ChaCha20-Poly1305 Parallel Decryption Benchmark Report

Benchmark source code: github.com/mjtomei/omerta_mesh/benchmarks/crypto

Summary

We benchmarked ChaCha20-Poly1305 parallel chunked encryption/decryption for the omerta_mesh tunnel protocol on Linux (20-core ARM) and Mac (8-core Apple Silicon). Key findings:

- **The Swift API is the bottleneck, not the crypto.** The ChaChaPoly Swift API allocates a `Data` object and performs ARC reference counting on every call. At 512B chunks (256 calls per 128KB block), this overhead consumes 74% of encrypt time on Linux. Calling BoringSSL's C API directly with pre-allocated buffers eliminates this, achieving 1146 MB/s encrypt at 512B chunks — a 2.6x improvement over the Swift API.
- **CryptoKit is unexpectedly slow.** On Mac, CryptoKit's unchunked peak (645 MB/s) is 2.5x slower than BoringSSL running on the same hardware (1616 MB/s). The cause is unclear — it may be additional validation, side-channel mitigations, or intentional rate limiting. It is not a hardware limitation.
- **Linux threads contend; processes don't.** All in-process parallelism strategies (pthread, GCD, TaskGroup, etc.) degrade to ~65% efficiency at 8 workers on Linux due to malloc arena contention from the Swift API's per-call allocations. Separate processes scale near-perfectly (96-100%). The direct BoringSSL API eliminates this contention even within threads.
- **512B chunks are viable with the direct API.** The remaining ~2x overhead at 512B (vs unchunked) is irreducible per-call crypto setup (Poly1305 key derivation, SIMD pipeline ramp-up). A future multi-buffer kernel could reduce this, but 1146 MB/s (9.2 Gbps) per core is sufficient for tunnel workloads. This is a tradeoff made for the potential of lower latency in more highly parallel implementations in the future.
- **Mac scaling is hardware-limited.** The 4P+4E core architecture caps 8-worker efficiency at ~62-72% regardless of API or strategy. No software change can improve this.

Background

The omerta_mesh tunnel protocol uses ChaCha20-Poly1305 (RFC 8439) with a chunked payload format and 512-byte chunks. This chunking enables parallel decrypt/authenticate using the standard library APIs without custom crypto code, since each chunk is an independent AEAD operation.

This report investigates the practical scaling behavior of parallel chunked encryption and decryption on two platforms.

Test Machines

	Linux	Mac
CPU	ARM (aarch64), NVIDIA GB10	Apple Silicon M-series
Cores	20 (homogeneous)	8 (4 Performance + 4 Efficiency)
OS	Ubuntu 24.04.3	macOS 26.2 (Tahoe)
Swift	6.2.3	6.2.3
Crypto lib	swift-crypto (BoringSSL), vendored CBoringSSL	CryptoKit (native), vendored CBoringSSL

Single-Core Throughput Baselines

First we present the single core, no chunking throughput on both machines we run our evaluations on. Note that CryptoKit is dramatically slower even though the hardware doesn't require it. This suggests CryptoKit may have artificial performance limitations.

Platform	Encrypt	Decrypt
Linux (Swift API)	1732 MB/s (13.9 Gbps)	1991 MB/s (15.9 Gbps)
Linux (Direct BoringSSL)	2041 MB/s (16.3 Gbps)	2033 MB/s (16.3 Gbps)
Mac (CryptoKit)	645 MB/s (5.2 Gbps)	659 MB/s (5.3 Gbps)
Mac (Direct BoringSSL)	1616 MB/s (12.9 Gbps)	1752 MB/s (14.0 Gbps)

A single core on either platform has enough throughput to handle a 1 Gbps network interface, even with a 2x slowdown from chunking overhead.

Worker Type Tests

The first test we ran was to understand what portion of the overhead we saw even with large blocks was coming from work dispatch mechanisms.

Each worker operates completely independently: 1. Allocates its own 128KB memory block 2. Pre-allocates a `SealedBox` array by encrypting once 3. In a timed loop (1000 iterations): encrypts all chunks, then decrypts all chunks 4. No shared state between workers — each has its own key, buffers, and boxes

Six parallelism strategies were compared: - **pthread**: Manual `pthread_create`/`pthread_join` - **process**: `posix_spawn` of child processes (re-exec with `--child-worker`) - **concurrentPerform**: `DispatchQueue.concurrentPerform` (GCD parallel-for) - **gcd.async**: `DispatchQueue.global().async` with `DispatchGroup` - **TaskGroup**: Swift structured concurrency (`withTaskGroup`) - **OperationQueue**: Foundation `OperationQueue` with `BlockOperation`

Metrics: - **Efficiency** = baseline_time / wall_time. 100% means N workers finish in the same wall time as 1 worker (perfect parallelism for independent work). - **Worker slowdown** = avg_worker_elapsed / baseline. 1.0x means each worker runs at full single-threaded speed.

Results: Strategy Comparison at 512B Chunks

Mac (8 cores: 4P + 4E)

Strategy	1w	2w	4w	8w
pthread	103%	96%	88%	61%
process	102%	97%	92%	66%
concPerform	102%	96%	88%	60%
gcd.async	101%	96%	89%	62%
TaskGroup	100%	96%	89%	61%
OpQueue	100%	96%	89%	61%

All strategies perform identically. The 4→8 worker cliff (89%→61%) corresponds exactly to the P-core→E-core transition. Worker slowdown at 8 workers is 1.55-1.61x regardless of strategy or chunk size, confirming this is a hardware constraint (DVFS / heterogeneous core speeds), not a software bottleneck. The 1 to 4 worker results also suggest that the cores have a boost mode that is being utilized for the smaller number of workers which will hurt strong scaling results but may be increasing energy efficiency if the boost is a result of overvolting.

Linux (20 homogeneous cores)

Strategy	1w	2w	4w	8w
pthread	96%	80%	73%	65%
process	**100%**	**100%**	**98%**	**96%**
concPerform	93%	81%	86%	65%
gcd.async	96%	94%	74%	65%
TaskGroup	99%	78%	75%	64%
OpQueue	95%	78%	72%	67%

Processes scale near-perfectly; all in-process strategies degrade to ~65%. The per-worker slowdown on Linux is chunk-size dependent (1.45x at 512B vs 1.07x at 64KB with 8 pthread workers), confirming the bottleneck is per-call overhead that contends across threads — most likely glibc malloc arena contention from the `Data` allocations inside `ChaChaPoly.open()`. The data suggests the only way to achieve good performance scaling with our installation of Swift on Linux is to use multiprocessing.

Analysis: Per-Chunk Overhead

A separate test measured the fixed cost per `ChaChaPoly.open()` call by comparing whole-payload decrypt to chunked decrypt at various sizes:

Platform	Fixed overhead per `open()` call	128KB whole decrypt
Linux	~450-500 ns	65 µs
Mac	~850-1000 ns	194 µs

At 512B chunks (256 calls per 128KB payload), this overhead is substantial: - Linux: 256 × 480ns = 123µs of overhead on a 65µs base = crypto work is only 34% of total time - Mac: 256 × 950ns = 243µs of overhead on a 194µs base = overhead exceeds the crypto work

The overhead is per-call, not per-byte, and includes: - Internal `Data` allocation for the plaintext return value - ARC atomic reference counting on `SealedBox` internals - Poly1305 key derivation and nonce setup

Chunk size impact on throughput

To evaluate how performance scaling changes with block size, we focus on results specific to multiprocessing since that is the only method of work dispatch that achieves close to optimal performance on both systems. We found that scaling behavior does not change much with block size, so we only include results for a single process in the table below.

Using child-reported per-operation times (excluding process spawn overhead), we measured aggregate encrypt and decrypt throughput. All numbers are normalized to the **Swift API peak baselines** (Linux: Enc 1732 MB/s, Dec 1991 MB/s; Mac: Enc 645 MB/s, Dec 659 MB/s).

Chunk	Linux enc %peak	Linux dec %peak	Mac enc %peak	Mac dec %peak
512	26%	34%	31%	43%
1024	40%	48%	49%	63%
2048	58%	67%	66%	77%
4096	78%	82%	83%	89%
8192	93%	90%	93%	96%

Chunk	Linux enc %peak	Linux dec %peak	Mac enc %peak	Mac dec %peak
16384	100%	94%	100%	101%
32768	106%	98%	103%	101%
65536	111%	99%	106%	103%

At 512B chunks, encrypt runs at only 26-31% of peak — nearly 3/4 of the time is per-call overhead. The knee is at 4KB-8KB (78-93% of peak). Values above 100% at large chunk sizes suggest the peak baseline (single 128KB AEAD) has slightly higher per-byte cost than smaller operations, possibly due to cache effects.

The current 512B default chunk size is too small for efficient use of the CryptoKit/swift-crypto API.

That said, even at 512B chunks a single core still has enough absolute throughput to handle a 1 Gbps interface (~450 MB/s encrypt on Linux, ~200 MB/s on Mac). The inefficiency is still worth addressing — unnecessary CPU cycles spent on encryption overhead drain battery life and reduce the compute available for other workloads like compute sharing.

Direct BoringSSL API

To test whether the per-call overhead is truly from `Data` allocation and ARC (rather than crypto setup), we bypassed the swift-crypto/CryptoKit API entirely by vendoring BoringSSL as a standalone C target (`CBoringSSL`) and calling `EVP_AEAD_CTX_seal`/`EVP_AEAD_CTX_open` directly. The direct API operates on caller-provided buffers with zero per-iteration allocation. This works on both Linux and Mac — no platform-specific hacks required.

The direct API eliminates all Swift-level allocation by encrypting/decrypting into pre-allocated caller-owned buffers. The improvement is largest at small chunk sizes where per-call overhead dominates, and diminishes at large chunk sizes where the crypto work dominates — confirming the overhead is `Data` allocation + ARC reference counting, not Poly1305 key derivation or nonce setup.

Peak Baselines

Platform	Encrypt peak	Decrypt peak
Linux	2041 MB/s	2033 MB/s
Mac	1616 MB/s	1752 MB/s

Linux (20 cores) — Single-Worker by Chunk Size

Percentages are relative to the Swift API peak baseline (Enc 1732 MB/s, Dec 1991 MB/s).

Chunk	Enc MB/s	Dec MB/s	Enc % Swift pk	Dec % Swift pk
512	1146	909	66%	46%
1024	1405	1183	81%	59%
2048	1689	1535	98%	77%
4096	1881	1791	109%	90%
8192	1954	1895	113%	95%
16384	1994	1955	115%	98%
32768	2025	2006	117%	101%
65536	2041	2025	118%	102%

Mac (8 cores: 4P + 4E) — Single-Worker by Chunk Size

Percentages are relative to the Swift API (CryptoKit) peak baseline (Enc 645 MB/s, Dec 659 MB/s). Values >100% show how much the direct BoringSSL API exceeds the CryptoKit baseline.

Chunk	Enc MB/s	Dec MB/s	Enc % Swift pk	Dec % Swift pk
512	841	712	130%	108%
1024	1088	958	169%	145%
2048	1351	1277	209%	194%
4096	1563	1517	242%	230%
8192	1648	1617	256%	245%
16384	1696	1674	263%	254%
32768	1730	1718	268%	261%
65536	1745	1747	271%	265%

On Mac, the direct BoringSSL API is dramatically faster than the CryptoKit Swift API — **2.5x for encrypt and 2.7x for decrypt** at the unchunked peak. Even at 512B chunks, the direct API achieves 130% of CryptoKit's unchunked encrypt peak — the smallest chunks with the direct API are still faster than CryptoKit's best case.