IMPLEMENTATION OF AN OPTIMIZED AUTHENTICATED ENCRYPTION SCHEME

Alessandro Giaconia, Diana Khimey, Filippo Visconti, Lorenzo Paleari

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

This paper presents the optimization performed on an authenticated encryption scheme, ChaCha20 for encryption and Blake3 for authentication. The focus is on enhancing performance through vectorization and parallelization, addressing the challenges of processing large data efficiently. Our results demonstrate significant improvements in throughput and efficiency, making our scheme highly suitable for modern high-performance applications that demand robust security and data integrity. The paper contributes an open-source implementation and offers insights into the scalability and performance of authentication and encryption in multi-core environments.

1. INTRODUCTION

Motivation. In an era characterized by the exponential growth of data and the escalating prevalence of digital communication, safeguarding sensitive information has become an indispensable priority. The continuous evolution of technology introduces new challenges in upholding the confidentiality and integrity of data exchanged over networks.

This project embarks on a journey focused on implementing a state-of-the-art, optimized, high-performance authenticated encryption scheme. The motivation driving this initiative arises from the necessity for robust cryptographic solutions that not only ensure security but also exhibit efficient runtimes. Authenticated encryption schemes serve as fundamental pillars in modern cryptography, offering a comprehensive solution for ensuring confidentiality, integrity, and authenticity of data. These schemes find wide-spread application in various contexts, including TLS, SSH, IPsec, and numerous others.

Given that authenticated encryption schemes are nowadays used in many critical applications, it is extremely important to provide the best performance possible, and it is necessary to avoid being limited by factors like file size.

For these reasons, our project leverages the strengths of ChaCha20 and Blake3, prioritizing efficiency and speed, and being targeted toward the most difficult cases, namely instances dealing with huge files.

Related work.

A significant study by R. Velea et al. [1] delves into the performance implications of parallelizing the ChaCha20 encryption cipher across multi-core CPUs and GPUs. The research, conducted using the serial implementation from the BoringSSL encryption library and accelerated through OpenMP and OpenCL, provides valuable insights into the efficiency gains achieved by parallelization. The study particularly highlights the reduction in power consumption with a parallel implementation, showcasing its impact on resource utilization.

It is worth noting that, currently, there is a distinct lack of open-source implementations focusing on the parallel aspects of ChaCha20. Our work aims to contribute to this field by providing an open-source parallel implementation of ChaCha20, addressing the existing gap in the availability of such resources.

For our Blake implementations, we followed closely the algorithm description provided in the original paper [2]. As a reference, we also used the single-threaded implementation provided by the paper's authors [3].

2. BACKGROUND

In this section, we give a brief overview of the background necessary to understand the rest of the report. In particular, we introduce the concepts of encryption and authentication, and we explain the algorithms we use.

Encryption. Encryption is a fundamental concept in information security that involves converting plaintext data into a secure and unreadable form, referred to as ciphertext. The primary goal of encryption is to protect sensitive information from unauthorized access or tampering during transmission or storage. This process ensures that only authorized parties with the appropriate decryption key can access and decipher the original data.

ChaCha20. ChaCha20, detailed in [4], serves as a symmetric key stream cipher with the dual purpose of ensuring data confidentiality and integrity. Recognized for its simplicity, speed, and resistance to cryptographic attacks, it proves to be an excellent fit for this project due to its high

parallelizability.

The functioning of the ChaCha20 algorithm involves taking a key, a nonce and a counter and subjecting them to a series of additions, rotations, and XOR operations. This process produces a keystream that is then XORed with the plaintext. These operations facilitate swift computation even on general-purpose hardware without specific instruction sets like AES-NI.

Today, ChaCha20 is widely used in various security protocols and applications, including TLS (Transport Layer Security) for secure communication on the internet. Notably, in TLS v1.3, it stands as the only alternative to the Advanced Encryption Standard (AES). This attests to its reliability and versatility in secure communication, highlighting its important role in today's cryptographic systems.

Authentication. Authentication is the process through which the integrity and origin of data are verified, to ensure that it remains untampered and confirming that it originated from the expected sender.

To achieve authentication, Message Authentication Code (MAC) is commonly employed. A MAC is a concise piece of information utilized to authenticate a message, working in conjunction with a secret key. The recipient of the message can validate its authenticity by recalculating the MAC and comparing it with the original MAC. This mechanism provides a secure means of ensuring that the received data has not been altered and originates from the expected source.

Blake3. We opted for Blake3 as the authentication component of our scheme, driven by its notable combination of high performance and robust security guarantees. Blake3 is inherently parallelizable, enhancing its suitability for our implementation. The implementation, carried out in the C language, employs two distinct approaches: one utilizing a stack for versatility, and the other employing a more efficient divide-and-conquer strategy, albeit with slightly reduced flexibility, but significantly improved performance.

At a high level, Blake3 processes input by dividing it into chunks of up to 1024 bytes and arranges them as the leaves of a binary tree. These chunks are then compressed using a compression function, conducting a series of operations on the input and producing a 64-byte output, the chaining value. The chaining value from each chunk becomes the input for the subsequent chunk, this process continues until all chunks are processed. The chunks effectively represent the leaves of a binary tree, and the tree is traversed from the bottom up. At each level, the chaining values of two children nodes are combined until reaching the root node. The final result is the chaining value of the root node, obtained through iterations of this process.

To obtain an output of arbitrary length, the last call to the compression function can be repeated with an increased counter, ensuring adaptability to various application requirements.

3. OUR PROPOSED METHOD

In this section, we dive into the details of our implementation. We describe how we implemented the algorithms, and we explain the optimizations we used to achieve high performance.

ChaCha20. Our implementation of ChaCha20 aligns with the specifications outlined in the RFC [5]. The algorithm involves processing a 64-bytes stream through a block function, where a fixed number of XORs, rotations, and additions are performed. Although the algorithm itself offers limited room for improvement due to its inherent simplicity, we pursued optimizations for single-core performance.

We began our optimization efforts with fundamental techniques, like inlining, which, to our surprise, proved more effective than using macros. We also delved into improving memory usage efficiency. However, these initial steps did not achieve the desired level of efficiency.

To further enhance performance, we performed vectorisation, specifically employing AVX-256 to optimize the block function. This proved to be a significant improvement, showcasing the algorithm's responsiveness to advanced optimization strategies.

We later moved into the parallel implementation, which leveraged the OpenMP library. Taking advantage of the independence of each block, parallelization required minimal modifications to the existing algorithm. However, optimizing for cache spatial locality demanded experimentation with various OpenMP scheduling options, adding an extra layer of complexity to the fine-tuning process. We experimented with all different scheduling options, namely: static, dynamic, and guided. In the static approach, the iteration space is divided into chunks of approximately equal size, with at most one chunk assigned to each thread. Dynamic, on the other hand, assigns a loop iteration (or a user-defined chunk) to each thread. When a thread finishes its assigned task, it requests a new one. Guided operates similarly to Dynamic but begins by assigning larger chunks and progressively reduces the dimension until reaching 1.

Blake3. In this subsection, we describe our implementations of the Blake3 algorithm, which are based on and closely follow the original Blake3 paper that describes the algorithm in detail [2]. Given their different strengths and characteristics, we also included a dispatcher that chooses the best implementation based on the input size, to always guarantee a correct output.

Divide-and-conquer version. The divide-and-conquer version of the algorithm, also referred as Full Tree version, relies on the premise that the entire input is accessible at the outset of the computation. This enables an equitable distribution of work among multiple threads, each operating on distinct portions of the input, and subsequently eliminating dependencies among them. The threads function concur-

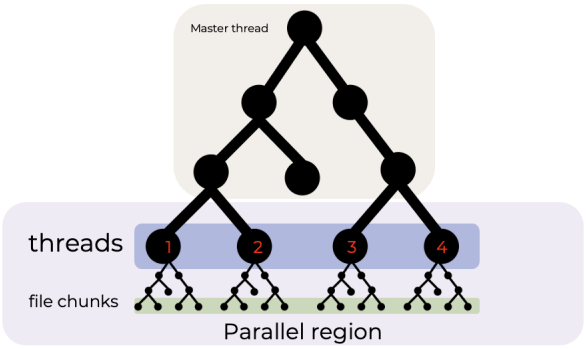

Fig. 1. High-level visualization of the divide and conquer approach

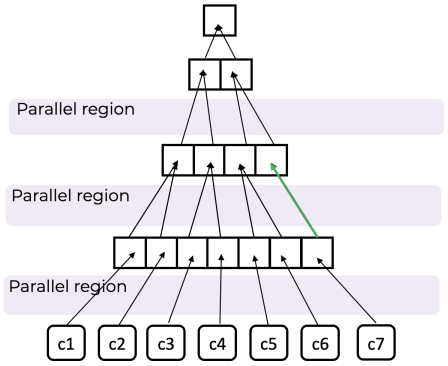

Fig. 2. High-level visualization of the parallel stack approach

rently until they reach the base case, wherein each thread is left with one node.

Upon reaching this base case, the threads transmit their individual results to the parent thread, which consolidates these outcomes and returns the final result. This parallelized approach enhances the algorithm's efficiency, leveraging the availability of the complete input early in the computation to facilitate concurrent and independent processing by multiple threads.

To facilitate parallelization in our algorithm, we leverage the Open-MP library, streamlining the process of distributing computation across threads once a parallel region is established. Additionally, we enhance performance by vectorizing the compression function, the most computationally intensive component of the algorithm, using 128bit vector instructions by partially adopting code from the reference implementation [3].

The primary advantage of this approach lies in its scalability, attributed to the even distribution of work among threads with no inter-thread dependencies. This characteristic allows us to achieve commendable speedups, as detailed in the results section. However, it is important to note that this approach, while highly scalable, has limitations in flexibility. Specifically, it necessitates the availability of the entire input at the commencement of the computation (which aligns with our requirements) and is most effective when the input conforms to a full binary tree structure.

Stack version.

As opposed to the Full Tree algorithm described above, this approach uses a stack to track the progress of parent and child chunks up to the root node. Instead of splitting the entire tree across threads, this algorithm parallelizes the compressions between different levels of the tree. Each thread can then access the chaining values of the child nodes stored on the stack, and store the result of the compression. To prevent false sharing and ensure proper workload distribution, each thread reads and writes from and to unique locations on the stack. As in the divide-and-conquer approach, parallelization is achieved through the use of the Open-MP library and by further performance improvements are achieved by vectorizing the compression function.

The primary benefit of this approach is that it simplifies the logic of compressing files that do not create a full binary tree. It does this by leveraging the stack, and continuously pushing single child chunks up the tree, as is highlighted by the green arrow in Fig. 2. However, it forces threads to continuously switch between parallel and sequential regions, such that the stack is synchronized between each tree level. These transitions cause significant overhead in this approach compared to the divide-and-conquer algorithm, but overall leads to simpler logic when dealing with imperfect file sizes.

4. EXPERIMENTAL RESULTS

Both encryption and authentication were tested on different versions to better detect implementation bottlenecks, initially working on single core and later improving the parallelization of the code to reach better speed up due to the multi-core environment. Every implementation has been tested with input size varying from 1MB to 128GB scaling with a factor of 2. This goes from small dimensions that fits perfectly inside the cache to huge dimensions, while still ensuring that the computation fits the available RAM.

Experimental setup. Our experiments were conducted on the Ault Cluster of the Swiss National Supercomputing Centre (CSCS). We specifically utilized a node with the following specifications:

- **Processors**: 2x AMD EPYC 7742 64 cores @ 2.25 GHz.

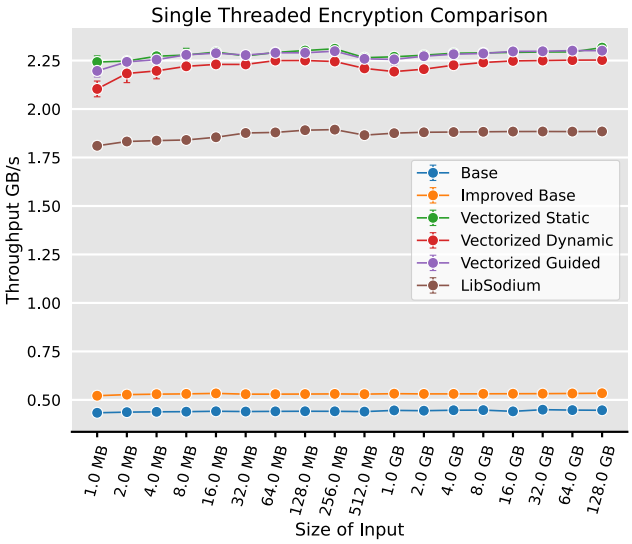

Fig. 3. Comparison of performance over changing input sizes of the single-threaded versions of our implementations and the reference implementation. Higher is faster.

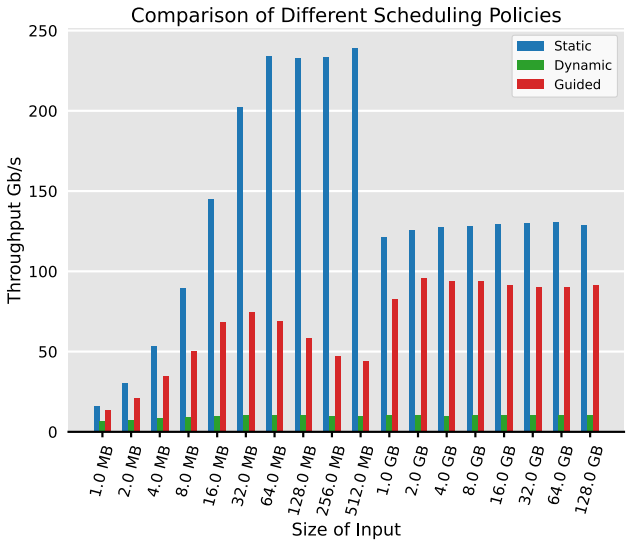

Fig. 4. Comparison of different OpenMP scheduling. Values represent the Throughput using 128 threads with the vectorized version of the code. Higher is faster.

- **Memory**: 256GB of RAM for each processor. 512GB Total. We tested RAM memory bandwidth using the STREAM benchmark[6], it showed a maximum of 130GB/s on a single processor and 270GB/s across the full node (see Appendix for full benchmark results A1).
- **L3 Cache**: Each AMD EPYC 7742 processor features a 256MB L3 cache, segmented into 16 blocks of 16MB each.
- **NUMA Configuration**: The node's dual-socket setup forms two NUMA (Non-Uniform Memory Access) regions, each with one processor and 256GB of RAM.

Compiler, version and used flags vary between authentication and encryption algorithms and were carefully chosen to achieve the best performance possible. In particular, some flags were taken from AMD's specialized guide on the EPYC 7002 processor family [7].

- **Blake**: `gcc 10.3.0, -O3 -march=native`
- **ChaCha20**: `gcc 13.2.0, -O2 -ffast-math -march=znver2 -mtune=znver2 -fopenmp -mfma -mavx2 -fprefetch-loop-arrays -param prefetch-latency=300`

ChaCha20 - Single-threaded comparison. Figure 3 showcases our single-threaded implementation's performance, comparing it to the `libsodium` library, one of the fastest crypto-libraries available.

Our results demonstrate a highly efficient vectorized single-core implementation that achieves a throughput of 2.32 GB/s improving of a factor of 5.30x the base version that sits at just 0.44GB/s.

Furthermore our vectorized implementation exhibits such efficiency to even surpass `libsodium` with a factor of 1.25x. Lastly we can observe, throughout all the input sizes, the throughput consistency showing an absence of performance degradation with bigger inputs.

ChaCha20 - Different scheduling. Utilizing OpenMP to parallelize our vectorized code comes with a question, which of the scheduling policies, mentioned and described in the OpenMP manual [8], is the best.

Figure 4 illustrates our findings, indicating that static scheduling outperforms the others. Our implementation encrypts input blocks independently using a fixed set of instructions, giving consistent encryption times per block. This behaviour benefits static scheduling, while dynamic and guided policies gives better results for tasks with variable processing times.

ChaCha20 - Final results. Lastly, we want to understand the limits of our best ChaCha20 implementation, pushing it to its limits by testing it against the full 128 core of the node. We now analyse and understand the results showed by Figure 5 in particular we concentrate on three interesting behaviours.

The initial observation in our study is the pattern of speedup (indicated in the graph legend within parentheses) for input sizes measured in GBs. Performance scale perfectly up until 16 Threads, reaching a sudden stop at a speedup of 28x when testing on 32 and 64 Threads and lastly, it gets to 55x when using 128 Threads. Given that the node consists of two CPUs, each with 64 cores, the increase in

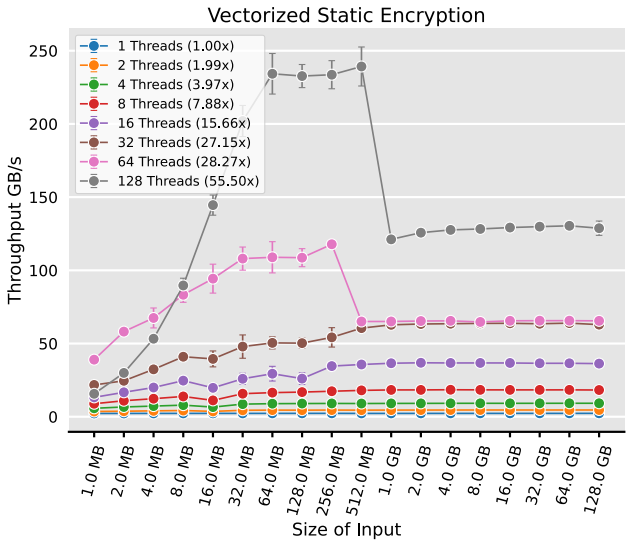

Fig. 5. Multi-threaded vectorization performance across varying input sizes. Between parenthesis the speedup achieved with an input size of 128GB. Higher is faster.

speedup when moving from 64 to 128 threads can be attributed to the utilization of both CPUs. This observation also suggests an absence of noticeable NUMA effects in this scenario. This can be explained by our ChaCha20 implementation: each block of input to be encrypted operates independently, allowing OpenMP to efficiently manage thread scheduling and data distribution across the two NUMA regions, therefore avoiding performance bottleneck. The observed pause in speedup at 28x can be explained using the results of our memory bandwidth benchmark. When using 64 cores, the maximum achievable memory bandwidth (MemBW) is 138GB/s. This number becomes significant when compared to the 65.7GB/s throughput achieved by our implementation at 64 cores. Considering the need to read, encrypt, and write back the input, the data transfer between RAM and CPU amounts to at least 131.4GB/s. This result come close the maximum bandwidth capacity. The slight discrepancy between these numbers can be attributed to difference in compilers, flags, and code, leading to the conclusion that we have likely reached the maximum available bandwidth.

Another noticeable anomaly consist in the huge throughput reached by 64 and 128 Threads with inputs ranging in the hundreds of MBs. New random inputs are generated each run, which are going to fit in last level cache if their dimension is less than 256/512MB (1 or 2 CPUs). This allows the processor to leverage the full MemBW just to write back the input. Indeed results show double the throughput reaching 240GB/s. Regrettably, as soon as we try to encrypt bigger data we will not fit in cache resulting in being constrained to half the throughput as explained before.

Lastly, we noticed a consistency difference between large (GB-sized) and smaller input sizes. Larger inputs take longer to compute ensuring consistent runtimes. On the other hand, smaller inputs take much shorter time to compute, in the order of few μs and thus even the slightest change in runtimes shows bigger inconsistency in the graph.

Blake - Single-threaded comparison. Through Figure 6, we will analyze how our implementations (`blake_f` represents the full-tree version, while `blake_d` represents the stack version) compare to the reference implementation (`blake_ref`), written in C by the authors of the original paper.

From the graph, the performance of our stack implementation closely follows the reference one, since the original approach also uses a stack. The full-tree version is slightly penalized by the overhead it introduces for very small inputs. However, on inputs larger than 4-16MB, it does achieve a better throughput. The increase in performance sits stably at around 16%.

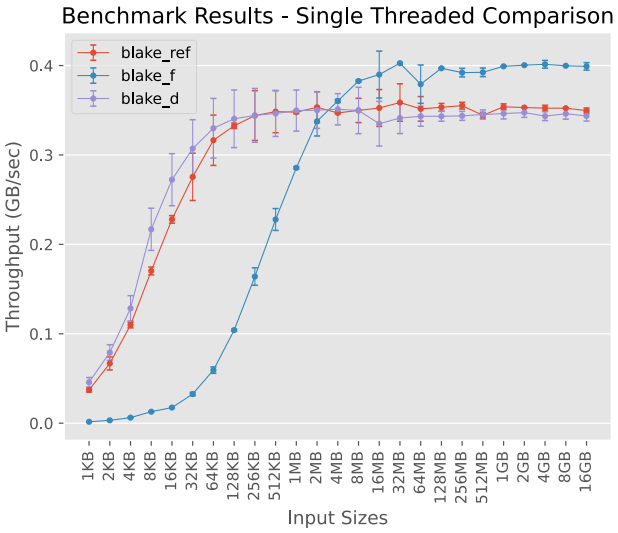

Fig. 6. Comparison of performance over changing input sizes of the single-threaded versions of our implementations and the reference implementation. Higher is faster.

Blake - Stack. In Fig. 7, we show the performance of the parallelized stack implementation for an increasing number of cores and input sizes.

The throughput performance scales roughly linearly to the number of threads, when the number of cores is below 32. However, as can be observed, speedups are negligible for a larger number of cores. This is likely due to the overhead of continuously creating and synchronizing a larger number of threads.

Blake - Full Tree. In Figure 8, we aim to analyze how our fastest implementation scales with the number of cores.

As it is possible to observe, there is a consistent increase

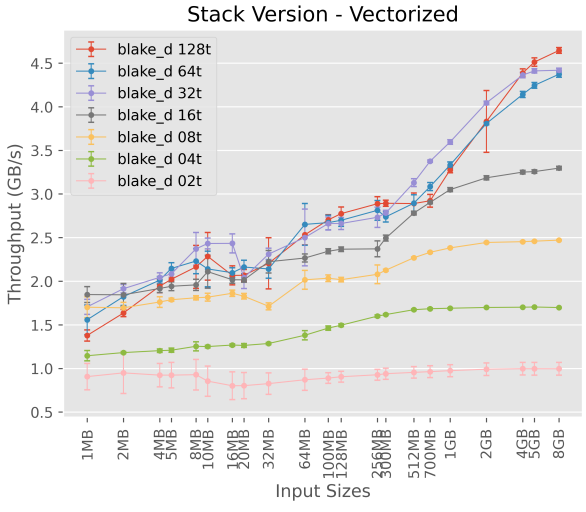

Fig. 7. Performance for growing input sizes of the parallelized and vectorized version of the stack approach. Speedup is shown for an increasing number of threads.

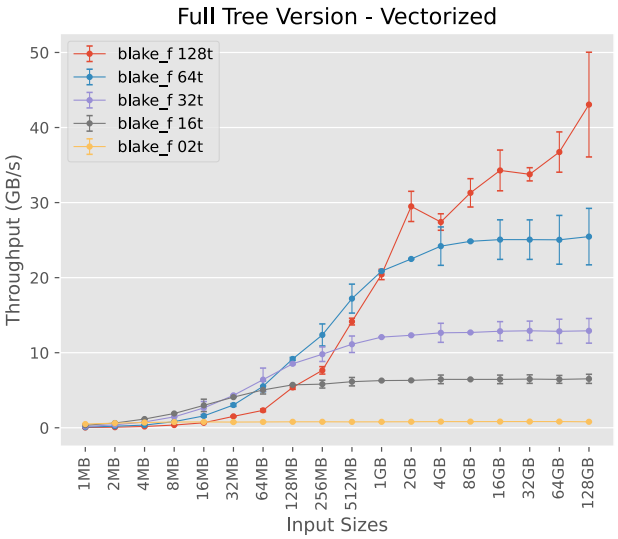

Fig. 8. Performance over changing input sizes of the multi-threaded, vectorized version of the divide-and-conquer implementation. Higher is faster.

of performance up until reaching 64 cores, which sees the throughput double as the number of cores doubles as well. However, going from 64 to 128 cores, the improvement is smaller, and the throughput fluctuates a bit. This is due to the fact that the machine has 2 64-core CPUs, and using both of them for the same program results in some overhead, likely due to slower memory accesses.

We achieved a maximum throughput of 39-50 GB/s, which, converted into bytes per cycle for reference, are equivalent to 18-23 B/C.

In the future, this issue could be fixed by changing the logic of the program in order to implement a multi-process computation, and spawning two different processes, task-setting them on one CPU each, such that we can have a finer control over memory traffic.

As a sanity check, we also bench-marked our code with the compression function disabled (the function body was emptied). This led to a final, ideal throughput of 120 GB/s for the 128 thread Full Tree Version.

5. CONCLUSIONS

This study introduced a comprehensive exploration into optimizing authenticated encryption through the implementation and enhancement of ChaCha20 and Blake3 algorithms.

The ChaCha20 single-threaded comparison displayed a highly efficient implementation, surpassing the `libsodium` library with a notable 1.25x speedup. The parallel version showcased exceptional scalability, achieving a peak throughput of 130 GB/s with linear speedups observed from 1 to 32 threads ultimately reaching the limit imposed by the CPU memory bandwidth. These results provide compelling evidence of the overall efficiency of our implementations.

In the realm of Blake3, we achieved single-thread performance comparable to that of the reference implementation. The multi-threaded stack approach to the algorithm demonstrates a substantial improvement in performance compared to the reference implementation, even while using the same stack structure to manage compression. The divide-and-conquer approach was shown to scale virtually indefinitely with the number of cores, indicating promising potential for uses with massive file sizes.

Future Work. The project lays the foundation for future enhancements and explorations. For ChaCha20, further investigations into alternative vectorization strategies and the utilization of advanced instruction sets (AVX-512) could yield additional performance improvements. Testing on newer architecture can also benefit the research and showcase the full scalability of our implementation without being limited by the memory bandwidth.

In the case of Blake3, scalability on systems with multiple CPUs are areas worth exploring. For further improvements, it would be interesting to explore how the encryption problems can be divided in order to take advantage of the stack version's simple logic as well as the scalability of the divide-and-conquer approach. Just as for ChaCha20, the use of vectorization strategies such as AVX-512 could lead to even further performance improvements for Blake3.

Moreover, a comprehensive security analysis of the implemented scheme is essential to ensure its robustness against potential vulnerabilities - such as side channel attacks.

6. REFERENCES

[1] R. Velea et al., "Performance of parallel chacha20 stream cipher," [Online] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=7507408, 2016.

[2] J. O'Connor et al., "Blake3 one function, fast everywhere," [Online] https://blake3.io, 2019.

[3] J. O'Connor et al., "Blake3 repository," [Online] https://github.com/BLAKE3-team/BLAKE3/, 2019.

[4] Daniel J. Bernstein, "Chacha, a variant of salsa20," [Online] https://cr.yp.to/chacha/chacha-20080128.pdf, 2008.

[5] Internet Research Task Force (IRTF), "Rfc 7539 - chacha20 and poly1305 for ietf protocols," [Online] https://datatracker.ietf.org/doc/html/rfc7539, 2015.

[6] AMD, "Stream benchmark," [Online] https://www.amd.com/en/developer/zen-software-studio/applications/spack/stream-benchmark.html, 2024.

[7] AMD, *High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors*, AMD, November 2020, [Online] https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/amd-epyc-7002-tg-hpc-56827.pdf.

[8] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, OpenMP, 4.5 edition, November 2015, [Online] https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

A. APPENDIX

1. Benchmarks

Single processor memory bandwidth

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 4 bytes per array element.
-------------------------------------------------------------
Array size = 3000000000 (elements), Offset = 0 (elements)
Memory per array = 11444.1 MiB (= 11.2 GiB).
Total memory required = 34332.3 MiB (= 33.5 GiB).
Each kernel will be executed 100 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 64
Number of Threads counted = 64
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 178468 microseconds.
   (= 178468 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:          141867.4     0.169484     0.169172     0.169902
Scale:         142325.9     0.168982     0.168627     0.169334
Add:           143459.4     0.251262     0.250942     0.251731
Triad:         143444.0     0.251348     0.250969     0.251694
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-06 on all three arrays
-------------------------------------------------------------
```

Full node memory bandwidth

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 4 bytes per array element.
-------------------------------------------------------------
Array size = 3000000000 (elements), Offset = 0 (elements)
Memory per array = 11444.1 MiB (= 11.2 GiB).
Total memory required = 34332.3 MiB (= 33.5 GiB).
Each kernel will be executed 100 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 128
Number of Threads counted = 128
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 83415 microseconds.
   (= 83415 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:          283249.0     0.084882     0.084731     0.085087
Scale:         284431.1     0.084562     0.084379     0.084783
Add:           286635.4     0.125829     0.125595     0.126814
Triad:         286662.6     0.125799     0.125583     0.126066
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-06 on all three arrays
-------------------------------------------------------------
```