CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Securing 5G URLLC

Exploring Secure Alternatives to Traditional Error Detection in Ultra Reliable Low-Latency Communication

Master's thesis in Computer science and engineering

Dennis Sehic and Talha Ahmad

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Securing 5G URLLC

Exploring Secure Alternatives to Traditional Error Detection in Ultra Reliable Low-Latency Communication

Dennis Sehic and Talha Ahmad

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Securing 5G URLLC
Exploring Secure Alternatives to Traditional Error Detection in Ultra Reliable Low-Latency Communication
Dennis Sehic and Talha Ahmad

Supervisor: Erik Ström, Department of Electrical Engineering
Examiner: Elena Pagnin, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Securing 5G URLLC
Exploring Secure Alternatives to Traditional Error Detection in Ultra Reliable Low-Latency Communication
Dennis Sehic and Talha Ahmad
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Fifth-generation (5G) wireless networks are networks that implement the 5G telecommunication standard. 5G networks feature a service called *Ultra Reliable Low-Latency Communication* (URLLC), which requires messages to arrive in under 1 millisecond. Resource-constrained devices may not be able to implement common security algorithms suggested for URLLC and will have to communicate insecurely. As it stands, the integrity of URLLC is most often achieved with a Cyclic Redundancy Check, which is malleable and can be generated for any arbitrary message by an adversary. In this thesis, a number of established security algorithms are benchmarked and evaluated for use in URLLC. Results show that Ascon and ChaCha20-Poly1305 are viable authenticated encryption algorithms, SipHash-2-4 is a viable integrity algorithm, and AES-GCM can be used for authenticated encryption only if hardware acceleration is available. The algorithms were tested against one set of standard encryption and integrity algorithms from fifth-generation (5G) wireless networks.

Keywords: Computer, science, Security, Cryptography, URLLC, 5G, Networks, IoT, engineering, thesis.

Acknowledgements

First and foremost we would like to thank our supervisor Erik Ström and our examiner Elena Pagnin, who have both been very involved in the project and supported it until completion. Naturally, we would also like to thank Carl Ridderstolpe and Tim Kinderby for providing feedback on our Thesis and pointing out flaws as the opposition.

Talha thanks Prof. Tischhauser from the University of Marburg in Germany for sparking his interest in IT security. Finally, Talha also thanks Ph.D. stuident Francisco Blas Izquierdo Riera for supporting him with his journeys in Computer Security during and beyond his academic career.

Dennis Sehic and Talha Ahmad, Gothenburg, 2025-09-05

Contents

List of Figures

List of Tables

Dictionary

3GPP	*3rd Generation Partnership Project.* Collaboration of global organization that develop and maintain technical standards for mobile communication.
5G	*Fifth-Generation Wireless Network.* This is currently the most broadly used mobile communication at the time of writing: 2025-09-05.
5G NR	*Fifth-Generation Wireless Network, New Radio.* A newer version of 5G.
AE	*Authenticated Encryption.*
AEAD	*Authenticated Encryption with Associated Data.*
AES-GCM	*Advanced Encryption Standard in Galois-Counter Mode.* AES is a standardized block-cipher, and Galois-Counter Mode is a way of chaining the block-wise encryption and also adding an integrity check.
AES-NI	*Advanced Encryption Standard New Instructions.* Hardware-Support for the Advanced Encryption Standard.
ARX	*Addition - Rotation - XOR.* Is a description for an algorithm that uses only the 3 types of instructions.
CC0	*Creative Commons.* Is an open-source license.
CPU	*Central Processing Unit.*
DES	*Data Encryption Standard.* Deprecated encryption function. Replaced by AES.
DoS	*Denial of Service.*
CLI	*Command Line Interface.*

eMBB	*enhanced Mobile Broadband.* One of three 5G services. Focuses on high data volume and user experience.
ETSI	*European Telecommunication Standards Institute.* One of the 3GPP organizations as well.
GNU Binutils	*Binary Utilities by the GNU Project.* Usually available on Linux.
HMAC	*Hashed Message Authentication Code.*
HRLLC	*Hyper Reliable Low-Latency Communication.* 6G counterpart to the 5G service URLLC.
IEEE	*Institute of Electrical and Electronics Engineers.* Known to publish professional and academic research.
IoT	*Internet of Things.*
ISC	*Internet Systems Consortium.* Another open-source license
IV	*Initialization Vector.* Random bitstring used in cryptographic algorithms.
MAC	*Message Authentication Code.*
MD5	*Message-Digest Algorithm 5.* Deprecated secure hashing algorithm, replaced by SHA.
mMTC	*massive Machine Type Communication.* One of three 5G services. Focuses on high numbers of users, usually machines.
NIST	*National Institute of Standards and Technology.*
Nonce	*Number used Once.* Random bitstring used in cryptographic algorithms.
OTP	*One-Time Pad.* Encryption scheme where an infinitely long random number is XOR'd with the message.
PERF	*PERFormance.* Process analysis tool for the Linux CLI.
PRF	*Pseudo-Random Function.*
PRP	*Pseudo-Random Permutation.*
RAM	*Random Access Memory.*

RDTSC	*Read Time-Stamp Counter.* An assembly instruction that reads the CPU's clock cycle count.
S-Box	*Substitution Box.* A substitution function used in cryptographic algorithms, expected to be non-linear for security.
SHA	*Secure Hashing Algorithm.* A family of standardized cryptographic hash functions: SHA-1, SHA-2, SHA-3.
UEA	*UMTS Encryption Algorithm.*
UIA	*UMTS Integrity Algorithm.*
UMTS	*Universal Mobile Telecommunication Systems.*
URLLC	*Ultra Reliable Low-Latency Communication.* One of 3 5G services. Focuses on low latency and reliability.
XOR	*exclusive-OR function.* Defined by the function $f(a,b) = a \oplus b$.

1 Introduction

1.1 Background

The increasing use of smart technologies, from health sensors to industrial automation systems to autonomous vehicles, often referred to as the Internet of Things (IoT), has driven the demand for low-latency and high-reliability communication infrastructures.

Fifth-generation (5G) wireless networks are networks that implement the 5G telecommunication standard. Developments in 5G wireless networks have attempted to solve this demand in a unique way. 5G consists of three main services: eMBB, mMTC, and URLLC. Enhanced Mobile Broadband (eMBB) is designed for high data speeds and high data throughput. massive Machine Type Communication (mMTC) is geared towards connecting a vast number of devices, often with minimal human interaction. The focus of this thesis is on Ultra-Reliable Low Latency Communication (URLLC) as it lacks standardized and tailored security algorithms.

Unlike conventional communication systems, URLLC is characterized by strict performance demands, including end-to-end latency often below 1 millisecond, and high reliability with an error rate lower than 10^{-5} [1]. It is not suitable for transmitting high volumes of data, but rather for mission-critical systems that require reliable real-time communication for use cases such as autonomous driving, healthcare, surgery, stock trading, and smart grids among many other scenarios [1].

1.2 Problem Statement

As the latency requirements of URLLC push message length to smaller sizes, conventional secure cryptographic algorithms generate too much overhead to be used, as stated in [2]. The security specifications in 5G New Radio (NR) implement well-researched and tested security protocols, but these face the same runtime overhead problems, making them unusable for many resource-constrained devices for use in URLLC.

To allow for secure communication in URLLC settings with resource-constrained devices, the following questions will be answered in this thesis, along with a main question:

- Which authentication and authenticated encryption (AE) algorithms are fast enough to be run on IoT devices in a URLLC setting? How fast are they on popular IoT devices?
- What is the security level that these algorithms provide? Can compromises be made on the security level in exchange for speed?
- **Main Question**: What authentication and authenticated encryption algorithms can be effectively implemented in 5G IoT devices within the constraints of URLLC networks?

In the future, URLLC will likely be used more due to the increased integration of technology into critical infrastructure, which requires a good level of security. Our focus on 5G is explained by its potential to be a stepping stone for future research in 6G networks. 6G is in its early development stage and will offer a service called Hyper Reliable Low-Latency Communication (HRLLC), with even higher reliability and lower latency. If it is at all possible to achieve secure communication in 6G, then the findings in 5G URLLC will be a stepping stone for this progress.

Another consideration is the lackluster security research in 5G URLLC, likely because the IoT is in its development stage. It is not far-fetched to think that research in 5G URLLC security will accelerate the widespread implementation of IoT devices using URLLC.

1.3 Overview

We were surprised by the low amount of public research done in the field of URLLC security. 3GPP does not allow the public to view their research [3]. Other sources do not yield useful results, as they are either a surface-level overview of restricted 3GPP research [4] or suggest using a concept called Physical Layer Security [5], which has assumptions that are unrealistic for current networks.

Results will be achieved by first benchmarking the algorithms and subsequently calculating the required processor speeds to justify their use in URLLC. This result will then be cross-referenced with available IoT devices today and their processor speeds. Additionally, RAM and flash memory usage will be measured to find out if they will be able to run on IoT devices in the first place. In this thesis, no physical IoT devices are used to measure the runtime of the algorithms. All experiments are carried out virtually. Algorithms are benchmarked for payload sizes often seen in URLLC services, which are between 1 - 32 bytes. This approach mimics how two entities would communicate virtually, and allows us to accurately predict the viability of a security scheme.

Other research has also benchmarked these algorithms, but most measurements are typically done for large volumes of data. The benchmarks for large data volumes can not be translated to performance on URLLC data lengths, as it masks the overhead an algorithm may have. Also consider the fact that most implementations are designed for large data volumes, and may for various reasons perform poorly for short messages.

2

Theory

Cryptography is the practice of protecting information from unauthorized access, tampering, reading, or manipulation. This is done by developing security schemes that satisfy a security notion based on logical reasoning and rigorous definitions of the security notion. A security notion can be quantifying the "likelihood of successfully hiding information from an adversary" or the "likelihood of denying a tampered message" of an algorithm. In order to allow legitimate users to achieve this security advantage, random numbers are used. Random number generators, or key generators are the source of secrets and therefore the heart of cryptography. It is assumed that the adversary does not know the secret, which is why the random number generation needs to be as unpredictable as possible to prevent secret guessing. These random numbers can be known as keys, nonces (number used once), or an IV (Initialization Vector) based on their purpose. Keys are usually kept secret, nonces and IVs can be made public and serve the purpose of ensuring resistance against various attacks.

Different algorithms have different security features and satisfy different security notions. The chosen algorithms and their roles are listed in Table 2.1. Justifications for our selection of algorithms are found in the chapter "Methodology".

Side-channel attacks are excluded, which are attacks that don't exploit the underlying cryptographic concept. When excluding side-channel attacks, all algorithms in Table 2.1 have no known vulnerabilities at the time of writing. This is important, because then the upper bound for the security estimate will be $2^{|k|}$, with k being the binary representation of the key. This estimate boils down to random guessing of the key.

Algorithm	Cipher	MAC
AES-GCM	✓	✓
SipHash	✗	✓
Ascon-AEAD	✓	✓
ChaCha20	✓	✗
Poly1305	✗	✓
HMAC-SHA256	✗	✓

Table 2.1: Listing of what functionalities certain algorithms provide. They may later be combined to provide more security features in one package.

2.1 Notation

A bitstring S is a concatenation of an arbitrary number of bits $s \in \{0,1\}$. The empty bitstring is denoted as ϵ. The length of a bitstring S is denoted as $|S|$, much like the cardinality of a set. The length of a bitstring S is defined as 0 if $S = \epsilon$. Otherwise, if S has at least one prefix bit S_{pre} and a postfix bitstring S_{post}, then it is defined as $|S| = 1 + |S_{post}|$. Bitstrings may be divided into *blocks*, each block has the same fixed length and is an ordered partitioning of consecutive bitstrings. $S_n, n \in \mathbb{N}$ denotes the n-th bit or the n-th block of S, depending on context.

$\{0,1\}^*$ denotes the set of all bitstrings of arbitrary non-negative length, while $\{0,1\}^n, n \in \mathbb{N}$ is the set of all bitstrings of non-negative length n. Z_n with $n \in \mathbb{N}$ represents the set $[0, n-1] \cap \mathbb{Z}$ as in modular arithmetic.

The binary function $\oplus$ is the binary exclusive-or (XOR) operation. If the XOR operation is used on hexadecimals instead, then it is defined as XOR-ing the binary representation of the hexadecimals in question. The binary function $\ll$ takes a bitstring a and an integer i, and results in bit-shifting a to the left i times. The leftmost i values are discarded and the new i bits on the right are zeros. Right shifting $\gg$ is defined accordingly to the right. The binary function $+$ is the modular addition. It takes two bitstrings $a, b \in \{0,1\}^n$, and returns the value $a + b \texttt{ mod } 2^n$ as a bitstring. When $+$ is not used with bitstrings, then its definition will be based on the usual mathematical context. The binary function $||$ is the concatenation function on bitstrings.

Let $\mathsf{M}, \mathsf{C}, \mathsf{K} \subseteq \{0,1\}^n$ be the messagespace, cipherspace and keyspace respectively.

2.2 Symmetric Encryption

Encryption is a process that transforms readable data known as plaintext, into an unintelligible string called ciphertext using an algorithm and a secret key. The goal is to protect the confidentiality of data. Let $v \in \mathbb{N}$ be the security level. A symmetric encryption scheme formally consists of three algorithms: The key generation algorithm $\mathsf{KeyGen} : v \to \mathsf{K}$, the encryption function $\mathsf{Enc} : \mathsf{K} \times \mathsf{M} \to \mathsf{C}$ and the decryption function $\mathsf{Dec} : \mathsf{K} \times \mathsf{C} \to \mathsf{M}$. Given $m \in \mathsf{M}, c \in \mathsf{C}, k \in \mathsf{K}$, the scheme must guarantee the following property for correctness:

$$\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$$

Encryption aims to primarily hide the original plaintext. That is why encryption schemes are expected to satisfy the following security notion.

Indistinguishability from random permutations: Let $p \in \mathbb{N}$ be polynomial in the security level v. Given a challenge ciphertext c_{chall}, and a polynomial number p of message-ciphertext pairs where $((m_0, c_0), (m_1, c_1), ..., (m_{p-1}, c_{p-1})) \in (\mathsf{M} \times \mathsf{C})^p$ with the constraints $\forall i \in Z_p : c_{\mathsf{chall}} \neq c_i$ and $\mathsf{Enc}_k(m_i) = c_i$, it is computationally hard to find a plaintext m_{chall} such that $c_{\mathsf{chall}} = \mathsf{Enc}_k(m_{\mathsf{chall}})$. Enc_k and Dec_k are the encryption and decryption algorithms that have k encoded into the algorithm as a constant, but can't be inspected by the adversary. These types of black-box algorithms are called *oracles*.

Alongside symmetric encryption, there is also asymmetric encryption. In symmetric encryption, the encryption and decryption key are the same. In asymmetric encryption, the encryption and decryption keys are unique. Symmetric encryption is more suitable for our purpose, as it is generally faster and is much more likely to satisfy the URLLC latency requirement.

In many cases, symmetric encryption will operate on a block of bits, which is referred to as a *block cipher*. Its counterpart is a *stream cipher*, which encrypts bit-by-bit.

2.3 Message Authentication Codes

A Message Authentication Code (MAC) is a cryptographic primitive that ensures the integrity and authenticity of a message. Again, let $v \in \mathbb{N}$ be the security level. A MAC scheme formally consists of a set of algorithms: The key generation algorithm $\mathsf{KeyGen} : v \to \mathsf{K}$, one tag-generation algorithm $\mathsf{MAC} : \mathsf{K} \times \mathsf{M} \to \mathsf{T}$ and one tag verification algorithm $\mathsf{Ver} : \mathsf{K} \times \mathsf{M} \times \mathsf{T} \to \{0, 1\}$ where T is the tag space. The verification algorithm returns 1 if tag verification is successful, and 0 otherwise. Given $m \in \mathsf{M}, k \in \mathsf{K}$, the scheme must guarantee the following property for correctness:

$$\mathsf{Ver}(k, m, \mathsf{MAC}(k, m)) = 1$$

MAC schemes want to ensure that a legitimate sender (the one holding the key k) is verifiable and that message tampering can be detected. That is why MAC schemes are expected to satisfy the following security notion.

Tag Unforgeability: Let $p \in \mathbb{Z}$ be polynomial to the securitz level v. Given a polynomial number p of message-tag pairs $((m_0, t_0), (m_1, t_1), ..., (m_{p-1}, t_{p-1})) \in (\mathsf{M} \times \mathsf{T})^p$ with the constraints $\forall i \in Z_p : \mathsf{MAC}_k(m_i) = t_i$, it is computationally hard to forge a new message-tag pair $(m_{\mathsf{new}}, t_{\mathsf{new}}) \in (\mathsf{M} \times \mathsf{T})$ such that $\forall i \in \mathbb{Z} : m_{new} \neq m_i$ and $\mathsf{Ver}_k(m_{\mathsf{new}}, t_{new}) = 1$. This is called *existential unforgeability* if the message-tag pairs can be chosen by the adversary and is a stronger security notion. Ver_k and MAC_k are the verification and tag generation oracles.

When combining Encryption-then-MAC verification, it is possible for communication partners to: make the ciphertext message unintelligible for illegitimate users *(via encryption)*, verify that the communication partner sent the message *(via MACs)*, and verify that the message contents have not been changed *(via MACs)*. In our implementations, an **Encrypt-then-MAC** combination is used, which is more secure. When considering the security estimate of a tag, the upper bound of the security estimate is $min(2^{|k|}, 2^{|t|})$ for a single message, with $k \in \mathsf{K}$ being the key and $t \in \mathsf{T}$ being the tag. With random guessing, if the tag is shorter than the key, then the tag is easier to guess.

2.4 Cryptographic Hash Functions

Cryptographic hash functions are generally used for signatures and checksums. In particular, they can serve as a tool for message authentication codes. A cryptgraphic hash function H is defined as $H : M \rightarrow \{0,1\}^d$, where $d \in \mathbb{N}$ is the digest length. The digest length is preset, therefore every hash value using a particular hash algorithm has the same length. Naturally, this means that *hash collisions* are unavoidable; that is, two input strings can have the same hash value. When hashing $2^d + 1$ strings, at least one hash collision is guaranteed. Cryptographic Hash functions are expected to satisfy the following security notions.

Preimage resistance: Given a hash function H hash value h, it is computationally hard to find an input $s \in \mathsf{M}$ such that $H(s) = h$

Second pre-image resistance: Given a string $s_1 \in \mathsf{M}$, it is computationally hard to find a unique string $s_2 \in \mathsf{M}$ such that $s_1 \neq s_2$ and $H(s_1) = H(s_2)$

Collision resistance: It is computationally hard to find a hash collision. In other words, it is hard to find two unique strings $s_1, s_2 \in \mathsf{M}$ such that $s_1 \neq s_2$ and $H(s_1) = H(s_2)$

With the above properties, it becomes hard for adversaries to forge tags if the hash function is used as part of the MAC scheme. In that scenario, the difficulty of forging tags is reducible to the difficulty of finding a pre-image resistance.

2.5 Advanced Encryption Standard - Galois/Counter Mode

The *Advanced Encryption Standard - Galois/Counter Mode* (AES-GCM) uses the AES block cipher in Galois Counter mode for both confidentiality and integrity. AES is a symmetric-key block cipher using 128, 192, or 256-bit blocks, and has been a standard set by the *National Institute of Standards and Technology* (NIST) since 2000 [6]. AES also happens to be a widely used algorithm that appears in TLS 1.3. As a consequence, many devices have started integrating dedicated hardware designed to execute AES operations with more speed and efficiency. This hardware is called *Advanced Encryption Standard New Instructions* (AES-NI) in Intel processors. Note that AES is known to have side-channel vulnerabilities [7], but these are mitigated by the dedicated AES-NI hardware.

2.6 Sponge construction

Sponge construction was developed by the Keccak team as a flexible cryptographic framework [8]. The construction can be boiled down to a randomized Finite State Machine. The flexibility comes from its property of being able to handle variable-sized inputs and outputs as bitstrings, as well as being able to construct various cryptographic schemes like key generation, stream ciphers, and MACs among other things.

Input and output size can be adjusted using a length parameter $l \in \mathbb{N}$, the security level can be chosen using a capacity $c \in \mathbb{N}$, and block size can be set using a bitrate $r \in \mathbb{N}$. Now, let the state size be defined as $b = r + c$. A sponge construction $\texttt{sponge}(f, r, \texttt{pad})$ utilizes a transformation or permutation function $f : \{0,1\}^b \to \{0,1\}^b$, a bitrate r and a padding function $\texttt{pad} : \{0,1\}^s \to \{0,1\}^{ri}, s > \mathbb{N}, i \in \mathbb{N}, ri > s$. The padding function exists to prepare the input to be split into r-bit blocks.

The construction is split into an absorption and squeezing phase. The purpose of the absorption phase is to continue taking input while permuting or transforming its inner state. The input message is processed with the *rate* r. Increasing r increases the speed at which the construction operates since it influences the number of blocks to be processed, which in turn influences the number of instructions and iterations. The capacity c is directly proportional to the attainable security level of the construction. c is the number of bits that do not interact with the input block at a given iteration and is therefore "hidden" from the adversarial attacker. Finally, the squeezing phase produces output at the same bitrate r and keeps permuting the inner state with the same security level c. The instructions executed in a sponge construction are shown in Algorithm 2.1

```
Input: f, r, pad, m
Require: r < b

// __________ Initialization __________
state[0:b] = 0 // 0-initialized state
n = 0

// __________ Absorption phase __________
// Initialize I to be r-bit blocks of pad(m)
while (n * r < length(pad(m)) - 1) {
    state[0:r] = state[0:r] xor I[n]
    state = f(state)
    n++
}

// __________ Squeezing phase __________
out = empty_string
while (length(out) < l) {
    out = out || state[0:r]
    state = f(state)
}

return first-l-bits(out)
```

Algorithm 2.1: Pseudocode for sponge construction

Implementations may also choose to output results on the fly during the squeezing phase. Note that the security of the construction heavily relies on the choice of the function f. The purpose of the sponge construction is to provide a "wrapper" for secure fixed-input to fixed-output functions to be used as a secure arbitrary-input to arbitrary-output function. This characteristic allows the construction of hash functions, MACs, and ciphers from a permutation or transformation function f. The security of an algorithm implementing the sponge construction is reducible to the security of the function f. The sponge construction inherits all security properties of f [8].

2.7 Duplex construction

The Duplex construction was presented as an alternative to the sponge construction for developing authenticated encryption schemes. Sponge constructions can act as a cipher/authenticated encryption scheme if $l > |M|$. However, it may not be necessary to have a separate absorption and squeezing phase, since one can simply interleave absorption and squeezing. Duplex constructions guarantee the same security promises as sponge constructions. A single interleaved absorption and squeezing action is referred to as a duplex. Originally, the duplex construction takes l as a list of lengths, which is as long as the number of blocks in `pad`(M). Its purpose is to allow more flexibility. In our project, simplifying is reasonable, so a single constant length l is used. The instructions executed in a duplex construction are shown in Algorithm 2.2

```
Input: f, r, pad, m, l
Require: l <= r < b

// __________ Initialization __________
state[0:b] = 0 // 0-initialized state

// __________ Duplexing __________
define duplex(f, pad, msg, l):
    Require: |msg| <= r
    state[0:r] = state[0:r] xor msg // Absorb
    state = f(state)
    return first-l-bits(state) // Squeeze

// __________ Process M __________
// Initialize I to be r-bit blocks of pad(m)
n = 0
while(n * r < length(pad(m))):
    duplex(f, pad, I[n], l)
    n++

return first-l-bits(out)
```

Algorithm 2.2: Pseudocode for duplex construciton

The advantages of using the Duplex construction are based on its natural ability to handle streaming data by giving an output as soon as it gets a block of input. If the message is separated into Associated Data and plaintext, then it is easy to identify its ability to handle the two parts of the message separately. One of its drawbacks is message prefixes: Two identical message prefixes result in the same ciphertext prefix. This forces the use of IVs and nonces in order to retain confidentiality. Much like the sponge construction, the security of an algorithm using the duplex construction is reducible to f. In particular, f should be indistinguishable from a random permutation.

2.8 SipHash

SipHash is a family of cryptographically secure hash functions designed for short messages. SipHash-2-4-128 outputs a 128-bit hash, or optionally a 64 or 32-bit hash when the alternative algorithms Siphash-2-4 or HalfSipHash are used. This work will denote SipHash-2-4-128 as either SipHash-2-4 or Siphash for brevity. It was designed for environments that require high performance. Compared to SHA-2 or SHA-3, it is significantly faster but is considered less secure as the security is directly proportional to the tag length [9]. Despite this fact, it is still considered to be computationally hard to find collisions, making SipHash a decent candidate as a MAC algorithm for URLLC. The probability of an attacker who searches 2^s keys to guess the right key is $2^s - 128$ [10]. One SipHash implementation always requires two parameters $c \in \mathbb{N}$ and $d \in \mathbb{N}$ and is referred to as SipHash-c-d. c refers to the number of *SipRounds* in the compression phase, d refers to the number of *SipRounds* in the finalization round. SipHash is specified as follows: [9]:

SipRounds: One SipRound applies additions and XORs between the v variables and rotates them by a certain amount of bits, as shown in Figure 2.1.
Initialization. Initialize the following four variables: $\forall\ 0 \leq i < 4 : v_i = k_i \oplus c_i$, where k_i is the key split into four parts, and c_i is the binary representation of the string *somepseudorandomlygeneratedbytes* split into four parts. The choice of this string was ad-hoc and only requires pairwise asymmetry among its parts.
Compression: After initialization, SipHash compresses the message into one of the variables iteratively: $v_3 = v_3 \oplus m_i$. Then, SipRound is applied c times, followed by $v_0 = v_0 \oplus m_i$ iteratively.
Finalization: Change the variable $v_2 = v_2 \oplus 11111111$, and apply d rounds of SipRound. Finally, output the hash as $v_0 \oplus v_1 \oplus v_2 \oplus v_3$.

These operations are interleaved in such a way that every variable's value influences another. Since SipHash uses these specific operations, it is considered an ARX algorithm (Addition, Rotation, XOR algorithm).

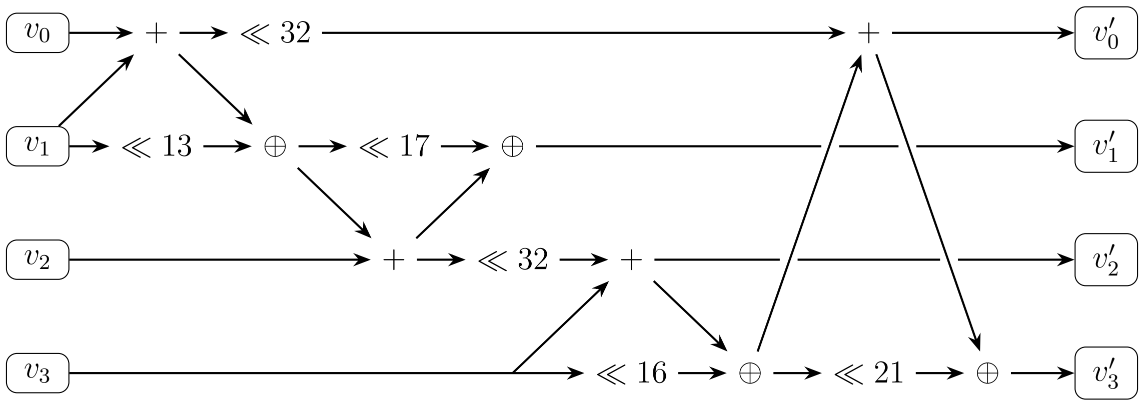

Figure 2.1: Operations performed in a single SipRound. It modifies a value v_i to a new value v'_i

SipHash-2-4 is used for this thesis, which is the default and recommended version. The number of rounds executed by SipHash can be lowered to fit higher performance needs, but this approach trades speed for a potential loss in security. Most of the runtime will be spent in SipRounds, as it contains the most instructions, so reducing the number of compression rounds c or finalization rounds d will lead to faster runtimes in exchange for possibly lower security.

2.9 Ascon

Ascon is a family of lightweight cryptographic functions, which recently became a standard chosen by NIST in 2023 [11]. Among the selection of cryptographic features, Ascon provides a lightweight authenticated encryption scheme **Ascon AEAD**. Here, lightweight means that Ascon uses simple construction, can operate efficiently with low power consumption and is performant while providing good security. Ascon uses the aforementioned duplex and sponge construction.

Ascon operates on a bit-state of $b = 320$, divided into the bitrate $r = 128$ and the capacity $c = 192$ by default. A single permutation round $\mathsf{Ascon}_p[1]$ works as follows:

Initialization: Define five 64-bit variables, which define the state at all times: $s_0||s_1||s_2||s_3||s_4 = \mathsf{state}$

Nonlinear Substitution: Applies a custom Substitution Box (S-Box) along the length of all variables s. Conceptually, it can be defined as iterating i between 0 and 63 and applying an substitution on the concatenation of the i-th bits of $s_0, s_1, ..., s_4$. However, the substitution is a function that consists of a series of XOR, AND, and right rotation applications. This can be parallelized, which makes Ascon faster.

Linear Diffusion: Does a set of rotated XOR operations as follows:

$$
\begin{aligned}
s_0 &= s_0 \oplus (s_0 \gg 19) \oplus (s_0 \gg 28)\\
s_1 &= s_1 \oplus (s_1 \gg 61) \oplus (s_1 \gg 39)\\
s_2 &= s_2 \oplus (s_2 \gg 1) \oplus (s_2 \gg 6)\\
s_3 &= s_3 \oplus (s_3 \gg 10) \oplus (s_3 \gg 17)\\
s_4 &= s_4 \oplus (s_4 \gg 7) \oplus (s_4 \gg 41)
\end{aligned}
$$

The three permutation steps above can be applied multiple times. Applying the permutation $n \in \mathbb{N}$ times is denoted as $\mathsf{Ascon}_p(n)$. Using this permutation, Ascon-AEAD is ready to be used as follows:

Initialization: Initialize state using key and nonce. Then, use the permutation $\mathsf{Ascon}_p(12)$.

Associated Data: Using the sponge-absorb construction, update the state with associated data as input. $\mathsf{Ascon}_p(8)$ is used as the permutation function.

Encryption/Decryption: On encryption, use plaintext as input in a duplex construction and extract ciphertext. On decryption, use ciphertext as input in a duplex construction and extract ciphertext. Use $\mathsf{Ascon}_p(8)$ as the permutation function.

Finalization: Use $\mathsf{Ascon}_p(12)$ to permute state one more time. XOR the final state with the key to get the tag. If on the receiving end, the calculated tag will be equality-checked with the received tag.

2.10 ChaCha20

ChaCha20 is a secure stream cipher that produces a keystream which is XORed with plaintext to produce a ciphertext. It uses a 256-bit key and a 96-bit nonce, a 32-bit block count parameter, and is designed using constant-time operations to avoid timing attacks [12], which means that the time-to-completion of various operations will not give information away that could be interpreted by an adversary. It is designed as an alternative to AES and does not require dedicated hardware like AES-NI, which is an instruction set found on modern x86 chips, to be fast [13].

2.11 Poly1305

Poly1305 is a one-time authenticator, taking a 32-byte one-time key and a message, producing a 16-byte tag which is used to authenticate the message [14]. The security of the algorithm is tied to the security of the chosen key generation algorithm, and the key generation algorithm can be changed without altering the rest of the algorithm. Assuming the attacker sees at most C authenticated messages, attempts at most D forgeries, has the advantage of at most δ in distinguishing the cipher, and the messages are at most length L, then the probability that any of the D forgeries are rejected is [14]:

$$1 - \delta - \frac{(1 - C/2^{128})^{-(C+1)/2} \cdot 8D\lceil L/16 \rceil}{2^{106}}.$$

Note that δ for AES has not been proven to be small, but years of scrutiny have given confidence in the underlying cipher in practice.

2.12 HMAC-SHA256

HMAC combines a cryptographic hash function with a secret key to provide integrity and authenticity. A key feature of HMAC is that it treats the hash function as a "black box", making it possible to change the hash function used depending on need [15]. HMAC is defined as follows:

$$\texttt{HMAC}(K, m) = \mathrm{H}\left((K' \oplus \texttt{opad}) \parallel \mathrm{H}\left((K' \oplus \texttt{ipad}) \parallel m\right)\right)$$
$$K' = \begin{cases} \mathrm{H}(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

H is the underlying hash function, `ipad` is the byte `0x36` repeated B times and `opad` is the byte `0x5C` repeated B times, where B is the same as the hash function's block size. In this work, we choose the hash function H to be SHA-256.

3 Methods

Algorithms that appeared suitable for lightweight encryption have been selected. The research was confined to IoT devices with 64-bit arithmetic, a transmission model, and an attacker model. The chapter also shows how the Benchmarking for the algorithms was set up and which attributes of an algorithm were measured. The custom benchmark itself is available at

https://github.com/Ryuzaou325/master-thesis

3.1 IoT Devices

IoT Device	Frequency	MIPS	RAM	Flash Memory
ESP32	160-240 MHz	200 - 400	520 KB	up to 4 MB
ESP32-S2	240 MHz	300	320 KB	up to 4 MB
ESP32-C2	120 MHz	unknown	272 KB	unknown
ESP32-C3	160 MHz	unknown	400 KB	up to 4 MB
ESP32-H2	96 MHz	unknown	256 KB	unknown
nRF54L15	128 MHz	192	256 KB	1.5 MB
nRF5340	64 or 128 MHz	92 or 192	64 - 512 KB	256 KB - 1 MB
nRF52833	64 MHz	80	128 KB	512 KB

Table 3.1: Chosen IoT devices and their performance

An IoT device is a computing unit with significantly less processing power than typical consumer electronics. These devices are characteristically lightweight, meaning they consume less power but are also highly resource-constrained. They are designed to be battery-efficient, small, portable, and cheap.

It is entirely possible for URLLC messages to be used with stronger computers like smartphone devices or dedicated workstations. However, IoT devices are becoming increasingly common while also being among the computationally slowest devices. If these devices are capable of integrating security into URLLC messages, then so are many, many other devices. Tested IoT devices are listed in table A.1, along with some of their relevant performance metrics. The most popular IoT device found at the time of writing is the ESP32, and tests also include a stronger device like ESP32-S2 and a weaker device like nRF52833.

3.2 Algorithm Choice

There are multiple types of cryptographic schemes and algorithms used for secure communication. Their purpose can be key exchange, mutual authentication, random number generators, encryption, integrity check, or authenticated encryption among others. The focus will be on integrity checks and encryption algorithms since these are the algorithms that ensure that the message can not be read and has not been altered. Such integrity checks will detect active attackers, assuming that the integrity check is considered to be computationally secure.

The algorithms tested were AES-GCM, HMAC-SHA256, ChaCha20, Poly1305, Siphash, the one-time pad (OTP), Ascon AEAD, and combinations of these. Notice that some algorithms are encryption algorithms, but they can be combined with an integrity algorithm in an encrypt-then-MAC scheme to provide both confidentiality and integrity. All functions have been implemented from a code repository that is either open-source, or licensed by an open-source institution, such as Creative Commons (CC0), or the Internet Systems Consortium (ISC) license. The algorithms will be benchmarked and compared to UEA2 (UMTS Encryption Algorithm 2) and UIA2 (UMTS Integrity Algorithm 2), where UMTS stands for Universal Mobile Telecommunication Algorithm. The former two algorithms are used in current 5G NR services but are licensed under the European Telecommunications Standards Institute (ETSI) and can only be used for non-commercial interests.

Siphash-2-4 [9] was chosen for its advertised usability as a cryptographic hash function. It is optimized for small message sizes. In the GitHub page for Siphash [10], there are multiple variants of Siphash. These differ based on compression rounds, finalization rounds, and MAC tag length.

Ascon AEAD became a standard chosen by NIST in 2023, and the developers of Ascon made their work publicly available [16]. The Ascon GitHub page already shows benchmarks for most of its functions on different CPUs, but not for messages that are as short as URLLC payloads.

Finally, the Libsodium library which is a popular cryptographic library, provides algorithms for ChaCha20, Salsa20, Poly1305, AES-GCM, and HMAC.

For testing against established standards, the algorithms UEA2 and UIA2- specified under the 3GPP standard [17] were used for comparison in the context of 5G NR standards. The metrics compared were instruction count, cycle count, flash memory size, and maximum RAM usage.

3.3 Performance Metrics

Benchmarking code was set up to measure the performance of implementations in a replicable way. For benchmarking, performance metrics that give us enough confidence to make statements about whether the implementations in question could run on IoT devices will be used. Performance metrics include:

- **Instruction Count** - Number of assembly instructions executed by the CPU.
- **Cycle Count** - The number of cycles a CPU needs to execute an implementation
- **Memory Requirements** - The amount of RAM and Flash memory an implementation needs to run successfully
- **Cycles Per Byte** - The total cycle count divided by the message length in bytes.

At first glance, instruction count should be the most accurate indicator of runtime due to its agnostic nature towards CPU power. However, CPU architectures, speculative execution, parallelization, out-of-order execution, and caching all influence the final running time of the code. An eye will be kept on the amount of stack memory needed by an implementation. In general, cryptographic algorithms will not allocate a lot of heap memory other than for key storage. Efficient algorithms should also aim to minimize stack memory usage, and it can be a limiting factor for IoT devices due to their nature to be restrictive.

3.4 Transmission Model

The transmission model for the thesis will be kept simple to make the analysis simpler. Nevertheless, certain assumptions will be made about the transmissions as per URLLC and 5G requirements and standards. MACs in URLLC can be of variable length, but this thesis sets a MAC length of 128 bits, giving us an upper bound of 128-bit security. None of the above algorithms produce a longer tag. This MAC length is considered to be resistant to brute-force attacks and collisions for this thesis. These assumptions are based on the scenario that no encryption or verification scheme is applied and that no malicious actor is present to interfere with transmission. It is assumed that:

- During transmission, noise may affect packet contents.
- Less than 1 out of 100000 packets are lost or dropped. Noise-induced errors may cause package drops. Packets that take longer than 1 millisecond to arrive also count as lost packets.
- The transmission is cellular and the device is always communicating to the same cell tower.
- Communication parties have a synchronized clock. Messages are accompanied by a timestamp, and the message is accepted only within a given timeframe.
- For securing messages, we have 0.2ms of runtime available. That means that URLLC messages could arrive in 0.8ms with a lower than 10^{-5} error rate.

The main goal of this thesis is to identify in what scenarios less than 1 out of 100000 packets are lost after applying encryption and verification algorithms. It may be the case that certain devices with low computing power cannot execute a secure implementation fast enough to stay within the 1 millisecond latency constraint dictated by URLLC.

Certain transmitted packets may be dropped due to interference from noise, in which case the MAC would flag the packet as invalid. An adversary that can block packets from being received by the recipient is not realistic, and will therefore not be considered. In the latter case, the dropped packet does not count towards the URLLC reliability requirement. Accepting the contents of the changed packets may have undesired consequences, which is the purpose of the integrity check. For simplicity's sake, it is assumed that the communicating parties are stationary and that the same cell tower is in communication with the same user. This model only has one user and one cell tower.

3.5 Attacker Model

Since the focus of the research is on securing the message in transmission, certain attacks will be excluded which will consist of attacks that do not target the underlying cryptosystem. Excluded attacks are social engineering attacks, phishing attacks, keyloggers, viruses, and worms among others. The core of the model will be the attacker that is between the sender and the receiver. Following this logic, the attacker will be given the following capabilities and restrictions. The attacker:

- Can listen to messages being sent.
- Can send messages to the sender and receiver
- Has polynomial computing power
- Has access to all algorithms being used as an oracle (known as Kerckhoff's principle: An attacker has access to everything but the key)
- Does not have access to secret information/keys, but may try to uncover them
- Can not prevent messages from reaching their intended recipient.
- Can not monitor, have access to, or take control of the physical communication devices (side channels)

The final two points allow us to focus on the analysis of the cryptographic primitives and eliminate side-channel attacks and scenarios where the DoS is not based on a weak cryptosystem. All tested implementations claim to be resistant to side-channel attacks, but this will not be part of the analysis in the thesis. The first four points make the adversary a realistic attacker, in the sense that the attacker only has public information but no private information about the system in play with reasonable computational power. Giving the adversary an oracle is more access than what a realistic adversary has, but the algorithms shown in the "Theory" section are resistant even against the stronger adversary.

3.6 Benchmarking

A program was made for benchmarking the various algorithms. The program takes the algorithm of choice, number of iterations, and message size in bytes as input parameters. For each iteration, the benchmark runs another loop and initializes a new random key, message, and nonce if applicable. Once the initialization is complete, the function RDTSC from the library x86intrin is used to start counting processor cycles, and once the iteration is complete the cycle count is saved for processing.

For each benchmark, a procedure that would be used by the sender and receiver in a real-world scenario is used, which varies between algorithms. For SipHash-2-4 this procedure consists of generating the tag using the message, and then generating it again and comparing the tags, alerting if a mismatch is detected. This is assumed in the thesis to give an accurate representation of the computations that would be used in a real-world scenario, excluding the propagation time of signals. A similar procedure is used for UIA2 and HMAC. For ChaCha20, Poly1305, ChaCha20-Poly1305, and AES-GCM, their available built-in functions for verification were used. For UEA2 and one-time pad, the time for encryption and decryption of a message was measured. For UEA2UIA2 and ChaCha20Siphash-2-4, a message was encrypted, a tag was generated, then the ciphertext-tag pair was verified and finally, the ciphertext was decrypted.

Instruction count was gathered using the PERF tool. The benchmarks were wrapped in a similar way as for the cycle count, where PERF was called at the beginning and end of each iteration. Flash memory size and RAM size were gathered by using the GNU Binutils "size" command in the Linux terminal on the executable of each algorithm after compilation.

The code distinguishes between two separate phases: the initialization phase and the cryptographic phase. If a benchmark was repeated *iteration* times, then the benchmark would have the structure shown in Algorithm 3.1.

The initialization phase included instructions that should happen even if the cryptographic primitive was not being used, or could be done in advance. These could include loading keys into cache memory, creating the message, and loading other constants. The cryptographic phase includes instructions that are believed to be done because of the cryptographic primitive and are likely not to happen in advance. This included generating a random IV/nonce, a call to a function that implements the cryptographic functions, and an integrity check if applicable.

Note that the benchmark measures what both the sender AND receiver would execute in a scheme. For instance, when integrity algorithms are benchmarked, the time to generate the MAC by the sender is summed with the time needed by the receiver to check the MAC. This approach allows us to simply add the measurements to the time needed to send, transmit, and receive the message in URLLC networks to receive the end result.

Measurements on the maximum amount of RAM used during certain code snippets

```
Input: iteration
Output: measurements

// Initialize measurement tally...
while (iteration > 0):
    // Initialize IVs, nonces...
    begin-measurement()
        crypto-phase-sender()
        crypto-phase-receiver()
    end-measurement()
    // Update measurement tally...
    iteration = iteration - 1

// Handle measurements, print results...
```

Figure 3.1: pseudocode for benchmarking

are hard to make because all tools that allow RAM checks tell us the maximum amount of RAM used from the beginning to the end of the process run. This is a hindrance because, within the benchmark, other variables, print statements, etc. may interfere with an accurate estimate. Therefore, in order to measure maximum RAM accurately, we create an executable file where only the security algorithm is being executed and nothing else and use the Linux command `size` on the file to get a more accurate estimate.

This command outputs a `text`, `data`, `bss`, and `dec` value for a file. The value `dec` = `text` + `data` is the flash memory usage of the file, and the value `data` + `bss` is the allocated RAM.

4 Results

This section presents the benchmarking results for the cryptographic algorithms discussed in the previous chapters. The algorithms' performance is evaluated based on the metrics cycle count, instruction count, and memory usage. The results are grouped into three categories: message authentication codes (MACs), encryption algorithms, and authenticated encryption (AE) schemes.

4.1 Measurements

Measurements were done for all algorithms specified in the Theory section and the Algorithm Choice subsection in Methods. When plotting graphs, they were divided into three groups of algorithms: MAC algorithms, ciphers, and authenticated encryption algorithms. Some authenticated encryption algorithms may use additional data but are given an empty string as additional data instead.

All measurements were averaged after 100 000 iterations to eliminate the influence of outliers. To summarize the following results: The one-time pad is the fastest Cipher. ChaCha20 is the fastest cipher that allows key reuse. SipHash-2-4 is the fastest integrity algorithm. Ascon, AES-GCM (with hardware acceleration), and ChaCha20-Siphash-2-4 are the fastest authenticated encryption schemes. The 5G NR standard algorithms are subpar; especially its MAC component.

To measure the total running time of an algorithm on IoT devices, we use the amount of MIPS (Million Instructions per Second) an IoT device can execute and multiply it by the Instruction count of the algorithm in question. This is more effective than using clock cycles because they have a high variance on different CPUs

4.1.1 MAC

For MAC algorithms there is SipHash-2-4, UIA2 (5G NR standard), Poly1305, and HMAC.

Figure 4.1 shows the performance of the integrity checks in cycles per byte. SipHash-2-4 has the fastest performance and also the lowest overhead, making it a very promising candidate. Poly1305 also shows good performance. Surprisingly, HMAC and UIA2 are much slower and have a much higher overhead.

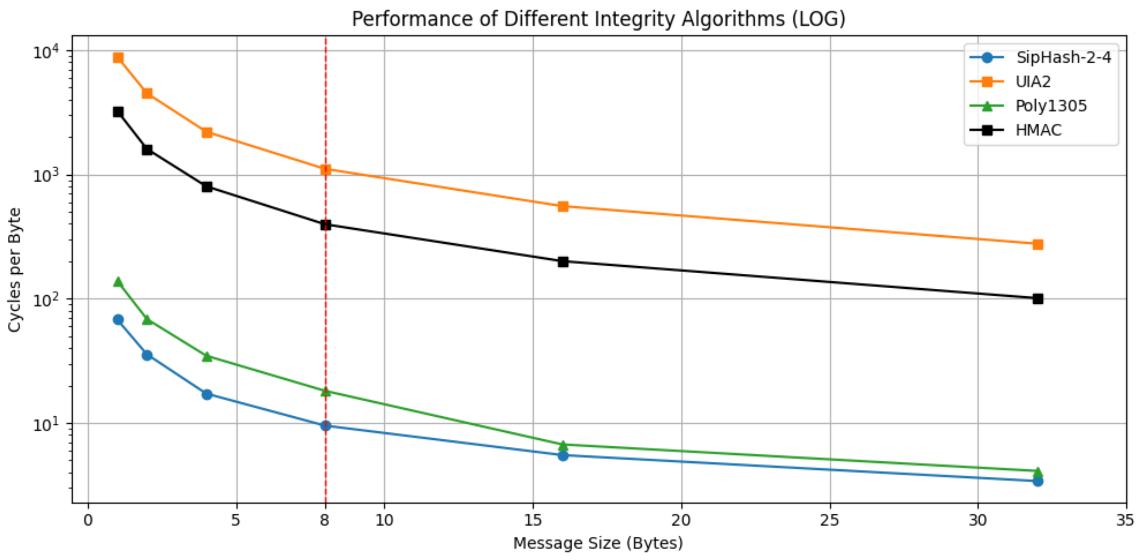

Figure 4.1: Performance in cycles per byte of authentication algorithms

Other performance metrics are shown in Figure 4.2 including instruction count, flash memory, and maximum RAM used. The instruction count correlates with the cycles cycle count measurements, which is no surprise. Differences in RAM and flash memory usage can be explained by the size of the internal state, number of constants defined, S-Box definitions, program length, etc.

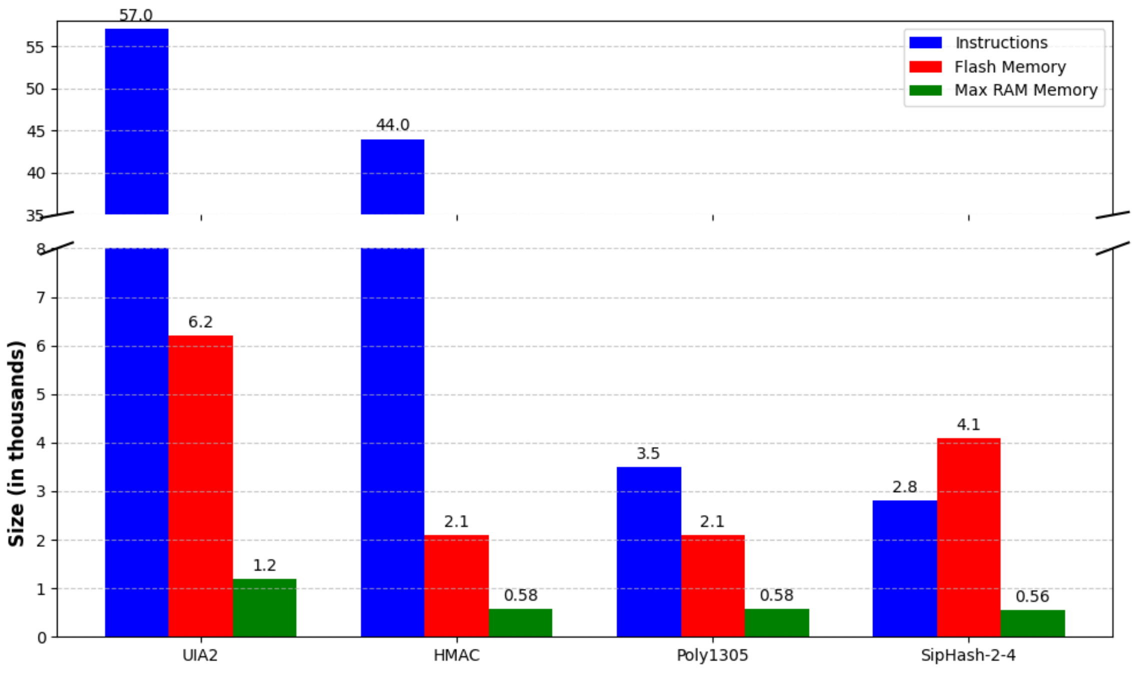

Figure 4.2: Instructions, flash memory and maximum RAM comparison of authentication algorithms

The following steps were included in the measurements to get the above results:

1. Random IV and nonce initialization if applicable (sender).
2. Generating the MAC (sender).
3. Verifying the MAC (receiver).

Finally, to measure the amount of time the algorithms would take on a specific IoT device, we take the performance metrics of algorithms in combination with the IoT device specifications to approximate computation time. The computation time is shown in Figure 4.3. In this figure, the red dotted line represents the 1ms URLLC constraint. The 0.8ms, 0.5ms, and 0.2ms limits are shown by the orange, green, and blue dotted lines respectively.

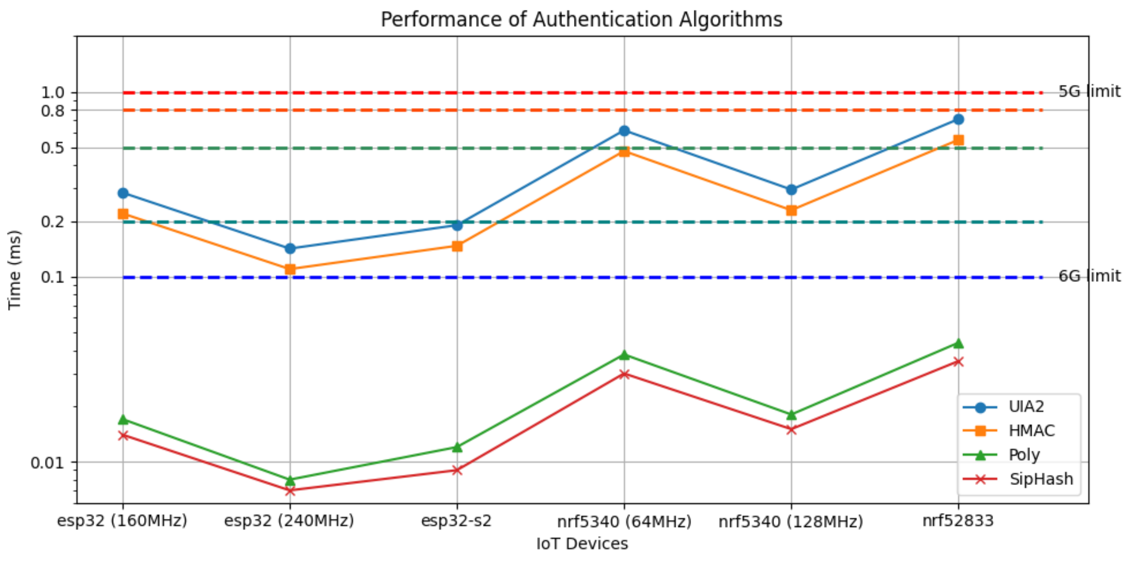

Figure 4.3: Performance of authentication algorithms on IoT devices based on instruction count

4.1.2 Ciphers

Encryption algorithms include ChaCha20, UEA2 (5G NR standard), and OTP. The steps used in the code to measure encryption algorithms' performance metrics are the same as the steps in MAC algorithms in the previous subsection.

Figure 4.4 shows the cycles per byte for each encryption method. Unsurprisingly, OTP is by far the fastest. Reading and writing values to and from variables takes more time than the encryption itself, which consists only of XOR operations. ChaCha20 performs well, beating UEA2, the 5G NR standard algorithm

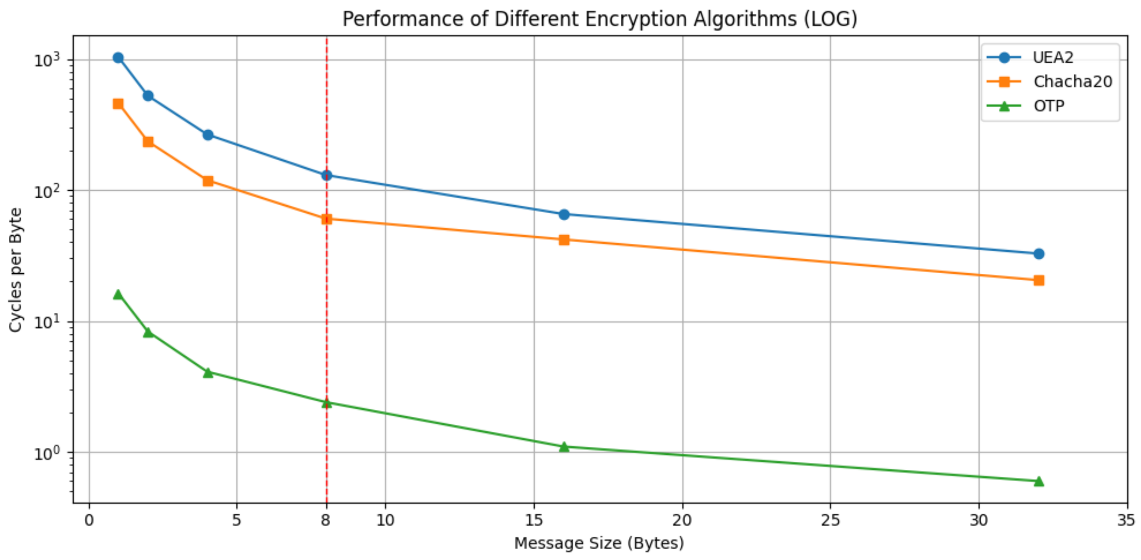

Figure 4.4: Performance in cycles per byte of encryption algorithms

Other performance metrics are illustrated in Figure 4.5, and show no irregularities. However, note that OTP has a technically low flash memory usage even if it has to store a theoretically infinitely long key. The flash memory should be as large as the key length currently being stored, and in practice, another secure communication channel can transfer keys between parties. This channel does not have to be on a URLLC service, and can instead be on an eMBB service which can transfer a very long key at once.

For encryption algorithms, no runtime calculations on IoT devices were made because the thesis focuses on the runtime of integrity and authenticated encryption algorithms. The purpose of measuring only encryption algorithms is to make a suitable encrypt-then-MAC combination that fits URLLC use cases and constraints.

Results from Figure 4.4 justify the decision to use ChaCha20 as an encryption scheme and results from Figure 4.1 justify the decision to use SipHash-2-4 as an integrity scheme in a custom encrypt-then-MAC combination.

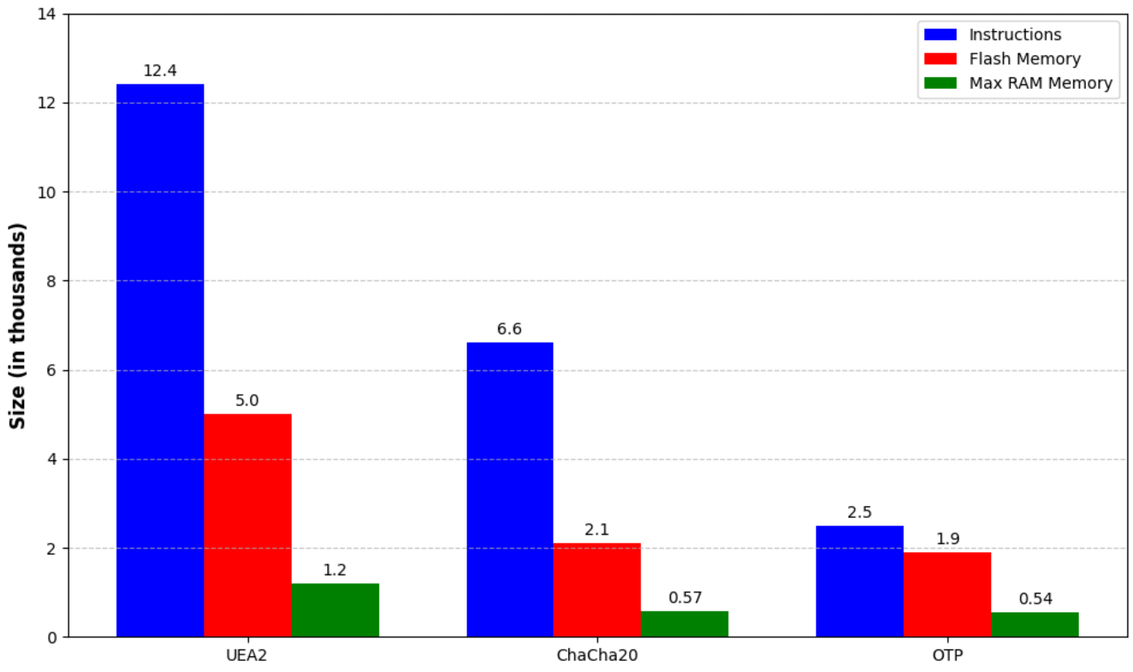

Figure 4.5: Instructions, flash memory and maximum RAM comparison of encryption algorithms

4.1.3 Authenticated Encryption

Our authentication algorithms include Ascon, AES-GCM, Chacha20-Poly1305, UEAUIA2 (5G NR standard), and ChaCha20-SipHash-2-4. The combination of ChaCha20-SipHash-2-4 is not a standard combination. The reason for choosing and measuring this combination is because they were the fastest algorithms in encryption and integrity respectively if the one-time pad is excluded. A combination of algorithms was sought that allowed the reuse of keys but was also fast. The one-time pad was predictably the fastest but requires a random key as long as all messages are sent in the lifetime of a conversation between two parties, as proven by Shannon [18].

We predict that combining any integrity algorithm with any encryption algorithm will give us a cycle count roughly equal to their cycle count summed up. This trend is seen on ChaCha20-SipHash-2-4, and it is also the reason that the one-time pad was chosen not to be combined with any other algorithm. The one-time pad has negligible overhead and negligible runtime, so any integrity algorithm combined with the one-time pad would give roughly the same runtime as the integrity algorithm itself.

Figure 4.6 shows that again, the 5G NR standard algorithm stands out as the slowest algorithm. However, all other authenticated encryption schemes show promising performance metrics even in Figure 4.7. Ascon is the algorithm with the lowest overhead, but AES-GCM overtakes Ascon with higher message lengths. One noticeable inconsistency is the AES-GCM instruction count, flash memory usage, and RAM usage. This can be explained by the hardware support that AES-GCM uses, giving it an advantage over other algorithms that don't have the hardware support.

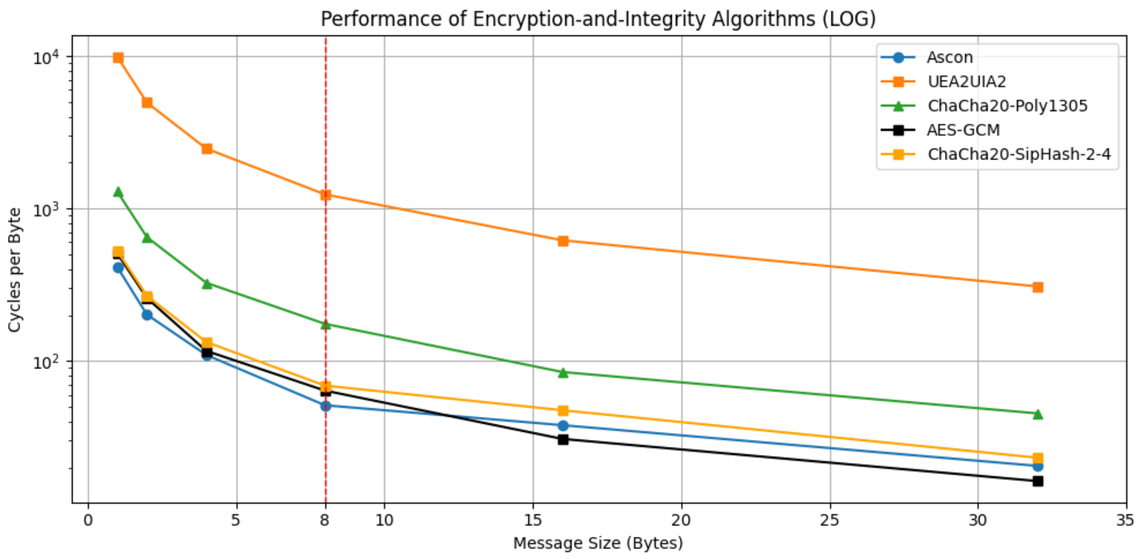

Figure 4.6: Performance in cycles per byte of authenticated encryption algorithms

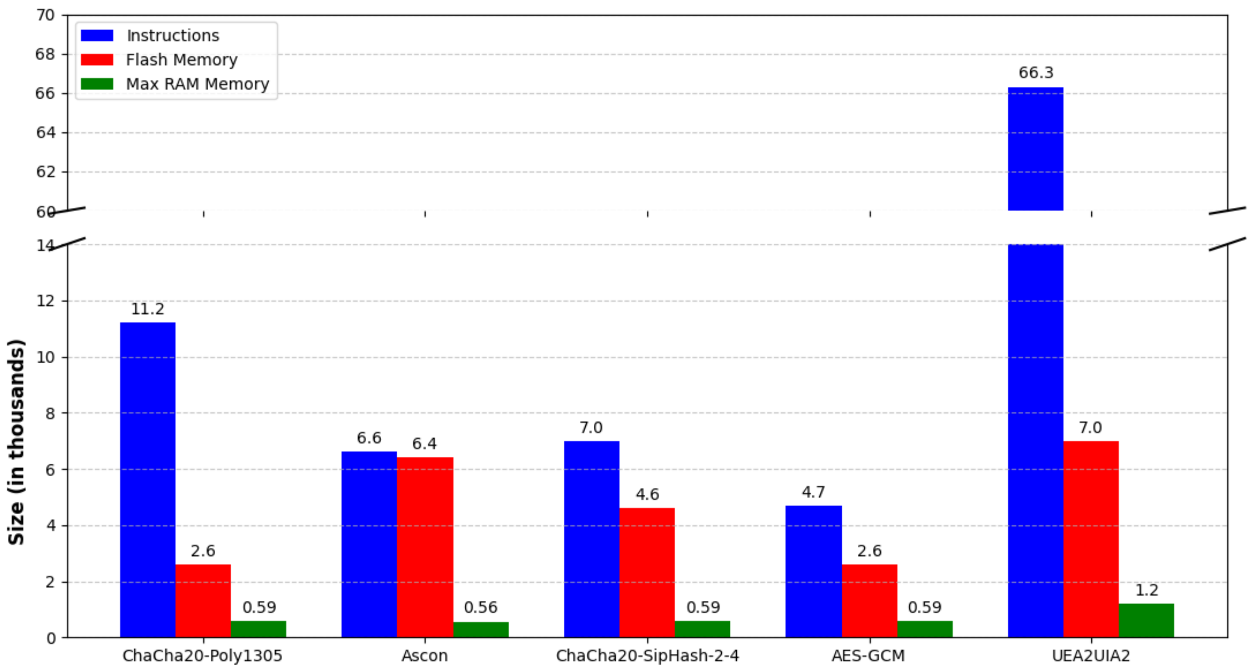

Figure 4.7: Instructions, flash memory and maximum RAM comparison of authenticated encryption algorithms

The above metric calculations were done using code that consisted of the following steps:

1. Random IV and nonce initialization (sender)
2. Encryption (sender)
3. MAC generation (sender)

4. MAC verification (receiver)
5. Decryption (receiver)

Finally, we show the most important result of the thesis in Figure 4.8. The graph shows how fast IoT devices are capable of running each algorithm. Three candidates even have runtimes below 0.1ms, which is good news for securing 5G URLLC services, and shows that it is indeed possible.

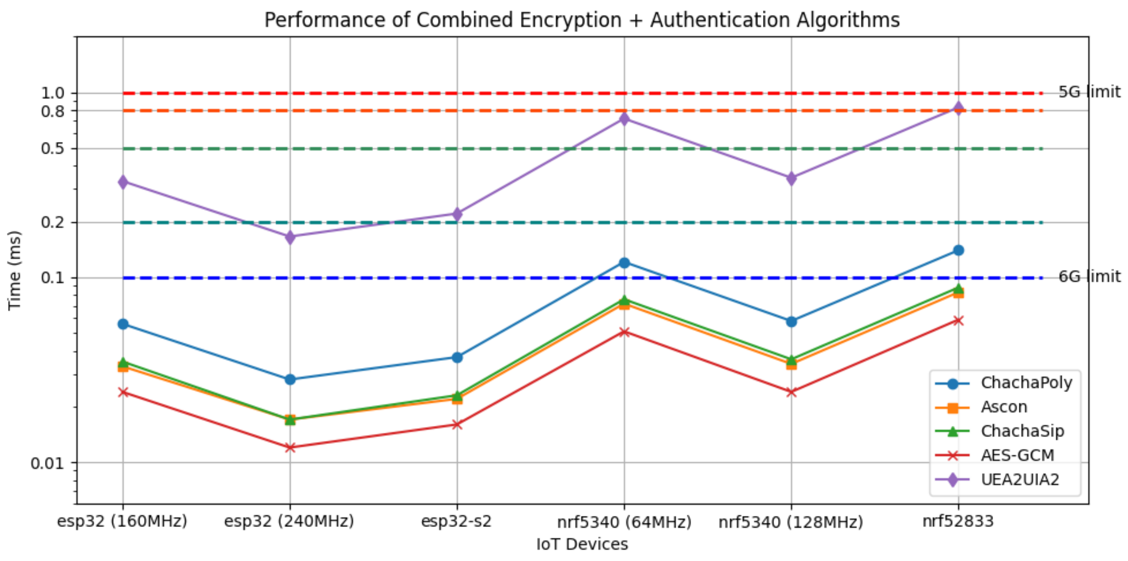

Figure 4.8: Performance of authenticated encryption algorithms on IoT devices based on instruction count

4.2 Table Summary

To conclude the chapter, the following three tables show a rough performance overview of every algorithm tested in this thesis. These tables list whether an algorithm provides secrecy or integrity, if it uses lots of storage and whether it is fit for use in a URLLC setting.

Algorithm	Secrecy	Integrity	Fast	Storage	URLLC-Capable
ChaCha20	✓	✗	✓	Small	✓
UEA2	✓	✗	✗	Large	✗
OTP	✓	✗	✓	Small	✓

Table 4.1: Encryption Algorithms

First, Table 4.1 shows encryption algorithms. Note that using these in their plain form makes them malleable and has no mechanic for verifying the sender's identity or the message's integrity. That is why they should be used in combination with MAC algorithms, whose properties are listed in Table 4.2.

Algorithm	Secrecy	Integrity	Fast	Storage	URLLC-Capable
SipHash-2-4	✗	✓	✓	Medium	✓
UIA2	✗	✓	✗	Large	✗
HMAC-SHA256	✗	✓	✗	Small	✗
Poly1305	✗	✓	✓	Small	✓

Table 4.2: Integrity Algorithms

An overview of authenticated encryption algorithms is shown in Table 4.3. At this point, it is important to point out that all algorithms tested do not exceed RAM or flash memory requirements of any IoT device, meaning all algorithms are able to be stored and run on every IoT device. Hence, a "small" or "large" storage size is only relevant if the IoT device is to store and run other processes.

Algorithm	Secrecy	Integrity	Fast	Storage	URLLC-Capable
ChaCha20-Poly1305	✓	✓	✓	Small	✓
Ascon	✓	✓	✓	Large	✓
ChaCha20-Siphash-2-4	✓	✓	✓	Medium	✓
AES-GCM	✓	✓	✓	Small	✓
UEA2UIA2	✓	✓	✗	Large	✗

Table 4.3: Authenticated Encryption Algorithms

5
Discussion

This section analyzes the performance results of the cryptographic algorithms shown in the previous section and discusses the advantages and limitations of these algorithms. Furthermore, the multiple time thresholds present in the graphs of IoT devices are evaluated to find which algorithms are realistically usable in constrained IoT devices.

5.1 Algorithms

Through Figure 4.3 we can see a distinct gap between the slower algorithms UIA2 and HMAC, and the faster algorithms Poly1305 and SipHash-2-4. Interpreting the different limits of 0.2, 0.5, 0.8, and 1.0 milliseconds that are present in the plot as situations where the cryptographic part of the communication protocol is more and more restricted, such as communication over long distances where propagation delays are noticeable, we can see that the slower algorithms may be usable in favorable situations with less constrained devices. To better guarantee that the protocol can run within the latency constraints on a wider range of devices, we see that faster algorithms are necessary as they sit below all boundaries. A similar pattern is seen in Figure 4.8.

SipHash has a larger flash memory footprint relative to its instruction size compared to every other tested algorithm. We believe this may be attributed to the large amount of constants that SipHash uses internally. These constants likely represent a form of precomputation that contributes to its fast runtime. However, SipHash does not use nonces, IVs, or any other freshness mechanism. As a result, each message must use a new key. Similarly, Poly1305 requires a new key each time. This downside of both algorithms may be mitigated by combining them with a stream cipher capable of key expansion, such as in ChaCha20-Poly1305.

The results of the one-time pad as a means of encryption show that it is the fastest of all tested encryption algorithms. Unfortunately, this method relies on keys of equal length to the message being provided which in many scenarios would require a tremendous amount of storage space dedicated to the keys. We decided to consider this method of encryption as the amount of data generated and sent as part of URLLC is significantly smaller than that of messaging scenarios such as 5G NR. The security of this method is purely tied to the randomness of the key, and as long as the keys follow a uniform distribution, perfect secrecy is achieved.

Ascon uses the same underlying permutation for both encryption and hashing. Using the same primitive reduces the size of the implementation. The 64-bit words and operations work well with the target 64-bit messages that this project focuses on. Furthermore, it can take advantage of parallelization and pipelining features in modern processors. Ascon is also not vulnerable to cache-timing attacks since it does not use lookup tables or branching. The fast runtime of Ascon is likely attributed to its small overhead, which comes from not needing an inverse of the permutations it uses.

5.2 Comparisons

A notable feature of ChaCha20 is that, although it remains competitive with AES in terms of speed, it does not rely on a special instruction set to achieve high performance. Its use of constant-time operations also makes it resistant to timing-based side-channel attacks. ChaCha20-Poly1305 may therefore be compatible with a broader range of IoT devices than AES-GCM which requires AES-NI or similar instruction sets to operate fast. The libsodium documentation recommends that after initially generating a nonce for ChaCha20, each subsequent nonce results from incrementing the past nonce for each message with the same key. Our tests with ChaCha20-Poly1305 and ChaCha20 generated a new nonce during each iteration. This means that the real-world results for that algorithm will likely be somewhat lower than what is shown on the graphs.

The combination of the fastest encryption algorithm and the fastest integrity algorithm, the one-time pad and SipHash-2-4 was not explicitly tested as the expected performance is equal to the sum of their runtimes. Furthermore, given how the one-time pad's encryption time is significantly lower than any other integrity function tested, a combined "OTP-Siphash-2-4" would be visually very similar to the integrity SipHash alone. Under this assumption, such a combination would still outperform most alternatives as shown in 4.6.

The UEA2, UIA2, and UEAUIA2 algorithms consistently perform slower than all other tested algorithms at the selected message sizes. This is likely due to the SNOW 3G algorithm they rely on. We believe the relatively high overhead of SNOW 3G's two S-boxes and the generation of a new keystream on every run significantly impacts performance. For larger message sizes than were tested here, we believe that the cycles per byte will decline to more competitive levels.

Since this project specifically targets the encryption and authentication of 64-bit messages with a 128-bit tag, the performance at 8-byte input lengths is most relevant. In this context, the slower algorithms UIA2, HMAC, and UEA2UIA2 perform an order of magnitude worse than faster alternatives, although their cycles-per-byte metrics may improve drastically outside the tested message sizes. Because of this, they are impractical for URLLC.

6 Conclusion

6.1 Conclusion

In this thesis, we evaluated the performance of various cryptographic algorithms for use in 5G Ultra Reliable Low-Latency Communication in the context of resource-constrained IoT devices. Our results show that the standard 5G NR security algorithms do not allow several tested devices to meet the latency requirements of URLLC, and devices weaker than these will certainly not.

Based on our benchmarking, we recommend SipHash-2-4 as a fast and lightweight integrity algorithm. For authenticated encryption, we found that ChaCha20-Poly1305, Ascon, and AES-GCM (using a special instruction set) are viable options. These algorithms meet the latency constraints on the tested devices and fit in the constrained storage.

These findings highlight the need for algorithms specialized for use in constrained environments where traditional security algorithms are not feasible. As communication systems increasingly rely on low-latency communication, lightweight algorithms become more important. By identifying and evaluating secure alternatives for URLLC, this thesis contributes to enabling secure communication in constrained-device systems.

6.2 Future Work

It is common for modern cryptographic algorithms to be tailored towards long messages like payloads in the popular eMBB services, which is why they need to be explicitly evaluated for URLLC. However, we believe that future communication will integrate more low-latency services. Future infrastructure is likely to integrate more automation and technology but may use resource-constrained IoT devices like the ones we tested. Therefore, the most obvious improvement in our research is to test more algorithms and schemes to see if they qualify for use in URLLC environments. This is also necessary because inevitably, more cryptographic algorithms will be invented due to the need for quantum-resistant cryptographic algorithms.

In this thesis, we made educated predictions on which algorithms are likely to work on specific IoT devices. The most logical next step is to physically acquire the tested IoT devices and see if our hypothesis holds. Actual results may deviate due to processor architecture, register size, time to fetch instructions, and efficiency implementations, but we don't believe that it will make a major difference.

Another important future research topic is to compile and analyze bit security estimates for each algorithm in this thesis. This includes summarizing the currently known cryptanalytic attacks on these algorithms and their estimated computational complexity. Such an analysis would provide another dimension when it comes to the choice of algorithm to implement, and whether an algorithm is deemed secure enough for a particular scenario.

Another interesting topic could be hardware support for cryptographic algorithms that are newer than AES. For instance, Ascon's developers have pointers on how to develop hardware architecture that would be able to execute Ascon AEAD instruction much faster [19]. As has been pointed out, the only reason AES-GCM can keep up with other algorithms speed-wise is due to hardware support. If hardware support was not present, AES-GCM would not be viable for URLLC as it would be orders of magnitude slower. Hardware support for already-viable algorithms would make them considerably faster and perhaps make them viable for sixth-generation (6G) Hyper Reliable Low Latency Communication (HRLLC)

Finally, we can not ignore the rise of 6G wireless technology, which is in the early development stage at the time of writing (2025-09-05). Many countries, organizations, and companies are collaborating to develop an even higher standard for communication. It has been proposed that 6G technology should include HRLLC, which has even stricter constraints. When the specifics of this new technology are known, the algorithms need to, once again, be tested under the new constraints.

Bibliography

[1] Z. Li, M. A. Uusitalo, H. Shariatmadari, and B. Singh, "5g urllc: Design challenges and system concepts," in *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, 2018, pp. 1–6. DOI: `10.1109/ISWCS.2018.8491078`.

[2] A. Pradhan, S. Das, M. J. Piran, and Z. Han, *A survey on security of ultra-/hyper reliable low latency communication: Recent advancements, challenges, and future directions*, 2024. arXiv: `2404.08160 [cs.CR]`. [Online]. Available: `https://arxiv.org/abs/2404.08160`.

[3] 3. Organization. "Study on the security of ultra-reliable low-latency communication (urllc) for the 5g system (5gs)." (2018), [Online]. Available: `https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3548` (visited on 06/02/2025).

[4] T. Yoshizawa, S. B. M. Baskaran, and A. Kunz, "Overview of 5g urllc system and security aspects in 3gpp," in *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2019, pp. 1–5. DOI: `10.1109/CSCN.2019.8931376`.

[5] R. Chen, C. Li, S. Yan, R. Malaney, and J. Yuan, "Physical layer security for ultra-reliable and low-latency communications," *IEEE Wireless Communications*, vol. 26, no. 5, pp. 6–11, 2019. DOI: `10.1109/MWC.001.1900051`.

[6] National Institute of Standards and Technology, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," National Institute of Standards and Technology, Tech. Rep. FIPS PUB 197, Nov. 2001, Updated: February 2023. [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf`.

[7] D. J. Bernstein, *Cache-timing attacks on aes*, `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, *Cryptographic sponge functions*, `https://keccak.team/files/CSF-0.1.pdf`, Accessed: 2025-03-26, 2011.

[9] J.-P. Aumasson and D. J. Bernstein, "Siphash: A fast short-input prf," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science, vol. 7668, Springer, 2012, pp. 489–508. DOI: `10.1007/978-3-642-34931-7_28`. [Online]. Available: `https://www.aumasson.jp/siphash/siphash.pdf`.

[10] J.-P. Aumasson and D. J. Bernstein. "Siphash implementation." (2016), [Online]. Available: `https://github.com/veorq/SipHash` (visited on 02/06/2025).

[11] National Institute of Standards and Technology, "Ascon-Based Lightweight Cryptography Standards for Constrained Devices," NIST Special Publication 800-232 (Initial Public Draft), Nov. 2024, Available: https://nvlpubs.nist.gov/nistpubs/SpecialF
232.ipd.pdf. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-232.ipd.pdf.

[12] Y. Nir and A. Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, Jun. 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://www.rfc-editor.org/info/rfc8439.

[13] D. J. Bernstein, "Chacha, a variant of salsa20," University of Illinois at Chicago, Tech. Rep., Jan. 2008. [Online]. Available: https://cr.yp.to/chacha/chacha-20080128.pdf.

[14] D. J. Bernstein, *The Poly1305-AES message-authentication code*, https://cr.yp.to/mac/poly1305-20050329.pdf, 2005.

[15] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 8th. Pearson, 2022, p. 832, ISBN: 978-1292437484.

[16] M. E. Martin Schläffer. "Ascon implementation." (), [Online]. Available: %5Curl%7Bhttps://github.com/veorq/SipHash%7D.

[17] ETSI/SAGE, "Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2," 3GPP Task Force, Tech. Rep. Version 2.1, Mar. 2009, Confidentiality and Integrity Algorithms Specification. [Online]. Available: https://www.etsi.org.

[18] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[19] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, *Ascon v1.2: Submission to nist*, Accessed: 2025-05-13, 2021. [Online]. Available: https://ascon.isec.tugraz.at/files/asconv12-nist.pdf.

A

Appendix 1: Cryptographic Schemes Cycles Per Byte

	1	2	4	8	16	32
UIA2	8846.6	4492.4	2206.4	1108.6	555.4	276.9
HMAC-SHA256	3246.4	1602.2	802.0	396.9	200.3	100.9
Poly1305	138.9	68.4	34.7	18.1	6.7	4.1
SipHash-2-4	52.2	26.0	13.3	6.9	4.1	2.8

Table A.1: Performance of integrity algorithms in Cycles per Byte. Y-Axis: Algorithm Name. X-Axis: Message Size in Bytes.

	1	2	4	8	16	32
UEA2	1037.7	523.3	265.2	129.9	65.5	32.7
ChaCha20	462.3	234.6	118.7	60.4	41.9	20.5
OTP	16.3	8.3	4.1	2.4	1.1	0.6

Table A.2: Performance of encryption algorithms in Cycles per Byte. Y-Axis: Algorithm Name. X-Axis: Message Size in Bytes.

	1	2	4	8	16	32
UEA2-UIA2	9891.7	4953.4	2473.0	1237.4	618.8	308.3
ChaCha20-Poly1305	1292.6	648.9	325.6	175.1	84.6	45.3
ChaCha20-SipHash-2-4	515.7	267.1	146.4	74.8	47.3	23.8
AES-GCM	507.3	258.5	116.4	63.9	30.8	16.3
Ascon	413.5	201.4	109.5	51.2	37.9	20.5

Table A.3: Performance of authenticated encryption algorithms in Cycles per Byte. Y-Axis: Algorithm Name. X-Axis: Message Size in Bytes.