Software-based parallel GHASH: a contiguous approach

Hayato Fujii[1], **Isabela Rossales**[1]

[1] KRYPTUS Information Security S.A. - Campinas, SP - Brazil

{hayato.fujii,isabela}@kryptus.com

Abstract. *Galois/Counter Mode (GCM) is a widely adopted block cipher mode of operation. Such degree of adoption has led to optimizations at every level, to the point that dedicated hardware support and specialized CPU instructions were developed, reducing the encryption process' computational overhead. In this proposal, multiple threads concurrently perform encryption and compute the GHASH function — an essential component of GCM — on disjoint segments of the input message. Upon completion, the intermediate authentication tags derived from each segment are combined into a single final authentication tag. This approach exploits a subtle aspect not explicitly addressed in the original specification, which presents GHASH as a strictly serial algorithm.*

1. Introduction

NIST's Special Publication 800-38D [Dworkin 2007] recommends the use of the Galois/-Counter Mode (GCM), an authenticated encryption with additional data (AEAD) mode of operation, combined with an approved block cipher, such as the Advanced Encryption Standard (AES), within U.S. federal agencies. This regulation reaches even to space applications, as noted by NASA JPL's David Foor in a call for comments during the first review of SP 800-38D [Foor 2025]. The recommendation reaches beyond government use; as an example, GCM is the only approved cipher mode for EAP-based cipher suites in WPA3-Enterprise deployments [Wi-Fi Alliance 2025].

The authentication aspect of the mode is provided by GHASH, a keyed hash function defined over $GF\left(2^{128}\right)$. When GCM is combined with a fast block cipher implementation, the former's computation emerges as the primary performance bottleneck due to the cost of operations in the finite field. Thanks to improvements in hardware design, this slowdown has diminished, as observed by [Gueron 2023].

In terms of parallelism, the "counter" aspect of the mode is easily implementable, as sequentially incremented counters are efficiently encrypted without critical paths. However, the authentication part does not benefit from the same strategy: the GHASH function is defined, in the original GCM specification, as a sequential algorithm, allowing limited opportunities to implement a parallel version in software.

As AES is commonly combined with GCM, modern (2010 onwards) Intel and ARM CPU cores often support hardware-accelerated instructions for both the block cipher [Akdemir et al. 2010] and polynomial multiplication [Gueron and Kounavis 2010] contexts in order to speed up implementations of such scheme; [Gueron 2023] reports speeds less than 1 cycle per byte encrypted for 8KiB messages in Intel platforms, while [Gouvêa and López 2015] reports 1.7 cycles per byte using an ARMv8-based CPU. In both cases, extensive use of the processor's pipeline greatly reduces the latency to encrypt larger messages.

This work began out of frustration exploring parallelism in GCM; a thread on the well-known development Q&A website[1] presents two conflicting answers: one stating that, "by looking at its scheme, no, the GHASH function is not parallelizable", and (the correct one) arguing that "yes, it is parallelizable, based on its mathematical construction". Indeed, one of the authors of this paper believed that GHASH was strictly serial before looking at the field multiplication and realizing that the GHASH function had a $GF(2^{128})$ polynomial underneath it.

1.1. Related Work

The original GCM specification [McGrew and Viega 2005] proposes that parallel hardware implementations of GHASH can divide even and odd input blocks between two multipliers, following a strategy similar to that outlined in the CWC-AES mode specification [Kohno 2003]. We explore this technique further in Section 2.

On the software side, the specification does not exploit parallelization, instead focusing on table-driven multiplication in $GF(2^{128})$. Intel's white papers, on the other hand, demonstrate how the x86 AES instruction set can be used to speed up AES-GCM implementations [Akdemir et al. 2010, Gopal et al. 2010]. These papers describe a method in which the powers of the GHASH subkey H modulo $P = x^{128} + x^7 + x^2 + x + 1$ are precomputed and used during encryption calls processing at least 64 bytes (i.e., four blocks if considering a 16-byte block cipher). This method supports parallel GHASH computation over four blocks and leverages the Aggregated Reduction Method, first proposed in [Jankowski and Laurent 2011], where modular reduction by P is deferred and performed only once every few blocks.

Beyond common CPUs, a parallelized GHASH implementation geared towards GPUs [Lee et al. 2025] uses a tree-like structure to compute the tag recursively, in order to avoid computing powers of H in an explicit manner. This method exploits the availability of many threads in a GPU by splitting the message in parts of up to 2^{10} blocks of 16-bytes. Computing GHASH of this single part involves spawning threads down enough to compute partial tags of two 16-byte blocks, then combining them up to the entirety of the submessage.

On the hardware side, implementations leveraging GHASH parallelism have been achieved on FPGAs [Abdellatif et al. 2012, Wang et al. 2010] and ASICs [Hoang et al. 2017, Zhang et al. 2009, Satoh et al. 2009]. Many of these employ several $GF(2^{128})$ multipliers operating in parallel and/or pipeline configurations, often using a Karatsuba-Ofman multiplier.

2. Galois/Counter Mode

Let $H = E_K(0^{128})$ be the *hash subkey*, where E_K is a NIST-approved block cipher under key K with a 128-bit block size, A the additional authenticated data (AAD), and C the ciphertext.

In the first step, the AAD and the ciphertext are padded with zeros so that their lengths are multiples of the block size. Let $u = 128 \cdot \lceil \text{len}(C)/128 \rceil - \text{len}(C)$ and $v = 128 \cdot \lceil \text{len}(A)/128 \rceil - \text{len}(A)$. Define a single message $M = A \parallel 0^v \parallel C \parallel 0^u \parallel [len(A)]_{64} \parallel [len(C)]_{64}$

[1] `https://crypto.stackexchange.com/questions/27466/is-aes-gcm-parallelizable`

and a block $S = \text{GHASH}_H(M) = X_{m+n+1}$, where X_i are 128-bit blocks, m is the number of blocks in A (rounded up), n is the number of blocks in C (rounded up).

Let X_0 be the *zero block*, 0^{128}. Then

$$X_i = (X_{i-1} \oplus M_i) \cdot H \quad \text{for } i = 1, \ldots, m+n+1. \tag{1}$$

The GHASH function computes:

$$\text{GHASH}_H(M) = M_1 \cdot H^{m+n+1} \oplus M_2 \cdot H^{m+n} \oplus \ldots \oplus X_{m+n+1} \cdot H, \tag{2}$$

where the $\oplus$ operation denotes XOR, the $\cdot$ operation denotes multiplication in the Galois Field $GF(2^{128})$, and H^i corresponds to the *i*-th power of H (e.g., $H^3 = H \cdot H \cdot H$).

Stepping back the application of Horner's rule[2], the authentication tag can be computed as a polynomial over a finite field, where coefficient values are blocks of the ciphertext (or associated data), and the variable is the hash subkey H. The degree of the polynomial is equal to the total number of input blocks — including authenticated additional data, ciphertext, padding and length blocks.

For the final calculation of the authentication tag, define $s = 128\lceil len(IV)/128 \rceil - len(IV)$ and a block, J_0, based on the initialization vector (IV) such that:

$$J_0 = \begin{cases} IV \parallel 0^{31} \parallel 1 & \text{if len}(IV) = 96, \\ \text{GHASH}_H(IV \parallel 0^{s+64} \parallel [len(IV)]_{64}) & \text{otherwise.} \end{cases} \tag{3}$$

Finally, the authentication tag is defined as $T = \text{MSB}_t(\text{GCTR}_K(J_0, S)) = \text{MSB}_t(S \oplus E_K(J_0))$, where t is the bit length of the tag, $\text{MSB}_s(X)$ a function that returns the s most significant bits of X and $\text{GCTR}_K(ICB, X)$ a CTR-mode function that starts at a specific initial counter ICB, under key K and processes the input string X.

The GHASH computation step (Equation 2) can be arbitrarily parallelized, as suggested in [McGrew and Viega 2005] and [Gopal et al. 2010]. For example, if we have two running threads (or multipliers), and M's size is 6 blocks, one thread/multiplier can work on even blocks M_2, M_4, M_6, and the other on the odd blocks M_1, M_3, M_5, since:

$$\begin{aligned} &(((((M_1 \cdot H \oplus M_2) \cdot H \oplus M_3) \cdot H \oplus M_4) \cdot H \oplus M_5) \cdot H \oplus M_6) \cdot H \\ =\ &[((M_1 \cdot H^2 \oplus M_3) \cdot H^2 \oplus M_5) \cdot H^2] \oplus [((M_2 \cdot H^2 \oplus M_4) \cdot H^2 \oplus M_6) \cdot H]. \end{aligned}$$

However, in order not to interleave the blocks of M, the parallelization can be done in a transposed form:

$$\begin{aligned} &(((((M_1 \cdot H \oplus M_2) \cdot H \oplus M_3) \cdot H \oplus M_4) \cdot H \oplus M_5) \cdot H \oplus M_6) \cdot H \\ =\ &[((M_1 \cdot H \oplus M_2) \cdot H \oplus M_3) \cdot H^4] \oplus [((M_4 \cdot H \oplus M_5) \cdot H \oplus M_6) \cdot H]. \end{aligned}$$

[2] $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_nx^n = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + x\,a_n(\ldots))))$

3. A Parallel AES-GCM Implementation

We choose AES-256 as our underlying block cipher, using the BearSSL[3] cryptographic library, which provides implementations based on lookup tables, bitslicing, and with specialized AES instructions. For AES in counter mode, we use BearSSL's `br_aes_ct64_ctr_run` to generate the keystream. For the GHASH computation, we rely on BearSSL's `br_ghash_ctmul64` to perform the finite field multiplications in $GF(2^{128})$.

The API of the implementation consists of three endpoints:

- `init`: initializes a context using a 32-byte AES256 key;
- `update`: takes the initialized context and a pointer to the plaintext and encrypts data. This is the main function, which partitions the plaintext into chunks, sets up parallel workers, and processes their outputs (see Algorithm 1);
- `finish`: finalizes the tag computation, handling any residual bytes not yet authenticated.

Algorithm 1 Multithreaded AES-GCM `Update` function

Input: PT, PT_len, ctx
Output: CT, CT_len, tag, ctx

Handle leftover untagged encrypted bytes from last `update` calls
Map:
Compute block count and distribute across threads, launch them
for each thread i **do**
  (inside thread)
  $CT_i, CT\_len_i = \texttt{AES\_CTR}(PT_i, PT\_len_i)$
  $partial\_tag_i = \texttt{GHASH}(CT_i, (CT\_len_i \mod 16), H)$
  Store untagged ciphertext bytes at ctx (last thread only)
end for
Wait for all threads
Reduce:
if PT has full blocks **then**
  $current\_tag \leftarrow 0, exp \leftarrow 0, last\_H\_power \leftarrow 0$
  for each thread i in reverse order **do**
    $H^{exp} = \texttt{h\_power\_inc}(ctx, exp, last\_H\_power)$
    $current\_tag = current\_tag \oplus (H^{exp} \cdot partial\_tag_i)$
    $exp = exp + CT\_len_i$
  end for
  if previous data was encrypted **then**
    $tag = current\_tag \oplus (last\_tag \cdot \texttt{h\_power\_inc}(ctx, exp, last\_H\_power))$
  else
    $tag = current\_tag$
  end if
end if
return success

This implementation does not support non-encrypted authenticated data once our application does not require it. However, support for it can be added by modifying the

[3] `https://bearssl.org/`

`update` function to not encrypt incoming data, handling only the GHASH computation. An additional function called `flip` can be used to signal the encryption context when additional data input is complete; at this point, further `update` calls must begin encrypting data.

To employ parallelism, we leveraged `libpthread`'s threading capabilities. In this aspect, our implementation is largely inspired by the MapReduce paradigm:

- While the original map function processes input data and generates (key, value) tuples, our mapping is simply the division of the input message (the plaintext) into chunks/segments, each of which will be assigned to a thread;
- After mapping is complete, the worker threads are initialized and dispatched. Once completed, all plaintext data provided via the `update` call has been encrypted, and each worker has also computed a partial tag;
- The reduce step consists of "merging" these partial tags by multiplying them with an appropriate power of the hash subkey H.

3.1. Map: thread workload distribution

Dividing the plaintext into equal-sized segments assumes that all worker threads will have identical workloads. However, this approach presents challenges for encryption: firstly, AES operates on 16-byte blocks, so if a segment's length isn't a multiple of 16, it must be padded with null bytes (`0x0`) to meet this requirement; secondly, GHASH also performs operations on encrypted 16-byte blocks: assigning a plaintext segment whose length is not a multiple of 16 to a worker thread can leave trailing encrypted bytes not yet covered by the tag.

In this scenario, it is preferable that only one thread encrypt full blocks plus some trailing bytes, while others process exclusively full, 16-byte blocks. The following procedure achieves this condition: let l be the length of the input and N be the number of available working threads, each one processing W_N bytes:

1. Compute how many full blocks of 16 bytes the message is composed of, excluding padding: $b = \lfloor l/16 \rfloor$;
2. If there are trailing bytes (i.e. $l - b \times 16 > 0$), increase b by 1;
3. Evenly divide b blocks for each of the N worker threads: $W_i = b/N \times 16$, with $i = 0 \ldots N-1$;
4. Distribute the remaining $b \mod N$ blocks to all workers *except to the last one*;
5. Remove a full 16-byte block from processing by the last worker and add the $l - b \times 16$ trailing bytes to W_{N-1}.

As an example, consider an `update` call of $l = 88$ bytes using $N = 4$. Ideally, each thread would encrypt 22 bytes, but we need to adjust the partial message lengths to be multiples of 16. Running the procedure above, we have W_0 and W_1 processing 32 bytes, W_2 16 bytes, and W_3 8 bytes. Note that, even with 8 bytes of payload, W_3 still has a practical workload of encrypting 16 bytes, plus the need to backup the trailing 8 bytes to the context's memory, as those aren't sufficient to run the field multiplication and compute a tag.

When an `update` call results in trailing bytes, subsequent `updates` must check if such backup buffer is empty. If not, the call must encrypt enough bytes to fill the buffer

and update the tag. Then, only if enough bytes are input, we can proceed to distribute the remaining input data to the worker threads.

3.2. Reduce: adding up partial tags

The intermediate tags derived from each ciphertext segment can be combined into a single authentication tag by multiplying them with an appropriate power of the hash subkey H and applying the XOR operation for each byte of the product.

In our implementation, up to 32 powers of H are precomputed during the encryption `init` function and stored in the context as the set $\{H, H^2, H^4, H^8, \ldots, H^{2^{31}}\}$. During the `update` phase, we employ a square-and-multiply strategy to compute arbitrary powers of H using these precomputed values, improving efficiency. For inputs exceeding 2 GiB – beyond the range covered by the cached powers – the algorithm falls back to a pure square-and-multiply computation without any table lookups.

Furthermore, the power computation function (`h_power_inc`, see Algorithm 1) is incremental: it reuses previously computed powers to reduce redundant operations, computing new values as $H^e = H^{e-\texttt{last\_e}} \cdot H^{\texttt{last\_e}}$. This optimization minimizes the number of field multiplications, particularly when successive calls require increasing exponents, which is our case.

Following Equation 2, let b_N be the number of blocks processed by a worker thread w_N, and $b = \lceil (m+n)/16 \rceil$ the total number of blocks encrypted by a single `update` call. The tag for this `update` call, based on the partial tags t_i, can be computed as

$$\text{tag} = \text{tag} \cdot H^b \oplus t_0 \cdot H^{b-b_{N-1}} \oplus t_1 \cdot H^{b-(b_{N-1}+b_{N-2})} \oplus \ldots \oplus t_{N-1} \cdot H^0. \tag{4}$$

While each factor of this sum could also be computed in each thread, resulting in better parallelism, one can also employ a tree-based sum of those factors to further increase performance. Since the number of parallel threads may be low, the complexity of running such algorithm for merge may be not justified.

4. Results

Experiments were conducted on a laptop running Ubuntu 20.04 equipped with an Intel Core i7-10510U processor (4 cores, 8 threads, base frequency 1.80 GHz, max turbo 4.90 GHz). The CPU governor was set to performance and benchmarks were executed using `clock_gettime` with the `CLOCK_MONOTONIC` source. All measurements were performed five times and averaged.

To establish a fair baseline, we developed a serial version of our implementation using the same BearSSL cryptographic primitives as the parallel version. Isolated benchmarks of BearSSL used functions indicate a throughput of approximately 624 MiB/s for `br_aes_ct64_ctr_run` and 3030 MiB/s for `br_ghash_ctmul64`.

Figure 1 shows the throughput in Mbps achieved by our parallel GHASH implementation across varying input sizes and thread counts. As shown, our implementation begins to outperform the serial baseline at approximately 8 KiB for configurations using 4 or more threads. Table 1 summarizes representative performance values for small, mid-sized, and large messages, comparing the serial version to the 4-thread and 8-thread configurations.

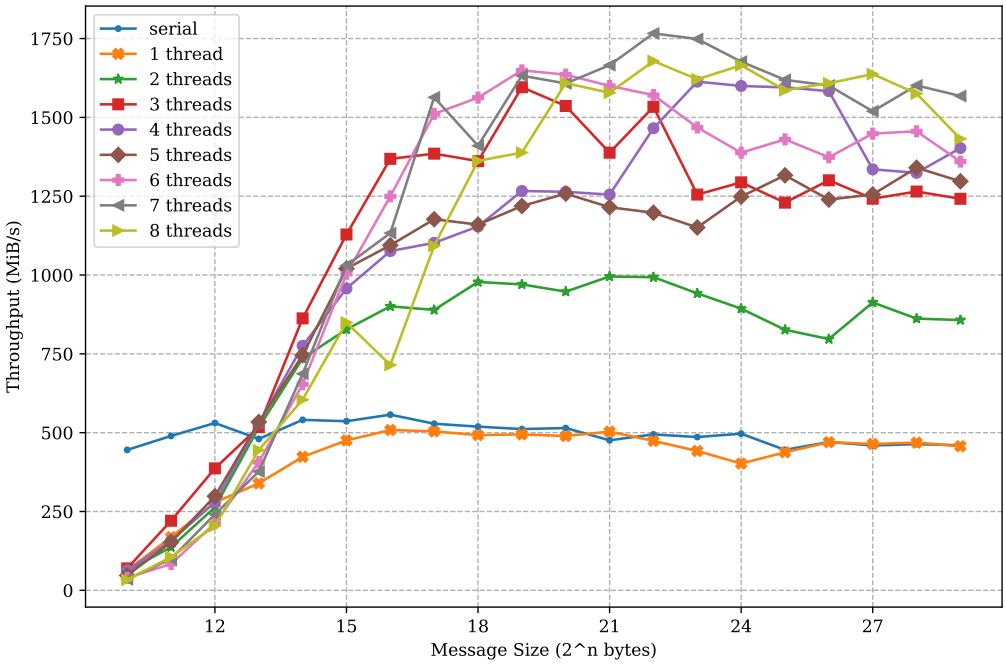

Figure 1. Throughput in MiB/s (vertical) for different message sizes (horizontal, log scale) and thread counts.

Table 1. Throughput (MiB/s) for selected input sizes using serial, 4-thread, and 8-thread configurations.

Bytes	Serial	4 threads	8 threads
1024	445.43	63.18	33.48
8192	480.18	529.05	444.88
8388608	486.17	1613.31	1620.74

5. Conclusion

This paper presented a little twist on GHASH parallelization: in a MapReduce style, parallelism is achieved by partitioning the plaintext into smaller chunks and joining the results in a straightforward manner. However, such reduction incurs the cost of computing powers of the hash subkey H, which requires multiple finite field operations. The overhead becomes particularly noticeable for smaller messages, as shown in Table 1.

Experimental results on a multi-core Intel platform show that our parallel implementation significantly outperforms the serial baseline for inputs larger than 8 KiB, achieving speedups of over $3\times$ for large messages using more than 4 threads. To the best of our knowledge, no freely available software implementation using this parallelization strategy exists; prior related work is mostly found in hardware-focused proposals.

Future work includes comparing this parallelization model with the traditional hardware-oriented approach suggested by [McGrew and Viega 2005], adding support for AAD, and leveraging the use of a thread pool persistent between `update` calls, as well as specialized instructions for both x86 and ARM architectures. Some improvements could also be made in the exponentiation strategy for powers of H, akin to [Lee et al. 2025].

5.1. Acknowledgments

The authors acknowledge the financial support granted by the Brazilian Ministry of Science, Technology and Innovation (MCTI), the Brazilian Space Agency (AEB) and the Financiadora de Estudos e Projetos (FINEP), under contract number 03.23.0113.00. Generative AI was used to compose this paper to improve clarity.

References

Abdellatif, K. M., Chotin-Avot, R., and Mehrez, H. (2012). Efficient parallel-pipelined GHASH for message authentication. In *2012 International Conference on Reconfigurable Computing and FPGAs*, page 1–6. IEEE.

Akdemir, K., Dixon, M., Feghali, W., Fay, P., Gopal, V., Guilford, J., Ozturk, E., Wolrich, G., and Zohar, R. (2010). Breakthrough AES Performance with Intel AES New Instructions. Technical report, Intel.

Dworkin, M. (2007). Recommendation for block cipher modes of operation: Galois/-Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards and Technology.

Foor, D. (2025). Pre-draft Public Comments on SP 800-38D Rev. 1. `https://csrc.nist.gov/files/pubs/sp/800/38/d/r1/upd/iprd/docs/sp800-38d-pre-draft-public-comments.pdf`.

Gopal, V., Ozturk, E., Feghali, W., Guilford, J., Wolrich, G., and Dixon, M. (2010). Optimized Galois-Counter-Mode Implementation on Intel Architecture Processors. Technical report, Intel.

Gouvêa, C. P. L. and López, J. (2015). Implementing GCM on ARMv8. In Nyberg, K., editor, *Topics in Cryptology — CT-RSA 2015*, pages 167–180, Cham. Springer International Publishing.

Gueron, S. (2023). Constructions based on the AES round and polynomial multiplications that are efficient on modern processor architectures. In *Third NIST Workshop on Block Cipher Modes of Operation*.

Gueron, S. and Kounavis, M. E. (2010). Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode. Technical report, Intel.

Hoang, V.-P., Nguyen, V.-T., Nguyen, A.-T., and Pham, C.-K. (2017). A low power AES-GCM authenticated encryption core in 65nm SOTB CMOS process. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 112–115.

Jankowski, K. and Laurent, P. (2011). Packed AES-GCM Algorithm Suitable for AES/P-CLMULQDQ Instructions. *IEEE Transactions on Computers*, 60(1):135–138.

Kohno, T. (2003). The CWC-AES Dual-Use Mode. Internet-Draft draft-irtf-cfrg-cwc-01, Internet Engineering Task Force.

Lee, J., Kim, D., and Seo, S. C. (2025). Parallel implementation of GCM on GPUs. *ICT Express*, 11(2):310–316.

McGrew, D. A. and Viega, J. (2005). The Galois/Counter Mode of Operation (GCM). Technical report, Cisco Systems Inc. and Secure Software.

Satoh, A., Sugawara, T., and Aoki, T. (2009). High-Performance Hardware Architectures for Galois Counter Mode. *IEEE Trans. Computers*, 58:917–930.

Wang, J., Shou, G., Hu, Y., and Guo, Z. (2010). High-speed architectures for GHASH based on efficient bit-parallel multipliers. In *2010 IEEE International Conference on Wireless Communications, Networking and Information Security*, pages 582–586.

Wi-Fi Alliance (2025). WPA3 Specification. `https://www.wi-fi.org/system/files/WPA3%20Specification%20v3.3.pdf`.

Zhang, C., Li, L., Xu, J., and Wang, Z. (2009). High-throughput GCM VLSI architecture for IEEE 802.1ae applications. In *2009 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 900–903.