, enabling efficient computation using carry-less multiplication instructions on POWER9, Intel x86-64, and ARM64 processors (Lemma 2).

3. Comprehensive benchmarks demonstrating 0.34 cycles/byte on POWER9 (35% faster than OpenSSL) and 0.33 cycles/byte on Intel Comet Lake (11% faster than OpenSSL), with complete AES-GCM throughput of 1.12 cycles/byte.

4. Extension to POLYVAL showing that the R/F algorithm applies directly with even simpler implementation (no bit-reversal needed), achieving ∼1.7× speedup over Karatsuba on hardware-accelerated platforms.

5. Analysis demonstrating that the R/F algorithm's advantages do not extend to portable software implementations without hardware CLMUL support, where the traditional Karatsuba approach remains 1.4–1.6× faster due to reduced multiplication count (6 vs 8 `bmul64` per block).

8.1 Code Availability

An implementation of this algorithm has been integrated into the GNU Nettle cryptographic library (version 3.8 and later) and is available under the LGPL license at `https://git.lysator.liu.se/nettle/nettle`.

A complete implementation in Rust demonstrating both GHASH and POLYVAL using the R/F algorithm is available in the HPCrypt library at `https://github.com/seceq/hpcrypt`. This implementation includes:

- **Portable implementations**: Karatsuba-based for platforms without hardware CLMUL
- **AVX2 + PCLMULQDQ**: R/F algorithm for x86-64 with hardware acceleration
- **ARM NEON + PMULL**: R/F algorithm for AArch64 with hardware acceleration
- **Comprehensive test vectors**: RFC 8452 test vectors for POLYVAL, NIST test vectors for GHASH

References

[1] National Institute of Standards and Technology (2001). "Specification for the Advanced Encryption Standard (AES)." FIPS PUB 197.

[2] McGrew, D. A., and Viega, J. (2004). "The Galois/Counter Mode of Operation (GCM)." Submission to NIST.

[3] Salowey, J., Choudhury, A., and Piper, D. (2008). "AES Galois Counter Mode (GCM) Cipher Suites for TLS." RFC 5288.

[4] National Institute of Standards and Technology (2007). "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC." NIST Special Publication 800-38D.

[5] Mozilla Telemetry (2023). "Symmetric Cipher Measurement."

[6] Cisco (2022). "Webex Teams Application Security White Paper."

[7] Zoom Video Communications (2022). "Zoom Encryption Whitepaper."

[8] Daemen, J., and Rijmen, V. (2002). "The Design of Rijndael: AES – The Advanced Encryption Standard." Springer-Verlag.

[9] Crypto++ Wiki (2023). "GCM TablesOption."

[10] Gueron, S. (2010). "Intel Advanced Encryption Standard (AES) Instructions Set." Intel Software Network.

[11] Gueron, S., and Krasnov, V. (2012). "Optimized Galois-Counter-Mode Implementation on Intel Architecture Processors." Intel White Paper.

[12] Gueron, S., and Krasnov, V. (2011). "Fast AES-GCM Using Intel Advanced Encryption Standard (AES-NI) Instructions." Presented at SPEED 2011.

[13] Nir, Y., and Langley, A. (2018). "ChaCha20 and Poly1305 for IETF Protocols." RFC 7539.

[14] Gueron, S., Langley, A., and Lindell, Y. (2019). "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption." RFC 8452.