Cha Cha Real Smooth: The Case for ChaCha20/Poly1305

CSE 253 - Final Project Report

Allen Aboytes
University of California, Santa Cruz
aaboytes@ucsc.edu

Abstract

We study the software performance of AES-GCM and ChaCha20/Poly1305 authenticated encryption algorithms. AES-GCM is the default choice for many protocols as an authenticated encryption algorithm. However, we argue that ChaCha20/Poly1305 is a reasonable alternative. We show that ChaCha20 provides competative performance with a vectorized implementation. We find that on a 64-bit x86 machine the software performance of ChaCha20 provides nearly double the performance of AES for long length messages. The performance gap grows wider with ChaCha20 vector optimizations.

1 Introduction

The confidentiality and integrity of data in transit must be protected. Internet Protocol Security (IPSec) is a network layer protocol that provides authentication and encryption of packets between two endpoints. IPSec is an open standard that supports various cryptographic algorithms used in its protocols to protect IPv4 traffic. Among these algorithms is AES-GCM [1, 2], an authenticated encryption (AE) algorithm that ensures the confidentiality and integrity of data. AES-GCM can also provide integrity for additional data that is not encrypted, making it an AEAD algorithm.

The most widely used AEAD algorithm is AES-GCM. In fact, it is the recommended AEAD algorithm for IPSec [3, 4], and the default on most systems. Although AES-GCM is perfectly secure, it relies on the security of AES. Devices in the network need other alternatives to AES-GCM for an AEAD.

While IPSec leans heavily towards AES-GCM for its combined confidentiality and integrity gaurentees, other internet security protocols have considered alternatives. The Transort Layer Security (TLS) protocol, a layer above IPSec, is one example that offers an option for an alternative, namely the ChaCha20/Poly1305 construction [5, 6]. Like AES-GCM, ChaCha20/Poly1305 provides 256-bit security as well as a 16-byte tag for integrity.

The combination of the ChaCha20 cipher [7] and Poly1305 MAC [8] as an AEAD not only provide security, but performance as well. ChaCha20 in particular was designed for parallelism to exploit single instruction, multiple-data (SIMD) instructions [9].

In this work we compare ChaCha20/Poly1305 to AES-GCM on a 64-bit x86 machine. Specifically, we look at the software performance of ChaCha20 in comparison to AES. We speed up ChaCha20 with vector instructions.

2 Background

In this section, we aim to provide a detailed background on the cryptographic algorithms studied in this work. We describe AES, ChaCha20, GCM, and Poly1305 individually and explain how to construct them for AEAD.

2.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric key block cipher. A symmetric key cryptographic cipher means that the same key used for encryption is used for decryption. It has a block size of 128-bits and supports key sizes of 128, 192, and 256 bits. Originally, when proposed as Rijndael in the AES competition, it supported various block and key sizes [10].

AES internally operates on a 128-bit state matrix. The state matrix is a 4x4 two dimensional array of bytes. It can be interepreted as an array of columns where each column represents a 32-bit word. Operations, such as addition, on the state happen over GF(2^8).

At its core, AES is a product cipher consisting of multiple layers of transformations. see Figure 1. The key addition layer, byte substitution layer, shift rows layer, and mix columns layer make up a round of AES. The security of AES is gaurenteed by the substitutions and permutations in a round that provide confusion and diffusion. Multiple rounds are requried for strong encryption, and the number of rounds depends on the size of the key.

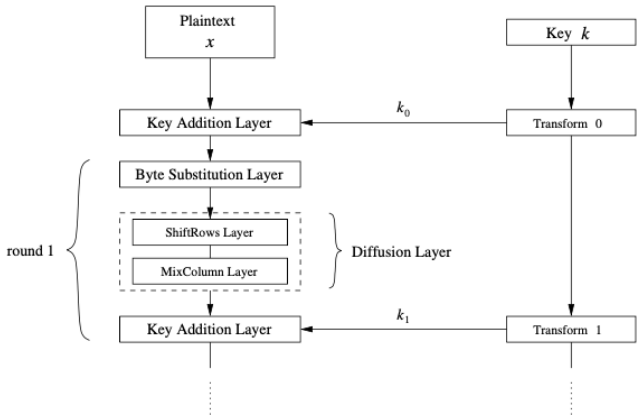

Figure 1: Internal structure of a round of AES [11].

$$\begin{pmatrix} 0x61707865 & 0x3320646E & 0x79622D32 & 0x6B206574 \\ key[0] & key[1] & key[2] & key[3] \\ key[4] & key[5] & key[6] & key[7] \\ counter[0] & counter[1] & nonce[0] & nonce[1] \end{pmatrix}$$

Figure 2: Initial state of ChaCha matrix [12]

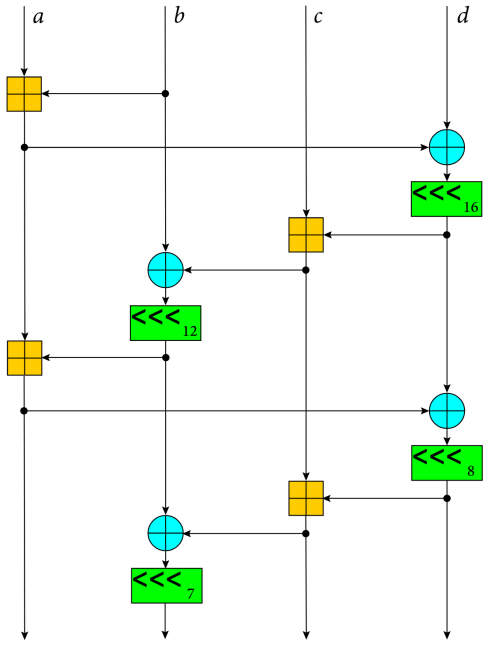

Figure 3: The ChaCha quarter round function [13]

The key addition layer adds a 128-bit round key to the state matrix. A long term secret key is used to create a key schedule. Addition is performed with XOR over GF(2^8). The byte substitution layer applies a non-linear transformation to each individual byte in the state matrix using a look up table (S-Box). The shift rows layer performs a left rotatation on each row. Each row is rotated by a certain number of bytes depending on the row number. The last transformation is mix columns which operates on the state by column. This layer treats each column as a 4 term polynomial over GF(2^8) which is muliplied by a special fixed polynomial.

AES is very secure and has gone over extensive analysis since it was standardized. It was designed to be efficient in hardware and software [10].

2.2 ChaCha

ChaCha is a 256-bit stream cipher based on the Salsa20 [9, 7] stream cipher. ChaCha conjecturely provides more diffusion by modifying the round function [7]. It processes messages in 64-byte blocks. Although Salsa20 supports 128-bit secret keys, and thus ChaCha, only 256-bit usage has been standardized [6].

The core of ChaCha is a hash function operated in counter mode. It operates on a 64 byte state which is interpreted as a 4x4 matrix of 32-bit words. The matrix is composed of a constant, a secret 256-bit key, a 64-bit nonce, and 64-bit counter (see Figure 2). This matrix is operated on by the quarter round function. The round in ChaCha calls the quarter round function 4 times. The quarter round function works on inputs independently, and modifies the matrix depending on the elements chosen.

The quarter round function is composed of 4 additions, 4 rotations, and 4 XOR's. This is the core of the ChaCha20 algorithm, Figure 3. This makes ChaCha a ARX cipher since it only relies on additions rotations, and XOR. This makes ChaCha very efficient since these instructions are fast on modern CPUs. ChaCha20 means that 20 rounds are required to provide its security gaurentees. Processing a whole ChaCha20 block would involve 16 additions, 16 rotations and 16 XOR's per round, totaling to 300 instructions each.

By design, ChaCha20 is very paralellizable. Each of the inputs for the quarter rounds in each round are calculated independently. This means that one can compute these values in parallel using SIMD instructions.

ChaCha20 is a relatively secure algorithm. Its predecessor, Salsa20, recieved widespread scrutiny as a participant in the eSTREAM project [14]. Other attention from the security community has came from the BLAKE hash algorithm. BLAKE is a has function based on ChaCha20 and was a finalist in the SHA-3 competition [15].

2.3 Galois Counter Mode

The Galois Counter Mode (GCM) is an authenticated block cipher mode of operation. It provides integrity by generating a 16 byte authentication tag based on a polynomial multiplier over GF(2^{128}) [2, 16]. It operates by first generating a one time use key (`H`) using the underling block cipher. Then it multiplies previously processed blocks using the `GHASH` function and adds in the cipher text which is just XOR over GF(2^{128}).

In the end it includes the length of the cipher text and length of any additional data in the tag.

2.4 Poly1305

Poly1305 is a message authentication code that is based on a large prime field [8]. It computes a 16 byte tag using a 128-bit one time key. It works by processing blocks as 16 byte little endian numbers. The one time key is split in half, `r` and `s`. The value `r` clears a set of predefined bits and is interpreted as a large little endian number. Poly1305 works by adding blocks as numbers, through some acucmulator value. Then that value is multiplied by `r` over the prime `p`. The prime `p` is 2^{130-5}, hence the name Poly1305. In the end, the value `s` is added to the result and the least significant bits are used for the tag [6].

2.5 Authenticated Encryption

There are various ways of constructing authenticated encryption schemes, however, only *Encrypt-then-Mac* is fully secure [17]. Constructing strong authenticated encryption with GCM or Poly1305 is simple. All GCM requires is a 128-bit block cipher which it used to operate on and generate a one time key used in the `GHASH` function. All Poly1305 requires is for a random secret, which could easily be generated in a similar fashion by the underlying encryption algorithm.

3 Evaluation

In this section, we describe our evaluation methods for evaluating ChaCha20/Poly1305 and AES-GCM, as well as AES and ChaCha20. All algorithms are used in 256-bit modes. We also provide details on how we implement these algorithms.

3.1 Testing Methodology

We record the CPU time of an encryption for various message lengths. The CPU time represents the overall latency of an encryption. We do not measure decryption since in the case of AES in counter mode and ChaCha20, the operations are symmetrical. We divide the message length in bytes by the time to obtain throughput. We collect 9604 samples which is enough samples to obtain a 95% confidence interval with a 1% error margin for a unknown population size. We make sure that all data is in the L1 data cache before taking a measurement. A single sample does not include any setup costs (e.g. key expansion). We use Google Benchmark [18], a mature microbenchmarking library writting in C++, to instrument the algorithms and gather timing information. The length of the input data used goes from 0 to 4096 in increments of 16. The input data for encryption is randomly generated using the Fortuna PRNG [19]. Random file generation is not measured.

3.2 Machine Configuration

We run all code on a virtualized 64-bit x86 machine. The virtual machine from DigitalOcean comes configured with Ubuntu 20.04 LTS on top of VT-x virtualized with KVM. The machine contains 8GB of RAM, 160 GB of virtualized disk, and an Intel Xeon Gold 6140 CPU with a 32 KiB L1 data cache. The CPU contains a skylake microarchitechure which suppoorts vector instructions. Hyperthreading is disabled. The amount of RAM and disk are of little importance since all out operations happen in the CPU caches, however, we mention them for completeness.

3.3 Software Implementation

We study the software performance of AES-GCM, ChaCha20/Poly1305, AES, and ChaCha20. Highly optimized variations of AES can be found in popular open source libraries, however only reference implementions for ChaCha20 are available. We implement vectorized versions of ChaCha20 [12] to include 128-bit and 256-bit instructions for the x86. They are implemented in C with the use of compiler intrinsics, and follow RFC [6]. These optimizations are compared with LibTomCrypt [20], an portable open source library that contains relatively high performance implementions of AES, ChaCha20, AES-GCM and ChaCha20/Poly1305. We also ended up implementing reference implementations of AES based on the FIPS 197 standard [1] and ChaCha20 which we discuss in Section 4 further detail. The source code for our vectorized ChaCha20 implementations can be found at `https://github.com/PandaZ3D/aead_perf`.

None of the implementations tested implement additional security protections (padding, XChaCha nonces [21], constant-time, etc.). Doing so often incurs more overhead, although, ChaCha20 by nature is constant time. Protections from side-channels are not considered. The only 'additional security' we study is the overhead from AE.

All source code implementations are compiled with gcc 9.3.0 and the `-O3` compiler optimization. LibTomCrypt is compiled with its default compiler flags which include `-O3`. The ciphers we implemented with SIMD instructions are compiled with `-msse -msse2 -msse3 -msse4.1 -mavx -mavx2` compiler flags. Note that using `-mavx2` with SSE2/SSE3 instructions optimizes them with AVX [12].

4 Results

This section shows the performance of the optimizations we implemented in comparison to LibTomCrypt [20]. All algorithms use a 256-bit key, and only the performance of encryption (and decryption) operations are measured. For each of these figures in this section, a lower latency is better.

4.1 Performance of Vectorized ChaCha20

Figure 4 shows the latency of our optimized ChaCha20 implementions in comparison to LibTomCrypt. We see that, generally wider bit instruction implementations yeild better performance for ChaCha20. Suprisingly, our 32-bit reference implementation does considerably better than the one found in LibTomCrypt. This could be due to some compiler optimizations or the fact that LibTomCrypt clears its internal buffers in between encryptions.

4.2 Two AES implementions

In Figure 5, we compare our naive 32-bit implmentation to the implementation in LibTomCrypt. Both algorithms are operated in counter mode. It is clear that the performance of our implementation does horrible on a 64-bit x86 machine. Our implementation follows the FIPS 197 specification [1] which directs an unknowledgeble reader to implement it using a simple 8-bit S-Box implementation. This is not efficient on 64-bit computers. A more efficient method, which LibTomCrypt uses and the original specification for Rijndael [10] mentions, is to use precomputed T-Boxes. These lookup tables speed up different steps of the round transformation which explains the large performance gap between our implementation and the one in LibTomCrypt.

4.3 Performance of AES and ChaCha20

We next compare software implementations of AES and ChaCha20, along with their AEAD modes. We see in Figure 6 the speed up of ChaCha20 128 is almost x2 faster relative to the base implmentation found in LibTomCrypt. In Figure 7 we use AES in counter mode and ChaCha20 as baselines to measure the performance of AES-GCM and ChaCha20/Poly1305. We see that the performance hit is negligable.

5 Discussion

Early results indicated that ChaCha20 was substatially faster than AES, however, this was due to our own inefficient implementation. The implemntation in LibTomCrypt uses T-Boxes for more efficient software computation. This outperformed our own implementation and closed some of the performance gap between AES and ChaCha20.

We present a select a number of packet sizes of interest. Table 1 and Figure 8 more intuiatively highlight the main differences between the ciphers. We also present the same data in Figure 9, except we measure the throughput of each cipher. The main observation we make is vectorized implementations of ChaCha20 perform worse than simple refence implementations for small block sizes. We only see the benefit of this method when computing multiple blocks in parallel.

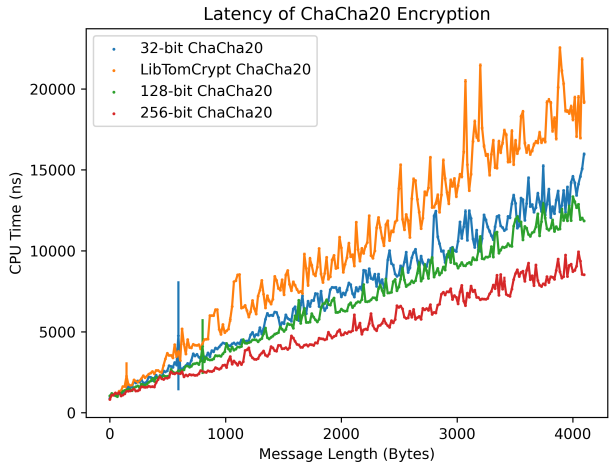

Figure 4: A comparison of ChaCha20 implementations. 32-bit ChaCha20 is a 32-bit portable reference implementation. Other n-bit implementations use SIMD instructions.

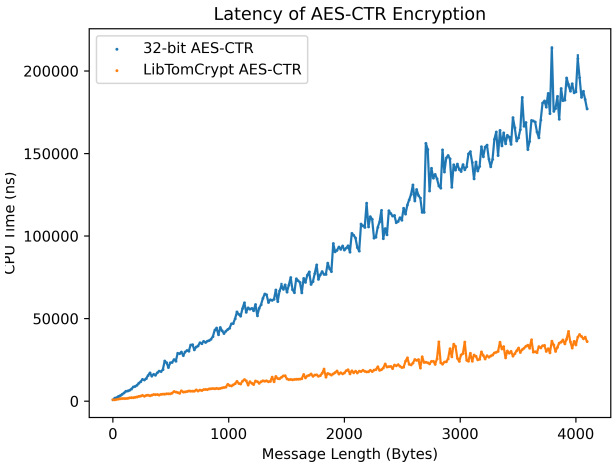

Figure 5: A comparison of two AES implementations operated in counter mode. The 32-bit AES-CTR takes a naive approach, using 8-bit S-Boxes.

From the results in Section 4, we noticed that the data was rather noisy. In particular, as the message length grew in size, it became more succeptible to system noise. We make note of these concerns.

5.1 Limitations of the Study

Although Hyperthreading on our virtualized system was disabled, for security, we did not have the right access permissions to disable turbo boost. Turbo boost is a feaure of Linux that dynamically scales the voltage and frequency of the CPU to gather better performance. It would have been easier to get

Table 1: Latency of Select Message Lengths

All algorithms measured except naive AES implementation. Latency in ns and length in bytes.

Algorithms	16	64	512	1024	2048	4096
32-bit ChaCha20	1105.85 ± 32.565	1060.5 ± 13.086	2908.3 ± 55.187	4147.07 ± 40.615	7585.2 ± 59.94	15981.2 ± 94.891
128-bit ChaCha20	1187.7 ± 29.237	994.108 ± 21.263	2247.44 ± 38.753	3948.23 ± 74.851	6289.98 ± 51.131	11847.7 ± 70.409
256-bit ChaCha20	1111.95 ± 34.461	1147.58 ± 30.072	2189.18 ± 48.566	3223.7 ± 44.897	4897.45 ± 51.105	8526.85 ± 72.992
AES-CTR [20]	846.739 ± 17.582	1462.65 ± 32.587	4796.21 ± 51.018	9171.27 ± 190.357	16505.6 ± 104.24	36026.7 ± 184.037
ChaCha20 [20]	1068.65 ± 12.605	1158.76 ± 10.467	3539.73 ± 45.707	5173.74 ± 56.531	9580.35 ± 92.081	19193.9 ± 119.549
AES-GCM [20]	1892.58 ± 26.77	2136.15 ± 33.322	6264.19 ± 63.141	12912.3 ± 105.056	19886.2 ± 112.191	39444.4 ± 196.108
ChaPoly [20]	1467.54 ± 42.381	1405.75 ± 26.891	4262.79 ± 48.791	6933.67 ± 70.454	12431.4 ± 79.573	26914.1 ± 162.831

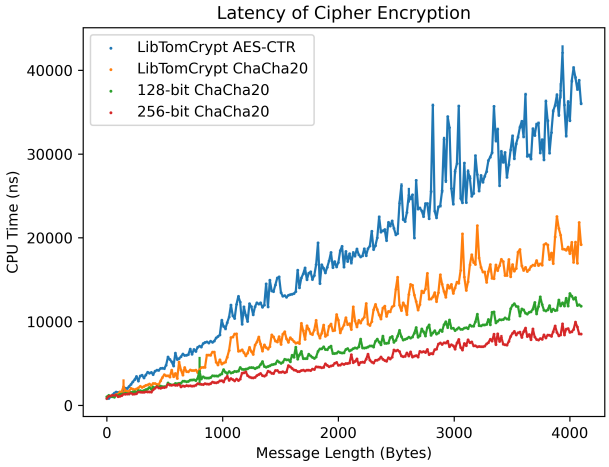

Figure 6: A comparison of AES in counter mode and ChaCha20, along with vector implementations for ChaCha20.

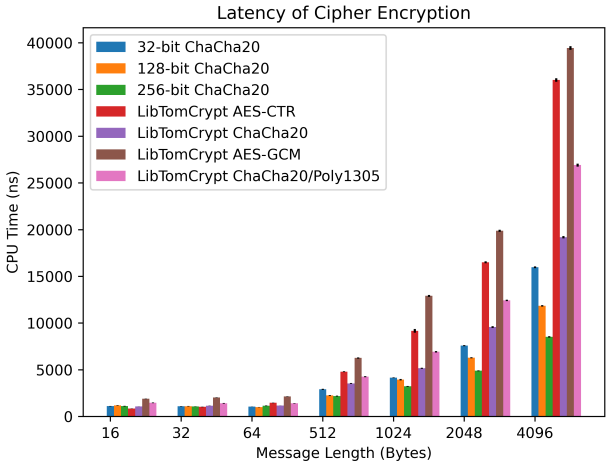

Figure 8: The latency of select message sizes for all ciphers measured. We omit our own AES implementation.

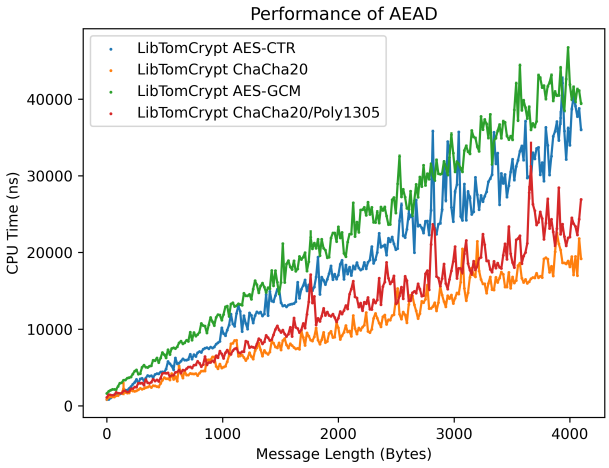

Figure 7: A performance comparison of AES-CTR and ChaCha20, with and without AE.

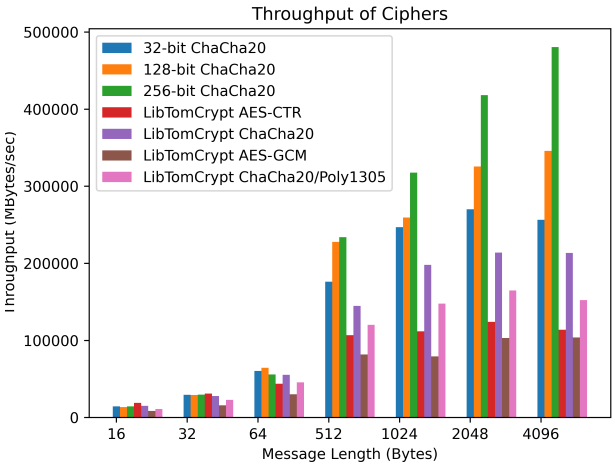

Figure 9: The throughput of select message sizes for all ciphers measured. We omit our own AES implementation. Lower is better

consitent results with a fixed CPU clock rate. Another concern is system noise from working in the cloud. Our machine configuration operated on a shared CPU, where although the OS might not deschedule us, the hypervisor might for QoS reasons.

Part of the problems with our study is that it is an incomplete evalutation. We could have implemented various other optimizations for the algorithms presented. Other optimizations for software AES [22], 512-bit vector ChaCha20 [12], vectorized Poly1305 [23], and AES-NI instructions could have been included. We leave this for future work.

We could have also tested more incrementally in that regard. The throughput of the MAC algorithms could also have been considered while holding the data size constant.

5.2 Lessons Learned

Optimizations. I learned a lot about how to use vector instructions. Implementing these algorithms taught me that programmers don't always need to use hand crafted assembly to get good performance. On a similar note, often accelerating a cryptographic algorithm involves either dedicated hardware circuits, parallelization, or both. There are a lot of software tricks one can utilize to get around obstacles in cryptography such as big number computations.

Benchmarks. I also invested a considerable amount of work learning how to benchmark software properly. Benchmarking is an art. Within a system there are various sources of noise, and that affects performance results greatly.

By reading the literature, I learned that most cryptographers measure latency using cycles per byte where execution time is measured in clock cycles. I tried to implement my own timing framework, but ultimitely ended up failing. There are existing benchmark frameworks specifically for cryptographic algorithms, but I could not get those to work. They were either old research projects or had a steep learning curve. This motivated my decision to use Google Benchark [18].

Crypto Implementations. The last lesson I learned was that getting cryptographic implementations right is hard. For something simple like ChaCha20, there is little programming time and more time invested in understanding the cipher. AES is quite complex with GF(2^8) operations. For both algorithms, as well as thier assiciated MACs, I understand most of the math, however, translating that to software was challenging at times. Usually reference implementations read like pseudocode, but perform poorly. Once you start worrying about performance and try to add optimizatoins, it becomes difficult to program. This is inherantly complex if the algorithm was not designed for parallelism, as is the case for AES or Poly1305.

When implementing these algorithms I looked at various open source libraries and public domain code [20, 24, 25, 26, 27] for guidance in addition to original design specifications and standards [10, 1, 7, 8, 16, 6].

6 Conclusion

In this work we showed that ChaCha20 offers considerable performance and is comparable to that of AES. We demonstrate this by implementing ChaCha20 with support from SIMD instructions in the x86 instruction set. We note that the benefit of using this design is to increase throughput for larger message lengths. A simple software implementation could have the advantage for small messages. The source code implementations for ChaCha20 are available at https://github.com/PandaZ3D/aead_perf

References

[1] *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. NIST, 2001.

[2] *SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. NIST, 2001.

[3] S. Frankel et al. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. RFC Editor, Feb. 2011.

[4] J. Viega and D. McGrew. *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)*. RFC 4106. RFC Editor, June 2005.

[5] Damien Miller. *ChaCha20 and Poly1305 in OpenSSH*. http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html.

[6] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. RFC Editor, June 2018.

[7] Daniel J Bernstein et al. "ChaCha, a variant of Salsa20". In: *Workshop record of SASC*. Vol. 8. 2008, pp. 3–5.

[8] Daniel J Bernstein. "The Poly1305-AES message-authentication code". In: *International Workshop on Fast Software Encryption*. Springer. 2005, pp. 32–49.

[9] Daniel J Bernstein. "The Salsa20 family of stream ciphers". In: *New stream cipher designs*. Springer, 2008, pp. 84–97.

[10] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael*. 1999.

[11] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[12] Martin Goll and Shay Gueron. "Vectorization on ChaCha stream cipher". In: *2014 11th International Conference on Information Technology: New Generations*. IEEE. 2014, pp. 612–615.

[13] https://en.wikipedia.org/wiki/Salsa20. Wikipedia.

[14] *eSTREAM: the ECRYPT Stream Cipher Project*. https://www.ecrypt.eu.org/stream/.

[15] *NISTIR 7896. Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. https://csrc.nist.gov/publications/detail/nistir/7896/final. NIST, 2012.

[16] David McGrew and John Viega. "The Galois/counter mode of operation (GCM)". In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.

[17] Mihir Bellare and Chanathip Namprempre. "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2000, pp. 531–545.

[18] *Benchmark*. https://github.com/google/benchmark.

[19] Niels Ferguson and Bruce Schneier. *Practical cryptography*. Vol. 141. Wiley New York, 2003.

[20] *LibTomCrypt*. https://github.com/libtom/libtomcrypt.

[21] S. Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD XChaCha20 Poly1305*. RFC Draft. RFC Editor, Jan. 2020.

[22] Emilia Käsper and Peter Schwabe. "Faster and timing-attack resistant AES-GCM". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 1–17.

[23] Martin Goll and Shay Gueron. "Vectorization of Poly1305 message authentication code". In: *2015 12th International Conference on Information Technology-New Generations*. IEEE. 2015, pp. 145–150.

[24] *Mbed TLS*. https://github.com/ARMmbed/mbedtls.

[25] *NaCl: Networking and Cryptography library*. https://nacl.cr.yp.to/.

[26] *wolfSSL*. https://github.com/wolfSSL/wolfssl.

[27] *Intel(R) Integrated Performance Primitives Cryptography*. https://github.com/intel/ipp-crypto.