Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis

Systems Group, Department of Computer Science, ETH Zurich

Benchmarking TLS Cipher Suites on FPGA

by

Colin Arnet

Supervised by

Prof. Gustavo Alonso and Zhenhao He

March 2023 - June 2023

Abstract

TLS is the most widely used protocol to secure web traffic. By encrypting and authenticating the data it provides confidentiality and integrity. From a set of different crytographic operations used in TLS, the cipher suites emerged to be the most used operations of the protocol, since they encrypt/decrypt the entire application data. Since the cipher suites are heavily used, they are a attractive target to offload on a FPGA device. The Vitis security library provides HLS implementations that can be loaded onto an FPGA. In this thesis, we evaluate the performance of these functions and compare the performance to the well established `OpenSSL` implementations. For the AES-GCM mode the hardware functions are on one level with the software implementations. On the other side for the AES-CCM mode, the hardware benchmark cannot keep up with the `OpenSSL` implementations. Nonetheless, running the cryptographic operations on a FPGA device, frees up CPU capacity for other tasks.

Contents

Chapter 1

Introduction

Security has always been a big concern in communication over the web. Sending plaintext over the network can give rise to major privacy issues, since an attacker can read any data that is sent. To address this issue, there are protocols that address this issue at different network layers. For the transport layer the Transport Layer Security (TLS) was introduced. TLS is a protocol that is built on top of the Transmission Control Protocol (TCP). TLS encrypts and authenticates the application data that is sent by an application. This way TLS ensures confidentiality and integrity. The Hypertext Transfer Protocol Secure (HTTPS) uses TLS to secure the application traffic. TLS is widely used and about 95% of the traffic that passes through Google is encrypted by TLS [6].

TLS is the de facto standard protocol that provides security on the web. Because of this, there is a lot of time invested in the improvement of TLS. The latest version TLS 1.3 [14] is a major revision of TLS 1.2 [15]. It improves security by removing the support for deprecated cryptographic algorithms still supported by TLS 1.2. It also simplifies the protocol by only supporting a handful of robust algorithms. Another big improvement over TLS 1.2 is that the handshake now runs in 1 round-trip time (1-RTT). There is also a option for a 0-RTT handshake, with which a previous session can be resumed without reestablishing the keys but it is coming with some security cutbacks. TLS 1.3 is being widely deployed and is on the way to being the standard security protocol of the web [8, 10, 9].

The last years heterogeneous systems have become more and more widely deployed. In addition to a general-purpose CPU they provide other devices like Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). FPGAs can be configured in any way a user needs. With a hardware description language (HDL) a user can define any circuit. The circuit can then be loaded and synthesised onto a FPGA device which can the circuit. FPGAs are therefore very flexible devices that can be used for any application.

Since TLS is a widely deployed protocol, it is a very attractive target for a hardware implementation. By offloading the computations of the protocol to a FPGA device, CPU capacity is freed up for other applications. This thesis has the goal to find the cryptographic operations that are heavily used by the TLS 1.3 protocol. After this, these operations are implemented with the help of Vitis High-Level Synthesis (HLS) tool and its security library. In a next step the performance of these implementations are evaluated and compared to the respective `OpenSSL` software implementation. The thesis should build a base for a further implementation of the complete TLS protocol in hardware.

The thesis is organized as followed. In chapter 2 we discuss related work, especially focussing on TLS implementations on unusual devices. The inner workings of TLS and its cryptographic operations are treated in chapter 3. The chapters 5 and 6 discuss the experiments and their results with a summary in the final chapter 7.

Chapter 2

Related Work

The performance of cryptographic operations has been evaluated on many different systems. Depending on the devices the computations must have different properties. For IoT devices, the computations must be lightweight, since the devices do not have much power. IoT devices mostly do not provide sufficient security, making it important to develop lightweight algorithms for these systems.

The National Institute of Standards and Technology (NIST) puts in a lot of effort in standardizing lightweight cryptographic algorithms [13]. The first round candidate algorithms were evaluated by Renner et. al. [17]. Blanc et. al [3] evaluated a subset of the second round candidates for wireless IoT networks.

In cloud-IoT applications a Dynamic Searchable Symmetric Encryption Scheme was evaluated by Ti et. al. [18]. In this case the data is encrypted and stored on a cloud server. A local device can then send a token to the server to get the encrypted data. Then the device decrypts the data locally. This ensures the secrecy of the data without filling up the small storage space of a IoT device.

Hardware acceleration is essential to offload the computations or to achieve better performance. Chang et. al. [5] detect the computation hotspots in the cryptographic computations and implement these in hardware. With this method they achieve a substantial speedup. Adams et. al. [2] achieve their speedup by offloading the computations on a RISC-V general-purpose GPU.

In recent years there has also been a trend to deploy FPGAs on a large scale. Microsoft's `catapult_v2` architecture [4] handles the devices as shared computing resources that can also be used by remote nodes. Furthermore the FPGAs can be reconfigured depending on the current usecase.

With EasyNet, He et. al. [7] implemented network functionalities for HLS to simplify the development of applications that need network access. Which shows a trend to offload computations away from general-purpose processors.

Chapter 3

Background

In this section we provide background information about TLS 1.3 and the crytpographic operations we chose to benchmark.

3.1 TLS 1.3

TLS 1.3 is defined in RFC 8446 [14] and is a complete overhaul of the protocol. Compared to TLS 1.2 [15], TLS 1.3 does not support deprecated algorithms anymore. It also supports less algorithms which reduces the complexity. In addition it improves the handshake from 2-RTT (round-trip time) to 1-RTT and even supports a 0-RTT option where old sessions can be resumed with the same key without the need to reestablish the key with a key exchange. In its base TLS 1.3 consists of two protocols. A handshake protocol and a record protocol.

3.1.1 Handshake protocol

In the handshake protocol the parties are authenticated, the cryptographic modes and parameters are negotiated and the keying material is established. To establish a key, the parties must perform a Diffie-Hellman key exchange. For this, each party must negotiate a algebraic group for the key exchange. For a Diffie-Hellman key exchange, each party A and B has a secret a and b respectively. The secrets are elements of a finite field group or a elliptic curve group which the parties negotiate beforehand. The groups are generated by a generator g[1]. Each party then computes its public key share g^a and g^b. They send each other their key-shares and then can compute their shared key respectively g^{ba} and g^{ab} which are equal. This procedure is secure because of the discrete logarithm problem for which it is difficult to compute a when only knowing g^a.

[1]Group-generator g: For any group element x there is a y where $g^y = x$

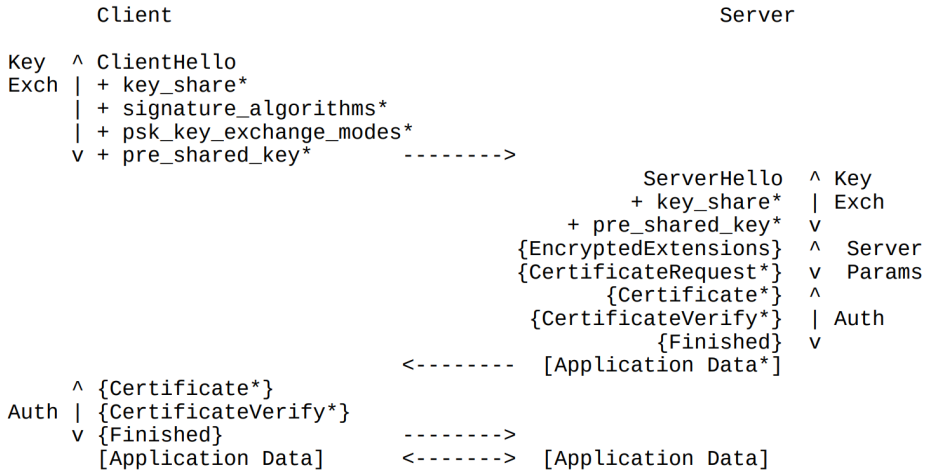

Figure 3.1: TLS 1.3 Handshake Protocol

In addition to the established key the server and the client must also agree on a symmetric cipher/hash pair and a signature algorithm.

To achieve this goal the client sends a `ClientHello` message to the server. The `ClientHello` message contains a random nonce, the offered protocol versions, a list of cipher/hash pairs and a set of Diffie-Hellman key shares incuding the groups. The server processes the `ClientHello` message by choosing a cipher/hash pair and the key share that will be used to establish the keying material. The server computes its key-share and sends it with all the choices back to the client labeled as a `ServerHello` message. In addition to this, the server must also send its signed certificate. The certificate must be signed with one of the signature algorithms supported by the client. The certificate is issued by a public-key infrastructure (PKI). If the server is not able to process the `ClientHello` message, the server sends a `HelloRetryRequest` to the client. This can happen when a client only sends key-shares from groups that the server does not support. The client is then forced to retry with a new `ClientHello` message. When the server successfully finds suitable parameters, the client verifies the signature and certificate and starts sending application data which is encrypted with the agreed upon key and cipher/hash pair. If everything works fine the handshake runs in 1-RTT.

In TLS 1.3, there is also an option for a 0-RTT handshake. If the keys are pre-shared from a previous session, then the session can be resumed with the same key. For this the client sends a pre-shared key label to the server. The server then can look for a suitable key and resume the session. The client is able to directly send application data with the pre-shared key.

TLS 1.3 Cipher Suites
TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_CCM_SHA256
TLS_AES_128_CCM_8_SHA256

Table 3.1: List of the TLS 1.3 cipher suites

3.1.2 Record Protocol

The record protocol starts after the handshake. With the chosen cipher suite the protocol encrypts and authenticates all the data that is sent by the application. Arriving data is decrypted with the same cipher suite and the integrity is checked. The supported cipher suites are listed in Table 3.1.

3.1.3 Cryptographic Operations

There are multiple types of cryptographic operations that are used in the protocol. The handshake performs a Diffie-Hellman key exchange which supports a set of 10 groups (5 finite field groups and 5 elliptic curve groups). For the certificate signature, TLS 1.3 supports 16 signature algorithms. And finally for the record protocol there are 5 cipher suites that are supported (Table 3.1). Each cipher suite is paired with a hash function that is used in the Hash-based key derivation function (HKDF). This function generates a suitable key with the shared material that was computed in the Diffie-Hellman key exchange. The cipher is a symmetric encryption scheme that uses the same key for the encryption and decryption. The cipher suites also authenticate each message and produce a tag that is used to check the integrity of the message upon arrival.

The cipher suites are the operations that are used the most. The other operations are only used during the handshake protocol which is only run in the beginning of the session. For this reason we decided to focus the benchmark on the encryption and decryption provided by the cipher suites.

3.1.4 Encryption Schemes

There are 5 cipher suites that are supported by TLS 1.3. They are all Authenticated encryption with associated data (AEAD) schemes. This means that the scheme computes a cipher text and additionally computes a message authentication code (MAC) that authenticates the data. With additional authenticated data (AAD) the security of the MAC is increased. The AAD can be seen as the context of the data. This way the receiver can check if the message comes from the correct context. The MAC is also called tag.

GCM

The Galois/Counter Mode (GCM) encrypts a series of counters with AES. The counters are based on a random initialization vector (IV). The encrypted counter are XORed with the plaintext which creates the ciphertext. The tag is created by multiplying the ciphertexts with a key-dependent constant (Figure 3.2. In TLS 1.3 the GCM tag always consists of 16 bytes. [16]

CCM

CCM is short for counter with cipher block chaining message authentication code or just counter mode with CBC-MAC. As a first step CCM computes the CBC-MAC of the message which creates the tag. Then the plaintext and the tag are encrypted using counter mode. In TLS 1.3 the CCM tag size can vary between 16 and 8. [12]

ChaCha20-Poly1305

ChaCha20 works in counter mode and generates a keystream that is computed with the key, nonce and counter. This keystream is then XORed with the plaintext to create the ciphertext. The ciphertext, the AAD and the key then form the input for the Poly1305 authentication function which computes the tag for the message. [11]

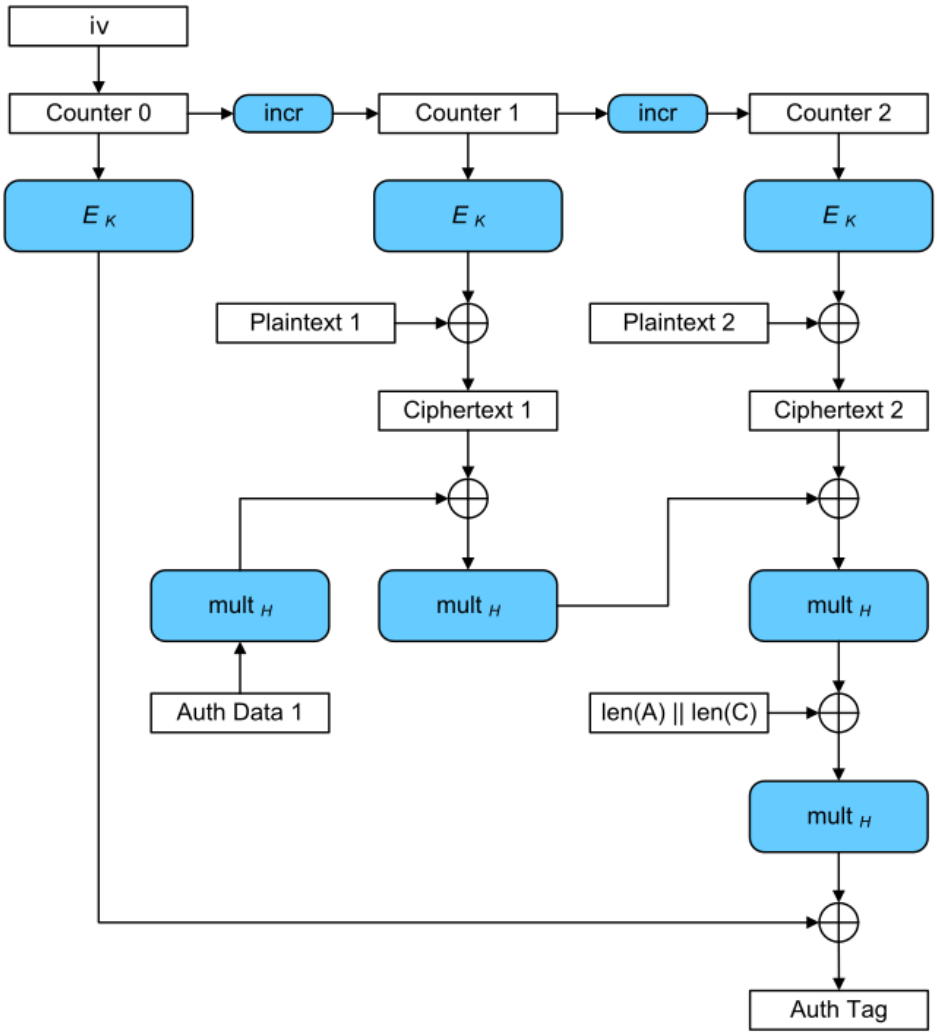

Figure 3.2: GCM encryption scheme (Image from Wikipedia [1])

Chapter 4

Implementation

In this section we will describe the implementation of the benchmark framework. We explain the basic structure with the software baseline benchmark and then describe the methods we used for the hardware implementation.

4.1 Software Benchmark

To be able to evaluate the performance of the hardware implementation, we created a baseline software benchmark. For the baseline we use the `EVP` functions of the `OpenSSL` library. `EVP` is a high-level interface for the `OpenSSL` cryptographic functions which makes them easier to use. For the benchmark we randomly generate messages, keys, initialization vectors (IV) and additional authenticated data (AAD). We run the benchmark with different configurations where the message size and the message numbers are varied. This allows us to evaluate the performance for smaller and bigger messages. The benchmark is run multiple times to statistically strengthen the results. Each run, we measure the execution time. With the execution time, we compute the throughput and the message rate. With a driver script we can run the benchmark with every possible configuration and save the data in `.csv` files. For the hardware benchmark, we created the same base structure and adapted the benchmark to run on the FPGA.

4.2 Hardware Benchmark

The behavior of the FPGA can be defined with a hardware description language (HDL) where a circuit is designed from scratch. This can become a very tedious process, especially for complex algorithms such as cryptographic operations. Fortunately AMD Xilinx provide the Vitis HLS (High-Level Synthesis) tool which greatly simplifys the process.

With the Vitis HLS tool the alogrithms can be implemented directly in `C/C++`. HLS functionality can be integrated by adding compiler directives. The program is then built with the `v++` compiler and synthesized for a specific FPGA device. HSL abstracts the circuit design away and makes FPGA configuration more accessible to developers [19]. Vitis also provides libraries that provide ready-to-use implementations of popular functions. In our case, we make use of the Vitis security library [20]. The library already contains implementations for all TLS cipher suites on which we will run the benchmark.

4.2.1 Implementation

For the benchmarks to run undisturbed, each cipher suite is implemented with a standalone benchmark program. Each benchmark program contains a host and a kernel component. The host component is run on the general-purpose processor. It generates the inputs and manages the kernel with the OpenGL API. The kernel is loaded onto the FPGA device and the host calls the kernel functions. In the end, the host stores the measured execution times into the respective `.csv` files for later data analysis.

The kernel contains the hardware implementation of the cipher. The specific function is implemented in the Vitis security library. In the building process the kernel is compiled into a `.xclbin` binary file which will be loaded onto the FPGA device. The CCM benchmarks are built with 300 Mhz clock frequency and due to more complexity the GCM benchmarks are built with 250 Mhz.

The main difference of the soft- and hardware implementation lies in the memory representations. The main work lies in providing the interface between the host and the kernel. While the software benchmark simply works with memory regions that are specified with pointers, the hardware benchmark mostly works with FIFO streams (Figure 4.1). A stream can be written and read one word at a time and works in a First-In First-Out (FIFO) fashion. For each stream a type must be defined which is the type of the words in the stream.

Before being able to fill up the streams, the inputs must be serialized into a continuous memory area. For this every parameter is organized as described in Figure 4.2. After this step the data can be loaded onto the device where the data can be read. For this the kernel parses the data and separates the data into FIFO streams that are later consumed by the function under test. The outputs are also FIFO streams that are again serialized. The serialized output is transferred back onto the host memory where it can parse out the resulting ciphertext and tag. The entire process is visualized in Figure 4.3.

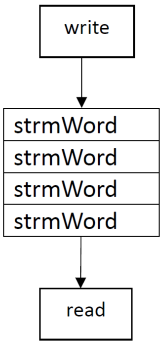

Figure 4.1: FIFO Stream

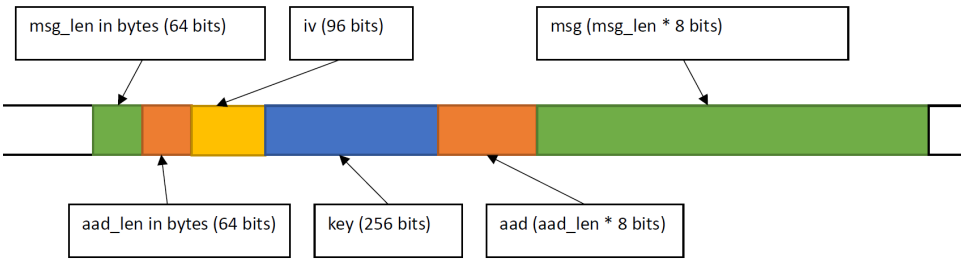

Figure 4.2: Serialized data for one message (example for GCM-256)

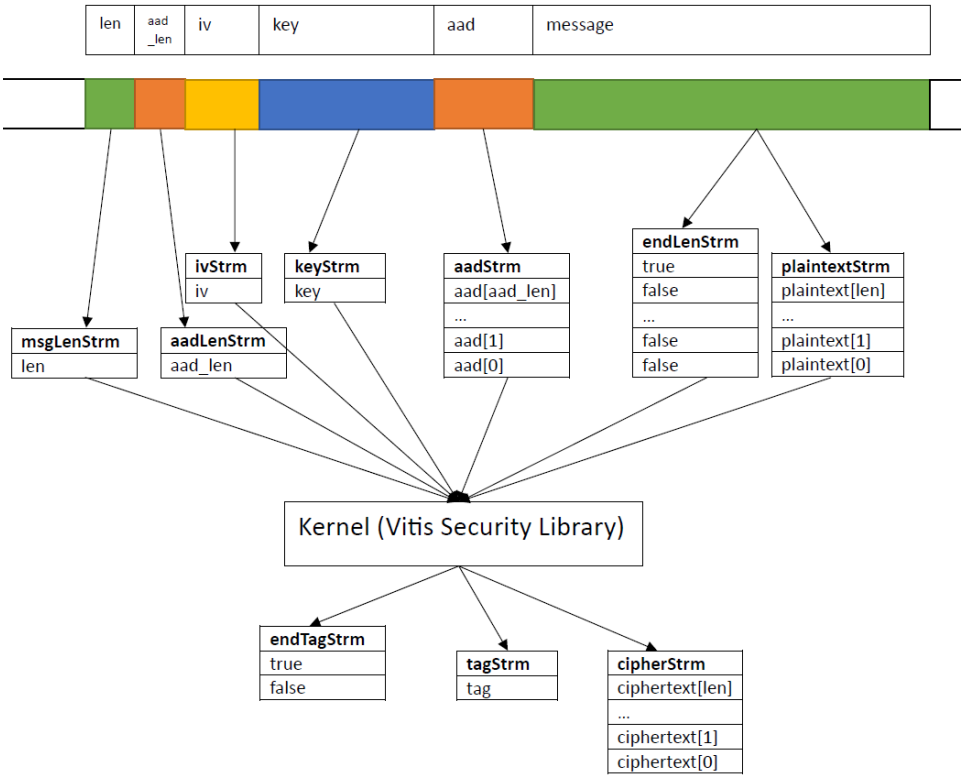

Figure 4.3: On top there is the serialized data as in Figure 4.2. The data is parsed into different streams that form the inputs to the kernel. The kernel is just the hardware implementation of the cryptogrpahic operation under test. The output are the ciphertext and tag as FIFO streams. The example is specific to GCM. For other ciphers the formatting is different but the core idea is the same.

Chapter 5

Experimental Setup

We run the benchmarks on one cluster node with one AMD Alveo U250 Data Center accelerator card with a Intel Xeon Processor (Cascadelake) host CPU. The software benchmark runs only on the general-purpose CPU and the hardware benchmark takes advantage of the FPGA to offload the cryptographic workload. The software benchmark runs on one CPU core and there is only one kernel loaded onto the FPGA. The benchmarks each are run 100 times for each of the 64 combinations of message sizes and message numbers. The options are 64, 128, 256, 512, 1024, 2048, 4096, 8192 bytes for the message sizes. The same goes for the message numbers. The biggest run goes through 8192 8192-byte messages.

Chapter 6

Experimental Results

In this section we take a look at the gathered measurements and compare the hardware version to the pure software version.

6.1 Data Analysis Approach

As described in section 5 the benchmarks are run 100 times for each combination of message sizes and message numbers. With the execution times measured in microseconds the throughput and the message rate are computed. The measurements lie mostly in the same range as can be seen in Figure 6.1. For this reason it is appropriate to compute the mean execution time for the further analysis. We look at both throughput and message rate. For each benchmark we fix the message number and show the trend with larger messages. The same process is done with fixed message sizes and varying message numbers.

6.2 AES-GCM

In this section we describe the behavior for the AES-GCM benchmarks. The behavior for the 128-bit and 256-bit block version are very similar. The encryption and decryption algorithms also show the same behavior. The hardware version reaches a maximal throughput of around 1.3 GB/s and the software implementation reaches a throughput of over 2.5 GB/s.

6.2.1 Throughput

For a fixed number of messages (Figure 6.2), the throughput increases exponentially on a log-scale. This means the performance grows linearly with the size of the messages. Therefore, smaller messages perform worse than large messages. The reason for this is that for each message a encryption

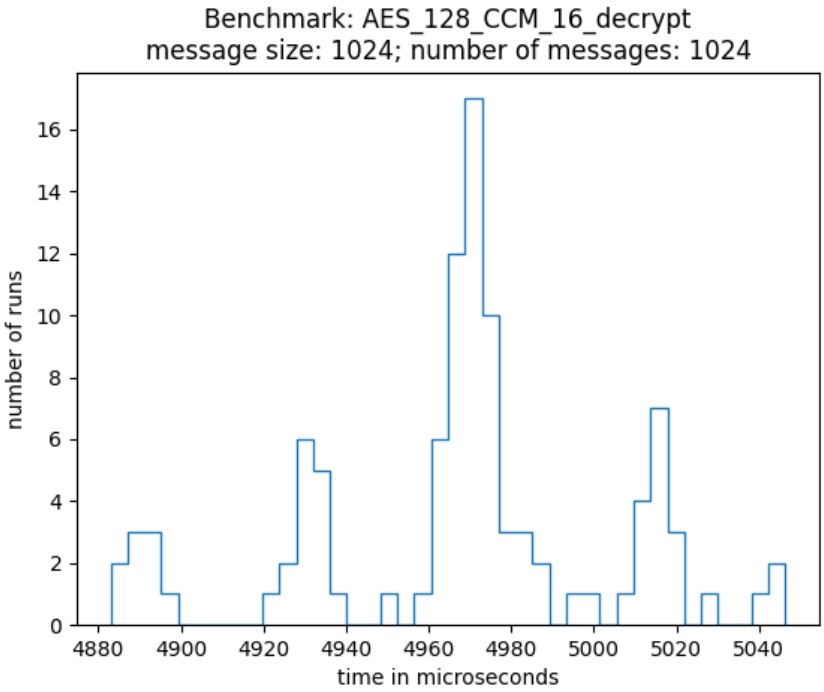

Figure 6.1: A histogram showing the distribution for the AES-128-CCM 16 decrypt hardware benchmark

operation is initiated. Each operation has some overhead where the computation is setup. So, in relation to smaller messages the overhead has a larger impact. This behavior can be observed for both the soft- and hardware benchmark.

The hardware benchmarks behave similar to the software version regarding their growth in throughput with larger message sizes. There is a bend in performance growth for messages larger than 2048 bytes where the performance stays constant for larger messages. The reason for this behavior lies in the implementation of the benchmark. As explained in section 4.2.1 the plaintext is written into a stream. In defining the stream, a size must be given which defines the number of words that can be stored in the stream at the same time. For all benchmarks we chose the size in a way that the largest tested message (which is 8192 bytes) fits completely into the stream. By increasing the message size the number of messages that fit into the plaintext stream decreases. Which means that not enough messages are stored in the stream ready for encryption which compromises the linear growth in performance for the hardware benchmark.

Another smaller difference between the two versions is that for smaller message numbers the performance in hardware is lower than software. But by increasing the number of messages, the hardware benchmark approaches and even surpasses the software benchmark. This can especially be seen when we fix the message size (Figure 6.3) and observe the throughput growth by increasing the message number. We can see that for large messages, the hardware benchmark performs much worse than the software version. This is another visualization of the bend in the performance growth we've seen above. For smaller messages we can see that the performance gradually approaches the software benchmark and even surpassing it in some cases. The fewer messages result in worse performance in hardware which is also a result of the overhead that comes in the hardware implementation. By fixing the message size, the throughput stays constant for the software implementation.

6.2.2 Message Rate

For a fixed number of messages (Figure 6.4) the message rate decreases linearly on a log-scale. This means the message rate decreases exponentially with increasing message sizes. Since larger messages take longer to encrypt not as many messages can be processed in the same time as for smaller messages.

For the message rates we can make similar observations as for the throughput. The behavior of both benchmarks is very similar. With smaller and lesser messages the hardware benchmark performs worse than the software version and improves in performance for larger tasks.

When fixing the message size (Figure 6.5) the software benchmark performs the same for any number of messages whereas the hardware version improves in performance with more messages provided. This further shows that the hardware benchmark is more comfortable with large tasks.

6.3 AES-CCM

In this section we describe the performance of the CCM benchmarks. We run the benchmarks for tag sizes 8, 12, 16 and observe the same behavior for the different tag sizes. The performance for encryption and decryption also behaves the same. The hardware version reaches a maximal throughput of around 200 MB/s and the software implementation reaches a throughput of over 1.2 GB/s.

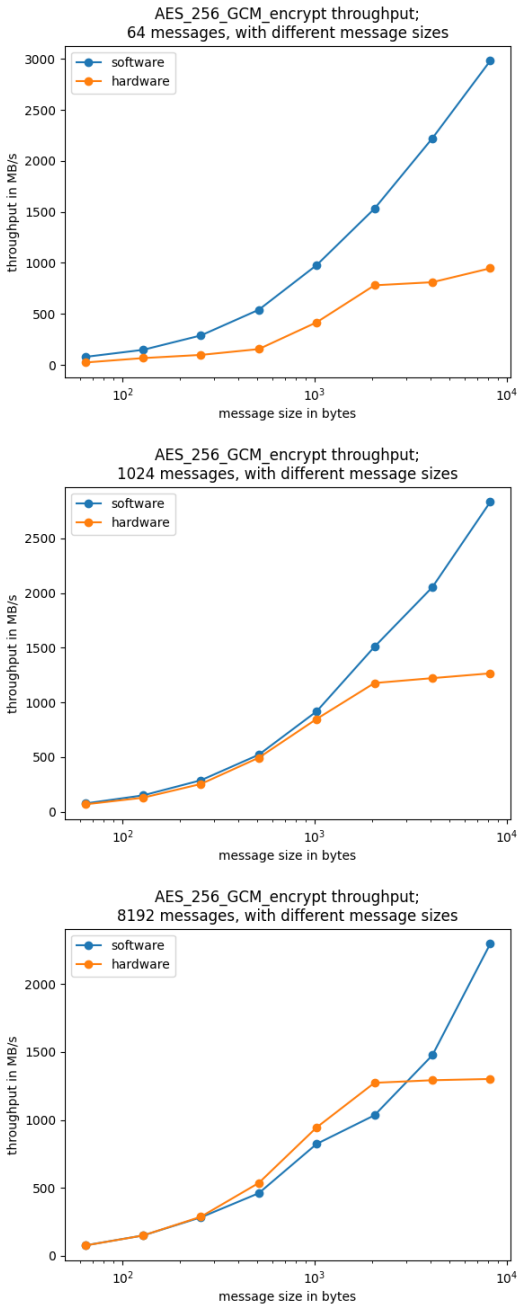

Figure 6.2: GCM throughput with fixed message number and increasing message size. 64, 1024, 8192 messages

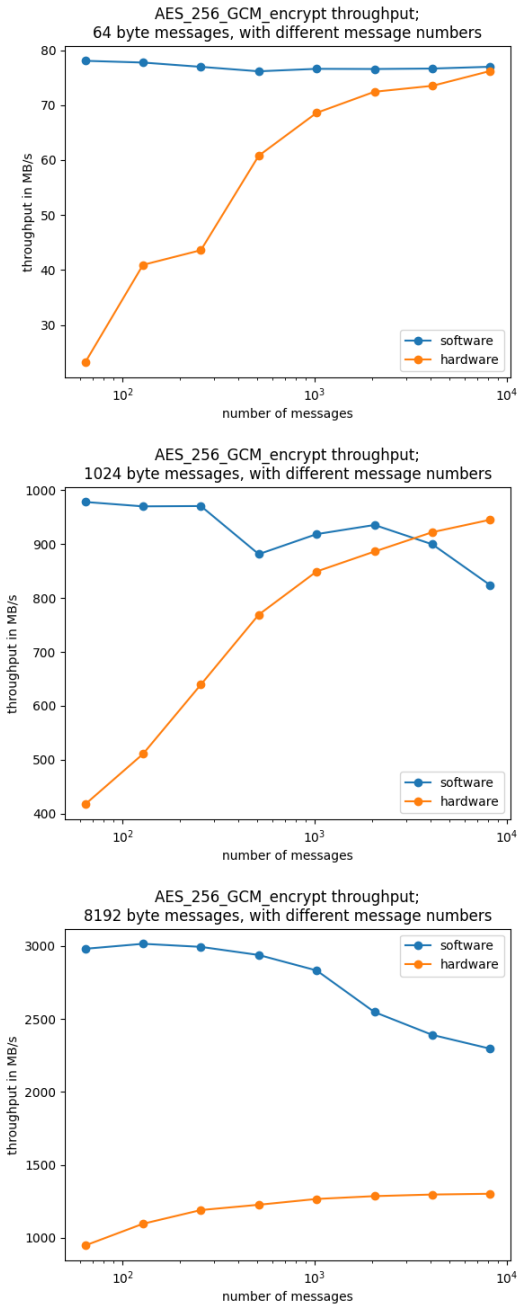

Figure 6.3: GCM throughput with fixed message size and increasing message number. 64, 1024, 8192 byte messages

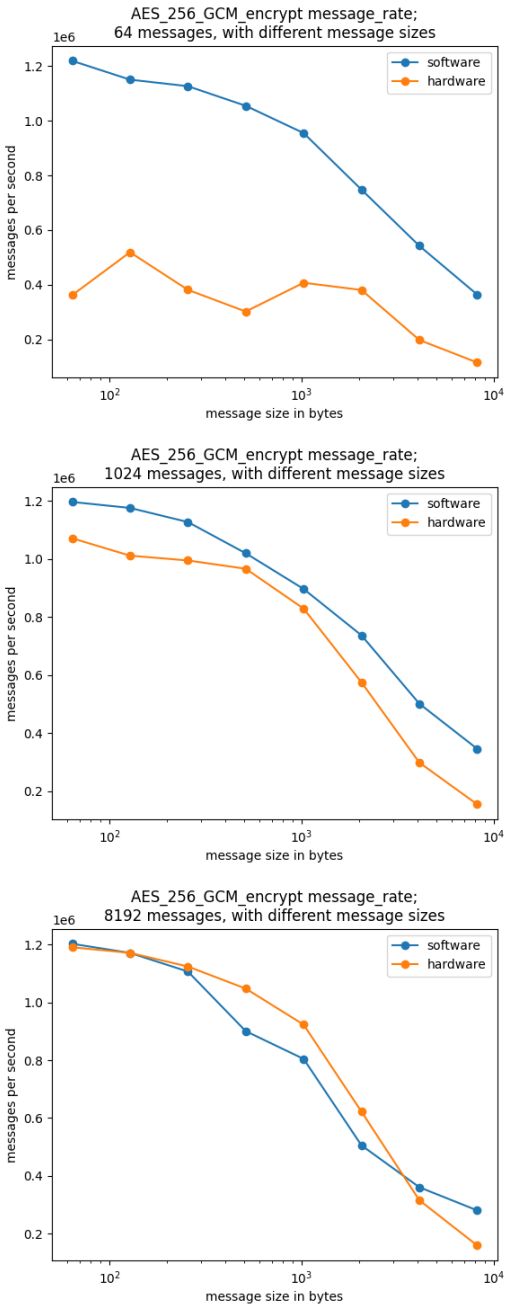

Figure 6.4: GCM message rate with fixed message number and increasing message size. 64, 1024, 8192 messages

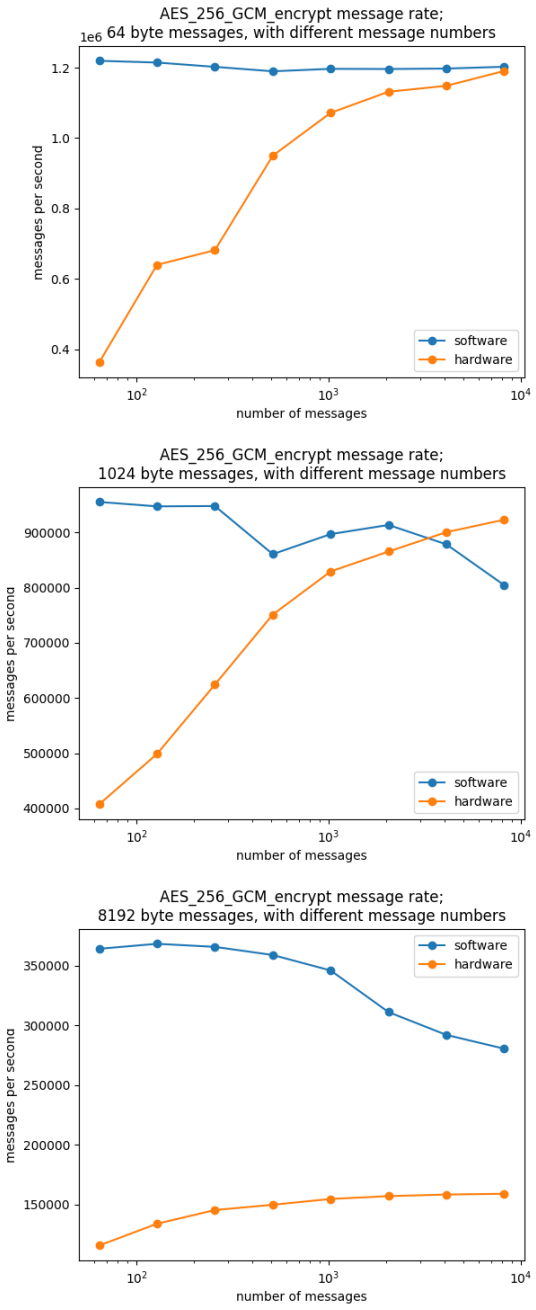

Figure 6.5: GCM message rate with fixed message size and increasing message number. 64, 1024, 8192 byte messages

6.3.1 Throughput

There is a big difference in performance between the software and hardware implementation. Both show linear performance growth in the log-scale. Which means a logarithmic growth in performance. The performance growth for the hardware lacks behind the software performance up to a factor of 5. For small messages the FPGA implementation is as performant as the software implementation but falls behind for larger messages. (Figures 6.6 and 6.7)

6.3.2 Message Rate

The same as in the GCM benchmark the message rate decreases with larger messages. With fixed message size and increasing message number the message rate stays constant in the software benchmark. In the same scenario the hardware performance increases steadily with the message number.

Overall, the message rate of the hardware benchmark also performs worse than the software implementation. The HLS implementation of CCM cannot keep up with the `OpenSSL` implementation. (Figures 6.8 and 6.9)

6.4 Resource Utilization

In table 6.1 we see the resource utilization of each benchmark. Each benchmark uses between 7.5 and 8.3% of the Look-Up Tables (LUT). The BRAM and DSP utilization is the same for all the programs. The reason for this are the identical memory requirements by the benchmarks. Since they all work on the same range of message sizes, they are all implemented with the same stream sizes. The differences in key size and tag size is not relevant because they do not make it necessary for larger word sizes. For example the tag size of 8 bytes is one word for a 128-bit wordsize tagstream. A tag size of 16 bytes also is one word in the tagstream.

For all benchmarks the LUT utilization is below 10%. The low resource utilization would allow to deploy multiple kernels on the same device. Either by allowing parallelism by deploying the same kernel multiple times or by broadening the available functionality by loading different kernels onto the device.

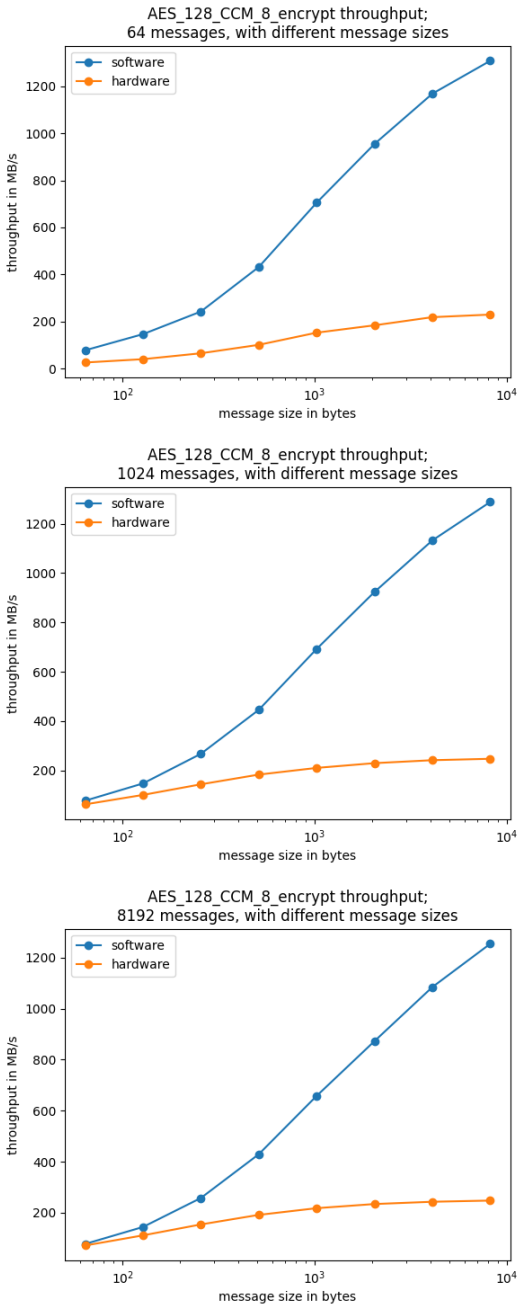

Figure 6.6: CCM throughput with fixed message number and increasing message size. 64, 1024, 8192 messages

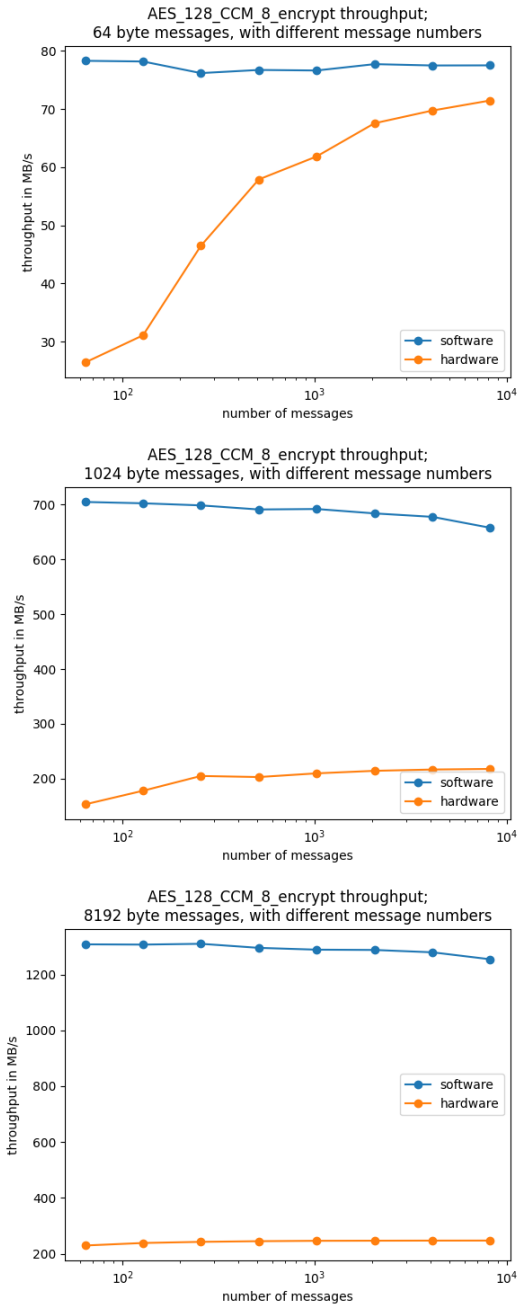

Figure 6.7: CCM throughput with fixed message size and increasing message number. 64, 1024, 8192 byte messages

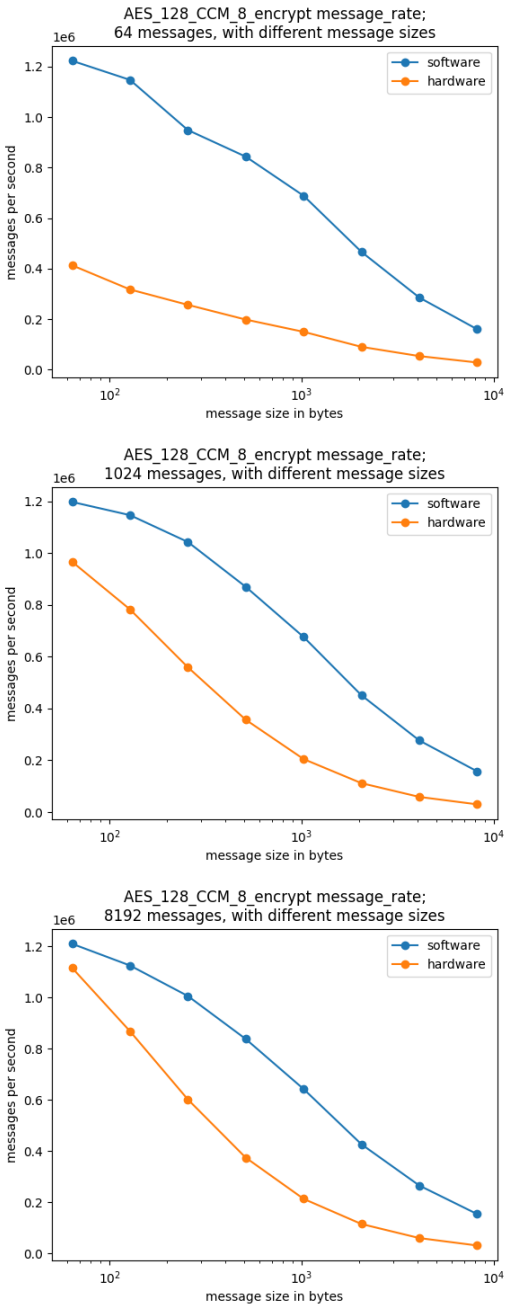

Figure 6.8: CCM message rate with fixed message number and increasing message size. 64, 1024, 8192 messages

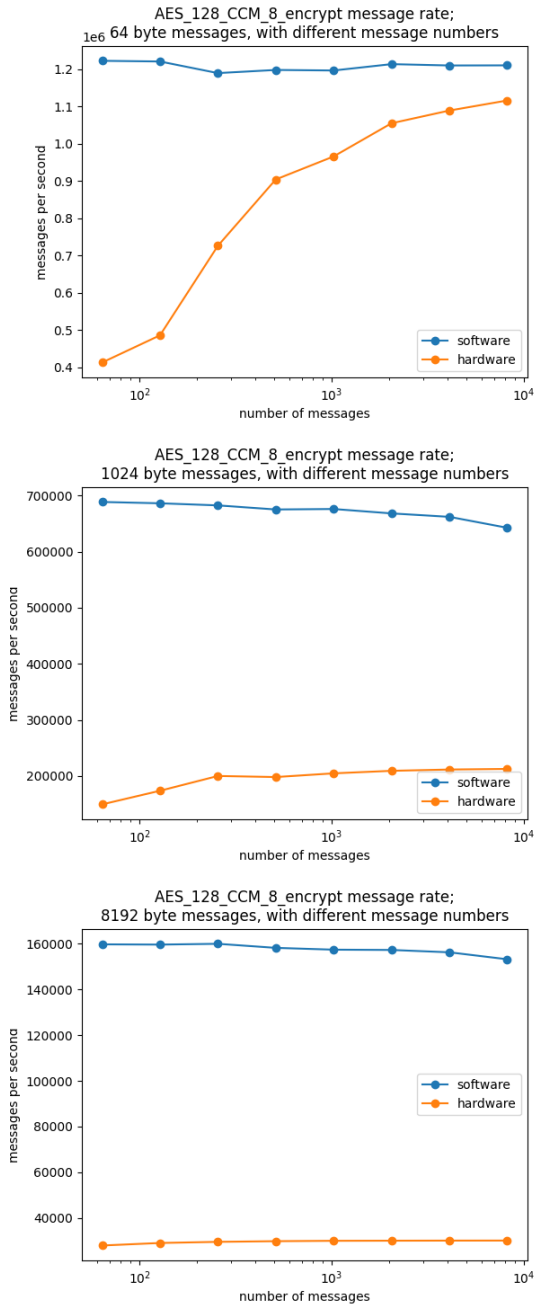

Figure 6.9: CCM message rate with fixed message size and increasing message number. 64, 1024, 8192 byte messages

	LUT (%)	BRAM (%)	DSP (%)
AES_256_GCM_encrypt	142984 (8.28%)	176.5 (6.57%)	7 (0.06%)
AES_256_GCM_decrypt	142945 (8.28%)	176.5 (6.57%)	7 (0.06%)
AES_128_GCM_encrypt	139054 (8.06%)	176.5 (6.57%)	7 (0.06%)
AES_128_GCM_decrypt	139142 (8.06%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_16_encrypt	130819 (7.58%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_16_decrypt	130821 (7.58%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_12_encrypt	129352 (7.49%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_12_decrypt	129342 (7.49%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_8_encrpyt	129183 (7.48%)	176.5 (6.57%)	7 (0.06%)
AES_128_CCM_8_decrypt	129223 (7.49%)	176.5 (6.57%)	7 (0.06%)

Table 6.1: Resource utilization for each kernel

Chapter 7

Summary and Conclusion

In this project we evaluate the performance of the cipher suites that are supported by the TLS 1.3 protocol. The cipher suites encrypt and decrypt the application data sent by the protocol and are the most used cryptographic operations in TLS 1.3. We evaluated the performance of two different implementations. A software implementation that is based on the `OpenSSL` library. For the hardware benchmark we make use of the Vitis security library that implements cryptographic functions with HLS. The computations are offloaded onto a FPGA device which frees up computation capacities on the CPU.

The AES-GCM encryption mode performs very similar for both implementations. For larger tasks the hardware benchmark surpasses the performance of the `OpenSSL` implementation. For the hardware benchmarks, enough memory must be specified to also perform well on large tasks or else it can't reach its full potential. At its peak the hardware benchmark reaches a throughput of 1.3 GB/s. Whereas the software implementation reaches a throughput of over 2.5 GB/s. For the AES-CCM encryption mode the hardware implementation performs up to 5 times worse than the software version and only reaches a throughput of 200 MB/s compared to 1.2 GB/s in the software implementation.

In the end, offloading cryptographic operations can result in better performance for some encryption schemes. Even though the performance cannot keep up for other operations, the main goal is to offload the computations on a separate device. This frees up capacity on the CPU which can be used for other tasks that are harder to offload to external devices.

7.1 Future Work

The goal for the future is to offload the entire protocol onto an FPGA device. The network functionalities are already provided by EasyNet [7]. A main challenge would be to implement the handshake protocol because of its complexity and high number of options. Also some cryptographic operations are not provided by the Vitis security library. Especially some signature algorithms and key exchange functionalities are missing.

Another direction would be to parallelize the computations on the FPGAs by deploying mutliple kernels on the same device. This way the performance could be increased massively. Looking at the resource utilization of the AES-256-GCM kernel, there would be space for 12 kernels on the same device.

The exact measurements and a complete collection of plots can be accessed through the `gitlab` repository.
`https://gitlab.inf.ethz.ch/zhe/crypto-bench-tls`

List of Figures

List of Tables

Bibliography

[1] Galois/counter mode - wikipedia. https://de.wikipedia.org/wiki/Galois/Counter_Mode. Accessed on 1st June 2023.

[2] Blaise Tine Hyesoon Kim Austin Adams, Pulkit Gupta. Hardware acceleration for cryptography algorithms by hotspot detection. 2021.

[3] Gouguec Kévin Blanc Soline, Lahmadi Abdelkader, Minier Marine, and Sleem Lama. Benchmarking of lightweight cryptographic algorithms for wireless iot networks. 2022.

[4] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. 2016.

[5] Liu Chen Chang Jed and Gaudiot Jean-Luc. Hardware acceleration for cryptography algorithms by hotspot detection. 2013.

[6] Google. Https encryption on the web. https://transparencyreport.google.com/https/overview?hl=en, 2023. Accessed on 31st May 2023.

[7] Korolija Dario He Zhenhao and Alonso Gustavo. Easynet: 100 gbps network for hls. 2021.

[8] Ralph Holz, Johanna Amann, Abbas Razaghpanah, and Narseo Vallina-Rodriguez. The era of tls 1.3: Measuring deployment and use with active and passive methods. 2019.

[9] Amann Johanna Razaghpanah Abbas-Jost Thomas Vallina-Rodriguez Narseo Holz Ralph, Hiller Jens and Hohlfeld Oliver. Tracking the deployment of tls 1.3 on the web: A story of experimentation and centralization. 2020.

[10] Doowon Kim Hyunwoo Lee and Yonghwi Kwon. Tls 1.3 in practice: How tls 1.3 contributes to the internet. 2021.

[11] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. Chacha20-poly1305 cipher suites for transport layer security (tls). RFC 7905, June 2016.

[12] David McGrew and Daniel Bailey. Aes-ccm cipher suites for transport layer security (tls). RFC 6655, July 2012.

[13] NIST. Lightweight cryptography. https://csrc.nist.gov/Projects/lightweight-cryptography, 2018. Accessed on 1st June 2023.

[14] Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, August 2018.

[15] Eric Rescorla and Tim Dierks. The transport layer security (tls) protocol version 1.2. RFC 5246, August 2008.

[16] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. Aes galois counter mode (gcm) cipher suites for tls. RFC 5288, August 2008.

[17] Enrico Pozzobon Sebastian Renner and Jürgen Mottok. Benchmarking software implementations of 1st round candidates of the nist lwc project on microcontrollers. 2019.

[18] Yu Chia-Mu Ti Yen-Wu, Wu Chia-Feng and Kuo Sy-Yen. Benchmarking dynamic searchable symmetric encryption scheme for cloud-internet of things applications. 2020.

[19] Xillinx. Vitis high-level synthesis. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.htmloverview, 2023. Accessed on 31st May 2023.

[20] Xillinx. Vitis security library. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-security.html, 2023. Accessed on 31st May 2023.