AUTHENTICATED ENCRYPTION: A COMPREHENSIVE BENCHMARK

Philipp Engljähringer, Michael Kürsteiner, Patrick Muntwiler, Johan Calle, Jakub A. Trzykowski

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In our study, we developed a benchmarking suite to easily compare the performance of multiple encryption, hashing, and authenticated encryption algorithms. Additionally, we created a tool for simple parallelization of authenticated encryption methods. Lastly, we measured the performance of various implementations and analysed their effectiveness.

1. INTRODUCTION

In today's world, the need for secure communication over the internet is gaining more relevance. A growing number of industries require services that rely on cloud infrastructures or tools that send data over insecure network channels. To make this process more secure, one can make use of cryptographic primitives, such as Authenticated Encryption with Associated Data (AEAD) [1].

Motivation. As the size of data exchanged increases, so does the demand for high performance algorithms enabling secure communication. Latency resulting from processing incoming packets over a connection is critical for performance. This is especially important in scenarios where large chunks of data need to be sent and processed timely and securely. There, encryption, decryption and authentication can become a serious bottleneck and negatively impact the performance of an application.

In order to improve performance of existing AEAD implementations, one needs to understand where their bottlenecks lay. To facilitate this, we created a tool that enables easy measurement of execution times of various AEAD algorithms and their isolated parts - encryption, decryption and authentication. The tool delivers a rudimentary set of plots to compare different implementations and gives an initial feedback, and raw measurements for more in-depth analysis. Furthermore, we provide functionality that allows to parallelize any AEAD implementation and measure the benefit of doing.

Related work. A lot of work in the area comprises benchmarks that investigate performance in specific scenarios, such as in embedded systems or in IoT. Most work is usually done on smaller data sizes, far below 1 GiB, and focuses on additional metrics that involve the level of security provided by the cryptographic algorithm [2, 3, 4]. Some work has been done in relation with cloud-based applications that observe large work loads comparable to the ones we use for our benchmarks [5]. Nevertheless, there has been little research that analyses the performance gained from parallelism. Additionally, none of these aforementioned papers analyse the components of AEAD algorithms in isolation, nor do they investigate new combinations that could result in potentially faster implementations. Our work runs benchmarks on a variety of AEAD implementations from popular libraries with different thread counts and block sizes, and analyses a novel AEAD implementation made out of a combination of enc/decryption and hashing components.

2. BACKGROUND: AUTHENTICATED ENCRYPTION

In this section we provide an introduction to Authenticated Encryption with Associated Data (AEAD) including its components and give a brief overview of how they work.

Encryption and Decryption. The term encryption refers to the process of altering a plaintext message to ensure confidentiality. Encryption has to be reversible in order to retrieve the original information, which is known as decryption. In general, the process computes the obfuscated information with what is known as a key, and only the parties with the right key should be able to read the message. Symmetric-key encryption uses the same key for both encryption and decryption.

An encryption algorithm may operate in different ways: if it operates on a single bit at a time it is called a stream cipher, whereas if it operates on a block of data it is called a block cipher [6].

The Advanced Encryption Standard (AES) is the most widespread encryption standard due to its security and high performance (including AES-specific hardware instructions) [7].

Hash function. A general hash function takes an arbitrary-length message and returns a fixed-length value.

As opposed to encryption, hashing should be irreversible. Hash functions should produce uniformly distributed values, should be deterministic, and the value returned should vary greatly when the message is modified [8].

Hash functions can be extended to satisfy certain cryptographic properties. Most importantly, a cryptographic hash function should be collision resistant (two different inputs should be unlikely to map to the same output) and preimage resistant (given the output, it should not be possible to determine the input) [6].

Authenticated Encryption (AE). AE simultaneously ensures confidentiality and authenticity of the transmitted data [9]. Confidentiality is provided by an encryption algorithm, and authenticity is provided in the form of a Message Authentication Code (MAC). The MAC is produced by a hash function which takes an initialization vector and either the plaintext message or the encrypted ciphertext, and produces a unique value, which can be compared between the sender and receiver to prove authenticity. Some AE schemes allow Associated Data (AEAD), which is not encrypted but is checked for authenticity using the MAC.

3. AEAD BENCHMARKING TOOL

To easily compare different implementations for AEAD, encryption or hashing, we developed a benchmarking harness. The purpose of this benchmarking harness is to allow implementations from different sources, potentially written in different languages, to all be benchmarked the same way, with minimal effort. Any language that can export a function with C ABI from a shared library can be benchmarked by our tool.

Implementation. The benchmarking harness is written in the Rust programming language, and it uses Criterion[1], a tool for creating benchmarks. The benchmark allocates 64 byte-aligned (cache alignment) buffers with pseudo-random data The tool also creates arrays for a key and an initialization vector (IV). By pre-providing these, we could exclude the time taken to generate the key and IV, since this time is not our focus of the benchmark.

After generating all the required data, the tool loads the functions to be benchmarked from shared libraries in a specified directory.

Our tool passes the function to be benchmarked to Criterion, which starts by doing a 5 second warm-up. Afterwards, the execution time of the function is sampled 100 times with one iteration of the function per sample.

Benchmarking modes. Several modes of benchmarking can be chosen, such as: performance with default settings, performance with different thread counts, and if applicable, different block sizes used by an implementation.

[1]Criterion - https://github.com/bheisler/criterion.rs

If a shared library exports a function to set the number of threads used by the AEAD function, then the benchmark can call this to compare different thread counts. For our benchmarks, we tested all thread counts up to the number of logical cores of the system.

Some implementations split the input data into blocks, and the size of these blocks can influence performance drastically. To find an optimal block size for a specific implementation, the shared library can export a function to set a new block size. Our benchmarking tool will then call this function to compare different block sizes. We tested block sizes equal to powers of two ranging from 1024 bytes to the size of the input, at a fixed thread count. The thread count was chosen to be half of the amount of logical cores of the system, since there was little scaling after that point for most implementations.

Function signatures. All of our functions utilize an input and an output buffer, which are large enough to not cause any out-of-bound accesses (e.g., 1 GiB + MAC). We ensure alignment to 64 bytes to prevent any potential performance differences from varying alignments. Since the buffers are non-overlapping, the pointers can be annotated with 'restrict'. Some encryption implementations provide a way to encrypt and decrypt data in-place, but we did not benchmark this operating mode.

All our function signatures take the input buffer, output buffer, and a size parameter denoting the number of bytes in the input. The functions also take two byte buffers containing data to be used as a key and an IV. Functions that output (or require) a MAC receive an additional output (or input) buffer with appropriate size.

All of these functions must be exported using the C ABI, i.e., by using 'extern "C"' in C++ or Rust and with name mangling disabled. For the encryption function of an AEAD implementation, the function signature looks like this (the other ones are very similar):

```
size_t encrypt(
  const uint8_t * restrict in,
        uint8_t * restrict out,
        uint8_t * restrict mac,
  const uint8_t * restrict key,
  const uint8_t * restrict iv,
  const size_t             bytes);
```

We decided to not include data that is only authenticated, but not encrypted (the "associated data" part of AEAD). Adding such data to the benchmark would not be too complicated, but it would introduce another size parameter. All in all, it should not have a significant effect on the overall performance, as AD is usually much smaller than the data. All our benchmarks were done with 1 GiB of encrypted and authenticated data, and no associated data.

Build system. To enable easy benchmarking, we created a Makefile[2] containing the required compilation flags. Each implementation has a Makefile that include the base Makefile, and uses the predefined variables from there. The Makefile then performs whatever steps necessary to create a shared library at the expected location. This allows our Rust build script to automatically compile each algorithm with the appropriate target architecture and compiler flags.

Comparing existing implementations. To decide on the single threaded encryption and hashing implementation to be used for our parallel AEAD, we first had to compare the provided functions by different existing implementations. We compared implementations from OpenSSL[3], Crypto++[4], RustCrypto[5] and some separate implementations like ChaCha20 or Blake3[6]. In the end we decided to AES_256_GCM from OpenSSL, Crypto++ and RustCrypto, and we tried to make some implementations of our own using the fastest existing encryption and hashing implementations. To ensure fairness between the 3 implementations, we chose the same settings for all of them: 256 bit (32 byte) AES, 16 byte GHash, 32 byte key, 12 byte IV.

4. PARSPLIT: MAKE ANY AEAD PARALLEL

We developed a way to parallelize any single-threaded AEAD implementation by splitting the input data into blocks, then dealing with each block in parallel.

Parallelizing an AEAD implementation requires parallelization of its two components: the encryption and the hashing. There are multiple ways of approaching parallelizing the encryption part. For a block cipher, the input is split into small fixed size blocks (e.g., 256 bits) and encrypted block-by-block. It might be possible to parallelize an algorithm within each block, but this requires knowledge about the inner workings of the block cipher. Years of work into optimizing these block cipher and hardware support, as well as the inherent limitation on maximal parallelism from the fixed block size, convinced us to look elsewhere for parallelization opportunities. Another approach would be to encrypt each of the fixed size blocks in parallel. This does work for some block cipher modes like Galois/counter mode (GCM), since each block there is independent. For other modes, encryption of a block depends on previous blocks, making it hard or impossible to parallelize. Finally, the encryption can be parallelized by splitting the entire data array into larger blocks (e.g., 16 KiB) and treating each as a separate input to the encryption algorithm. This method works for any block cipher mode, since each of the larger blocks is encrypted completely independently of the others.

[2]Make - `https://www.gnu.org/software/make/`
[3]OpenSSL - `https://www.openssl.org/`
[4]Crypto++ - `https://github.com/weidai11/cryptopp`
[5]RustCrypto - `https://github.com/RustCrypto`
[6]Blake3 - `https://github.com/BLAKE3-team/BLAKE3`

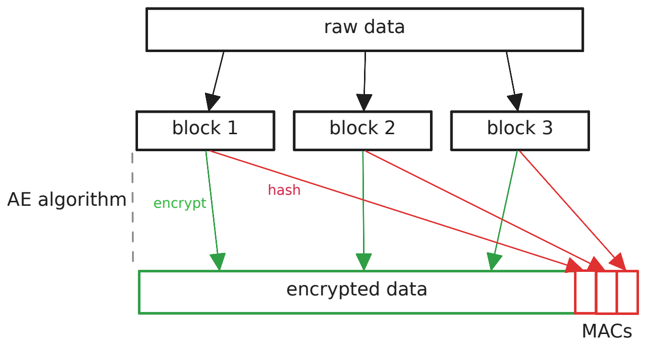

Fig. 1. Diagram representing encryption process used by ParSplit. Decryption is symmetric to it.

For the hashing part of the AEAD, there are also some issues caused by dependency chains. The hashing operation performed on blocks of input is often not associative or commutative, so these blocks cannot be handled out of order. Some hashing algorithms such as Blake3 already provide a parallel implementation, which could be combined with a parallel encryption scheme to create a parallel AEAD. The method of splitting the input into larger blocks discussed for encryption also works for hashing, by treating each block as a separate input to a hashing algorithm.

We decided on pursuing the last method, by splitting the entire input into fixed size blocks and handling each of them independently. Since there is no change or even knowledge of the inner workings of any AEAD implementation required, we can benefit of years of optimization by using the fastest existing single-threaded implementation with our method.

Treating each block as a separate input will lead to one MAC getting generated per block, which leaves the question of how to handle those. The simplest method is to concatenate all these separate hashes together, leading to one large hash. The drawback of this is that the MAC is no longer a fixed size, but dependent on the input, leading to an O(n) space overhead. Appending all the encrypted blocks and the concatenated hash leads to an algorithm like in figure 1.

Decryption is handled almost identically, the block is decrypted by itself and the MAC is compared against the corresponding value in the block of concatenated hashes.

For a 16 KiB block size and a 32 byte hash, the overhead for the hash in relation to the input size is 0.1953125% (1 / 512). Large block sizes or shorter hash values will reduce the size overhead. If this overhead is unacceptable, then there are two possibilities: The concatenated hashes can be hashed in the end into one value as an additional step, or the hashes can be merged in parallel using a Merkle tree [10].

In a Merkle tree, the hashes are repeatedly concatenated pairwise and then hashed, following the structure of a binary tree. At the root of this tree, only 1 hash value will remain. This procedure can be done in parallel, and could be integrated into our ParSplit method.

Our attempts to write an efficient parallel implementation of such a Merkle tree were sadly unsuccessful, so we decided to store all hashes concatenated together as one big block instead. We discovered towards the end of the project, that Blake3 internally uses as Merkle tree, which enables a parallelized implementation. Studying this implementation could help in designing an efficient Merkle tree for our ParSplit method.

Our implementation of ParSplit was done in Rust, using the Rayon[7] library. Rayon provides ways to easily parallelize loops, similar to OpenMP. ParSplit statically links to a single-threaded AEAD implementations, making use of the common function signatures we already use for the shared libraries.

Due to our design choices for ParSplit and the benchmarking tool, adding new implementations was quite easy, which lead to us adding a lot of different implementations to compare. In the end we had 9 versions of the ParSplit, 13 single threaded AEADs, 24 encryption methods (mostly versions of AES) and 22 hashing methods (mostly versions of SHA, Blake, GHash). Most of them were a lot slower in our initial benchmarks compared to the implementations we present in this report. Plots with the result of some of the other implementations can be found in the appendix.

5. EXPERIMENTAL RESULTS

Experimental setup. We ran our benchmarks on two machines, with 24 and 64 logical cores, respectively. We will focus on the data from the 64 core CPU, and put the results from the 24 core CPU into the appendix.

Main benchmark environment:

Our primary benchmarking system consisted of an AMD EPYC 7742 64-Core Processor at 3.3 GHz (no boost) with 512 KB cache and 512 GB memory. The C and C++ code for the benchmarks was compiled with GCC (version 8.4.1 20200928) with these flags: `-o3 -march=native -mavx -mavx2 -mfma`. All Rust code was compiled with rustc (version 1.74.1 (a28077b28 2023-12-04)), with these compilation flags: `opt-level=3 lto="fat" target-cpu=native`. The target OS was Rocky Linux 8.4. The libraries for the cryptographic operations had the following versions: OpenSSL 3.2.0 (23 Nov 2023), Crypto++ 8.9.0, RustCrypto aes-gcm 0.10.3, Blake3 1.5.0 The parallelization was done with Rayon version 1.8.0, and the benchmark used Criterion-rs 0.5.1.

Results. We split our findings into 4 categories: Single-threaded AEAD encryption and hashes, multithreaded AEAD encryption, speed-up of AEAD encryption and multithreaded AEAD encryption for different block sizes. In order to provide a compact overview, we do not include the plots for the AEAD decryption experiments. The results from these benchmarks are almost identical to the results of their respective encryption counterparts and do not yield any new interesting insights for the focus of this report. We used a simple algorithm that takes the XOR of the data with a key as our performance baseline for comparison.

Single-Threaded AEAD Encryption and Hashes. The goal of Fig. 2 and 3 is to compare the execution times of the fastest AEAD encryption and hash implementations, obtain a baseline to calculate the overhead produced by our ParSplit implementations and show the proportions that are taken up by the three components (encryption, decryption and authentication) needed in AEAD. The data depicts the median of 100 samples per algorithm. Additionally, we add the execution time of the ParSplit implemented version of the algorithm with one thread in order to visualise the overhead produced by our parallelization approach. Between samples, the variability of execution times is so low that we decided to exclude this information (i.e. box plots with quantiles or confidence intervals). OpenSSL AES GCM has the fastest AEAD encryption and decryption with 310 ms and 306 ms as well as the fastest hashing implementation with 144 ms.

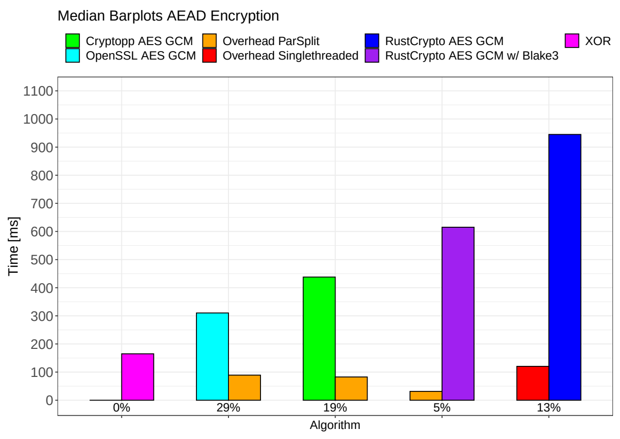

Fig. 2. Median-based execution times of AEAD encryption implementations. Next to every bar is the overhead needed to run the same algorithm modified with our ParSplit approach, with a thread count of 1. This is not the case for RustCrypto AES GCM, here our ParSplit approach with 1 thread is faster than the native implementation. Below the bars are the overheads represented as percentage share.

Multi-Threaded AEAD Encryption. In Fig. 4 we show the absolute execution times of the fastest multi-threaded ParSplit implementations. Fig. 4 will help us to give the speed-ups in Fig. 5 more meaning and depicts

[7]Rayon - `https://github.com/rayon-rs/rayon`

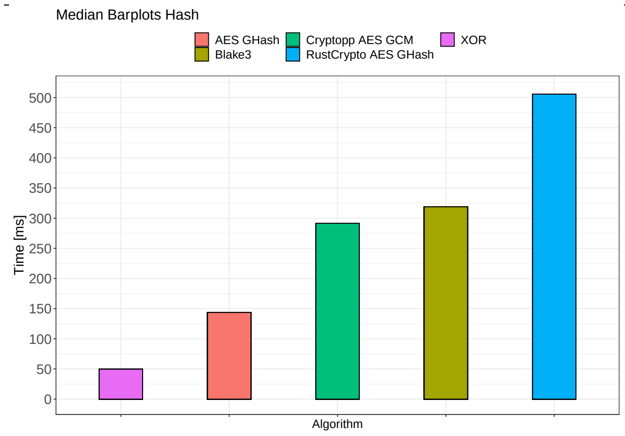

Fig. 3. Median-based execution times of hash implementations.

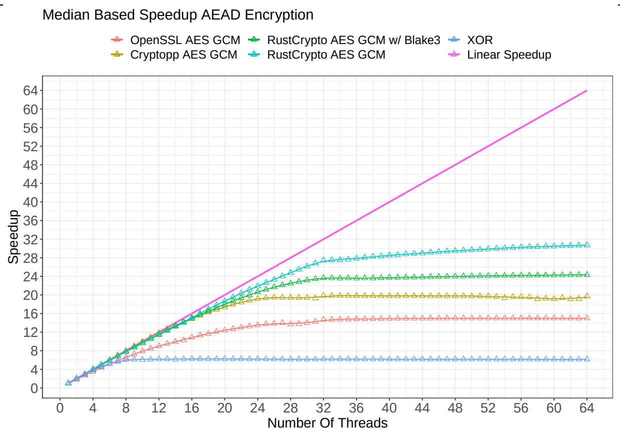

Fig. 5. Speed-ups of the tested multithreaded AEAD encryption implementations.

the benefits of higher thread counts as well as showing the fastest implementations at specific thread counts. Note that the tests with thread count 1 are using the ParSplit implemented version of the corresponding algorithm with just one thread. For every thread count and every implementation, we collected 100 samples. Crypto++ AES GCM is the fastest algorithm reaching its peak at about 28 threads with 26.83 ms for its AEAD encryption and 27.47 ms for its AEAD decryption.

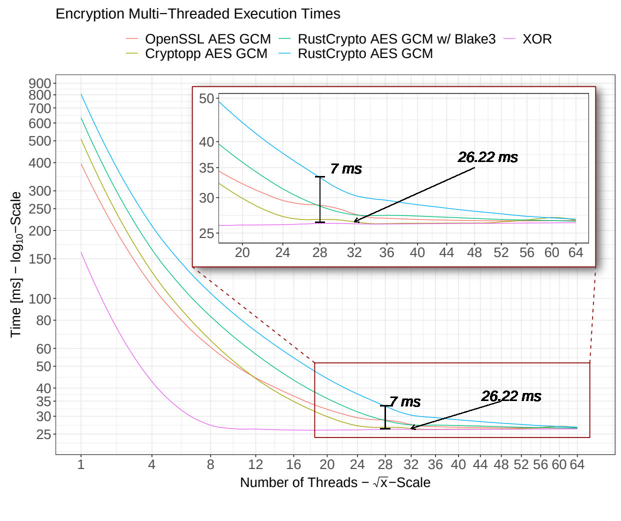

Fig. 4. Execution times of our multithreaded AEAD encryption benchmarks. The x-axis and the y-axis are square root and $\log_{10}$ scaled respectively to improve visibility of the data.

AEAD Encryption Speedup. Next, we show achieved speed-ups from our multithreaded AEAD encryption benchmarks in 5. This gives us insights which algorithms benefit the most from ParSplit parallelization. Speed-ups were calculated based on the median execution times out of a 100 runs per algorithm and thread count. The theoretically maximally achievable linear speed-up serves as a baseline for comparison. The most speed-up is achieved by RustCrypto AES GCM. However, its native implementation is the worst optimized one, hence it profits the most.

Multi-Threaded AEAD Encryption for Different Block Sizes. Lastly, Figure 5 shows the benefit of choosing the right block size at a fixed thread count of 32 for every benchmark we ran. Like in our other experiments, we ran a 100 samples per block size and per algorithm. The best block sizes are in the range from 8 to 32 KiB.

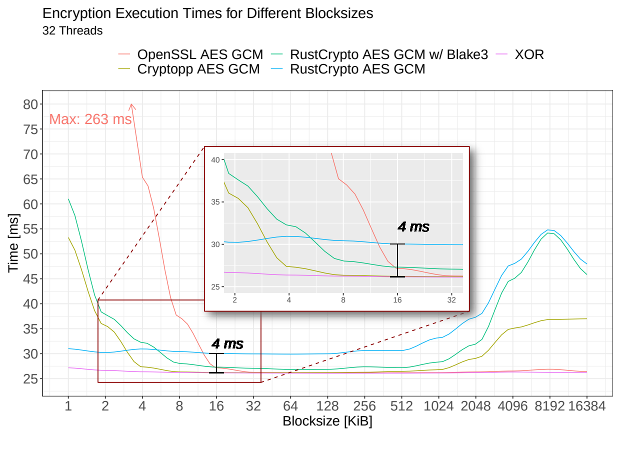

Fig. 6. Execution times of our multithreaded AEAD encryption benchmarks for varying block sizes and with 32 threads.

6. CONCLUSIONS

To put it all into perspective, we have created a flexible and extensible benchmarking suite for cryptographic algorithms such as enc/decryption, MACs, or Authenticated Encryption. Using it, we have performed a variety of experiments to determine which algorithms (and implementations) perform well on large data samples. Our results can aid programmers and security analysts in choosing the most applicable solution for their needs. Furthermore, we will now highlight the bottlenecks in the contemporary implementations, which can subsequently shape new research into how to overcome those.

Encryption, Decryption. Hashing. AES 256 GCM turned out to have the fastest encryption, decryption and hashing implementations in the single threaded setting. Encryption and decryption took 310.22 ms and 306 ms, Hashing took 144 ms for 1 Gib. We account this to the fact, that OpenSSL is known to make better use of specialized instructions for Hardware Acceleration and CPU cache optimizations, it provides low-level assembly implementations for cryptographic algorithms and specific architectures and the GCM algorithm itself is designed to be relatively efficient compared to other modes of operation. In addition, GCM also allows efficient parallelization due to its algorithmic structure [11].

In the parallel setting AES 256 GCM is slightly faster than Crypto++ AES GCM, however after 12 threads Crypto++ AES GCM takes the lead by a couple of ms and reaches its peak at about 28 threads with a total execution time for AEAD encryption of 26.83 ms and for AEAD decryption of 27.42 ms. It is worth mentioning the official implementation of Blake3 provides a parallel version, which yielded very impressive performance. It could hash the data in 13.05 ms, corresponding to a throughput of 76.92 GiB per second. This is due to the inherent Merkle Tree structure of the algorithm, that enables efficient hashing in a multithreaded setting. However, one needs to put in more effort in order to combine it with other components of AEAD together with our ParSplit solution, but nevertheless, it is a promising candidate to improve performance of existing AEAD implementations.

AEAD. As we have seen, all implementations used by our ParSplit are eventually bounded by memory accesses (flattening of the speed-up curve in 5). This largely depends on the system's memory bandwidth. With our implementation, this happened with about 8 threads for the trivial XOR algorithm, and with about 28 for the next fastest algorithm (AES_256_GCM from OpenSSL).

It is worth mentioning that for all, switching to ParSplit resulted in variable amount of overhead in single threaded mode. This was negligible for XOR, around 28.8% for OpenSSL out of a total execution time of 310.22 ms, 18.9% Crypto++ out of 437.94 ms, and around 5.1% for RustCrypto AES GCM with Blake3 out of 615.11 ms. Note that we roughly observed -13% overhead for the RustCrypto AES GCM ParSplit implementation out of 945.95 ms on our primary benchmark platform. We couldn't reproduce this on our secondary benchmark platform. Our hypothesis is, that the compiler might have found an optimization for this specific implementation but not for the other ParSplit implementations with the hardware used on the primary benchmark platform.

Since the implementation of GHash in RustCrypto was slower than Blake3, we created a version of the RustCrypto AES_256_GCM where we replaced GHash with Blake3. This change sped up this implementation from taking 944.95 ms to only taking 615.11 ms (1.6257 GiB per second). This change would very likely not yield any benefits in the OpenSSL implementation, since their version of GHash is faster than Blake3 already.

Future ideas. An opportunity we have identified is to consider using in-memory computing [12]. Especially the symmetric encryption should be a good candidate, as these usually consist of relatively simple operations such as XORs, shifts and shuffles. A similar thing is already being done with AES, which has dedicated instructions, however, on the CPU level. We believe that systems that need to process large amounts of encrypted data should consider this a worthwhile research.

On the other hand, if the authenticated encryption is going to primarily involve network communication, it might be worthwhile to offload the work to the networking hardware.

As for our benchmarks, more prospective work can be done by combining the GHash of OpenSSL with Crypto++ AES GCM and the multithreaded Blake3 implementation with the OpenSSL AES GCM and the Crypto++ AES GCM implementations together with our multithreaded ParSplit approach. Here it would be interesting to see, how the parallel implementation of Blake3 could be combined in an efficient manner with our ParSplit solution.

7. REFERENCES

[1] Phillip Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2002, CCS '02, p. 98–107, Association for Computing Machinery.

[2] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel, "A survey of lightweight-cryptography implementations," *IEEE Design Test of Computers*, vol. 24, no. 6, pp. 522–533, 2007.

[3] Heiko Bühler, Andreas Walz, and Axel Sikora, "Benchmarking of symmetric cryptographic algorithms on a deeply embedded system," *IFAC-PapersOnLine*, vol. 55, no. 4, pp. 266–271, 2022, 17th IFAC Conference on Programmable Devices and Embedded Systems PDES 2022 — Sarajevo, Bosnia and Herzegovina, 17-19 May 2022.

[4] Soline Blanc, Abdelkader Lahmadi, Kévin Le Gouguec, Marine Minier, and Lama Sleem, "Benchmarking of lightweight cryptographic algorithms for wireless iot networks," *Wireless Networks*, vol. 28, no. 8, pp. 3453–3476, Nov 2022.

[5] Stefan Contiu, Emmanuel Leblond, and Laurent Réveillère, "Benchmarking cryptographic schemes for securing public cloud storages," in *Distributed Applications and Interoperable Systems*, Lydia Y. Chen and Hans P. Reiser, Eds., Cham, 2017, pp. 163–176, Springer International Publishing.

[6] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 5th edition, 2001.

[7] Morris J Dworkin, *Advanced Encryption Standard (AES)*, May 2023.

[8] Urs Wagner and Thomas Lugrin, *Hash Functions*, p. 21–24, Springer Nature Switzerland, 2023.

[9] J. Black, *Authenticated encryption*, p. 11–21, Springer US.

[10] Ralph C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*, p. 369–378, Springer Berlin Heidelberg, 1988.

[11] David Mcgrew and John Viega, "The galois/counter mode of operation (gcm)," 02 2004.

[12] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

Appendices

A. ADDITIONAL PERFORMANCE PLOTS

Results shown here were measured on the same 64 core system as presented in the report.

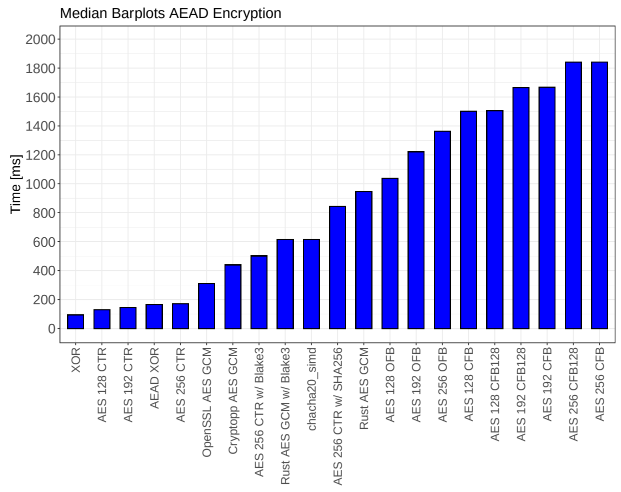

Fig. 7. Execution times of all encryption implementations.

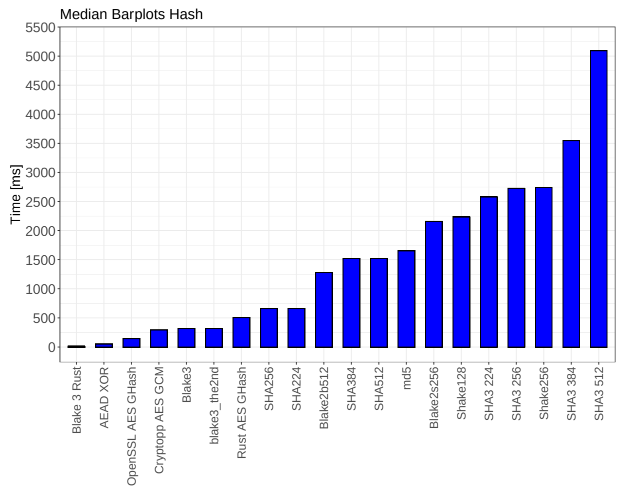

Fig. 8. Execution times of all hash implementations.

B. SECOND BENCHMARK PLATFORM

Environment. Here we will place data from our secondary benchmark environment, which consists of an AMD Ryzen 9 5900X 12-Core Processor at 4.2 GHz (no boost) with a Cache size of 512 KB and 32 GB Memory. The C and C++ code for the benchmarks was compiled with GCC (version 11.4.0) with these flags: `-o3 -march=native -mavx -mavx2 -mfma`. All Rust code was compiled with rustc (version 1.74.1 (a28077b28 2023-12-04)), with these compilation flags: `opt-level=3 lto="fat" target-cpu=native`. The target OS was Ubuntu 23.10.1. The libraries for the cryptographic operations had the following versions: OpenSSL 3.2.0 (23 Nov 2023), Crypto++ 8.9.0, RustCrypto aes-gcm 0.10.3, Blake3 1.5.0 The parallelization was done with Rayon version 1.8.0, and the benchmark used Criterion-rs 0.5.1.

Performance data.

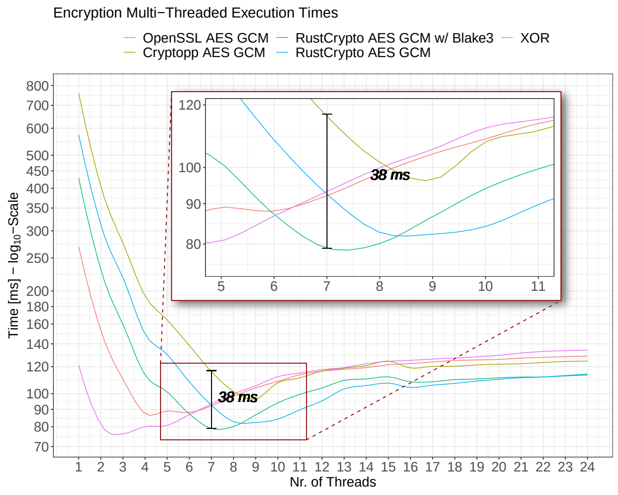

Fig. 9. Execution times for AEAD encryption implementations on second benchmark platform.

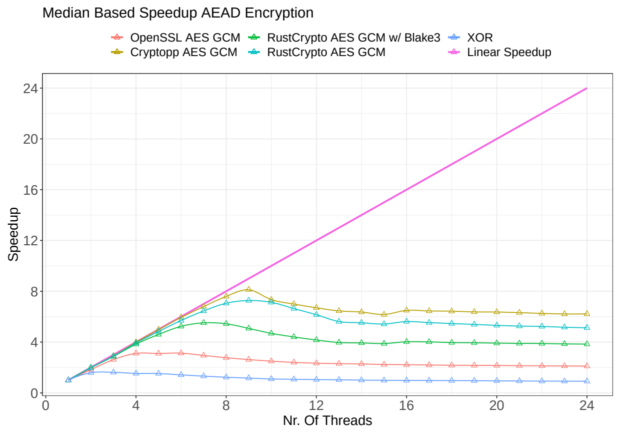

Fig. 10. Speedups of AEAD encryption on second benchmark platform.

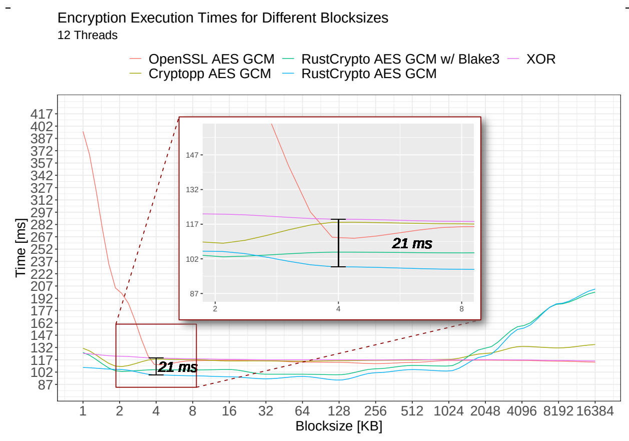

Fig. 11. Execution times for different blocksizes with 12 threads on second benchmark platform.

C. PERFORMANCE ODDITY WITH SPECIFIC THREAD COUNTS

On both of our benchmarking platforms, we detected an interesting performance oddity when running with specific thread counts, most of them an odd number of threads. For these thread counts, the data clearly showed two specific modes in the runtime results, while for other thread counts there was only every one mode.

As one example we can take AEAD encryption with ParSplit with Crypto++ AES_GCM on our 64 core machine, using 5 threads:

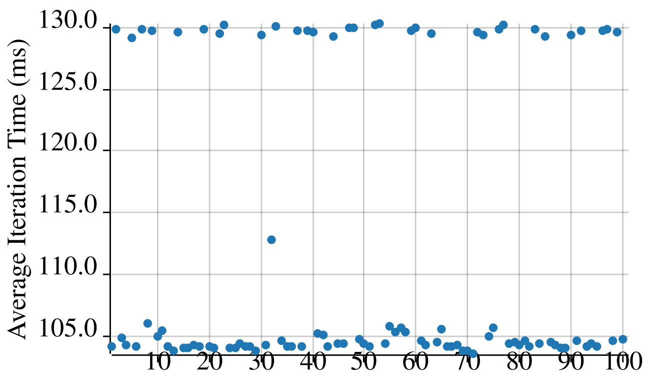

Fig. 12. Execution times of 100 samples with 5 threads running ParSplit with Crypto++ AES_GCM

As can be clearly seen, there are two distinct modes in the execution time of the benchmark. Ignoring the one outlier in the middle, the upper mode has a median time of 129.72 ms with 32 of 100 samples, while the lower mode has 104.342 ms with 67 samples. This corresponds to a factor of 1.243 in the execution time between the two modes.

The same behavior was detected on our 24 core benchmarking machine, and with many different thread counts such as 3, 5, 6, 7, 9, 10. This behavior was still present for higher thread counts, but almost all samples there were in the lower mode.

The 24 core CPU we used (AMD R9 5900X) consists of 2 chiplets with 6 cores each, all with 2 logical cores each. We theorized that this behaviour might be caused by one of these, but disabling one chiplet, hyperthreading, or both, did not change this behaviour. Any boosting behaviour of this CPU was also disabled, and the frequency was locked to 4.2 GHz. In any case, the frequency would have to be either drop to 3.4 GHz or increase to 5.2 GHz during the benchmark to explain a difference with factor 1.24, which does not seem plausible.

A quick benchmark on Windows 10 showed the same behaviour like the two Linux system, so it is probably not operating system specific.

We think that the most likely source of this behavior is `Rayon`, the parallelization library we used for ParSplit. `Rayon` uses a thread pool and splits the tasks to be performed by the threads by using work stealing. It is possible that this implementation in `Rayon` has issues with specific thread counts, but more testing is required to confirm this. Potential tests could include implementing ParSplit with threads manually or in another language using `OpenMP` instead of `Rayon`.