Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Håkon Tjomsland

Confidentiality and Integrity for Sensor Data in the Cloud-IoT Architecture

Master's thesis in Communication Technology and Digital Security
Supervisor: Colin Boyd, NTNU IIK
Co-supervisor: Håvard Skåra Mellbye, Disruptive Technologies
June 2022

NTNU
Norwegian University of
Science and Technology

Håkon Tjomsland

Confidentiality and Integrity for Sensor Data in the Cloud-IoT Architecture

Master's thesis in Communication Technology and Digital Security
Supervisor: Colin Boyd, NTNU IIK
Co-supervisor: Håvard Skåra Mellbye, Disruptive Technologies
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Norwegian University of
Science and Technology

Title:	Confidentiality and Integrity for Sensor Data in the Cloud-IoT Architecture
Student:	Håkon Tjomsland

Problem description:

Cloud-IoT is an emerging paradigm involving IoT sensors sending data through cloud services to end-users. The cloud services can collect, handle, and analyze sensor data from IoT sensors, which reduces the complexity for the end-user. However, this solution introduces security problems because the cloud service should check the integrity of the data without having access to it. End-users must trust the sensor data sent from IoT sensors after being transmitted through different systems, services, and communication channels. This project investigates suitable methods to achieve integrity for a cloud service while ensuring integrity and confidentiality for end-users.

Motivated by the mentioned challenge, this thesis involves analyzing encryption modes, solution design, high-level implementation, and validation to suggest a solution to the problem. The project will explore challenges and solutions for encryption in IoT devices and research and analyze encryption modes suitable for such an architecture. Authenticated Encryption (AE) algorithms are the main candidate solutions for this project as they achieve both integrity and confidentiality. This thesis will assess independent keys for controlling integrity and confidentiality and the challenges involved in key management. The result of this thesis will be a contribution towards solving the integrity and confidentiality challenges in the mentioned cloud-IoT architecture.

Date approved:	2022-02-25
Responsible professor:	Colin Boyd, NTNU IIK
Supervisor(s):	Håvard Skåra Mellbye, Disruptive Technologies

Abstract

Internet of Things (IoT) devices are widely established and useful in various industries. IoT sensors can retrieve information such as temperature, humidity, and proximity. Because IoT sensors collect a large amount of data that is difficult to manage locally, the cloud-IoT architecture has arisen. Sensors transmit data to a cloud, where it can be stored and analyzed. Allowing cloud services to handle this makes it easy for end-users to retrieve sensor data. Even though this design has numerous advantages, it also introduces security issues regarding the data. How can end-users trust the security of sensor data sent through different communication channels, systems, and services? Additionally, as the cloud handles and stores the sensor data, it must know that it communicates with a sensor and receives actual data and not noise. Thus, the cloud should be able to verify the data's accuracy and consistency.

This master thesis aims to achieve confidentiality and integrity of sensor data for end-users while ensuring integrity for the cloud party in the cloud-IoT architecture. The cloud-IoT paradigm's challenges are the vast attack surface and the constrained devices. IoT sensors usually have limited computational power, memory space, battery power, and bandwidth. The restrictions make it hard to implement security mechanisms while also fulfilling cost and performance criteria. This project has followed the design cycle, where research and design have been the main focus of designing a solution suitable for the cloud-IoT architecture. Due to uncertainties of newly developed lightweight algorithms and the lack of standardized lightweight encryption schemes for IoT sensors, the main focus has been Authenticated Encryption with Associated Data (AEAD) algorithms. The thesis has looked at different approaches to achieving the confidentiality and integrity requirements.

Different solutions are proposed, involving double AEAD, AEAD with additional MAC, and splitting keys in ChaCha20-Poly1305 and EAX. Splitting keys lets sensors exchange one key, where it is split into an integrity part and a confidentiality part. If the cloud only receives the integrity part, it can verify data without decrypting it. The different solutions are compared and discussed in terms of performance, security level, and constraints. Also, the thesis discusses how key management is affected by the different solutions. Due to the many advantages of the ChaCha20-Poly1305 solution with split keys, it was validated through implementation and simulation to predict how it will interact in the real-world cloud-IoT architecture.

Sammendrag

Tingenes internett, kjent som *Internet of Things* (IoT) på engelsk, er vidt etablerte enheter som er nyttige i ulike bransjer. IoT-sensorer kan hente informasjon som temperatur, fuktighet og berøring. Siden IoT-sensorer samler inn store mengder data som er vanskelig å administrere lokalt, har sky-IoT-arkitekturen oppstått. Sensorer overfører data til en nettsky, hvor det kan lagres og prosesseres. Å la skytjenester håndtere sensorene, gjør det enkelt for brukere å motta sensor-data. Selv om denne arkitekturen har mange fordeler, introduserer den også sikkerhetsproblemer. Hvordan kan sluttbrukere stole på sikkerheten til sensor-data sendt gjennom ulike kommunikasjonskanaler, systemer og tjenester? I tillegg, når skyen håndterer og lagrer sensordataen, må den vite at den kommuniserer med en sensor og mottar faktisk data og ikke støy. Dermed bør skyen kunne verifisere dataens nøyaktighet og konsistens.

Denne masteroppgaven har som mål å oppnå konfidensialitet og integritet på sluttbrukernes sensor-data samtidig som den sikrer integritet for sky-enheten i sky-IoT-arkitekturen. Sky-IoT-paradigmets utfordringer er den enorme angrepsoverflaten og de simple enhetene. IoT-sensorer har vanligvis begrenset beregningskraft, minneplass, batteristrøm og båndbredde. Restriksjonene gjør det vanskelig å implementere sikkerhetsmekanismer samtidig som de oppfyller kostnads- og ytelseskriterier. Dette prosjektet har fulgt en designsyklus, hvor forskning og design har vært hovedfokus for å designe en løsning som passer for sky-IoT-arkitekturen. På grunn av usikkerhet knyttet til nyutviklede lettvektsalgoritmer og mangelen på standardiserte lettvektskrypteringsordninger for IoT-sensorer, har hovedfokuset vært på algoritmer av krypteringsformen autentisert kryptering med tilknyttet data (AEAD). Oppgaven har sett på ulike tilnærminger for å oppnå konfidensialitet og integritet.

Ulike løsninger er foreslått i denne oppgaven, blant annet løsninger som involverer dobbel AEAD, AEAD med ekstra MAC, og splitte nøkkel i ChaCha20-Poly1305 og EAX. Å splitte nøkler lar sensorer utveksle én nøkkel, der den er delt inn i en integritetsdel og en konfidensialitetsdel. Hvis skyen bare mottar integritetsdelen, kan den verifisere data uten å dekryptere dem. De ulike løsningene sammenlignes og diskuteres med hensyn på ytelse, sikkerhetsnivå og begrensninger. I tillegg diskuterer oppgaven hvordan nøkkel-håndtering påvirkes av de ulike løsningene. På grunn av de mange fordelene med ChaCha20-Poly1305-løsningen med delte nøkler, ble den validert gjennom implementering og simulering for å forutsi hvordan den vil samhandle i den virkelige sky-IoT-arkitekturen.

Acknowledgements

This thesis was submitted concerning finalizing a master's degree in Communication Technology and Digital Security at the Norwegian University of Science and Technology in Trondheim. The thesis was written in the spring of 2022 and built on a preceding project [Tjo21] from the fall of 2021.

I want to thank my supervisors, Colin Boyd and Håvard Skåra Mellbye, for guidance and valuable feedback along the way. Also, I would like to thank TikZ [Jea16] for the vector graphics used or modified to create cryptographic figures.

Last but not least, I would like to thank my partner and family for their love and support. Also, gratitude goes out to all my friends for some enjoyable years in Trondheim.

Håkon Tjomsland

Contents

List of Figures **xi**

List of Tables **xiii**

List of Symbols **xv**

List of Acronyms **xvii**

1 Introduction **1**

1.1 Cloud-IoT 1
1.2 Motivation 2
1.3 Objective 4
1.4 Methodology and Thesis outline 5

2 Background and Related Work **7**

2.1 Potential attacks and threats 7
2.2 Resource limitations in IoT sensors 9
2.3 Current research 11
2.3.1 Lightweight cryptography 12
2.3.2 Lightweight cryptography today 13
2.4 Cryptography 15
2.4.1 Symmetric-key cryptography 15
2.4.2 Public-key (asymmetric-key) cryptography 21
2.5 Message Authentication Codes (MACs) 23
2.5.1 Hash-based Message Authentication Code (HMAC) 24
2.5.2 CBC-MAC and CMAC 25
2.5.3 Poly1305 26

3 Key Management **29**

3.1 Cryptographic keys 29
3.1.1 Key lifecycle 30
3.2 Key generation 32
3.3 Key distribution 34

3.3.1 Alternative 1: Key Wrapping 35
3.3.2 Alternative 2: ECDH Key Exchange 35
3.3.3 Alternative 3: Key Evolution scheme 36
3.3.4 Key distribution in the cloud-IoT architecture 37

4 Authenticated Encryption 39
4.1 Authenticated Encryption (AE) 39
4.1.1 CBC and HMAC 40
4.1.2 Authenticated encryption in the cloud-IoT architecture . . . 41
4.2 Authenticated Encryption with Associated Data (AEAD) 44
4.2.1 CCM 46
4.2.2 GCM 47
4.2.3 EAX 52
4.2.4 ChaCha20-Poly1305 54
4.3 Implementations on IoT sensors 56
4.4 Comparison of the AEAD schemes 57

5 Design 61
5.1 Achieving confidentiality and integrity 61
5.1.1 Using two independent keys for different operations 62
5.1.2 Splitting a key into integrity key and confidentiality key . . . 63
5.2 Suggestions for different solutions 64
5.2.1 Double AEAD 64
5.2.2 AEAD with additional MAC 66
5.2.3 ChaCha20-Poly1305 with split keys 68
5.2.4 EAX with split keys 74
5.3 Discussion of the solutions 75

6 Implementation and Validation 81
6.1 Implementation 81
6.1.1 ChaCha20 and Poly1305 81
6.1.2 ChaCha20-Poly1305 with split keys 82
6.2 Running the code with test vectors 84
6.2.1 Authenticated encryption 84
6.2.2 Authenticated decryption 86
6.3 Simulation and validation 87
6.4 Performance 89

7 Discussion and Conclusion 91
7.1 Discussion 91
7.2 Conclusion 93
7.3 Further work 95

List of Figures

1.1 Cloud-IoT architecture [Tjo21]. 2
1.2 Confidentiality and integrity in the cloud-IoT architecture. 4

2.1 Tradeoff in cryptography for IoT sensors. 10
2.2 Illustration of the initial ChaCha20 state. 17
2.3 A ChaCha quarter round. 18
2.4 The round function in Advanced Encryption Standard (AES). 19
2.5 Cipher Block Chaining (CBC) mode encryption. 20
2.6 Counter (CTR) mode encryption. 20
2.7 Illustration of Hash-based Message Authentication Code (HMAC). . . . 25
2.8 Illustration of CBC-Message Authentication Code (MAC). 26

3.1 The four phases of the key-management lifecycle, including key states. . 32

4.1 Encrypt-and-MAC, MAC-then-Encrypt, and Encrypt-then-MAC [Tjo21]. 40
4.2 CBC-HMAC encryption. 42
4.3 CBC-HMAC decryption. 42
4.4 Distribution of keys using CBC with HMAC 43
4.5 Authentication and encryption of data in Authenticated Encryption with Associated Data (AEAD) [Tjo21]. 44
4.6 The encryption process of AEAD algorithms. 45
4.7 Illustration of the Counter mode with CBC-MAC (CCM) mode. 47
4.8 The Galois/Counter Mode (GCM) mode using CTR encryption and GHASH authentication. 48
4.9 Illustration of GCM encryption. 49
4.10 Illustration of Galois Message Authentication Code (GMAC) authentication. 51
4.11 Encrypt-then-Authenticate-then-Translate (EAX) mode authenticated encryption. 53
4.12 The ChaCha20-Poly1305 algorithm [ChaPol]. 55

5.1 Confidentiality and integrity in the Cloud-IoT architecture. 62

5.2 Key distribution when splitting keys into confidentiality and integrity parts. 63
5.3 Architecture with double AEAD encryption. 65
5.4 Keys and algorithms needed to protect the data in this architecture. . . 66
5.5 Authenticated encryption with GCM mode and GMAC authentication. 67
5.6 Flowchart of the authenticated encryption in the cloud-IoT architecture with ChaCha20-Poly1305. 69
5.7 Authenticated encryption in the modified ChaCha20-Poly1305 algorithm using two keys, K_m and K_c. 72
5.8 MAC verification in the modified ChaCha20-Poly1305. 73
5.9 The modified ChaCha20-Poly1305 decryption algorithm using two keys. 73
5.10 The modified EAX decryption algorithm using two keys. 75

6.1 Screenshot of the output after the simulation of scenario **S1** is run. . . . 88

List of Tables

2.1 Operations of a ChaCha20 quarter round. 17
2.2 Block cipher modes of operation in symmetric-key encryption. 19
2.3 The use of public-key cryptography. 22
2.4 Key size equivalency in symmetric and asymmetric algorithms. 22

3.1 Security strength of cryptographic algorithms. 34
3.2 Comparison of key distribution alternatives Key Wrapping, ECDH Key Exchange, and Key Evolution. 38

4.1 Summary of the comparison of CCM, EAX, GCM, and ChaCha20-Poly1305 (Cha-Pol). 59

5.1 The inputs of the modified ChaCha20-Poly1305 algorithm 70
5.2 The inputs of the modified EAX algorithm 75
5.3 Average throughput (MiB/s) for AEAD encryption of different packet sizes [SLdP+19b]. 78
5.4 Comparison of key sizes, block-cipher calls, and ciphertext expansions of the design solutions from Chapter 5. 80

6.1 Performance time of authenticated encryption and authenticated decryption in seconds. 89

List of Symbols

AD Associated Data.

C Ciphertext.

IV Initialization Vector.

K Cryptographic key.
K_c Confidentiality key.
K_m MAC key.
KS Keystream.

M Message.
μC Microcontroller.
μP Microprocessor.

N Nonce.

P Plaintext.

T Authentication tag (MAC tag).

List of Acronyms

3DES Triple Data Encryption Standard.

AD Associated Data.

AE Authenticated Encryption.

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

CBC Cipher Block Chaining.

CBC-MAC Cipher Block Chaining-Message Authentication Code.

CCM Counter mode with CBC-MAC.

CMAC Cipher-based Message Authentication Code.

CTR Counter.

DH Diffie-Hellman.

DSA Digital Signature Algorithm.

E&M Encrypt-and-MAC.

EAX Encrypt-then-Authenticate-then-Translate.

ECB Electronic Code Book.

ECC Elliptic-Curve Cryptography.

ECDH Elliptic Curve Diffie-Hellman.

ECDSA Elliptic Curve Digital Signature Algorithm.

EtM Encrypt-then-MAC.

GCM Galois/Counter Mode.

GMAC Galois Message Authentication Code.

HMAC Hash-based Message Authentication Code.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IPsec Internet Protocol Security.

IV Initialization Vector.

KDF Key Derivation Function.

KEK Key-Encryption-Key.

MAC Message Authentication Code.

MCU Microcontroller Unit.

MITM Man-in-the-Middle.

MtE MAC-then-Encrypt.

NIST National Institute of Security and Technology.

NTNU Norwegian University of Science and Technology.

OMAC One-key MAC.

PPRF Puncturable Pseudorandom Function.

PRF Pseudorandom Function.

RAM random access memory.

RFC Request For Comments.

ROM read-only memory.

RSA Rivest-Shamir-Adleman.

SSH Secure Shell Protocol.

SSL Secure Sockets Layer.

TLS Transport Layer Security.

WPA2 Wi-Fi Protected Access 2.

Chapter 1
Introduction

This chapter will introduce the project and explain the overall goal and process of the master thesis. The first section introduces the cloud-IoT paradigm and presents the challenge this thesis will focus on solving. Next, the chapter will explain the motivation and utilization of this project. In the light of the challenge and the motivation, the chapter will define the scope and state the research questions this thesis will answer. Lastly, the chapter presents the process of this work and the thesis outline to give the reader a good overview of what lies ahead.

1.1 Cloud-IoT

Most people have heard of the term "Internet of Things" (IoT) today. IoT refers to physical devices that use internet access to collect and distribute data. IoT devices are becoming increasingly popular, and experts predict that their number will continue to increase in the near future. 5G technology is projected to expand the popularity of IoT gadgets even further [WCS+18] as it introduces more use areas.

IoT sensors capture data such as temperature, proximity, humidity, water detection, and touch events from their surroundings. The sensors produce a large amount of data that consumers can use in various ways to improve the efficiency, safety, and sustainability of buildings and infrastructure. Smart homes and smart cities rely on IoT sensors to benefit industries, healthcare, corporate offices, transportation, regional enterprises, and agriculture.

An IoT sensor usually includes a Microcontroller Unit (MCU). This little unit is a small computer that runs on a small chip. Because of their small sizes, they have many limitations, such as limited computational power and memory. These limitations reduce the number and type of applications and cryptographic algorithms supported in IoT sensors. Section 2.2 will go into more detail about such constraints.

Cloud computing is a good way to analyze data, handle and manage data streams,

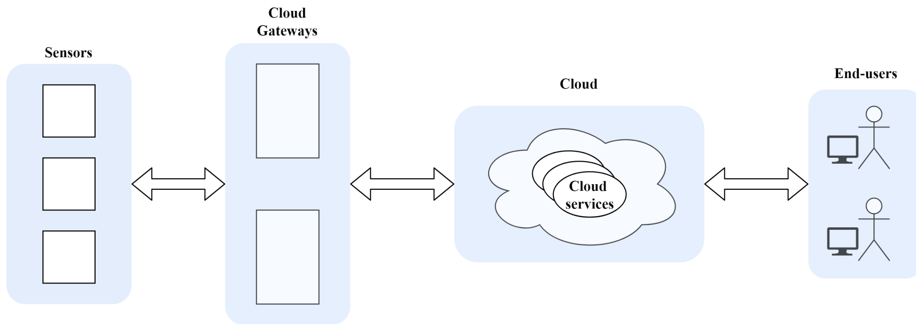


Figure 1.1: Cloud-IoT architecture [Tjo21].

and increase availability to end-users [LL15]. Without cloud computing, one must create a local system where all sensor communication occurs locally, such as in a building. This option is usually not favored because it demands a lot of power, maintenance, and resources. In cloud computing, sensors communicate and deliver sensor data to the cloud. Third parties often run the cloud solution, where services can process, analyze, and store this sensor data. The cloud-IoT architecture allows end-users to fetch and read the sensor data from the cloud. The system simplifies the retrieval and management of all sensor data for private individuals and businesses. Many applications exist for the cloud-IoT solution, leading to more optimal homes and workplaces. Figure 1.1 illustrates a typical architecture for such a system. The cloud gateways are an intermediate part that provides connectivity and allows the sensors to communicate transparently with the cloud.

1.2 Motivation

Despite its mentioned benefits, the cloud-IoT paradigm also introduces new issues. Data privacy, authorization, confidentiality, authentication, and access control are examples of such problems [MA20]. The attack surface in such a system is quite extensive, and one must implement security mechanisms to avoid attacks such as packet sniffing, Man-in-the-Middle (MITM), and spoofing attacks. Section 2.1 will elaborate on possible attacks and threats. This project will focus on protecting the security of sensor data throughout the communication and address the following problem: how can an end-user trust data from a sensor bought off the shelf after passing through multiple communication channels, migrations, caches, and storage to the customer's systems?

When it comes to information security, one often addresses the CIA Triad to secure data [PJW10]. It focuses on three principles: Confidentiality, Integrity, and Availability. To avoid the attacks mentioned, one must address confidentiality and

integrity. The report will not focus on availability because it concerns the system's uptime, which requires different mechanisms. The project will focus on confidentiality and data integrity to protect sensor data. In this report, data integrity will involve both data authenticity and integrity. There are sometimes different interpretations of the terms, but they are usually implemented in the same way in cryptography.

In the cloud-based IoT architecture, sensor data are transmitted over the air and stored in databases. Further, end-users can retrieve sensor data from the cloud. With robust encryption schemes, the sensor data can be transferred to the customer securely, as attackers do not have the resources to decrypt the ciphertexts sent over the air. Additionally, by introducing a MAC, we ensure message integrity.

There is another challenge as well. In addition to fulfilling encryption and integrity for the end-to-end communication between the sensor and the end-user, the cloud service must also verify the data integrity. Since the cloud service handles the data streams from IoT sensors, it must know that it communicates with a sensor and receives actual data and not noise. Also, if an attacker performs a MITM attack where it masquerades as a cloud gateway, it could send corrupted data to the cloud. The cloud stores the data, and end-users can use it as a backup service. If the cloud stores a lot of encrypted data without being able to read the data or verify the integrity, it could, in the worst case, be storing manipulated data without being aware of it. If that is the case, no one discovers the data manipulation before an end-user tries to validate the data integrity. This is especially problematic if the data is stored in the cloud over a longer period before the end-user extracts the data. If the information is manipulated or affected by noise, the parties must discard the data. Such scenarios can potentially lead to the loss of critical data. As sensors do not have the capability of storing the data, it will be gone forever. Also, the cloud service may require metadata from sensors to perform data processing. Such metadata can be information about sensors, type of data, or timestamps. If the cloud cannot read the sensor data because of encryption, it must rely on metadata sent in the clear. Due to threats like MITM attacks and spoofing that can tamper with or fake the metadata, the cloud party should verify the integrity to ensure data accuracy and consistency. An integrity check in the cloud would assure the parties that the data is correct and prevent the cloud entity from processing or storing corrupt data or metadata.

The motivation is to check the data integrity twice, once for the cloud service and again for the end-user. Overall, the system should meet the confidentiality and integrity requirements listed below.

Confidentiality and integrity requirements:

1. Sensor data should be end-to-end encrypted between the sensor and the end-user (confidentiality).

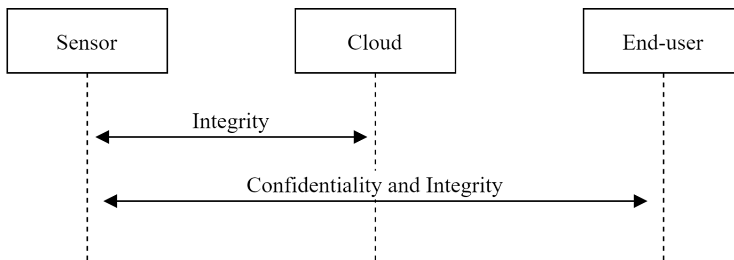


Figure 1.2: Confidentiality and integrity in the cloud-IoT architecture.

2. The end-user must be able to validate the consistency, trustworthiness, and accuracy of the sensor data it receives (integrity).
3. The intermediate entity, the cloud, must be able to validate the consistency, trustworthiness, and accuracy of the sensor data it receives without reading or disclosing sensitive data (integrity).

A solution that meets all three requirements would let the end-user trust sensor data sent through multiple communication channels, systems, and storage on its way from a sensor to the end-user. Also, it would let the cloud party securely handle the sensor data.

1.3 Objective

This project aims to ensure that a third party can maintain a cloud service that handles many users' sensor data while guaranteeing data security. The sensor data must be protected for the end-user to trust the data received. Therefore, the thesis will research ways to meet the confidentiality and integrity requirements mentioned above in the cloud-IoT architecture. Figure 1.2 illustrates how communication should be secured. While improving the security, the system should be easy to use both for end-users and management in the cloud. The end-users should retrieve the sensor data while the managers monitor and maintain the system.

The research questions from the project are maintained from [Tjo21] and are presented below. The overall research question in this thesis is:

RQ: How can end-users trust the security of IoT sensor data sent through different communication channels, systems, and services?

The research question can be divided into smaller and more specific research questions that will be addressed in this thesis:

RQ1: What type of cryptography and which algorithms should be used?

RQ2: How can the chosen solution fulfill integrity for a third party in the cloud and achieve integrity and confidentiality for end-users?

RQ3: How does key management affect possible solutions?

RQ4: What advantages and disadvantages does the solution introduce compared to other schemes?

The main focus of the master project is a design problem, designing a solution that solves the problem stated in this report. The thesis will address the research questions mentioned above and answer them. As authenticated encryption includes confidentiality and integrity, it is the main focus of this thesis.

1.4 Methodology and Thesis outline

This section describes the methodology and presents the outline of the thesis. It explains how the thesis will answer the research questions and discusses the purpose behind every chapter. The chapters describe the method of work of this thesis in chronological order. The following procedures are covered in this project: research and preparation, design, comparison, implementation, and validation. All chapters build on the previous ones and contribute to answering the research questions.

This report consists of 7 chapters in total. This thesis is a continuation of a pre-project [Tjo21]. Because of this, small parts in Sections 1.1, 2.1, 2.2, 4.1, and 4.4 are taken from or build on the project preceding this thesis. Figures 1.1, 4.1, and 4.5 originate from this project.

In order to answer the research questions **RQ1-RQ4** and fulfilling the confidentiality and integrity requirements from Section 1.2, there is a need for some background knowledge. Chapter 2 will research the state-of-the-art of IoT devices and cryptography. The chapter will explore constraints and threats that must be considered throughout the thesis and look at cryptographic primitives to solve the integrity and confidentiality challenges. To find what algorithms are most suitable for IoT sensors, symmetric-key encryption, asymmetric-key encryption, and MACs will be researched and discussed. The chapter will explain the most important algorithms that the thesis works on in the next chapters. The goal of Chapter 2 is to give sufficient knowledge that the next chapters can build on.

Chapter 3 aims to answer research question **RQ3** concerning the key management in the cloud-IoT architecture. Key management has many concerns and is a broad subject. Therefore, only the most relevant concerns in the scope of this thesis are

covered. The chapter discusses the key generation and distribution in detail with different concerns regarding the cloud-IoT architecture.

After studying algorithms, key management, and the state-of-the-art for IoT sensors, Chapter 4 starts adapting and applying knowledge and standards to the cloud-IoT situation. The chapter addresses authenticated encryption as it is one of the main focuses of this thesis. Authenticated encryption can combine separate confidentiality and integrity algorithms or be AEAD algorithms that achieve both in the same operations. This chapter will introduce the most common algorithms relevant to fulfilling the confidentiality and integrity requirements. Additionally, the chapter discusses the possibility of separating keys in AEAD so the actors can check the integration and confidentiality separately. It seems interesting to investigate if this is possible and useful. The preferred solution should not break the standards as it could reduce the security strength of the algorithms. Chapter 4 aims to learn about the properties of different authenticated encryption approaches and what is suitable for the cloud-IoT architecture to fulfill the integrity and confidentiality requirements. At the end of this chapter, the thesis has answered **RQ1** and provided knowledge towards answering **RQ2**

Further, Chapter 5 will propose different designs to fulfill the confidentiality and integrity requirements. The chapter will discuss different solutions and their advantages and disadvantages. As a continuation of Chapter 4, the chapter will look at authenticated encryption and solutions separating keys of AEAD algorithms, all this with **RQ3** and the key management aspects from Chapter 3 in mind. The chapter will compare the different approaches and discuss tradeoffs between the suggested solutions. As a result, Chapter 5 answers both research questions **RQ2** and **RQ4**.

Lastly, Chapter 7 will sum up the work in this thesis and explain how the thesis has fulfilled all research questions. The chapter will discuss and reflect on the work done and propose future work that would be interesting to research.

Chapter 2

Background and Related Work

This chapter looks at the state-of-the-art when it comes to security in IoT sensors and the cloud-IoT paradigm. Discussion of both challenges and current research in this field occurs in the following sections. As the thesis aims to secure sensor data, this chapter will first look at challenges and constraints to outline the current situation and investigate recent research and solutions. Since the cloud-IoT architecture includes many possible exploitations if not secured properly, the first section will look at the attack surface in the system. Further, the next section discusses the resource limitations in IoT sensors. Such sensors are usually resource-constrained devices, which introduces complexity when implementing cryptographic schemes in this architecture. Lastly, the chapter will focus on current research and related work to find the best approach to achieve confidentiality and integrity for the sensor data.

2.1 Potential attacks and threats

There are many possible threats in a cloud-IoT architecture. In such an architecture, attack points could be local user devices, IoT sensors, cloud gateways, and the cloud service, consisting of databases, administration, and virtual machines. Typical attacks on such a system could be packet sniffing, MITM attacks, spoofing, and cloud attacks [ROC+20]. Additionally, one must consider side-channel attacks that target the implementation of cryptographic algorithms. The following section will further describe the attacks to understand what must be prevented.

Packet sniffing Packet sniffing occurs when attackers intercept data over communication channels. Packet sniffing is of significant risk if the data is sent in cleartext or encrypted using weak encryption mechanisms that the attacker could break.

Man-in-the-Middle (MITM) attack We have a MITM attack if an attacker is positioned somewhere between the IoT sensor and the end-user during transmission. This scenario allows an attacker to eavesdrop and potentially tamper with the data.

A MITM attack can, for instance, happen if IoT sensors communicate with the cloud through cloud gateways. Here, an attacker could pretend to be a cloud gateway and alter the data sent from sensors toward the end-user. Another possibility is that employees working with monitoring or development in the cloud service could read and tamper with the data before sending it to the end-user.

Spoofing A spoofing attack happens if a user or a program impersonates somebody else by falsifying data. Here, an attacker can pretend to be another user or IoT device by altering the MAC or IP address and retrieving or sending confidential data. Also, in the case of nonce reuse, an attacker can pretend to be a valid user by replaying a message. Section 2.4.1 explains the latter case in more detail.

Cloud attacks Attackers could get unauthorized access to data or resources in the cloud by exploiting vulnerabilities in APIs or other services. Another possible cloud attack could be if a legitimate cloud user acts maliciously and leak credentials or other important data.

Side-channel attacks Side-channel attacks exploit information acquired from the system. Side-channel attacks target specific implementations and not abstract algorithms. Attackers can perform physical information leakages and exploit information from, for instance, electromagnetic radiation, power consumption, or timing information [Sta10]. Such attacks do not require tampering with the sensor but can occur by only observing the behavior of the devices. Side-channel attacks can also be invasive. If attackers physically acquire a cheap sensor, they might extract cryptographic keys from the sensor. IoT sensors usually do not have physical protection. Extracting keys is not an easy job, but it is possible. Therefore, one should consider this when assessing potential attacks and threats. If someone steals a long-term key, it will disclose non-authorized data that should be kept secret from the attacker. However, this depends on the type of sensor, and the consequences vary. If sensor data are encrypted using a scheme that ensures forward secrecy, the session keys will not be compromised if the attacker obtains the long-term key. Forward secrecy will be discussed further in Chapter 3.

This project will consider attacks targeting packets sent over the air through the cloud until it reaches the end-user. Such attacks are packet sniffing, MITM attacks, and spoofing. The mapping of potential risks underlines the importance of finding the answer to the question; how can the end-user trust the sensor data received? One must create a system that prevents attackers and unauthorized users from accessing or tampering with sensor data.

2.2 Resource limitations in IoT sensors

To ensure that the data is protected against both passive and active threats, it should be encrypted and authenticated. Implementing cryptographic algorithms is a challenging task for IoT devices because of their resource limitations. The capabilities of a sensor depend on the MCU. The microcontroller determines capabilities such as cryptographic operations, performance, and memory. Therefore, possible challenges with encryption schemes in IoT devices can be limited computational power, little random access memory (RAM) and read-only memory (ROM), low battery power, real-time response, and low bandwidth [TRK20] [SSM+17]. Not all sensors even have an internal clock, which can be a challenge in cryptography as the sensors cannot relate to time. The challenges exist because IoT sensors are physically small, low-power technologies with limited resources. The smaller the sensor, the lower the computational power, the smaller battery, and thus less extensive support for advanced encryption schemes. The list below states possible challenges for constrained devices such as IoT sensors.

- **Limited computational power:** Many IoT devices struggle with performance as they use low-speed processors. Therefore it is hard to perform expensive operations in terms of computation power.
- **Limited memory space:** Because of resource limitations, tiny devices have limited memory storage. Limited memory makes it hard to run applications and computations that require a lot of memory storage.
- **Low battery power:** IoT devices are low-power devices with limited battery life. The devices must save as much energy as possible and only be active when necessary. One should therefore avoid performing operations that require a lot of battery power.
- **Real-time response:** IoT sensors that serve real-time applications must respond quickly. Then it is essential to increase the throughput and lower the delay. One should therefore avoid time-consuming operations.
- **Low bandwidth:** Sometimes, IoT networks operate in an area of low bandwidth. Therefore, one must consider the data size to ensure that the most important data gets through.

Ensuring sufficient security for IoT devices is challenging because of several conflicting requirements. Achieving good performance, attaining a battery life of many years, and reaching a sufficient security level requires different measures and implementations. One should consider this while also contemplating the tradeoff between security, cost, and performance. An IoT sensor with adequate security

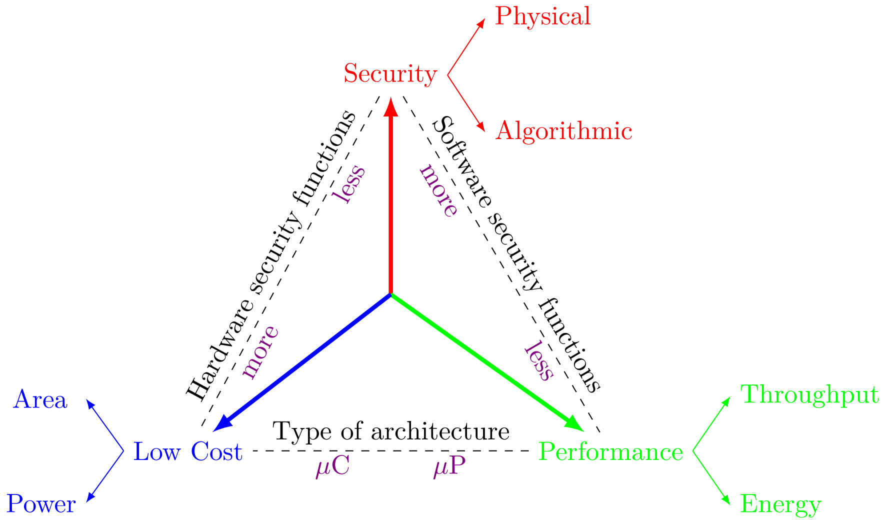


Figure 2.1: Tradeoff in cryptography for IoT sensors.

implementation might withstand attacks but can sometimes be too expensive in cost and performance. In contrast, a cheap sensor with a weak security implementation could have great performance metrics but are vulnerable to attacks. Figure 2.1 illustrates tradeoffs for IoT sensors in terms of cost, security, and performance. The type of architecture distinguishes between μC and μP. The first is based on a microcontroller (μC) with peripherals on the same chip. The latter architecture uses a microprocessor (μP), a more powerful CPU with external peripherals.

When considering a tradeoff between security, cost, and performance in resource-constrained devices, one should consider the following properties according to National Institute of Security and Technology (NIST) [MBTM17]:

Physical properties:

- *Area:* The physical area that is necessary to implement cryptographic primitives. One considers both the complexity and the size of digital electronic circuits. It is desired to have a low physical area.
- *Memory:* The amount of ROM and RAM that is needed. The lower the memory consumption, the better.
- *Implementation:* Algorithms can be implemented on software, hardware, or both. The best approach depends on the use case, resources available, the desired functionality, and what characteristics to focus on.

– *Energy:* The energy consumption in digital circuits should be as low as possible.

Performance properties:

– *Latency:* The latency is measured in seconds and should be minimal.

– *Throughput:* The amount of data processed per clock cycle or time unit. The throughput should be as high as possible.

– *Power:* The power consumption defines how much power is needed to use the circuit and perform operations. This property should be as low as possible.

Security properties:

– *Minimum security strength:* One must consider the number of operations required to break the algorithm. The security strength depends on the algorithm and cryptographic key.

– *Attack models:* Such cryptanalysis considers different cryptographic attack scenarios where the attacker has various system access. One should consider attack models such as known-plaintext and chosen-ciphertext attacks.

– *Side-channel resistance requirements:* As mentioned in Section 2.1, the attacker can extract secrets from devices in side-channel attacks. One should address and prevent as many side-channel attacks as possible by implementing sufficient physical, technological, and algorithmic security.

The challenge is to find cryptographic schemes that require low computational power and consume the least possible energy while simultaneously providing sufficient security with acceptable performance. Such issues are researched extensively in the cryptographic community.

2.3 Current research

The rapid growth of the IoT devices has made it hard to keep up with security. There is a need to develop new cryptographic algorithms to ensure effective end-to-end communication in general IoT communication. The algorithms must be suitable for low-power devices and fulfill all security requirements. Research regards both new algorithms and adaptations of existing algorithms. The research that focuses on cryptography for constrained devices with big limitations is *lightweight cryptography* [BP17]. The term lightweight has many meanings and may differ when looking at

algorithms with different properties designed for different environments. However, the main focus of lightweight cryptography is that such encryption methods feature low computational complexity.

2.3.1 Lightweight cryptography

The problem today is a lack of standardized lightweight encryption schemes designed for IoT sensors. Although some regular algorithms can perform lightweight operations, there is a need for dedicated lightweight algorithms. Most cryptographic algorithms are designed for desktop and server settings, not resource-constrained environments. There is a lot of research and suggestions regarding lightweight algorithms. Still, it is hard to know what schemes are secure and how interoperable they are in different protocols and systems. Many divergent use cases, environments, constraints, and algorithms make lightweight cryptography hard. This section will look at various aspects of lightweight cryptography and outline today's state-of-the-art.

One can implement lightweight cryptography in both hardware and software. The implementation of cryptographic primitives relies on different metrics. Hardware implementations depend on throughput, latency, memory, and power consumption. Software implementations depend on code size, throughput, and memory (RAM) consumption [BP17]. Cryptographic algorithms run on the MCU in a software implementation, while some operations can be hardcoded in the circuit in hardware implementation. Generally, cryptography implemented on hardware is the best approach for performance.

There are several categories of lightweight cryptography. This thesis will divide lightweight cryptography into two sub-categories: lightweight implementation of regular algorithms and dedicated lightweight algorithms. Constrained devices today often use regular standardized algorithms if they support lightweight implementations. AES is an example of this. Standard algorithms that support lightweight implementations will be discussed later in this chapter. Although AES is one of the most used approaches today, dedicated lightweight algorithms are needed. Both organizations and researchers are currently working towards standardizing more lightweight algorithms.

The start of lightweight cryptography

Lightweight cryptography is not a new phenomenon but has been researched for many years. Several algorithms designed in the 80s and 90s were lightweight and aimed at devices with low computational power, such as cell phones, car keys, and satellite phones. A5/1 and A5/2 were stream ciphers used in the cellular phone standard GSM. They were considered lightweight ciphers but are now broken. IoT devices are usually more constrained today than cell phones and require even lighter

cryptography. Nevertheless, even though devices have become smaller, the sensors and MCUs have improved, and their capabilities have evolved due to Moore's law [ML16].

In early cryptography, it was normal to keep the algorithms secret. However, the algorithms were often leaked or reversed, which led to people breaking them. Today, one no longer keeps the algorithms secret as cryptography usually follows Kerckhoffs's principle. This principle says that the system should still be secure if the public knows everything about a cryptosystem except the key [Sma16]. This approach has introduced interoperability and created a more open cryptographic community resulting in many cryptographic proposals and standards.

2.3.2 Lightweight cryptography today

In the last years, academics have published many lightweight algorithms. There are lightweight encryption schemes proposed both for cloud computing and IoT devices. Some suggestions for IoT environments are a lightweight attribute-based encryption scheme [YCT15], Hybrid Lightweight Algorithm (HLA) [SSM+17], and lightweight encryption for smart homes using identity-based encryption [SBSD16]. Many proposals introduce advantages, but some also bring disadvantages, such as poor scalability or the introduction of new attacks.

There are a lot of public institutions that work on lightweight cryptography. ISO/IEC has published standards that involve lightweight cryptography. The ISO-standard *ISO/IEC 29192-2:2019* includes algorithms such as PRESENT and CLEFIA [ISO29192]. PRESENT is an ultra-lightweight block cipher developed based on hardware optimization, designed for low-power devices where chip efficiency is needed [BKL+07]. However, there are several attacks introduced on the ultra-lightweight block cipher [BN14] [Lee14]. Sony developed CLEFIA intended for digital rights management, such as advanced copyright protection [Sony]. Additionally, Cryptography Research and Evaluation Committees (CRYPTREC), started by the Japanese Government, established a Lightweight Cryptography Working Group in 2013 [Gro+17]. The group aims to help users make technical decisions and select appropriate lightweight algorithms in constrained environments. CRYPTREC aims to recommend standardized algorithms, not to standardize lightweight algorithms formally.

In 2013, the National Security Agency (NSA) released two lightweight block ciphers called Simon and Speck. The purpose of the algorithms was to secure constrained environments [BSS+13]. National Security Systems (NSS) use Simon and Speck in highly constrained environments where using AES-256 is not possible or supported [NSAcyber]. In 2014 NSA attempted standardizing Simon and Speck, but ISO did not approve it due to uncertainty about whether it was trustworthy.

NSA refused to answer some questions regarding the algorithms. It was speculations that the NSA knew about weaknesses and exploitations that they kept secret from the public. NSA later put the algorithms in the public domain and published over 70 papers to conclude that the algorithms are secure [RDJ+17] [NSAcyber]. In 2018, Simon and Speck were included in the ISO standards *ISO/IEC 29167-21* [ISO29167-21] and *ISO/IEC 29167-22* [ISO29167-22]. Both are air interface standards for radio frequency identification (RFID) devices.

There are many literature proposals regarding lightweight cryptography, but not many systems use lightweight algorithms today. Some common cryptographic standards have use cases overlapping with use cases of lightweight cryptography. NIST has approved two cryptographic standards for block ciphers in resource-constrained environments: AES and Triple Data Encryption Standard (3DES) [MBSM16]. NIST is working on a deprecation timeline for 3DES, where they recommend that all users migrate to AES [CSRC17]. Therefore, one should avoid 3DES. Despite much research on new lightweight algorithms, AES is still the popular choice for constrained devices. The algorithm uses operations seen as lightweight and is therefore suitable for IoT sensors. AES is standardized by ISO and recommended by several public institutions such as NIST, CRYPTREC, and NESSIE (New European Schemes for Signatures, Integrity, and Encryption). Protocols such as Bluetooth, Wi-Fi Protected Access 2 (WPA2), and Zigbee (IEEE 802.15.4) use AES frequently [IEE20][LDS09][BP17]. AES is standardized for use in most areas because of its versatility, as many platforms and different use cases support the algorithm. Therefore, many implementations are currently building on the algorithm in constrained environments rather than implementing a newly proposed algorithm.

The work towards standardizing more lightweight cryptography

Work is being done to create and standardize lightweight block ciphers with smaller block sizes, smaller key sizes, simpler rounds, simpler key schedules, and minimal implementations [SSM+17]. NIST is currently working on finding lightweight algorithms suitable for constrained environments such as sensor networks. They examine both new AEAD schemes and new hashing schemes. In March 2017, NIST published a report on lightweight cryptography where they stated the following [MBSM16]:

> *The landscape for lightweight cryptography is moving so quickly that a standard produced using the competition model is likely to be outdated prior to standardization. Therefore, the most suitable approach for lightweight cryptography, in terms of timeline and project goals, is to develop new recommendations using an open call for proposals to standardize algorithms.*

According to the mentioned report, NIST wants to create a portfolio of lightweight algorithms and modes suitable for use in constrained devices. The organization intends to bind algorithms to profiles, where each profile contains metric ranges and algorithms goals. Such profiles would simplify implementation as different algorithms correspond to profiles that describe various characteristics. NIST has been working with a selection process of lightweight algorithms for the last few years. The agency has been through 2 rounds of reviews and announced in 2021 [TMC+21] that it is starting on the final round with ten finalists remaining.

When it comes to lightweight cryptography today, one can easily be confused. There are many literature proposals, but none are currently widely used. One should not implement a scheme that is not proven secure by trusted authorities or lacks trial or research. As there are no clear recommendations or standards in lightweight cryptography that stand out and are tailored to the problem stated in this report, it is reasonable to look at the lightweight implementation of regular algorithms. This is because they are more widespread than other proposed lightweight schemes, and the world's leading organizations and institutions focusing on cryptography recommend such algorithms.

2.4 Cryptography

This section will look at standardized algorithms widely used in cryptosystems today. The aim is to filtrate out the algorithms that are not suitable for constrained devices and examine those that could meet the integrity and confidentiality requirements stated in Section 1.2. It is important to get an overview of the cryptographic algorithms to answer the research questions this thesis addresses.

In modern cryptography, one distinguishes between asymmetric-key cryptography and symmetric-key cryptography. Symmetric-key cryptography uses one shared key to encrypt and decrypt messages, while entities in asymmetric-key cryptography possess one public and one private key. Asymmetric-key cryptography is also known as public-key cryptography. This section will look at both mentioned types of systems.

2.4.1 Symmetric-key cryptography

In symmetric-key cryptography, the sender and receiver must possess the same key. The reason is that the same key both encrypts and decrypts a message. Symmetric-key algorithms achieve confidentiality, which makes the private data unreadable to attackers. There are two ways to implement symmetric-key algorithms: stream cipher or block cipher. The first one processes the data bit by bit, while the second process data blocks. Both techniques will be described, but firstly the cryptographic term *nonce* will be explained, as symmetric cryptography diligently uses nonces.

Nonce The nonce (N) is a unique, random, or pseudorandom number. The nonce is short for *"number used once"* assigned to the data to be protected. A nonce is needed as it introduces randomness and prevents replay attacks. A nonce must only be used once per key. Otherwise, attackers can, at worst, replay, decrypt or spoof messages [VP17] [CWE-323]. If systems allow for nonce reuse, an attacker could replay different messages because the messages are valid more than once. Consequently, an attacker can intercept communication between two parties and spoof messages by sending the messages masqueraded as one of the valid users. Cryptographic algorithms often use the nonce as an Initialization Vector (IV) in the initial state. If algorithms use the same nonce and key on several messages, the attacker can decrypt a message in the worst case. Let us assume a stream cipher that encrypts two plaintexts, P_1 and P_2, into C_1 and C_2 using the same K and N. The encryption performs the following operation: $C = (P \oplus (K, N))$. The symbol $\oplus$ represents the logical operation XOR (exclusive OR). Then, an attacker can perform the following computation: $C_1 \oplus C_2 = (P_1 \oplus (K, N)) \oplus (P_2 \oplus (K, N)) = P_1 \oplus P_2$. As the encryption uses the same nonce, there is no randomness in the encryption when using the same key. The result of XOR-ing two ciphertexts is the same as XOR-ing two plaintexts since XOR-ing K and N with oneself becomes the same. If the attacker in the example knows P_1, it is possible to disclose other plaintexts such as P_2.

Stream ciphers

A stream cipher can encrypt and decrypt a plaintext by XOR-ing the plaintext with a pseudorandom bit-stream, called keystream, bit by bit. Pseudorandom defines numbers generated by a deterministic algorithm but look statistically random. The definition of a stream cipher is:

$$P_i \oplus KS_i = C_i \tag{2.1}$$

The P_i is the i-th bit of the plaintext, the KS_i represents the i-th bit of the pseudorandom keystream, and C_i is the i-th bit of the ciphertext. Of the many stream ciphers published throughout the years, *ChaCha20* [LCM+16] has become a much-used stream cipher in known protocols such as Transport Layer Security (TLS).

ChaCha20 is a stream cipher that generates a keystream block and XOR it with the plaintext to generate the ciphertext. In order to understand the algorithm, we first have to look at how it generates the keystream. ChaCha20 calls a function referred to as the *ChaCha20 block function.* This function takes a key (K), nonce (N), and a block counter (ctr) parameter as input and outputs 64 pseudorandom bytes, as illustrated below:

$$\textit{ChaCha20_block_function}(K, N, \textit{ctr}) \rightarrow \text{64 byte block of random bytes}$$

c	c	c	c
k	k	k	k
k	k	k	k
ctr	n	n	n

Figure 2.2: Illustration of the initial ChaCha20 state.

Table 2.1: Operations of a ChaCha20 quarter round.

$a + b$	$d \oplus a$	$d <<< 16$
$c + d$	$b \oplus c$	$b <<< 12$
$a + b$	$d \oplus a$	$d <<< 8$
$c + d$	$b \oplus c$	$b <<< 7$

The block function performs several *quarter rounds* on the ChaCha state. A ChaCha20 state has 16 values of 32 bits represented in a 4×4 matrix. The 16 values of the initial ChaCha20 state come from the following: the four first values are constants (c), the next eight comes from the key (k), the thirteenth value is the block counter (ctr), and the last three are from the nonce (n). Figure 2.2 illustrates the initial ChaCha20 state presented as a matrix.

A quarter round operates on 4 of the 16 values with operations involving addition, XOR, and bit rotation. Each value in the matrix will go through 20 rounds; hence the name ChaCha20. This means that 80 quarter rounds are performed in total (each round operates on four integers, and there are 16 integers, $20 \cdot 4 = 80$). Figure 2.3 illustrates a ChaCha quarter round, where a_i, b_i, c_i, and d_i is the four 32-bit values. The operations in the figure is $\boxplus$ = addition, $\oplus$ = XOR, and $<<<$ = bit rotation to the left. Table 2.1 lists all operations performed in a quarter round.

When the ChaCha20 block function outputs a 64-byte string of pseudorandom values, we have a key block that can be XOR-ed with a plaintext block to create the ciphertext. Key blocks can be concatenated into a keystream if the block function is called several times and the counter parameter *ctr* increases every time. The ChaCha20 encryption function performs the operation described in Equation 2.1.

Block ciphers

Block ciphers are suitable and much used for IoT devices [WDH+19]. Block ciphers encrypt and decrypt data in blocks of fixed lengths. As mentioned earlier, the most used and approved 128-bit block cipher is AES. The algorithm works on a 4x4 array of bytes, performing four steps in 10-14 rounds on the bytes of data. The number of rounds depends on the size of the key. The four steps are XOR-ing the output with a

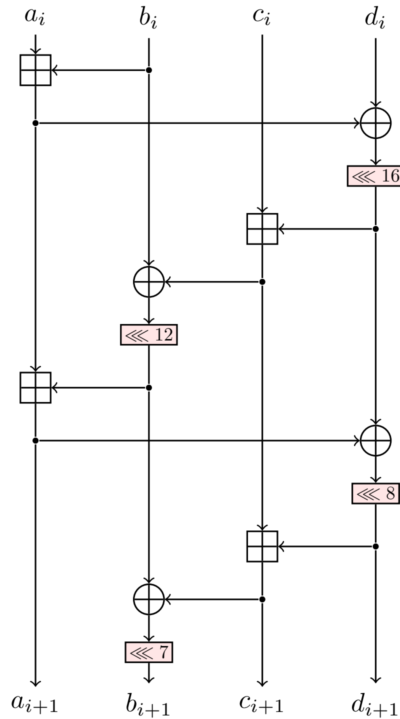


Figure 2.3: A ChaCha quarter round.

byte of the round key from the last round (AK), substitution of bytes (SB), shifting rows (SR), and mixing columns (MC). Figure 2.4 shows the four steps in an AES round. The widespread usage of the algorithm is because of the low-computational operations and its hardware support in processors. A hardware implementation of AES has performance advantages in terms of power and calculation [HH18], which makes it suitable for constrained devices such as IoT sensors. There are also other block ciphers that might be an alternative. Such block ciphers are GIFT-COFB, Romulus, and TinyJAMBU. They are all qualified for the last round of the NIST lightweight cryptography competition [TMC+21].

The last paragraph discussed the advantages of AES. However, one important detail is that the encryption only takes a single block at a time. This restriction reduces the flexibility, as all messages must be of the same size. Also, the same plaintext will always lead to the same ciphertext for the same key if someone maps plaintexts and ciphertexts together. The use of *block cipher modes of operation* will avoid such problems. Such modes are used together with a block cipher to ensure confidentiality, integrity, or both. Modes of operation can operate on several blocks

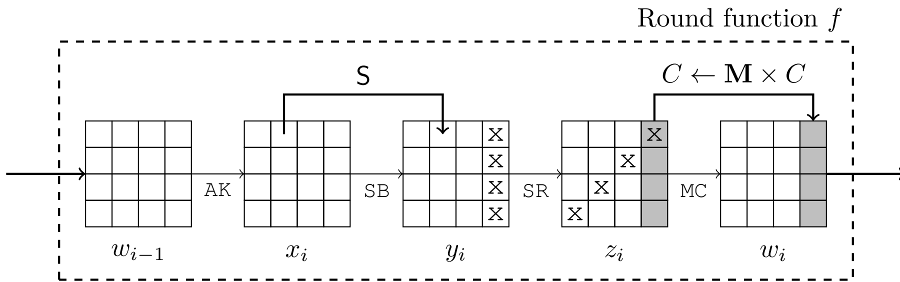


Figure 2.4: The round function in AES.

Table 2.2: Block cipher modes of operation in symmetric-key encryption.

Mode of operation	Confidentiality	Integrity	Status today
ECB	X		Not secure
CBC	X		Dropped by TLS 1.3
CTR	X		Underlying algorithm used by other schemes
CCM	X	X	Used in TLS 1.2 and 1.3
GCM	X	X	Used in TLS 1.2 and 1.3

in series and introduce a nonce to make identical messages distinguishable. Using block cipher modes of operation is therefore preferred. There exist a lot of different modes. Some of the most known are Electronic Code Book (ECB), CBC, CTR, CCM, and GCM. Table 2.2 gives a short overview of the mentioned modes.

CBC mode is a classical encryption mode much used in block ciphers. Figure 2.5 shows how CBC encryption works. Plaintext P is divided into blocks $P_0, P_1, ..., P_n$, which is encrypted, and XOR-ed with the previous encrypted block. The result is an unreadable ciphertext. The mode of operation uses a unique random value, IV, in the first block to make every message unique. Known protocols such as TLS 1.2 use CBC mode encryption. Additionally, the authentication scheme CBC-MAC uses CBC mode to ensure integrity. This scheme generates an authentication tag based on CBC encryption. Section 2.5 will discuss Cipher-based Message Authentication Code (CMAC), as the section will look at authentication schemes.

Another much-used block cipher mode is the *CTR mode*. The block cipher mode of operation is a synchronized stream cipher. CTR generates a keystream by using a nonce (N) and a counter (Ctr) that is increased for every operation. As the counter is incremental, the sender and receiver must be synchronized and use the same counter

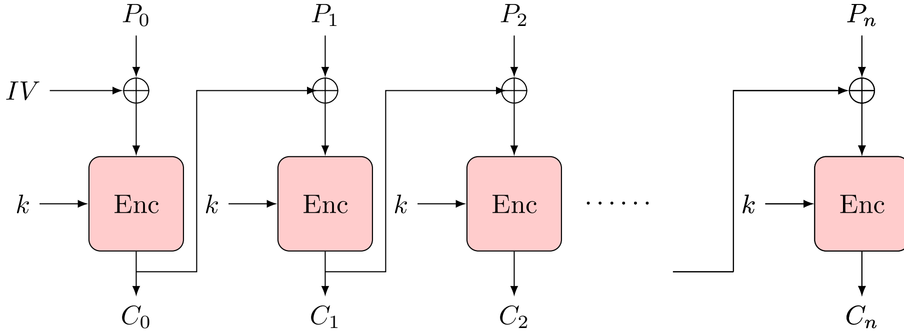


Figure 2.5: CBC mode encryption.

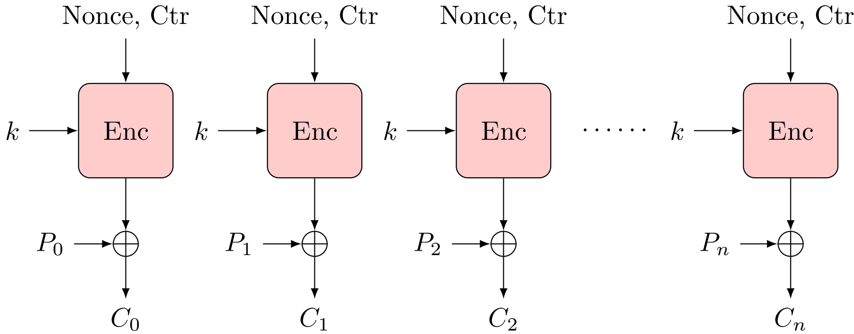


Figure 2.6: CTR mode encryption.

parameter. The keystream is generated as follows: $KS = Enc(K, N \mid\mid Ctr)$. The symbol || means joining strings end-to-end, denoted as concatenation. Further, the encryption is performed by XOR-ing the keystream and the plaintext: $C = KS \oplus P$. The mode allows for encryption and decryption in parallel, and it is possible to decrypt specific plaintext blocks without decrypting the whole stream, which can be a useful feature. Figure 2.6 illustrates the CTR mode encryption with plaintext blocks $P_0, P_1, ...P_n$ and ciphertext blocks $C_0, C_1, ...C_n$. CTR mode is not much used as a stand-alone encryption scheme, but as an underlying algorithm used in authenticated encryption modes such as CCM and GCM. These modes ensure both integrity and confidentiality. They are both approved by authorities and defined for established cryptographic protocols like TLS 1.2 and 1.3. This thesis will investigate some of these block ciphers further in Chapter 4.

Block ciphers and stream ciphers are reasonable approaches to protect sensor data. However, there are some challenges regarding key management. Since symmetric-key cryptography requires both parties to possess the same key, there is a need to

implement a secure solution where the sender and receiver exchange or agree upon using the same key.

2.4.2 Public-key (asymmetric-key) cryptography

In public-key cryptography, each entity possesses two cryptographic keys. One key is publicly available for everyone, referred to as the public key. The second key is kept secret for everyone except the key owner and is known as the private key. Typical encryption encrypts data using the receiver's public key. Then, the only person who can decrypt the following ciphertext is the entity owning the private key corresponding to the public key used. Therefore, the receiver is the only person that can obtain the plaintext.

In addition to encryption, public-key cryptography allows for digital signatures and key management. For the key management, parties can use public-key cryptography to agree upon a shared key for symmetric-key encryption. The sender and receiver can securely agree upon a symmetric key over an insecure channel using their public and private keys. Digital signatures can achieve message integrity by signing a message with one's private key. The signature allows the receiver to verify the message using the sender's public key. Public Key Infrastructure (PKI) is a typical key management system in public-key encryption. The infrastructure uses digital certificates and signatures for authentication and integrity. However, such an infrastructure is too costly for IoT sensors.

Many public-key algorithms are pretty divergent as they depend on different mathematical problems. One can divide public-key cryptography into three categories: finite-field, integer factorization, and elliptic curve.

Finite-Field Cryptography (FFC) Algorithms using finite-field cryptography are Digital Signature Algorithm (DSA) and Diffie-Hellman (DH). Finite field cryptography builds on mathematical problems in a finite field. A finite field means a finite number of elements in a set where subtraction, addition, division, and multiplication are defined. DH and DSA are based on modular exponentiation and the discrete logarithm problem. DH key exchange is a key agreement method that allows parties to agree on cryptographic keys over an insecure channel. The use of DSA ensures authentication and integrity to messages in public-key cryptography.

Integer Factorization Cryptography (IFC) The Rivest-Shamir-Adleman (RSA) algorithm is a much-used algorithm based on the integer factorization problem. The sender in RSA encryption can generate a public key based on two large prime numbers. The key is available to everyone, but the two prime numbers are the secret. The public key encrypts messages, and only receivers that know the large prime numbers

Table 2.3: The use of public-key cryptography.

Algorithm	**Data encryption (confidentiality)**	**Digital signature (integrity)**	**Key exchange protocol**
DH			X
DSA		X	
RSA	X	X	X
ECDH			X
ECDSA		X	

can decrypt the message. Without knowing the secret prime numbers, one must factor the product of the two prime numbers to obtain the secret numbers. As the prime numbers are very large, this is not possible today. In addition to encrypting data, the algorithm can transmit shared keys between parties using symmetric-key cryptography.

Elliptic-Curve Cryptography (ECC) Elliptic Curve Cryptography uses elliptic curves, an algebraic structure over finite fields. Two known algorithms based on the algebraic structure are Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve Diffie-Hellman (ECDH). The algorithms are variants of the DSA and DH algorithms, using only elliptic curve cryptography instead. ECDSA is a digital signature algorithm used for integrity and authentication, while ECDH is a key agreement protocol where parties can agree upon a shared secret key.

The algorithms in asymmetric cryptography are used for different purposes. Table 2.3 summarizes the algorithms and their use. There is only one algorithm for data encryption of the mentioned algorithms in public-key cryptography: RSA. However, RSA can be quite slow. Therefore, one favors using the algorithm for key exchange, not data encryption. Another factor to take into account is the key size of RSA. For instance, the algorithm requires a key size of 3072 bits for the same security level as AES gives for a key of 128 bits. Table 2.4 shows what key sizes different algorithms need for equivalent security.

Table 2.4: Key size equivalency in symmetric and asymmetric algorithms.

AES	**DSA & DH & RSA**	**ECDSA & ECDH**
128 bits	3072 bits	256 bits
192 bits	7680 bits	384 bits
256 bits	15360 bits	512 bits

As the table illustrates, the key size of public-key cryptography algorithms is very large compared to symmetric-key cryptography. Many public-key algorithms

introduce too big key sizes and complexity for resource-constrained devices like IoT sensors. This means that algorithms such as RSA are too complex for IoT sensors. The table shows that ECC uses key sizes that are more suitable for constrained devices. For instance, a 256 bits key provides adequate security as AES-128. Thus, ECC algorithms could be an alternative.

One important feature of key exchange protocols in public-key cryptography is *forward secrecy*. This feature ensures that keys used for sessions agreed upon with a key exchange protocol will not be compromised if the private key used in key exchanges is compromised. Only ephemeral DH and ECDH implementations provide forward secrecy of the above-mentioned key exchange protocols. The keys must be ephemeral, meaning that keys are generated for each session and discarded when the entities end the session. Chapter 3 will discuss forward secrecy and key exchange protocols in more detail.

Public-key cryptography is under the threat of quantum computers in the future. As mentioned earlier, the public key algorithm depends on different mathematical problems. Today the problems are too hard to solve, but if quantum machines become a reality, they will be able to perform calculations that could solve the mathematical problems [PQC]. NIST is currently working towards standardizing post-quantum cryptography because large-scale quantum computers will break many of the current cryptosystems.

The obvious choice to achieve confidentiality of sensor data is using symmetric-key cryptography. AES with a block cipher mode of operation and the ChaCha20 stream cipher are suitable approaches. Chapter 4 will research the symmetric encryption algorithms that allow for authenticated encryption in more detail. Since symmetric-key encryption depends on a shared key between the involved parties, it is reasonable to look at public-key cryptography for the key exchange. Chapter 3 will discuss the key management and distribution.

2.5 Message Authentication Codes (MACs)

The previous section looked at algorithms that achieve confidentiality, and now it is time to look at schemes that ensure data integrity. When sending sensor data over networks, some properties should be in place to avoid malicious actions from attackers. One must ensure that messages have not been tampered with during transmissions and verify what entity has sent a message. Data integrity fulfills these properties and can be implemented using a MAC, AEAD, or digital signature. This section will look at how MACs ensure data integrity.

A MAC is a function with two inputs, a key (K) and a message (M), where

the output is a MAC tag (T). The sender's authentication tag is produced in the encryption phase and added to the ciphertext. The receiver then recomputes this tag and compares it with the one received from the sender. If the recomputed tag is the same as the one received, the receiver knows that the content of the message is not modified. The equation below illustrates the generation of such a tag:

$$\text{MAC}(K, M) \rightarrow T$$

This section will look at two different ways to generate a MAC tag, using a hashing algorithm or a block cipher mode of operation. HMAC, CBC-MAC, and Poly1305 will be discussed. They are suitable approaches for IoT sensors because block ciphers and hashing algorithms have a low cost in terms of computational power.

2.5.1 Hash-based Message Authentication Code (HMAC)

HMAC is a much-used MAC construction based on an iterated hash function. The hash function uses a corresponding key and message as input and outputs a hash known as the authentication tag. The hash can be of different sizes depending on the hashing algorithm used. According to Request For Comments (RFC) 2104 [KBC97], the definition of the HMAC function is as illustrated below:

$$\text{HMAC}(K, M) = Hash\Big((K \oplus opad) \mid\mid Hash(K \oplus ipad \mid\mid M)\Big) \rightarrow \text{MAC tag}$$

The K is the key, and M is the message that is getting authenticated. The *opad* and *ipad* are two fixed strings. They are abbreviations for outer and inner padding. The two strings are XOR-ed with K to derive two different versions of the key that are computational independent. *Hash* is the cryptographic hash function. Figure 2.7 shows a visual presentation of the HMAC algorithm.

The HMAC algorithm is often referred to as HMAC-X because there are different hash functions one can use to achieve the authentication tag. Some of these are MD5, SHA-1, SHA-2, and SHA-3. However, not all are recommended for use today. Hash functions such as MD5 and SHA-1 are deprecated because of weaknesses [Tur11] [BR18]. The algorithms are insecure because there exist collisions of hash computations [SBK+17] that attackers can use to forge MAC tags and digital signatures. For HMAC to be secure, the hash function must have strong unforgeability. It should be computationally impossible for attackers to find a new message-tag pair after a chosen-message attack [BN00]. Such an attack is where an adversary can obtain pairs of a message and MAC, where the MAC is valid for the message. A valid user has not created the pair in the chosen-message attack, but it can allow the adversary to retrieve information and learn about the MAC system.

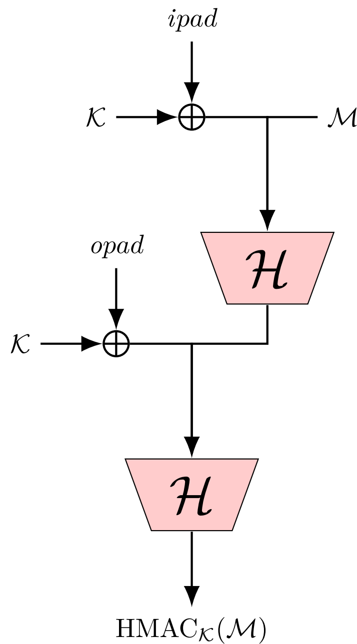


Figure 2.7: Illustration of HMAC.

2.5.2 CBC-MAC and CMAC

CMAC is a MAC based on the symmetric-key block cipher CBC mode. As described in Section 2.4.1, CBC mode encrypts chains of plaintext blocks, where each block depends on the previous encrypted block. Since the encryption of the blocks is interconnected, a change in one random bit in one of the plaintext blocks will change the output of the last encrypted block, M_n. This feature is the basis of the MAC as the last encrypted block is used as the authentication tag. Figure 2.8 shows how CBC calculates a MAC tag for a message M. The IV in CBC-MAC is always set to zero, $IV = 0$.

Today, CBC-MAC is only secure when used in CCM mode. NIST recommends not to use CBC-MAC as an authentication mode outside the CCM context [Dwo04]. The reason is that when used on variable-length messages, it is vulnerable to length extension attacks [Nan09]. However, CMAC is a much-used authentication scheme based on CBC-MAC that is safe to use independently. The CMAC builds on CBC-MAC but adds additional processing at the end of CBC-MAC, on message block M_n. CMAC derives two subkeys (K_1 and K_2) from the main key and uses the subkeys to

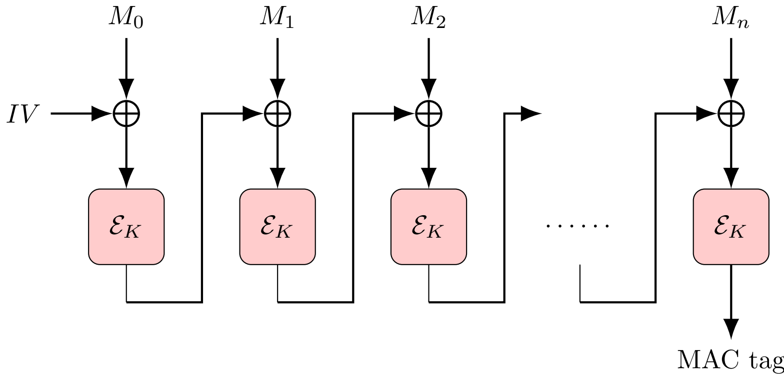


Figure 2.8: Illustration of CBC-MAC.

encrypt M_n into M_n':

$$M_n' = \begin{cases} K_1 \oplus M_n & \text{if } M_n \text{ is a complete block} \\ K_2 \oplus (M_n||10...0) & \text{if } M_n \text{ is not a complete block} \end{cases}$$

The computation of M_n' makes the authentication scheme safe on all variable-length messages. Therefore, CMAC would ensure the integrity of messages. The CMAC is also known as OMAC1. OMAC1 is a MAC proposal equivalent to CMAC, but CMAC is the cipher-based MAC specified and approved by NIST [Dwo16].

2.5.3 Poly1305

There are two versions of the Poly1305 MAC: one published by Bernstein requiring AES [Ber05] and one standardized in RFC 8439 Poly1305 [NL18]. ChaCha20 encryption is usually combined with Poly1305 rather than AES encryption in authenticated encryption. Additionally, the Poly1305 version described in RFC 8439 is used in known protocols such as TLS. Because of this, this thesis will focus on the latter, the variant standardized by Internet Engineering Task Force (IETF). Poly1305 is a one-time authenticator that uses a one-time key to produce a tag based on a message. The authenticator takes a 256-bit one-time key K_{otk} and message M as input and outputs a 128-bit MAC tag T:

$$\text{Poly1305}(K_{otk}, M) \rightarrow T$$

The MAC scheme divides the message into 16-bytes blocks handled as coefficients in polynomial operations. The one-time key should be generated pseudorandomly and divided into two parts: $K_{otk} = K_r, K_s$, where each part is 128 bits, and the pair (K_r, K_s) is unique. The algorithm clamps K_r, meaning that some fixed bits

of K_r are set to zero. The MAC scheme evaluates the polynomial at the value of the clamped K_r. The algorithm sorts the values K_r and K_s in little-endian order in the polynomial computations. This sorting means that the least significant value is placed first in the sequence. The calculations are performed modulo $2^{130} - 5$, where an accumulator calculates the sum of the polynomial evaluations. Lastly, the accumulator adds the value of K_s. The authentication tag is the 128 least significant bits, sorted in little-endian order, of the accumulator result.

This chapter aimed to look at the background and related work regarding the security of IoT sensors. We have seen that IoT sensors are constrained devices with many limitations, such as limited memory and low computational power. As the cloud-IoT architecture consists of many elements, the attack surface is quite large, and one must prevent attacks such as MITM, packet sniffing, and side-channel attacks. Lightweight cryptography is the best approach to achieving integrity and confidentiality for constrained devices. Since there is a lack of standardizations in this field, today's best practice is to look at lightweight implementations of current regular algorithms. This chapter has discussed different cryptographic primitives and their use, and as evidenced, symmetric-key algorithms and MAC functions are a good approach to secure sensor data.

Chapter 3

Key Management

This chapter is about key management. The thesis aims at securing sensor data with symmetric-key cryptography. Such cryptography can only be accomplished if the parties involved obtain a shared secret, a symmetric key. The key must be kept secret from everyone except the sender and receiver of the encrypted message. Therefore the two parties must agree upon a shared key to use in the communication. The key agreement must be done over a secure channel because an adversary might eavesdrop on the communication channel. For parties to obtain cryptographic keys, one must consider key management. Key management is about managing the cryptographic keys used in the system. It involves the generation, destruction, exchange, replacement, and storage of keys. This chapter will research key management and discuss relevant perspectives in the scope of this thesis to learn how key management affects possible solutions. This chapter will firstly examine cryptographic keys and their lifecycle. Further, sections will look at key generation and distribution. The different approaches will be assessed and discussed concerning the cloud-IoT architecture.

3.1 Cryptographic keys

There exist several different keys for diverse purposes. In SP 800-57 [BD15], a recommendation for key management, NIST specifies 19 different key types. In general, one distinguishes between public, private, and symmetric keys. The keys are designated for different use, such as key-wrapping, signatures, key transport, data encryption, key agreement, and authentication.

One can divide cryptographic keys into two categories; *session keys* and *long-term keys*. Long-term keys are static and used over a long period. Long-term keys can derive or protect the distribution of session keys. Long-term keys are commonly public and private key pairs in asymmetric cryptography but can also be symmetric. Session keys are derived for a session and used in a shorter period. Session keys are relevant for protecting the session between a sensor and the end-user. As session

keys preserve the secure communication between two entities, the keys are usually symmetric.

The security of long-term keys is crucial. If a long-term key gets compromised, an attacker can sometimes behave like the key owner and compromise previously used session keys, resulting in an attacker decrypting all previous communication based on the long-term key. If a session key is compromised, the consequences are less crucial. An attacker can decrypt the messages belonging to the key of that session but not previous sessions. As mentioned in Section 2.4.2, some key agreement protocols have forward secrecy to protect the long-term keys. This feature ensures that session keys are not compromised if the long-term key is compromised [PBM00].

Although it is normal to distinguish between long-term and session keys, the distinction has lately been more fluid. One also has intermediate keys somewhere between the two categories. For instance, in Section 3.3.3 later in this chapter, the thesis will discuss a scheme that evolves the long-term key for every session. Is it still a long-term key if one evolves the key per session? It is not always easy to distinguish the two categories, and sometimes keys are in the middle. Still, one should be aware of the distinction and avoid that a key compromise reveals previous sessions.

3.1.1 Key lifecycle

Key management is about handling the lifecycle of cryptographic keys. Strong algorithms are not good enough unless sufficient key management is in place [BBB+06]. The lifecycle of a key includes the following steps: generation, distribution, storage, usage, and destruction. This section will look at the different steps in a key lifecycle and discuss how they affect solutions to this thesis.

Key generation and distribution Key generation is about securely generating long-term and session keys. Key distribution ensures that parties agree upon and receive the necessary keys. Authenticated encryption relies on secure key distribution and that no other party than the ones involved gets hold of the keys. Section 3.2 and 3.3 will look into key generation and distribution.

Key storage In order to maintain secure communication, the distributed keys should be stored securely. An encryption application should store and manage the keys. The encryption application can either be the cloud party, the end-user, or a trusted third party. Keys should be stored such that the cloud party can retrieve the integrity key and the end-user retrieve a key or keys for integrity and confidentiality. IoT sensors should only store active session keys because of their limited memory.

Key usage A cryptographic key must have a special purpose, such as signatures, encryption, or key derivation. Generally, an entity should not use a key for several cryptographic purposes since it weakens security. However, one considers authenticated encryption as one process that provides several services, which is reasonable. In addition to specifying its use, a key should have a defined operational crypto period. According to NIST, the crypto period is when a cryptographic key is authorized for use [BD15]. The recommended crypto period depends on the key type, costs, and sensitivity of the data or keys to be protected. Key exchanges are very expensive in terms of resources for an IoT sensor, and the sensor should maintain a session as long as possible. A session key can last a sensor's lifetime if one does not terminate the session to avoid expensive costs. However, the longer the crypto period of a session key, the more damage is done if the key is compromised. The choice of the crypto period is a tradeoff that the actors must consider.

Key destruction A key is archived if it is no longer in operation. When archiving a key, the key management securely stores the key long-term. An archived key can be reactivated or used to decrypt previously encrypted data. Because of the constrained resources, sensors should delete session keys when a session is terminated. The cloud, end-user, or a third party should keep the keys as long as a party in the system stores encrypted sensor data, for instance, the cloud storing backup data. At the end of a key's lifecycle, key destruction occurs. The destruction happens when the parties consider a key useless, and there is no interest in keeping the key. That could be when the cloud deletes the backup data.

According to SP 800-57 [BD15], the key-management lifecycle consists of four phases: pre-operational phase, operational phase, post-operational phase, and destroyed phase. In the pre-operational phase, keying materials are generated and distributed. The keys are not ready for cryptographic operations before the operational phase. The IoT sensor stores the keying material on its RAM during operational use. In this phase, a key is either active or suspended. The active state protects information in terms of encryption or signatures. A key is suspended if there is a suspicion of key compromise. It is possible to restore a suspended key to an active state if the situation is safe. Otherwise, it can be deactivated, compromised, or destroyed, depending on the situation. When keys no longer are in normal use, they are in the post-operational phase. Here, the key is either deactivated or compromised. If the key is deactivated, keying material can still be accessed and used to process protected info. From the deactivated and compromised states, keys can only end up in the destroyed phase. In the final state, keys are destroyed and not available anymore. Figure 3.1 illustrates the four phases and the flow between them. Included in the phases are the possible states for the cryptographic keys.

The question of who should handle the key management in the cloud-IoT archi-

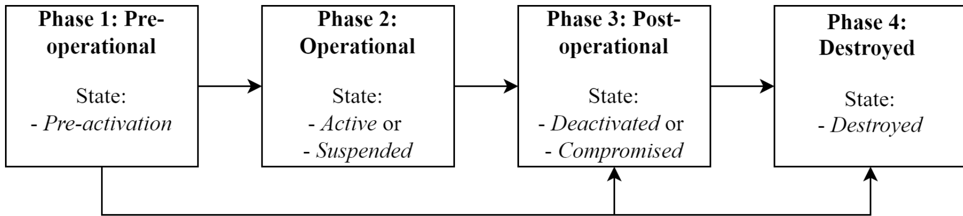


Figure 3.1: The four phases of the key-management lifecycle, including key states.

tecture is complex. If the cloud party handles the key management, will the end-user trust it? Is the end-user itself able to handle the key management, or should a trusted third-party get involved?

- One solution is that the end-user handles the key management and gives the cloud party access to necessary key material. The end-user could give the cloud access to an integrity key that the cloud party can extract. However, this approach would make the cloud coupled to the solution of the end-user. If there are many end-users with different solutions, it is not scalable.
- Another solution is that the cloud possesses all keys in the beginning and sends the necessary keys to the end-users. When the cloud has distributed the necessary keys to end-users, it could delete the keys it is not supposed to obtain. This approach is more scalable than the first solution, but it assumes that the end-user trust the cloud party.
- A third solution is to involve a third party, but this could increase the complexity.

The best solution depends on different scenarios, threats, and tradeoffs. This thesis will not go into more detail as the main task of this thesis is to focus on achieving authenticated encryption between the IoT sensors, end-users, and the cloud party. Cryptographic keys must be generated and distributed securely to meet the confidentiality and integrity requirements stated in Section 1.2. The scope of this thesis considers key generation and distribution the most important key management steps. Therefore, Sections 3.2 and 3.3 will discuss key generation and distribution in the cloud-IoT architecture.

3.2 Key generation

Cryptographic keys must be generated using an approved method. According to SP800-57 [Bar20], there are different ways to generate keys:

A: Parties can generate keys by using random number generators. Such generators output sequences of random bits that can be used as keys.

B: One can derive keys from another secret value, such as a master key or key derivation key. A Key Derivation Function (KDF) derives and outputs keys.

C: To generate a new shared secret, one can use key agreement techniques like Diffie-Hellman and MQV [BCK+17]. This approach handles both key generation and key distribution.

The approaches are suitable for different types of cryptographic keys. Generating long-term keys is usually done with the method described in approach **A**. Keys are generated randomly with random bit generators such as a True Random Number Generator (TRNG) or Deterministic Random Bit Generator (DRBG) [BK+07]. A TRNG produces a random sequence of bits based on an unpredictable physical process, while the DRBG approximates a TRNG using mathematical operations. The generators output a string of random bits that can be seen as a long-term key. In the cloud-IoT architecture, trusted authorities should generate the long-term keys randomly. Long-term keys for IoT sensors could be generated as described in key generation approach **A** and assigned during production to ensure a secure generation. If sensors obtain long-term keys during production, they do not have to implement and use resources to generate long-term keys. This solution allows the sensors to possess authenticated long-term keys when booting up.

The approaches in **B** and **C** are suitable approaches to generate session keys. Here, the parties involved agree upon a key that they will use for the current session. For session keys, key generation and distribution are often combined. Section 3.3 will discuss the two approaches in more detail. The most important thing is that a trusted key manager or third party is behind the generation when generating keys. Also, the keys must be of a key strength that gives sufficient protection. The key strengths vary depending on what cryptographic algorithms systems use.

One can achieve different security strengths for different cryptographic algorithms and key sizes. Since IoT sensors have constrained resources, it is desired to obtain sufficient security strength with small keys. Security strength measures the strength that a cryptographic algorithm achieves and specifies the number of operations required to break the algorithm. For instance, the AES-192 algorithm provides a security strength of 192 bits, meaning that it takes an attacker approximately 2^{192} operations to break the algorithm. Algorithms approved by NIST have estimated maximum security strengths of 112, 128, 192 or, 256 bits. Some algorithms have a maximum security strength of 80 bits, but they are not sufficient anymore and therefore deprecated [Bar20]. Table 3.1 shows the security strength of different cryptographic algorithms. It is based on the table in section 5.6.1.1 in the NIST

Table 3.1: Security strength of cryptographic algorithms.

Security Strength	**Symmetric Key Algorithms**	**Finite Field Cryptography**	**Integer Factorization Cryptography**	**Elliptic Curve Cryptography**	**Current status**
$\leq$ 80 bits	2DES[1] (K=112)	K_{pub}=1024 K_{pri}=160	K = 1024	K=160-223	Not acceptable
112 bits[2]	3DES[3] (K=168)	K_{pub}=2048 K_{pri}=224	K = 2048	K=224-255	Not recommended
128 bits	AES-128 (K=128)	K_{pub}=3072 K_{pri}=256	K = 3072	K=256-383	Sufficient security
192 bits	AES-192 (K=192)	K_{pub}=7680 K_{pri}=384	K = 7680	K=384-511	Sufficient security
256 bits	AES-256, ChaCha20 (K=256)	K_{pub}=15360 K_{pri}=511	K = 15360	K=512+	Sufficient security

[1] Deprecated in 2003 (SP 800-131A) [2] Not acceptable from 2031 (SP 800-131A) [3] Deprecated through 2023 (SP 800-131A)

publication SP 800-57, a recommendation for key management [Bar20]. The table shows the security strength of both symmetric and asymmetric algorithms. The symbol K represents the recommended key size, and if there are recommendations to distinguish between the key size of the public and private key, they are given as K_{pub} and K_{pri}, respectively. Only the last three rows in the table are sufficient security today.

3.3 Key distribution

After a trusted party has generated long-term keys, key distribution is the next step in key management. The goal is to establish a shared secret, known as the session key, between the sender and receiver before using symmetric encryption. If a third party gets hold of the secret key, the confidentiality breaks as the attacker can eavesdrop and decrypt the data without anyone noticing. Therefore, parties must securely agree upon and distribute keys. This section will explain different techniques for this: *key transport* and *key agreement*.

Key agreement Key agreement is a process where both the sender and receiver are part of the negotiation. Both parties contribute to the process of generating a shared secret. DH is the most known key agreement scheme. Key generation and key distribution are included in key agreement techniques.

Key transport Key transport schemes differ from the key agreement as only the sender selects keying material for the secret key and sends this securely to the receiver. Depending on the key transport scheme, the involved parties can distribute a secret key in several ways. When the initiating party generates keying material, it must send the secret to the receiver using a key distribution approach. Such approaches can be distributing keys through physical security procedures, key wrapping schemes, or a key transport scheme based on public-key cryptography such as RSA.

This section will consider three key distribution alternatives that use either key agreement or transport techniques. The objective is to discuss key management aspects and review if different approaches affect possible solutions in this thesis. All alternatives use pre-shared keys distributed between the parties in advance. A pre-shared key can be the long-term key for the IoT sensors generated and shared during production, as mentioned in Section 3.2. Pre-shared keys are appropriate for resource-constrained devices with limited CPU power and in environments where it is easier to configure them rather than introducing certificates [ET05].

3.3.1 Alternative 1: Key Wrapping

Key wrapping is a key transport scheme where one entity generates and sends a secret key to the receiving entity. Key wrapping is possible if the devices already possess a long-term encryption key securely obtained during production. Entities can send keys or key shares over an insecure channel by wrapping the secret using encrypting with the long-term key. Such a key is known as the Key-Encryption-Key (KEK). Both the wrapping and unwrapping processes of key data require the same KEK. According to SP 800-38F, there are two symmetric key-wrapping schemes: Key Wrap (KW) mode and Key Wrap With Padding (KWP) mode [Dwo12]. Both KW and KWP are modes of operation of the AES algorithm which protect the integrity and confidentiality of cryptographic keys. The advantage of such key wrapping schemes is that they only require symmetric cryptography operations. These are usually cheaper than asymmetric operations and thus preferable for constrained devices. Despite advantages in terms of cryptographic operations, there are disadvantages to this approach. The security of key data depends on the KEK, and attackers could obtain all key data protected with the KEK if disclosing the key-encryption key [SH02]. Therefore, this approach would not fulfill forward secrecy.

3.3.2 Alternative 2: ECDH Key Exchange

As mentioned in Section 2.4.2, the DH key exchange allows parties to exchange a secret key over an insecure channel securely. A variant of DH using pre-shared keys [BH09] [PH05] is a suitable approach for key distribution in the cloud-IoT architecture. The DH variant is defined for TLS and uses a pre-shared key to authenticate the

DH key exchange. This lets the sensors avoid using other computationally expensive public-key algorithms for authentication. The asymmetric operations in DH still need to be performed, but using pre-shared keys makes it more suitable for constrained devices. One important aspect of DH key exchange is the possibility of providing forward secrecy. Achieving forward secrecy happens if one uses a new DH private key for each session. Generating a new key for each session is referred to as ephemeral DH. If a long-term key is later compromised, attackers can only decrypt the corresponding session, not previous sessions. One can implement DH key exchange based on discrete logarithms or elliptic-curve cryptography (ECDH). Table 3.1 shows that it is more suitable for constrained devices as it uses a smaller key than DH for the same security strength. For a security level of 128 bits, the ECDH requires a key of at least 256 bits, while DH needs a key of 3072 bits.

3.3.3 Alternative 3: Key Evolution scheme

The last alternative covers key exchanges based on key evolution. SAKE (Symmetric-key Authenticated Key Exchange) is a proposed authenticated key exchange protocol that relies on a key evolution scheme and a resynchronization technique [ACF19]. The protocol only uses symmetric-key operations to achieve forward secrecy, a suitable approach for constrained devices. The result is that the parties can share evolving symmetric keys, known as the session keys. The protocol builds on shared master keys. Each party possesses two master keys, one for computing session keys (K_1) and another for authentication and resynchronization (K_2). The main master key, K_1, derives session keys using a Pseudorandom Function (PRF) with some random values as input. In order to obtain forward secrecy, the master keys are frequently updated using the key-evolving technique. Both parties use a function that cannot be inverted to derive new master keys from the old ones. The two master keys are updated simultaneously, so the evolution of K_1 and K_2 follows each other.

Several constructed protocols for symmetric key exchange deal with key evolution. A paper about authenticated key exchange protocols using lightweight operations based on pre-shared keys proposes protocols suitable for the architecture stated in this paper. LP3 and PP2 are some of the protocols suggested [BDdK+21]. Both achieve forward secrecy, and, like SAKE, these protocols are based on the approach where keys are derived and then evolved. The protocols use either linear or non-linear key evolution. Session keys are derived from a pre-shared long-term key before the long-term key evolves. Since the pre-shared long-term keys can be generated and distributed on IoT sensors during production, this approach is suitable for the Cloud-IoT architecture. LP3 is a protocol from the mentioned paper that uses linear key evolution to derive session keys which involves three messages between sender and receiver. Both parties possess a static key for MAC and a key derivation key for the evolution. The protocol computes session keys and derives key derivation

keys using a PRF. Every time a session key computation occurs between the parties, the key derivation key and a counter value are updated, allowing a party to catch up, evolving its key if they are out of sync. PP2 is a non-linear key evolution that uses a Puncturable Pseudorandom Function (PPRF). Such a function is based on hash functions and thus considered lightweight. PPRF is non-linear and can handle multiple concurrent sessions between two parties. However, the PPRF requires storage that can be challenging for some IoT sensors, and sensors in this scope do not necessarily require several concurrent sessions.

3.3.4 Key distribution in the cloud-IoT architecture

The three approaches are suitable for implementation in the cloud-IoT architecture. However, the best solution depends on the system's priorities as the alternatives give different advantages and disadvantages. One of the most important factors considered is forward secrecy. Alternative **1** does not ensure forward secrecy, alternative **2** ensure forward secrecy if ephemeral DH is implemented, and the last alternative do ensure forward secrecy. Because of this, alternatives **2** and **3** are the most favorable. However, as discussed in Section 3.1.1, if avoiding expensive costs is the main priority, one could use a session key for the whole lifetime of a sensor (as long as the session is not terminated). If one intends to use the same session key for as long as possible to avoid expensive operations, forward secrecy is not that important. In terms of data security, using a session key over a long time is not recommended because all data encrypted with the session key is disclosed if an attacker obtains the key. Assuming forward secrecy is a priority, alternatives **2** and **3** are the best. Table 3.2 illustrates the most significant differences between the alternatives.

Alternative **2** regarding DH is the most used today. TLS supports different DH modes, where ECDH with pre-shared keys is one of the modes [Res18]. As this is a widely used standardized scheme, it is natural to implement this approach. ECDH with pre-shared keys requires asymmetric operations every time the sensor and other involved parties establish a new session key. The frequency of deriving session keys depends on the length of a session. However, to fulfill the confidentiality and integrity requirements stated in Section 1.2, this thesis considers introducing several keys. Several keys would lead to more DH exchanges which are expensive for IoT sensors. Another aspect of alternative **2**, as mentioned in Section 2.4.2, is that DH is one of the asymmetric-key algorithms that will be vulnerable to quantum attacks when quantum machines become a reality. It is uncertain when this will occur, and cryptographic organizations consider the algorithm secure today. However, NIST is currently in the process of standardizing post-quantum cryptography [PQC]. Classic McEliece, CRYSTALS-KYBER, NTRU, and SABER are key-establishment algorithms considered for standardization in post-quantum cryptography [PQC-R3].

Table 3.2: Comparison of key distribution alternatives Key Wrapping, ECDH Key Exchange, and Key Evolution.

Features	Key Wrapping	ECDH	Key Evolution
Supports pre-shared keys	Yes	Yes	Yes
Key transport scheme	Yes	No	No
Key agreement scheme	No	Yes	Yes
Forward secrecy	No	Yes[4]	Yes
Symmetric cryptography	Yes	No	Yes
Asymmetric cryptography	No	Yes	No
Quantum attack secure	Yes[5]	No	Yes[6]

[4] Forward secrecy only with ephemeral DH. [5] With appropriate security parameters. [6] With appropriate security parameters.

Alternative **3**, using key evolution, is also a reasonable approach in the cloud-IoT architecture. Key evolution is cheaper than DH as evolving keys only use lightweight operations such as hash computations. Utilizing key evolution would also open up for a sensor to have two encryption streams: one to the end-user and one to the management service in the cloud. If a sensor maintains these two sessions simultaneously, the cloud can decrypt one stream but not the other one going to the end-user. This could be an interesting approach if using key evolution.

This chapter has looked at key management and focused on key generation and distribution. Key management in the cloud-IoT architecture is complex, and choosing the best alternative depends on threat scenarios, security level, algorithms, devices, and the system. The best approach for IoT sensors is to generate and distribute long-term keys during production. When exchanging session keys, pre-shared keys with ECDH or key evolving schemes are preferred. ECDH is the most widely used and the natural choice, but key evolution schemes are an interesting approach that systems potentially use more in the future. Both methods allow for pre-shared long-term keys. As this chapter has discussed, computing session keys can be expensive, especially when using asymmetric operations. The number of key exchanges and large key sizes are two important factors for constrained IoT sensors. The more keys introduced to the system, the more resources are required for the sensors to store, derive, and agree upon keys. The thesis will consider this complexity when designing solutions to authenticated encryption in the cloud-IoT architecture.

Chapter 4
Authenticated Encryption

The previous chapters have researched cryptographic primitives, algorithms, and key management suitable for IoT sensors. The thesis has explained that symmetric cryptography and MAC schemes requiring low-computational operations are ideal for protecting data in the cloud-IoT architecture. As the challenge in this thesis is to achieve integrity and confidentiality for sensor data, this chapter focuses on authenticated encryption. Up to this point, the thesis has discussed background knowledge and different standards suitable for IoT sensors. Starting with this chapter, the thesis will now apply and adapt the knowledge and standards to the scenarios in the cloud-IoT architecture. In addition to looking at how one can protect the information in end-to-end communication, the chapter will discuss how an intermediate party can verify the integrity without decrypting the data. Small modifications of the algorithms are considered to meet all requirements. The preconditions for this chapter are proper key management with secure key exchange.

4.1 Authenticated Encryption (AE)

Authenticated Encryption (AE) algorithms protect sensor data in terms of confidentiality and integrity. As TLS is a trusted and widely used protocol in many applications, this chapter will discuss AE approaches used in TLS. AE can typically perform two functions: encryption and decryption. The encryption function takes plaintext and a key as input and returns a ciphertext, while the decryption function takes the ciphertext and corresponding key as input and outputs the plaintext. Combining encryption with a MAC, discussed in Section 2.5, would ensure confidentiality and integrity. As the project preceding this thesis presented [Tjo21], three approaches combine encryption with MAC generation to achieve AE:

- **Encrypt-and-MAC (E&M):** The plaintext is encrypted, and a MAC is produced based on the plaintext message. The result is a ciphertext, and a MAC tag sent to the receiver.

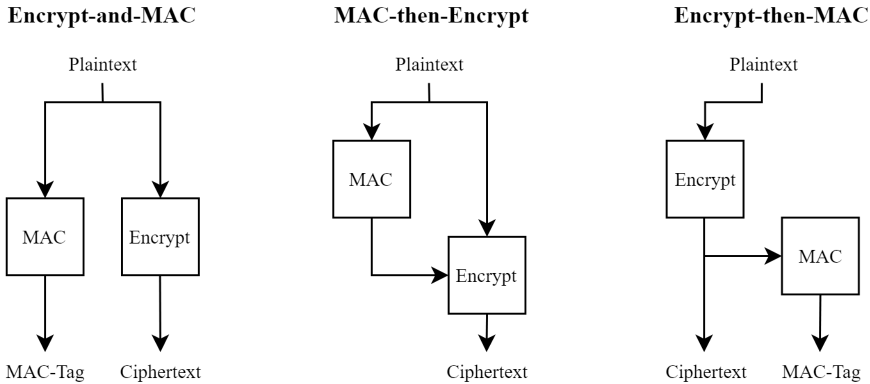


Figure 4.1: Encrypt-and-MAC, MAC-then-Encrypt, and Encrypt-then-MAC [Tjo21].

- **MAC-then-Encrypt (MtE):** The MAC is produced based on plaintext. Further, the MAC is appended to the plaintext, and they both are encrypted. The result is a ciphertext sent to the receiver.
- **Encrypt-then-MAC (EtM):** The plaintext is encrypted, and the MAC is produced based on the ciphertext. The MAC and the ciphertext are both sent to the receiver.

Figure 4.1 illustrates the authenticated encryption approaches. They all achieve confidentiality using an encryption algorithm and generate a MAC to ensure message integrity. This chapter will discuss the three techniques to learn suitable approaches for the integrity and confidentiality requirements in the cloud-IoT architecture.

4.1.1 CBC and HMAC

Combining CBC and HMAC is an approach to authenticated encryption as it achieves both confidentiality and integrity [DR08]. CBC provides confidentiality, and HMAC ensures integrity. CBC mode and HMAC have been described in Section 2.4.1 and Section 2.5.1, respectively, and are suitable for IoT sensors because of their low-cost operations.

MtE vs. EtM

CBC with HMAC can be implemented in both EtM, and MtE approaches. The EtM approach is preferred today, but Secure Sockets Layer (SSL) and TLS have used the MtE method over a long period to ensure AE. However, the MtE approach

has been proven to not be optimal after discovering several vulnerabilities. EtM, on the other hand, has been proven to be the most secure of the three AE approaches [BN00][Kra01]. TLS used a MtE approach that could more accurately be called MAC-then-Pad-then-Encrypt. This approach starts with padding the plaintext to the block size of the encryption function. Block ciphers are encrypting blocks of a fixed length, so if the plaintext is smaller than the block size, the blocks are filled up by applying bits referred to as padding. The sensitive data is encrypted using CBC mode after applying the necessary padding to the plaintext.

The weaknesses in the MtE approach used in TLS can result in padding oracle attacks [Vau02]. A padding oracle reveals information about the padding in ciphertexts during decryption. A chosen ciphertext can tell if the encrypted plaintext has valid padding or not if one interprets the responses. The result is attackers retrieving information to differentiate between invalid and valid padding for any ciphertext. Attackers get hold of such padding oracles by exploiting side channels such as error messages and timing responses. A padding oracle attack can lead to attackers decrypting data without knowing the encryption key. If an attacker can find out if a given ciphertext has the correct padding when decrypted, it is possible to crack the CBC encryption. Finding the valid padding is done by observing the error messages. When the attacker obtains the exact padding, the attacker can work byte-for-byte by starting with the last byte. It is only necessary to perform XOR calculations to decrypt the byte as this is how the decryption operation works in CBC mode. Despite using the integrity check, the exploit is possible because the authenticated encryption approach performs padding after the MAC generation. The result is a MAC that fails to prevent the attack because the validation happens after the padding in the decryption and not before, as it should have to avoid padding oracles retrieving information about the padding.

Because of the vulnerabilities in the MtE approach of CBC with HMAC, IETF later published an RFC [Gut14], a document for Internet standards, about the problem. The document included an improved security mechanism regarding replacing the MtE approach with the EtM approach. The more secure approach would mitigate weaknesses in TLS. Attacks such as The Lucky Thirteen [AP13], a well-known padding oracle attack based on timing attacks, were mitigated. Unlike MtE, the EtM method drops a tampered ciphertext before any padding information leaks because the MAC is validated first. The EtM is therefore not vulnerable to padding oracle attacks.

4.1.2 Authenticated encryption in the cloud-IoT architecture

In this project, the goal is to verify the integrity without allowing decryption of the ciphertext. That means that keys should be divided into an integrity key and an encryption key, so decryption and integrity verification can be done independently

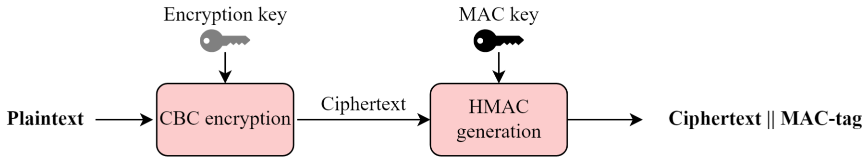


Figure 4.2: CBC-HMAC encryption.

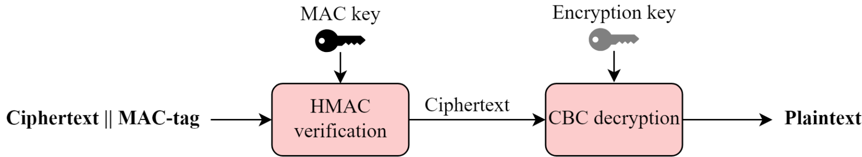


Figure 4.3: CBC-HMAC decryption.

of each other. When sensors transmit sensor data, it is first pushed to the cloud before end-users obtain the data. Therefore, the cloud should be able to verify the integrity without decrypting the data. Since the MAC verification must occur before decryption, only one of the three approaches is suitable, namely EtM. It is the only approach suitable because the authentication tag is generated based on the ciphertext. The cloud can re-generate the MAC tag using the ciphertext as input while not disclosing any sensitive data. Further, the end-user would possess both the integrity key and confidentiality key to verify and decrypt the data. The decryption of the ciphertext only takes place if the integrity is verified first. Because of this, it seems reasonable to choose algorithms based on the EtM approach where the integrity can be verified independently of the encryption.

The EtM approach of CBC with HMAC is a good alternative to authenticated encryption in the cloud-IoT architecture. In this approach, HMAC generates an authentication tag based on the CBC mode encryption of the plaintext, preventing attackers from tampering with the ciphertext. For instance, if an attacker tampers with the ciphertext, the authentication tag generated by the sender does not correspond with the newly tampered ciphertext and will therefore be seen as invalid. Because of this, an attacker can only get away with tampering with the ciphertext when knowing the HMAC key to generate a new tag corresponding with the tampered ciphertext. Figure 4.2 shows how CBC mode encryption and HMAC can be used in the EtM approach to ensure data confidentiality and integrity. Figure 4.3 illustrates the HMAC verification and decryption at the receiver's side. There are two keys in this system, and both parties possess a copy of each key.

CBC with HMAC is an AE scheme that would allow separating the integrity and confidentiality operations. Figure 4.4 illustrates how the distribution of keys

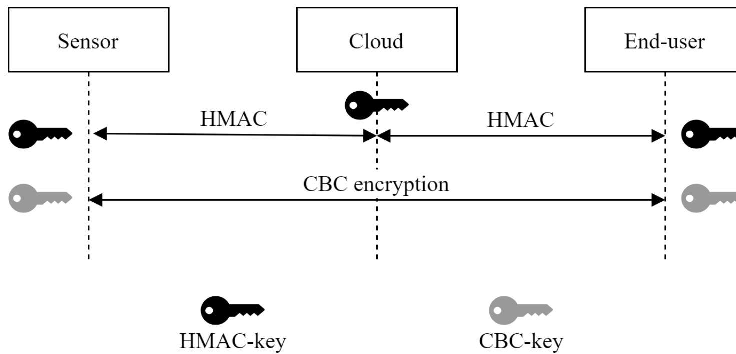


Figure 4.4: Distribution of keys using CBC with HMAC

should take place. The confidentiality key encrypts the plaintext, and the MAC tag is generated based on the ciphertext with the integrity key. If the cloud party only obtains the integrity key, the party can verify the data integrity when data is pushed to the cloud. If the end-user possesses both keys, it can verify the integrity and decrypt the data. Although it seems like a suitable approach for this architecture, there are indications that known protocols are moving away from CBC with HMAC.

SSL/TLS has been using CBC mode and HMAC-X algorithms for data confidentiality and integrity in versions ranging from SSL 2.0 to TLS 1.2. However, the newest version of TLS, TLS 1.3, no longer defines CBC with HMAC-X as an alternative to authenticated encryption. Algorithms such as CBC and non-AEAD ciphers do not exist in TLS 1.3 [Eri18]. Modes with integrated MACs, such as CCM and GCM mode, have replaced these schemes.

Despite that TLS has recommended EtM as an option for AE, the recent recommendation is to move away from this approach and use AEAD algorithms instead. TLS version 1.3 has excluded all symmetric encryption algorithms considered legacy [Eri18]. TLS 1.2 still supports AES-CBC with HMAC, but only if implementations eliminate timing side-channel attacks like Lucky Thirteen. CBC must be implemented carefully to be secure, and implementations have been demonstrated to be vulnerable several times. Also, different variants of the Lucky Thirteen attack have been published since the first attack, which requires implementation fixes. The possibility of attackers introducing new attacks could be problematic if sensors use an implementation of CBC with HMAC that is later compromised and require a fix. If sensors are out in the field, changing the implementation can be comprehensive and costly. In conclusion, a secure implementation of CBC with HMAC in the cloud-IoT architecture is a possibility, but known protocols do not see it as the future in AE.

As TLS 1.3 only uses AEAD algorithms, it is clear that this is considered the

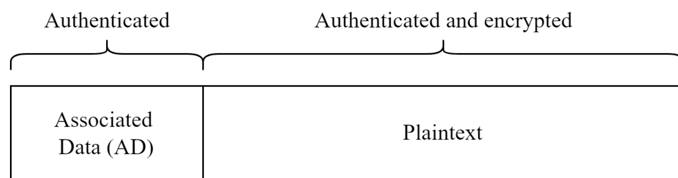


Figure 4.5: Authentication and encryption of data in AEAD [Tjo21].

best approach. Changing to AEAD schemes like AES-GCM would also mitigate the Lucky Thirteen attack [SHS15]. The AEAD algorithms handle both encryption and MAC simultaneously, which recently have become much used and accepted. Because of possible vulnerabilities with CBC mode, the many pitfalls of combining separate encryption and integrity algorithms, and the current focus on AEAD algorithms, it seems most sensible to look at the AEAD algorithms for possible solutions to the problem stated in this report.

4.2 Authenticated Encryption with Associated Data (AEAD)

AEAD is a type of AE that achieves confidentiality and integrity in the same algorithm. AEAD involves sending encrypted data with related metadata sent in clear, referred to as Associated Data (AD). AEAD schemes encrypt the plaintext and verify the integrity of both the AD and the ciphertext. The AD is sent in the clear because it consists of information that should be visible to everyone who receives or intercept the packet. It is useful for network packets where headers such as addresses, protocol versions, and port numbers are visible for routers that forward the packets. Including the AD in the integrity computations ensures that no one changes the information. The AD is optional and does not have to be included. Figure 4.5 illustrates a typical AEAD packet.

AEAD algorithms take plaintext (P) and AD as input. When encrypting data and generating authentication tags, there is a need for a key (K) and a nonce (N) known by the sender and receiver. The key ensures encryption and decryption, and the nonce is used as an initialization vector. Common for all AEAD algorithms described in this chapter is that the input and output are the same, even though the operations of the algorithms differ. The expression below illustrates the encryption process for AEAD algorithms:

$$\text{AEAD}(K, N, P, AD) \rightarrow C \mid\mid T$$

The output of the AEAD algorithms is the concatenated ciphertext C and

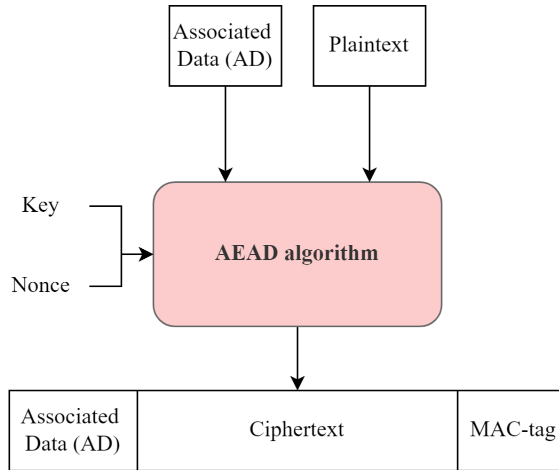


Figure 4.6: The encryption process of AEAD algorithms.

authentication tag T. Figure 4.6 illustrates the encryption process, where the result of the algorithm is a packet sent to the receiver.

The decryption process goes the opposite way of the process in Figure 4.6. The decryption consists of verification and decryption. The AD, C, and authentication tag T are the input, and the plaintext is the output. The key and nonce are the same as in encryption for the same data. The decryption process only returns plaintext if the MAC tag is correct. Otherwise, the algorithm returns an *INVALID* message, and the plaintext is not disclosed. The decryption process is illustrated below:

$$\text{AEAD}(K, N, C, AD, T) \rightarrow P \quad \text{or} \quad INVALID$$

AEAD algorithms are more beneficial for IoT sensors than separating algorithms for authentication and encryption because they are simpler to deal with than combining several algorithms. Implementing only one algorithm requires fewer resources, is simpler to implement, and securely combines encryption and authentication. In contrast, if one must implement two separate algorithms, it increases the chance of critical errors and incorrect implementations when combining them. In this thesis, one should keep in mind that it is desired to verify the integrity without decrypting the data. Therefore, while AEAD algorithms combine authentication and encryption, it is applicable to choose an algorithm where one can verify the integrity separately. If the cloud could detach and check the MAC isolated in AEAD modes, it could solve the problem. One must have separate keys for integrity check and encryption for this to be possible. The thesis will look at if it is possible to split the key or

generate two keys, one for authenticated encryption for the end-user and one for the integrity of the cloud party. This section will look at different AEAD algorithms and discuss such properties.

In ISO/IEC 19772:2020, there are three standardized AEAD modes of operation: GCM, CCM, and EAX [ISO19772]. They are all modes of operation for block ciphers in symmetric-key cryptography. NIST has approved two of the three AEAD algorithms: CCM and GCM. In addition to the schemes mentioned, there is a much-used AEAD stream cipher in TLS 1.3 called ChaCha20-Poly1305. This section will look at the above-mentioned AEAD schemes and discuss approaches involving separating the integrity check.

4.2.1 CCM

CCM combines (CTR) mode with Cipher Block Chaining-Message Authentication Code (CBC-MAC). The CTR mode, described in Section 2.4.1, ensures confidentiality, while CBC-MAC, described in Section 2.5, ensures integrity. One often refers to CBC-MAC as CMAC. The AEAD mode is a MtE approach with some modifications [Šve16]. Although it is a MtE approach, the AEAD algorithm is not vulnerable to padding oracle attacks or exposed to similar vulnerabilities as the MtE implementation of CBC with HMAC. The algorithm is protected against the types of invalid messages that attackers will try to create to retrieve information. The AEAD mode is proven secure as a whole unit and securely combines confidentiality and integrity [Dwo04] [Jon02] [WHF03]. The authenticated decryption performs decryption and verifies the integrity before returning any information. If the data is not authentic, the algorithm only outputs an INVALID message that does not leak any useful information for attackers.

The CBC-MAC is produced based on the plaintext, and the counter CTR mode is used on the MAC and plaintext to generate a ciphertext [Dwo04]. Figure 4.7 shows the authenticated encryption in CCM mode. There is one AD block (AD_1) and two plaintext blocks (P_1 and P_2) in the illustration. The AEAD algorithm takes in K, N, P, and AD. As CCM mode is based on a secure MtE approach, it starts with generating a MAC based on N, P, and AD [Hou07]. Further, CTR mode encryption takes place on the MAC and the plaintext, which results in the ciphertext. In the illustration, the ciphertext is the concatenation of C_1, C_2, and C_T sent to the receiver. The decryption process consists of decryption and verification in that given order. The application of CTR mode on the ciphertext decrypts the MAC and plaintext. Further, the CBC mode is applied to the plaintext and AD to recompute the tag and verify the correctness of the MAC. The operation only returns the plaintext if the tag is correct.

As described above, separating keys into integrity keys and confidentiality keys

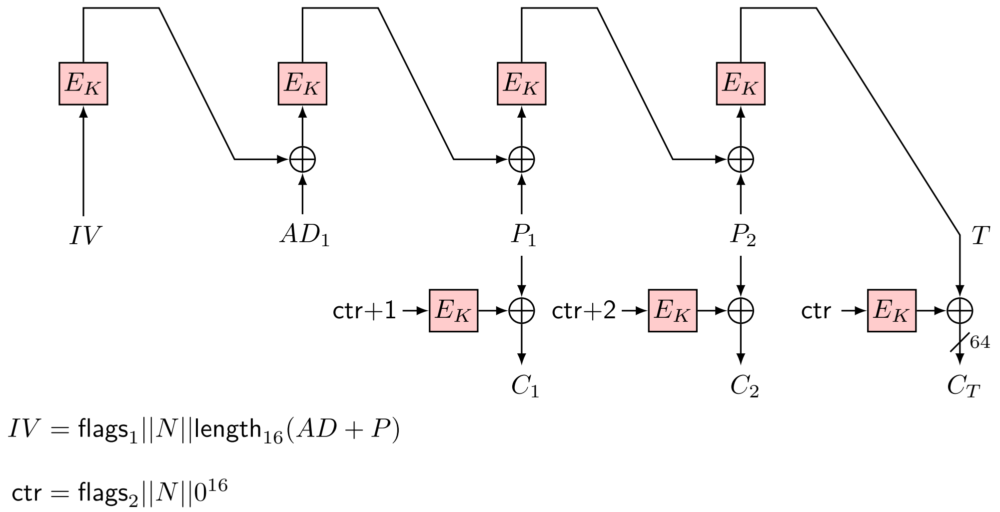


Figure 4.7: Illustration of the CCM mode.

would be an appropriate approach to fulfill the integrity and confidentiality requirements. However, this is problematic in CCM, where the MAC is generated first and encrypted with the plaintext. Given the example in Figure 4.7, the tag is calculated using CMAC: $C_i = E_K(M_i \oplus C_{i-1})$, where message M consists of AD, P_1, and P_2, and the tag is the result of $T = MSB_t(C_3)$. MSB_t is a specified number of the t most significant bits of the output. This tag is encrypted with the plaintext to obtain the ciphertext: $C = (P \oplus MSB_t(S))||(T \oplus MSB_t(E_K(ctr)))$, where $S = E_K(ctr + 1)||E_K(ctr + 2)$. For a party to verify the integrity of the ciphertext, the authentication tag is obtained by calculating $T = LSB_t(C) \oplus MSB_t(E_K(ctr))$. LSB_t is a specified number of the t least significant bits. This is only possible by possessing the key K, but by obtaining this key, one can also decrypt the ciphertext: $P = MSB_t(C) \oplus MSB_t(S)$. The MtE approach makes it problematic for the cloud party in the architecture to verify the MAC. Since the MAC is encrypted, the entity must decrypt the message before verifying the MAC. This means that the intermediate part must possess the decryption key as well. If the party must possess both keys, this architecture's confidentiality and integrity requirements are not fulfilled using CCM.

4.2.2 GCM

GCM is a block cipher mode that ensures data confidentiality by encrypting the plaintext with a variant of the CTR mode. GCM also uses a universal hash function, called GHASH, defined over a binary Galois field, to authenticate the data [Mor07].

Figure 4.8 shows a simplified illustration of the GCM mode using CTR encryption and GHASH authentication to illustrate how they are combined.

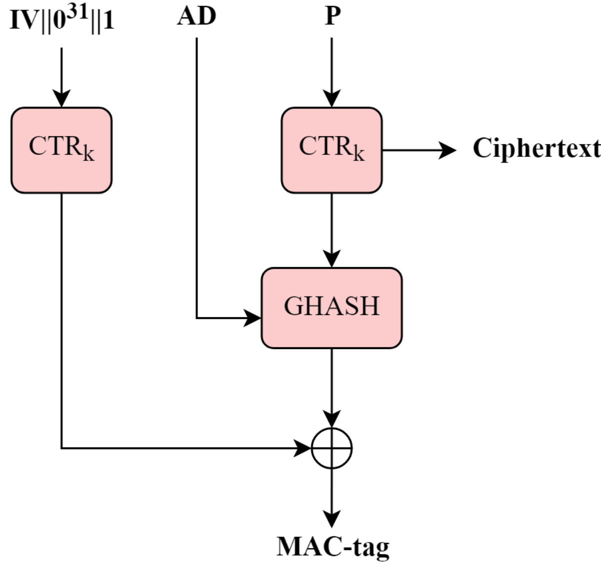


Figure 4.8: The GCM mode using CTR encryption and GHASH authentication.

Figure 4.9 shows a more detailed illustration of the authenticated encryption in GCM mode. The illustration shows two blocks of plaintext (P_1 and P_2) and one block of authenticated data (Auth Data$_1$). The upper half that results in two blocks of ciphertexts is the encryption part, while the lower half shows the generation of the authentication tag. If the IV is 96 bits long, which is the recommendation for AES-GCM, the $Counter_0$ is defined as $Counter_0 = IV \mid\mid 0^{31} \mid\mid 1$. In the figure, the *incr* operation increments the rightmost 32 bits with modulo 2^{32}. $mult_H$ means multiplication in the GF(2^{128}) field with a hash key H, where the polynomial used in GCM is: $x^{128}+x^7+x^2+x+1$. The $mult_H$ operations are part of the GHASH function, which generates an authentication tag by putting data blocks in the function.

The GHASH function takes the AD, C, and a hash key H as input. The hash key is generated by encrypting 128 bits of zeros: $H = Enc_k(0^{128})$. The GHASH function is defined as follows: $GHASH(H, AD, C) = X_{m+n+1}$, where m is the number of AD blocks, and n is the number of ciphertext blocks. Equation 4.1 shows the definition of X_i [MV04]. The $\cdot H$ operations in the equation are the same as the $mult_H$ operations in Figure 4.9. They both indicate multiplication with H in $GF(2^{128})$.

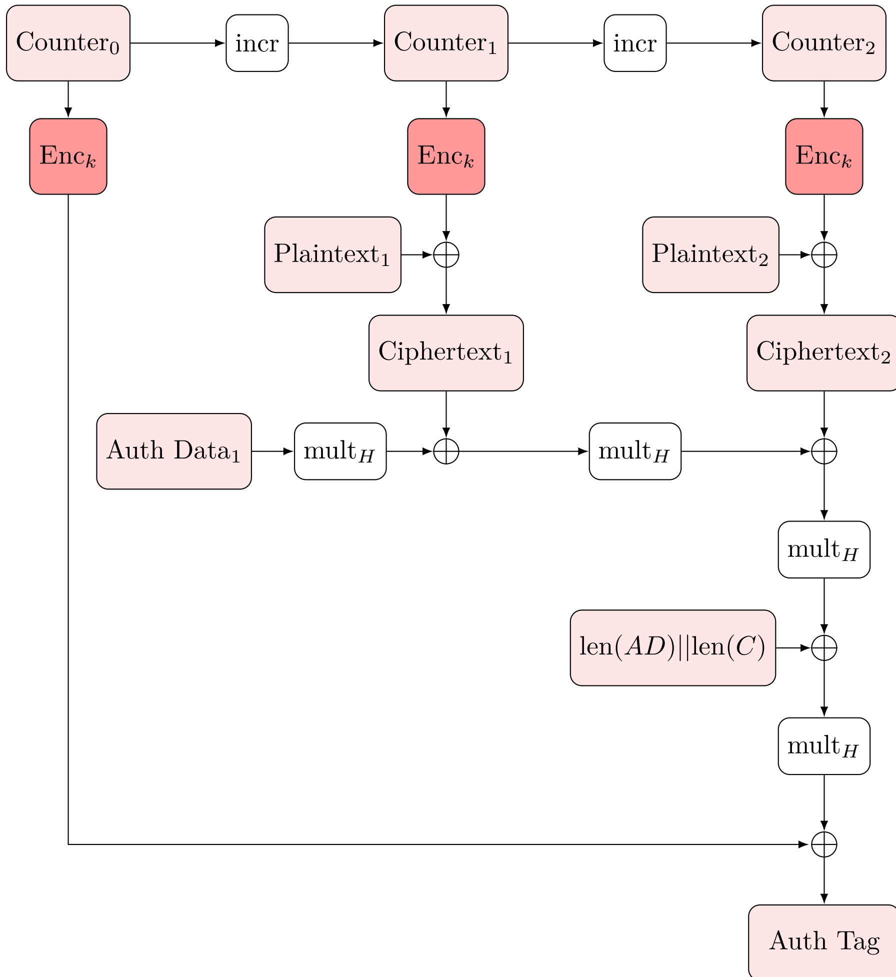


Figure 4.9: Illustration of GCM encryption.

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus AD_i) \cdot H & \text{for } i = 1, ..., m-1 \\ (X_{m-1} \oplus (AD_m || 0^{128-v})) \cdot H & \text{for } i = m \\ (X_{m-i} \oplus C_i) \cdot H & \text{for } i = m+1, ..., m+n-1 \\ (X_{m+n-1} \oplus (C_m || 0^{128-u})) \cdot H & \text{for } i = m+n \\ (X_{m+n} \oplus (\text{len}(AD) || \text{len}(C))) \cdot H & \text{for } i = m+n+1 \end{cases} \quad (4.1)$$

The u is the length of the final block of ciphertext, $u = len(C) \ mod \ 128$, and v is the length of the final block of AD, $v = len(AD) \ mod \ 128$. Lastly, after the GHASH operations, the output of the GHASH is XOR-ed with the first encrypted counter, $Enc_k(Counter_0)$, as seen in Figure 4.9. The result of the computation is the authentication tag, described as *Auth Tag* in the illustration. The authentication tag is defined by: $T = MSB_t(GHASH(H, AD, C) \oplus Enc_k(Counter_0)$, where MSB_t is the t most significant bits of the output. After performing the GHASH operations and XOR-ing the output with $Enc_k(Counter_0)$, the authentication tag is defined as the t leftmost bits of the result.

GMAC

A practical thing with GCM is the authentication-only variant, referred to as GMAC. This variant does not take confidentiality into account, only data authentication. The advantage of GMAC is that it does not call the block cipher for each block of data. The implementation of GMAC only authenticates the AD by generating and verifying the authentication tag [Mor07]. Figure 4.10 illustrates the GMAC operations. In the illustration, three AD blocks are considered. Note that the length of the ciphertext, in this case, will be zero. GMAC is, like GCM, considered a mode of operation for an underlying block cipher. The expression below shows the generation of the authentication tag T on message M in GMAC. Message M consists of C and AD, but in GMAC, C is an empty string, $C = \{\}$.

$$GMAC(K, IV, M) = GHASH(H, AD, C) \oplus Enc_K(Counter_0) \rightarrow T$$

One practical feature of GMAC is the possibility of using it as an incremental MAC. This feature means that when computing a message M, one can compute a new message M' similar to M with a computational power proportional to the hamming weight between M and M'. The hamming weight between M and M' can be seen as the number of non-zero bits when the two messages are XOR-ed. To find the reason for this, one must look at GHASH. The function GHASH(H,AD,C) is linear in the AD and C. Let us say that we have two associated data strings,

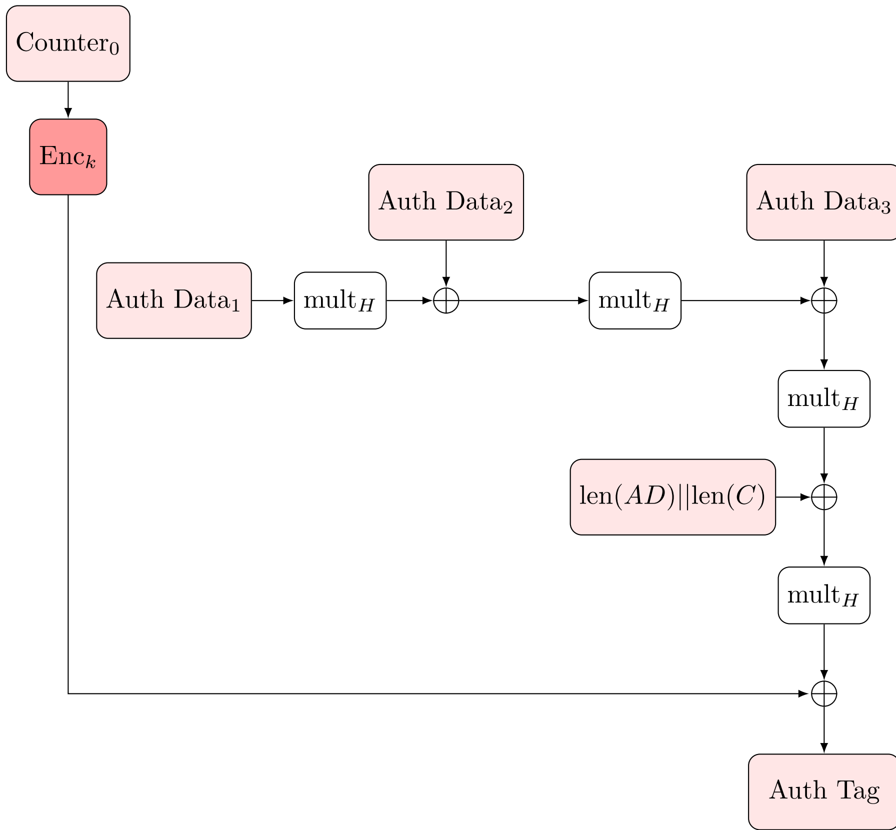


Figure 4.10: Illustration of GMAC authentication.

AD and AD', and two ciphertexts, C and C'. Given that len(AD)=len(AD') and len(C)=len(C'), we have [McG05]:

$$GHASH(H, AD, C) \oplus GHASH(H, AD', C') = X_{m+n+1}$$

$$= GHASH(H, AD \oplus AD', C \oplus C')$$

Using the latter property can reduce the amount of computation in some cases. Let us assume that we have calculated an authentication tag T based on message M: $T = GMAC(K, IV, M)$. Here, M includes both the ciphertext and the AD. Further, we want to calculate a new tag T' on message M', a message quite similar to M. Given that $\text{len}(M) = \text{len}(M')$, one can reduce the computation by only computing:

$$T' = T \oplus Enc_K(Counter_0) \oplus Enc_K(Counter'_0) \oplus GHASH(H, M \oplus M')$$

The latter example shows a case when messages are of fixed length. There exist similar advantages for prepending or appending data to messages as well. Incremental

GMAC makes an effective approach to authenticating large data sets because the incremental MACs make it easy to produce new authentication tags when performing minor changes to the data set. If, for instance, IoT sensors send data sets with small changes in measurements, this feature could be useful.

Splitting keys into integrity key and confidentiality key is problematic in GCM. In the GCM mode, the authentication and encryption are combined. If the cloud party receives a ciphertext and MAC tag T from a sensor, it must be able to verify the authentication tag without decrypting the ciphertext. It must recompute the authentication tag T' and check if $T = T'$ to verify the integrity. To generate T', the following computation must be performed: $T' = MSB_t(GHASH(H, AD, C) \oplus Enc_k(Counter_0)$. Here, we can see that the decryption key is needed to verify the MAC. Both $H = Enc_k(0^{128})$ and $Enc_k(Counter_0)$ can only be computed using the decryption key. Therefore, separating the confidentiality and integrity parts is not an intuitive solution in GCM. However, if one disregards the last part of the MAC computation where the MAC tag is XOR-ed with $Enc_k(Counter_0)$, the algorithm follows the EtM approach.

The MAC tag is generated and verified based on the ciphertext. Therefore, it would be interesting to see if one can solve the problem by modifying small parts of GCM. Introducing a MAC key, K_m, and making the GHASH function use this key instead of the encryption key could be an alternative. The following computation would generate the hash key H: $H = Enc_{k_m}(0^{128})$. However, since the result of the MAC computation is XOR-ed with $Enc_k(Counter_0)$ in the last operation, this is not enough. The reason is that the $Enc_k(Counter_0)$ operation is computed with the encryption key, and if this key is needed for the integrity check, the cloud can also use it to decrypt the data. One could remove the latter operation to fix this, but this would not be a preferred solution as it breaks the standard and could reduce the security strength. Another more reasonable approach would be to perform the encryption of $Counter_0$ with the MAC key K_m instead of the encryption key: $Enc_{k_m}(Counter_0)$. If this is the case, the MAC tag would be computed by the following computations: $T = MSB_t(GHASH(H, AD, C) \oplus Enc_{k_m}(Counter_0)$, where $H = Enc_{k_m}(0^{128})$. It seems interesting to investigate GCM with split keys as described above. However, as this approach breaks the standard, it is uncertain if it will affect the security strength of the algorithm. This would require security analysis. Because of other interesting solutions, the thesis will not take this further. However, it would be fascinating to investigate this approach in further work.

4.2.3 EAX

EAX uses CTR mode described in Section 2.4.1 for data encryption and an algorithm called OMAC^s for data authentication. OMAC^s is the same as CMAC, described in

Section 2.5, but with additional padding of two values in the last block. Also, the authentication adds a value s, which is encoded into an n-bit binary string as an extra argument to the MAC computation [Šve16] [BRW04]. The s-values used in the OMAC computations of the nonce N, AD, and plaintext P are respectively 0, 1, and 2. The proposal of the EAX mode worked as an alternative to CCM. The general idea was to fix the limitations identified in CCM and make it more flexible with the possibility of processing messages of arbitrary lengths and pre-process fixed headers [BRW03].

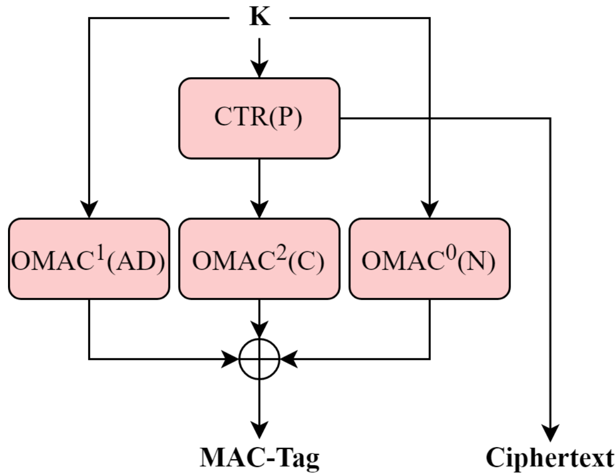


Figure 4.11: EAX mode authenticated encryption.

The encryption and decryption functions have the same inputs and outputs as the other AEAD algorithms. However, the operations of EAX differ from the others. Computing $C = \text{CTR}(K, \text{OMAC}_K^0(N), P)$ encrypts the ciphertext, and the authentication tag is generated by the following computation: $T = MSB_t(\text{OMAC}_K^0(N) \oplus \text{OMAC}_K^1(AD) \oplus \text{OMAC}_K^2(C))$. MSB_t is the t most significant bits of the output. Figure 4.11 illustrates the authenticated encryption process in EAX mode. EAX is based on a generic composition scheme [BRW04] and can be seen as an EtM approach. As the MAC is generated based on the ciphertext, it could be suitable for this thesis.

Splitting keys in EAX into confidentiality key and integrity key could be possible. If the cloud party only possesses the integrity key, it could verify the MAC tag with the following computation: $T' = MSB_t(\text{OMAC}_K^0(N) \oplus \text{OMAC}_K^1(AD) \oplus \text{OMAC}_K^2(C))$. The operations would only require C, N, and AD as input, in addition to the key. The end-user could verify the integrity with the same computation and use the

confidentiality key to decrypt the data by computing $P = \text{CTR}(K, \text{OMAC}_K^0(N), C)$. This will be discussed in Section 5.2.4.

4.2.4 ChaCha20-Poly1305

ChaCha20-Poly1305 is an AEAD algorithm consisting of the ChaCha20 stream cipher and the Poly1305 authenticator. TLS 1.3 introduced the algorithm as one of the recommended cipher suites to implement [Eri18], and it is a good alternative for resource-constrained devices. The ChaCha20 stream cipher was introduced in Section 2.4.1. The algorithm uses the ChaCha20 block function to perform quarter rounds on a 4×4 matrix of 32-bit values. The output of the block function is a 512-bit string of pseudorandom values that can be concatenated into a keystream if the block function is called several times. ChaCha20 encrypts a plaintext by XOR-ing it bitwise with the generated keystream: $C_i = KS_i \oplus P_i$. Section 2.5.3 described Poly1305 which generates a 128-bit MAC tag based on the AD, plaintext, padding, and length values [Pro14]. Poly1305 uses a one-time key K_{otk} that must be generated pseudorandomly and divided into two parts, K_r and K_s. The AEAD algorithm is an EtM approach as the ciphertext generation occurs first, and the computation of the authentication tag is based on the ciphertext.

The pseudorandomly one-time keys in Poly1305 can be computed in several ways, for instance, by encrypting a nonce using AES. One convenient property when using ChaCha20 and Poly1305 together is using the above-mentioned *ChaCha20 block function* to generate the one-time key, $K_{otk} = K_r||K_s$, that Poly1305 needs [NL18]. To generate one-time keys for Poly1305, one needs the same parameters used to generate a keystream in Chacha20: a 256-bit key K, a block counter ctr, and a nonce N. When generating the one-time keys, the ctr is set to zero. The *ChaCha20 block function* performs quarter rounds to generate random bytes in the same way as described before. When the function outputs 512 bits of random values, the first 128 bits will be the K_r value, and subsequently, the next 128 bits will be the K_s value. The last 256 bits are not used. The expression below illustrates how the one-time key, $K_{otk} = K_r||K_s$, is generated.

$$\textit{ChaCha20 block function}(K, N, ctr) \rightarrow K_r \;||\; K_s \;||\; \text{256 remaining random bits}$$

After using the ChaCha20 block function to generate the pseudorandom one-time key consisting of K_r and K_s, one can use the one-time key to generate the authentication tag in Poly1305. The AEAD algorithm ChaCha20-Poly1305 consists of three steps that are performed in the following order:

1. Generate the one-time key, $K_{otk} = K_r \;||\; K_s$, for Poly1305 using the Chacha20 block function.

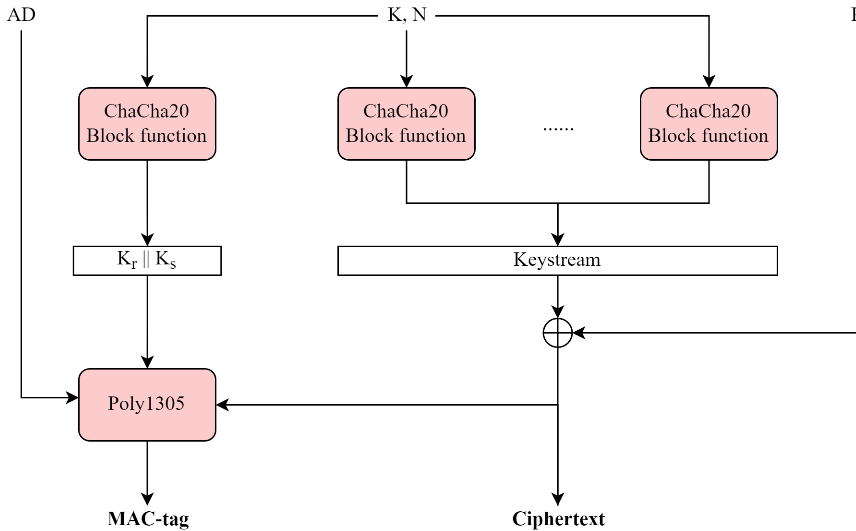


Figure 4.12: The ChaCha20-Poly1305 algorithm [ChaPol].

2. Generate a keystream and encrypt the plaintext by using the Chacha20 encryption function.

3. Generate an authentication tag with Poly1305 using the one-time key K_{otk} generated in step **1**.

Figure 4.12 shows an illustration of the ChaCha20-Poly1305 algorithm. The authenticated encryption key K in ChaCha20-Poly1305 is not a one-time key, and it is used throughout a session. Only the one-time key in Poly1305, $K_{otk} = K_r \mid\mid K_s$, must be unique for every MAC computation.

ChaCha20-Poly1305 is proven secure, according to a security analysis performed by Gordon Procter [Pro14]. The analysis presents a security reduction and demonstrates that the combination of ChaCha20 and Poly1305 results in a secure authenticated encryption scheme. The scheme is Indistinguishably under both Chosen Plaintext Attacks (IND-CPA) and Chosen Ciphertext Attacks (IND-CCA), making it hard for attackers to distinguish pairs of ciphertexts and plaintexts. The security analysis assumes that the nonce is unique every time the same key is used. If the same nonce and the same key are used on two different plaintexts, it can lead to attackers decrypting plaintexts, as described in Section 2.4.1.

Splitting the keys in ChaCha20-Poly1305 seems possible as one can combine and split the integrity and confidentiality operations. The algorithm uses the EtM

approach, and it looks suitable for fulfilling the confidentiality and integrity requirements described in this thesis. Because of this, splitting keys in ChaCha20-Poly1305 will be discussed further in this thesis.

4.3 Implementations on IoT sensors

All AEAD algorithms discussed can be implemented on IoT sensors. The block cipher modes of operation CCM, EAX, and GCM are usually combined with AES. The algorithms support implementations in both software and hardware. The implementations are quite similar, but because of the finite field multiplication in GCM, the mode of operation requires more code for software implementation and gates for hardware implementation than, for instance, CCM. Therefore, constrained devices often use AES with CCM encryption if reducing the implementation cost is the priority. However, GCM is usually preferred if there is a need for high-speed data transfers. The $GF(2^{128})$ multiplication can be optimized in several ways depending on the implementation. Implementing finite field multiplication either in hardware or software with table-driven operations generally results in a good performance.

Several techniques to increase the efficiency of the block cipher modes exist in general processors used in laptops and desktop computers, such as x86 processors. A carry-less multiplication instruction set extends the instruction set used in microprocessors from semiconductor companies such as AMD and Intel [SKP20]. The PCLMULQDQ instruction can speed up the GHASH part in GCM as the multiplication of large finite fields is performed more efficiently. In 2010 Intel published a white paper about how Intel's proposed AES New Instructions (AES-NI) and the PCLMULQDQ instruction would optimize the AES-GCM algorithm for implementation and efficiency [GK10]. There are many different ways to optimize the GCM mode of operation. Processors like x86 are not a common architecture in low-power IoT devices. Low-power microcontrollers often do not support these instructions that optimize the algorithms for constrained devices. The implementation depends on the microprocessors, but special hardware and computed lookup tables are ways to accelerate the process [SKK10]. Generally, implementing the AEAD scheme in hardware or co-constructing hardware and software will result in higher throughput and lower energy consumption [NBT20].

ChaCha20-Poly1305 differs as it is designed for high-performance software implementations. The software implementation requires few resources and consists mainly of inexpensive operations, making it suitable for IoT sensors. Comparing ChaCha20-Poly1305 with an AES-GCM implementation without hardware acceleration shows that the ChaCha20-Poly1305 is usually faster [NL18]. Studies [SLdP+19a] [SSS17] have shown that implementations of ChaCha20-Poly1305 are a good alternative for resource-constrained devices.

As the section describes, one can implement all four AEAD algorithms on constrained devices, but they require different resources and can be implemented differently. The best choice of algorithm depends much on the IoT sensor's capabilities.

4.4 Comparison of the AEAD schemes

AEAD schemes are the best approach to achieve integrity and confidentiality for IoT sensor data in the cloud-IoT architecture. The question is what AEAD scheme to take into account. This section will compare and reflect on the mentioned AEAD schemes and discuss their features. CCM, EAX, and GCM are all AEAD algorithms, but NIST has only approved CCM and GCM. Even though the proposal of EAX was said to improve CCM, GCM outcompeted EAX as it introduced more advantages. CCM was already widely established, and it was a need for a high-speed algorithm in terms of performance. Since GCM is the only block cipher mode that supports parallelization, it was chosen and standardized by organizations such as NIST. However, EAX might be a suitable approach in this thesis when considering splitting keys as a practical mechanism. Since the three different block cipher modes might look very similar at first glance, the following section will compare them to overview similarities and differences. The comparison is based on the NIST documentation and some other papers that compare the schemes [Dwo04] [Mor07] [SKK10] [Šve16] [BRW04].

There are many similarities when looking at CCM, EAX, and GCM. All are provable secure modes of operations defined for block ciphers. CCM and GCM have a block size of 128 bits, while EAX is defined for block sizes of arbitrary length. However, EAX is normally combined with AES, using a block size of 128 bits. The block cipher modes are all provably secure and ensure integrity and confidentiality with one key. Another similarity is that they are two-pass schemes. Each block performs two passes, one for confidentiality, and the other for integrity, making the modes referred to as two-pass schemes [JdOB+11]. Unlike other modes such as ECB and CBC, there is no error propagation in CCM, EAX, and CCM. Error propagation means that an error in one bit can lead to consequential errors in other bits. Error propagation in the three modes of operation is impossible because the AEAD schemes abort decryption if bits are changed. The integrity check takes place before returning the decrypted ciphertext.

The ChaCha20-Poly1305 scheme is both similar and dissimilar to the block cipher modes. The biggest difference is that the algorithm is a stream cipher, while the others are block cipher modes of operation. When it comes to similarities, ChaCha20-Poly1305 is, like the others, provable secure [Pro14], defined as one key algorithm, and avoids error propagation. Also, it is an online scheme, like EAX and GCM. The two latter features come from the algorithm being a stream cipher. ChaCha20-Poly1305 is a stream cipher that can encrypt the plaintext and AD of arbitrary length by

XOR-ing the plaintext with the keystream. It does not have to divide the data into blocks. Also, a one-bit error only affects one bit. The similarities of the AEAD algorithms are listed below.

Common features between the AEAD algorithms:

- Authenticated Encryption with Associated Data algorithms.
- Provable secure.
- Requires one key.
- No error propagation.
- Two-pass scheme.

Despite many similarities, there are also many differences when studying CCM, EAX, and GCM. The two latter modes perform the authentication based on the ciphertext, but GCM additionally encrypts the authentication tag. CCM uses the opposite approach as it computes the MAC firstly and then encrypts the MAC and plaintext. Additionally, the modes require different operations. For the authenticated encryption, CCM requires two block cipher operations per block of plaintext and one encryption operation for each block of AD. EAX requires the same, but also a block cipher operation for the nonce. In GCM, a block of authenticated encryption requires one encryption operation and one GF(2^{128}), while authenticated AD only requires the GF(2^{128}) operation. Lastly, the GCM mode needs one Galois field multiplication and one encryption operation in the final stage [SKK10].

GCM has the most features, EAX has the second most, while CCM has the fewest. This paragraph will discuss the differences between GCM and CCM as they differ the most. GCM is parallelizable, online, and has an incremental MAC. Those properties do not CCM have. Parallelizable means that it can simultaneously process things in parallel, such as performing several block cipher operations. The online feature regards the ability to encrypt and authenticate data of arbitrary length. The feature is not possible in CCM, as the scheme must know the length of the plaintext and AD in advance. Also, the GCM mode is generally faster than the CCM mode, which could be an important aspect of real-time applications. In addition to the parallelization, GCM is faster because 1 AES operation and 1 GHASH operation in GCM are faster than 2 AES operations in CCM. GHASH is generally faster than AES. However, this is only if the GHASH implementation is done on hardware or software with table-driven operations. The latter approach requires precomutation with the key and memory storage of the lookup table. Lastly, the GCM mode is more flexible as it can achieve AE, AEAD, or ensure authenticated AD (referred to as GMAC). However, it is possible to use CCM mode with an empty payload, as

Table 4.1: Summary of the comparison of CCM, EAX, GCM, and ChaCha20-Poly1305 (Cha-Pol).

Feature	**CCM**	**EAX**	**GCM**	**Cha-Pol**
AEAD algorithm	Yes	Yes	Yes	Yes
Block size	128-bit	n-bit	128-bit	N/A
Provable secure	Yes	Yes	Yes	Yes
One key algorithm	Yes	Yes	Yes	Yes
Avoids error propagation	Yes	Yes	Yes	Yes
Parallelizable	No	No	Yes	Yes
Online scheme	No	Yes	Yes	Yes
Pre-process static AD	No	Yes	Yes	No
Accept IV of arbitrary length	No	Yes	Yes	No
Incremental MAC	No	No	Yes	No
Specified in TLS 1.3	Yes	No	Yes	Yes

NIST specifies in SP 800-38C [Dwo04]. The list below shows the properties of GCM that are not supported in CCM.

Features supported in GCM that is not supported in CCM:

- Parallelizable.
- Online (can process plaintext and AD of arbitrary length).
- Incremental MAC.
- Authentication-only variant (GMAC)[1].
- Can accept IVs of arbitrary lengths.

EAX is closer to GCM as it has the same features except for being parallelizable and having incremental MAC. One useful attribute of GCM and EAX is that they can pre-process static AD. This can be useful for IoT sensors if, for instance, the sensor includes fixed headers or metadata. Then, the authentication of this header can be performed only once, which reduces the number of authentication operations. Table 4.1 summarizes the comparison of CCM, EAX, GCM, and ChaCha20-Poly1305 properties.

It seems like the only AEAD algorithms that allow for verifying the integrity without being able to decrypt the ciphertext are the EAX mode and ChaCha20-

[1] CCM mode can work as an authentication mode if it is used on messages where the payload is empty [Dwo04].

Poly1305. In CCM and GCM, the algorithms can not verify the MAC without using the decryption key. This differs in EAX and ChaCha20-Poly1305, as the MAC is generated based on the ciphertext. Because of this, the two algorithms could be suitable for the approach involving splitting a key into confidentiality and integrity parts, where the cloud party could verify the integrity using the integrity key.

This chapter has studied authenticated encryption. We have looked at CBC with HMAC, CCM, EAX, GCM, and ChaCha20-Poly1305. The algorithms use different operations, but all should be possible to implement on IoT sensors with the proper implementations. This chapter has shown that the best approach to authenticated encryption is AEAD algorithms combining integrity and confidentiality. As discussed in Section 4.1.1, protocols such as TLS are currently moving away from non-AEAD algorithms such as the CBC with HMAC approach. AEAD algorithms are, therefore, the main focus of this thesis.

The choice of algorithm that could solve the problem stated in this thesis depends on its design and flexibility. As this thesis will focus on splitting a key into an integrity key and a confidentiality key, choosing an algorithm where the MAC verification occurs before the decryption seems reasonable. Put another way; we want an AEAD algorithm that follows the EtM approach. As this chapter has discussed, only EAX and ChaCha20-Poly1305 seem to use this technique. Therefore, it seems reasonable to focus on these algorithms. The ChaCha20-Poly1305 is the most established today. Protocols such as TLS, Internet Protocol Security (IPsec), and Secure Shell Protocol (SSH) use the AEAD algorithm. EAX, on the other hand, is not that widely used today. When released, it was out-competed by GCM. Due to the arguments presented, it seems reasonable to focus on the ChaCha20-Poly1305 algorithm and research whether it is possible to split the key, fulfilling the confidentiality and integrity requirements stated in this thesis.

Chapter 5 Design

Up to this point, the thesis has researched AEAD algorithms and key management. After examining algorithms, constraints, and requirements, it is time to continue the work from Chapter 4 and look at how confidentiality and integrity should be handled and integrated between the parties involved in the cloud-IoT architecture. This chapter aims to find a solution that fulfills integrity for a third party in the cloud and achieves integrity and confidentiality for end-users. The chapter will discuss cryptographic keys in AEAD algorithms and propose different solutions, where ChaCha20-Poly1305 with split keys is the main focus. Lastly, the chapter compares and discusses the different approaches to find the best solution for the cloud-IoT architecture.

5.1 Achieving confidentiality and integrity

The communication of sensor data between sensors and end-users should be end-to-end encrypted and authenticated. End-users are the only party that should be able to decrypt and verify sensor data. When sensors perform measurements, the sensor data is pushed to the cloud for storage and processing. The cloud should handle data while guaranteeing that payload content is legitimate and not disclosed; therefore, there is a necessity for an additional integrity check. As Figure 5.1 illustrates, the goal is to ensure integrity between sensors and the cloud while achieving confidentiality and integrity for the end-user.

As discussed in Chapter 3, there is a need to define cryptographic keys for the cloud-IoT architecture. The cryptographic keys must be specified to meet the confidentiality and integrity requirements stated in Section 1.2. Specifying keys can be done in several ways. One can use two independent keys for different operations; one used in the cloud and one at the end-user. Otherwise, one can split a key into an integrity part, and a confidentiality part where the former would fulfill integrity and the latter ensure confidentiality. The next section will discuss the two alternatives.

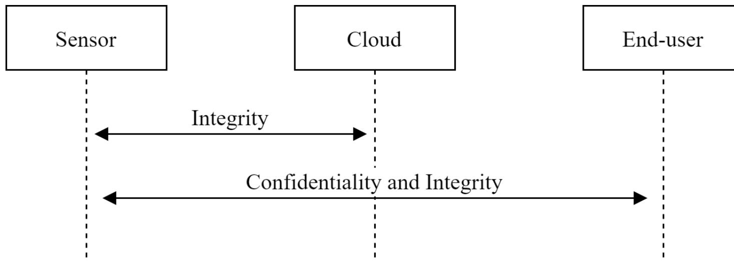


Figure 5.1: Confidentiality and integrity in the Cloud-IoT architecture.

5.1.1 Using two independent keys for different operations

One can use two separate keys, one key for the cloud and one for the end-user, where the keys are specified for different computations. The first key would ensure authenticated encryption for the end-user, while the second key would let the cloud verify the received data to confirm that it is trustworthy.

Double authenticated encryption

If two independent keys are the desired approach, the data could be encrypted twice. Double encryption would allow the cloud to decrypt the first layer of ciphertext and verify the integrity when sensors push the data to the cloud. Further, the end-user could decrypt the second layer, which discloses the sensitive sensor data. Section 5.2.1 will discuss a solution involving double AEAD.

Authenticated encryption with a separate MAC

The other approach is to use a separate MAC in addition to the authenticated encryption, where sensors could perform a distinct MAC generation based on the AEAD of the sensor data. This technique is more feasible as it requires fewer computational operations than encrypting twice. The approach would allow the cloud to verify the messages with its separate MAC key, while the end-user would use its key to perform authenticated decryption with its AEAD key. Section 5.2.2 will go into detail about such a solution.

Using two separate independent keys for the two above techniques is not the best approach when considering scalability, cost, and performance. It introduces additional operations, and IoT sensors must perform two key exchanges for every session, one for each key. As mentioned in Chapter 3, key exchanges are expensive for IoT sensors, especially when using public-key cryptography. Therefore, it is not an optimal solution for sensors.

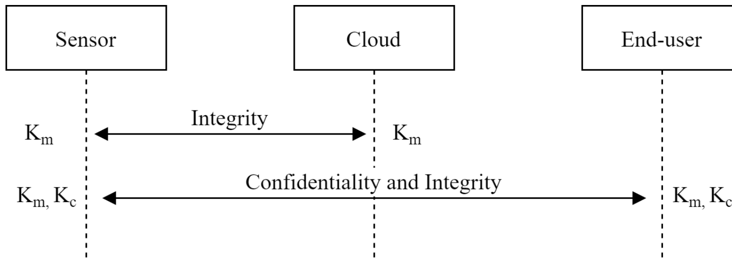


Figure 5.2: Key distribution when splitting keys into confidentiality and integrity parts.

5.1.2 Splitting a key into integrity key and confidentiality key

The other possibility is using one key that can be split into two parts, one for integrity and one for confidentiality: $K \rightarrow (K_m, K_c)$. In this case, the K_m is the MAC key, and the K_c is the confidentiality key. Figure 5.2 illustrates the distribution of keys. K_m can be given to the cloud while the end-user possesses both keys. This way, the cloud would verify the integrity, while the end-user would be able to both confirm the integrity and decrypt the secret data. Sections 5.2.3 and 5.2.4 will discuss such solutions.

A challenge is figuring out how one can split the keys. One alternative is that the sensor split the key and the parties involved agree upon the necessary key parts with the sensor. This method would involve two key exchanges for sensors per session, which is expensive. A better alternative is that the sensor and the entity responsible for key management exchange a key value, $K = K_m||K_c$, where the first part is K_m, and the last part is K_c. The key management entity can send the different parts to the parties that should obtain them. Alternatively, the entities must agree upon a common "master key" and derive the integrity and confidentiality keys from the master key. The latter approach would involve a KDF. Such functions are based on PRFs and can derive keys or stretch them into longer keys. The NIST recommendation SP 800-108 [Che+08] approves using known algorithms such as HMAC, CMAC, and CTR mode as the PRF in the KDF. These are considered lightweight and much cheaper in terms of operations than asymmetric key exchange algorithms. Additionally, if sensors already use authenticated encryption that involves such algorithms, there is no additional cost to implement a PRF.

The cost of splitting keys depends on the approach, key sizes, and what parties handle the key management and the trust relations, as discussed in Section 3.1.1. If the end-user trust the cloud party and the real threat is external attackers, the cloud entity should handle the key split and key management for end-users. Otherwise, if end-users do not trust the cloud entity, the end-user must involve a trusted third

party or handle the key management itself. Regardless of who handles the key management, if the sensor and the entity responsible for key management agree upon one key and split the key into confidentiality and integrity parts, there is only a need for one key exchange for the sensors. The entity can further derive and distribute the necessary key parts to the other entities. Although the sensors must perform costly operations, it is less expensive as it requires one key exchange rather than two.

5.2 Suggestions for different solutions

As concluded in Chapter 4, one can use AEAD algorithms to ensure confidentiality and integrity in IoT devices. The thesis has discussed using two independent keys or splitting a key into integrity and confidentiality parts. This section will continue the work from Chapter 4 and explore and propose cryptographic design solutions for the cloud-IoT architecture to fulfill requirements from Section 1.2. The designs are quite different in implementation, resource usage, and key management. The design solutions this section will propose are suitable for different situations depending on trust relations, requirements of the constrained devices, and metadata requirements.

5.2.1 Double AEAD

Double AEAD involves performing authenticated encryption on the data twice. In such a case, there is a need for two separate keys. This section will refer to the two AEAD keys as K_1 and K_2. The IoT sensor would possess both keys, the end-user would obtain K_1, and the cloud would need K_2. The keys are defined below:

- K_1 : Encrypt data between sensor and end-user.
- K_2 : Encrypt data between sensor and cloud.

In the authenticated encryption, the sensor first encrypts data with K_1, and the output is encrypted once more with K_2. The following expression demonstrates double encryption:

$$Enc(K_1, P) = C_1 \rightarrow Enc(K_2, C_1) = C_2$$

The double-encrypted ciphertext, C_2, is sent to the cloud party from the sensor. As the cloud possesses the key K_2, it decrypts C_2 with the following computation:

$$Dec(K_2, C_2) = C_1 \text{ or } INVALID$$

The cloud verifies the integrity and decrypts the first layer of the ciphertext. Despite decrypting the first layer, there is no disclosure of sensitive data because of the double encryption. Further, the cloud sends the ciphertext C_1 to the end-user. This

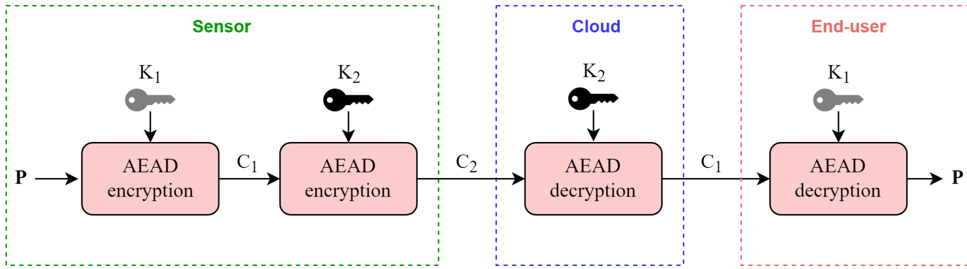


Figure 5.3: Architecture with double AEAD encryption.

party possesses K_1 and can decrypt C_1: $Dec(K_1, C_1) = P$ or $INVALID$. Figure 5.3 illustrates the process of end-to-end encryption with double encryption.

Sensors can perform double AEAD encryption with algorithms that allow for implementations on IoT sensors. Section 4.3 discussed the implementation of AEAD algorithms on IoT sensors. If the sensors have hardware accelerations and support AES, CCM or GCM are the best approaches. Deciding between the two depends on what requirements are important for the solution. If features such as parallel encryption and good performance are important, and the sensor implementation allows for efficient multiplication in the Galois field, GCM is the best approach. Otherwise, CCM could be the best solution as it is cheaper to implement. However, if hardware acceleration is not an alternative for the sensors, a software implementation such as ChaCha20-Poly1305 is a suitable approach. It all depends on the environment, equipment, requirements, and the situation.

The positive aspect of double encryption with AEAD algorithms is that it allows the cloud party to receive encrypted metadata from the sensor. Sensitive metadata such as sensor information, data type, timestamps, and status messages can be encrypted between the sensor and the cloud. Encrypting metadata is the best solution to avoid attackers sniffing such information. Otherwise, the metadata discloses information about the sensor, which might lead to side-channel attacks or breaking privacy policies. AEAD algorithms are secure, and if the sensor data is encrypted twice with AEAD, the sensor data is well protected. However, there are negative sides to this approach. The solution is not optimal for IoT sensors. It requires many cryptographic operations, consumes much battery, and requires key management of several keys. As the encryption takes place twice for each data packet, it drains the battery much faster. Therefore, this approach is only in favor if securing data transmissions is the main priority. If the IoT sensors have sufficient resources to perform double encryption, and the reduced lifetime of sensors is not that critical, it could be a suitable approach. Otherwise, one should consider other

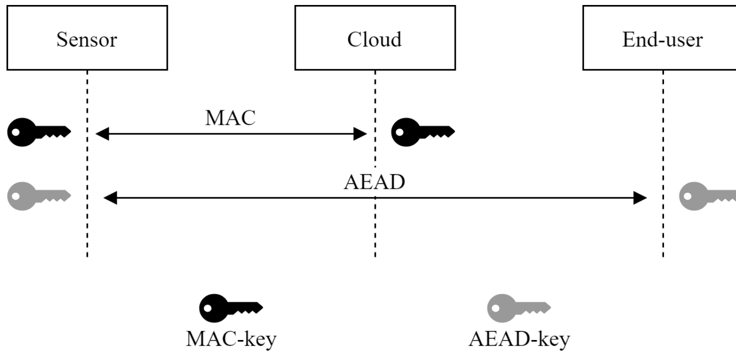


Figure 5.4: Keys and algorithms needed to protect the data in this architecture.

solutions.

5.2.2 AEAD with additional MAC

Another reasonable approach is to use an AEAD algorithm for authenticated encryption of the sensor data while introducing a MAC for the integrity check in the cloud. The sensor can use an AEAD algorithm to protect the sensor data between the sensor and the end-user. If the sensor has metadata or other information the cloud wants to obtain, it can be included in the authenticated AD. This approach does not keep the metadata confidential, but the integrity prevents attackers from tampering with the metadata. For the cloud party to verify the integrity, it can verify the MAC upon the AEAD encryption. In this way, one protects the sensor data when pushing it to the cloud for storage and processing. Figure 5.4 illustrates the parties that should possess the different keys. The keys are defined below:

- $\boldsymbol{K_C}$ **:** Key for combined confidentiality and integrity (AEAD).
  - Must be known by IoT sensor and end-user.
  - This key ensures authenticated encryption of the sensor data.
- $\boldsymbol{K_M}$ **:** MAC Key for integrity.
  - Must be known by IoT sensor and cloud service.
  - This key allows the cloud party to verify the integrity of the AEAD encrypted sensor data and ensure that it stores legitimate data.

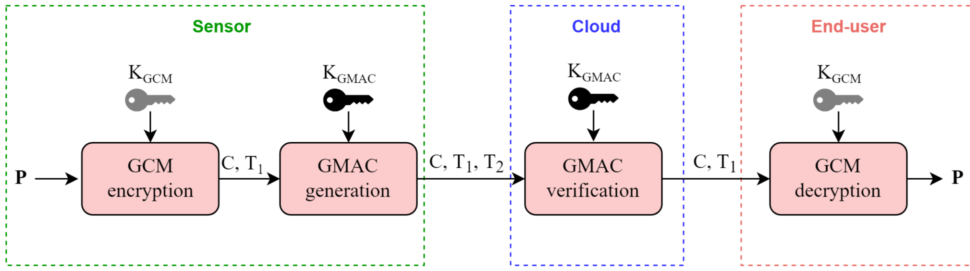


Figure 5.5: Authenticated encryption with GCM mode and GMAC authentication.

GCM mode and GMAC

GCM with additional GMAC is one approach of AEAD with additional MAC. GMAC is the authentication-only variant of GCM, introduced in Section 4.2.2. Using GMAC, it is possible to generate a MAC upon a message by only passing AD into the algorithm. If the sensor first encrypts sensor data with AEAD and passes the ciphertext and any metadata into the GMAC, it will generate an authentication tag on the GCM output.

This approach is possible with the two defined keys, K_C and K_M. The sensor first performs authenticated encryption using GCM mode: $GCM(K_C, IV, P, AD) \rightarrow C||T_1$. Further, the output of the authenticated encryption, including metadata MD, is the input of the GMAC computation. The input of the GMAC will be referred to as message M, where $M = C||T_1||MD$. The GMAC computation is: $GMAC(K_M, IV, M) \rightarrow T_2$. The sensor sends the outputs of GCM and GMAC to the cloud party: C, T_1, and T_2. The cloud can verify the authentication tag T_2 with the MAC key K_M. The validation occurs with the following computation: $GMAC(K_M, IV, M) = T_2'$. If $T_2 = T_2'$, the cloud can process the encrypted data and read the related metadata. The end-user receives C and T_1 from the cloud and can use K_C to perform authenticated decryption: $GCM(K_C, IV, C, AD, T_1) \rightarrow P \text{ or } FAIL$. The entity obtains the sensor data if the authentication tag is verified. Figure 5.5 illustrates the authenticated encryption with GCM and GMAC between the IoT sensor, cloud, and end-user.

CCM mode and CMAC

AEAD with additional MAC works the same way for CCM and CMAC. The two keys, K_C and K_M, would be used in the same manner described for GCM with GMAC. The only difference, in this case, is that one uses CCM for authenticated encryption and CMAC for the additional authentication tag. CMAC is the authentication-only variant of CCM as it uses CCM with an empty payload.

The choice between GCM and CCM depends on the implementation and application. One must look at the constrained devices and decide on the most suitable implementation. One certain thing is that if AES-GCM or AES-CCM is implemented, it costs nothing more in terms of implementation to introduce an additional MAC (GMAC or CMAC). The MAC schemes are already a part of the AEAD algorithms and perform the same operations. However, the additional MAC increases the cost of battery power, computations, and memory, as some additional operations take place per message. However, one must decide on this tradeoff, and if the cloud party need to verify the integrity, this could be a suitable approach.

Compared with double encryption, one can reduce the number of encryption operations needed per message with n operations when using AEAD with additional MAC instead of double AEAD encryption, where n is the number of blocks required. For instance, if one encrypts a plaintext in CCM that requires two blocks, double CCM encryption requires eight encryption operations, while CCM with CMAC requires six encryption operations. For GCM, double GCM encryption of two data blocks requires six GHASH and six encryption operations, while GCM with GMAC requires six GHASH and four encryption operations. This example did not include AD, and the number of operations can vary depending on the AD included and the length of messages and values. For instance, when the ciphertext from the first authenticated encryption is expanded with the generated MAC tag, it can increase the input length with an additional block in the second AEAD/MAC computation.

5.2.3 ChaCha20-Poly1305 with split keys

Section 4.2.4 discussed the ChaCha20-Poly1305 algorithm. The AEAD algorithm encrypts data using ChaCha20 and generates a MAC tag with Poly1305. This section will propose a modified version of ChaCha20-Poly1305 that meets the confidentiality and integrity requirements for the cloud-IoT architecture. Since the algorithm follows the EtM approach, it allows for verifying the MAC without decrypting the ciphertext. Therefore, the idea is to let the intermediate part, the cloud, verify the MAC without reading the sensitive data. When the end-user obtains encrypted data, it can verify the MAC and decrypt it. The described flow is summarized in Figure 5.6, which illustrates a flowchart of the desired authenticated encryption using the ChaCha20-Poly1305 algorithm.

Split a key into integrity and confidentiality keys

In order to realize the desired flow described in Figure 5.6, it must be possible for the cloud party to verify the authentication tag but impossible to decrypt the data. Having one integrity key and one confidentiality key is reasonable for ChaCha20-Poly1305. This section defines a MAC key, K_m, and a confidentiality key, K_c. As

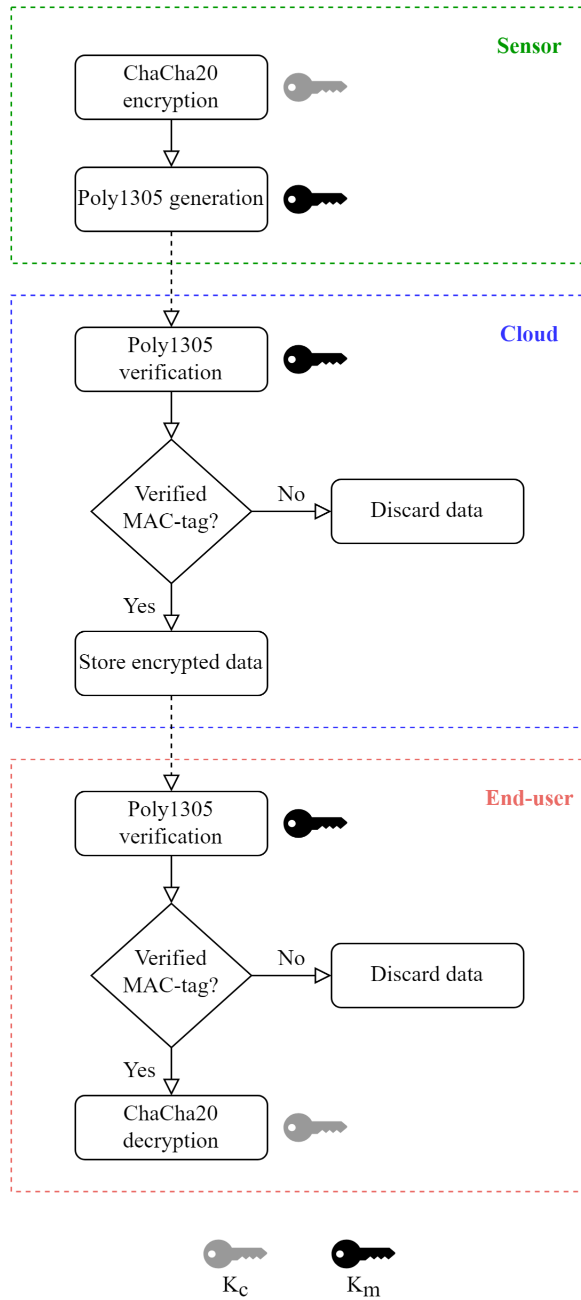


Figure 5.6: Flowchart of the authenticated encryption in the cloud-IoT architecture with ChaCha20-Poly1305.

Table 5.1: The inputs of the modified ChaCha20-Poly1305 algorithm

Input values	Length
Integrity key K_m	256-bit
Confidentiality key K_c	256-bit
Nonce	96-bit
Plaintext	Arbitrary length
AD	Arbitrary length

one can see in Figure 5.6, the illustration includes the key parts and their use. The two key parts are described below.

- $\boldsymbol{K_c}$ **:** Confidentiality key.
  - Must be known by IoT sensor and end-user.
  - The key must be able to encrypt the plaintext and decrypt ciphertext so that the sensor data is kept confidential for everyone except those who possess the key.
- $\boldsymbol{K_m}$ **:** MAC key.
  - Must be known by IoT sensor, cloud service, and end-user.
  - The key must generate and verify MAC tags so that everyone who possesses the key can check if the message is validated.

Since the key is divided into two parts, the algorithm needs to consider an extra input. Table 5.1 summarizes all the modified ChaCha20-Poly1305 algorithm inputs and their corresponding lengths. The inputs are K_m, K_c, nonce, plaintext, and AD. The combined key size of K_m and K_c is quite big and not optimal for all sensors. However, splitting keys would allow one key exchange between the sensor and the trusted key management entity. Also, it is possible to reduce the size. For instance, one can reduce the integrity key in size to 128-bits. The algorithm only uses the integrity key to generate the one-time key, $K_{otk} = K_r || K_s$, used in Poly1305, which can be done in other ways requiring smaller keys. The ChaCha20 block function requires a 256-bit key to output a 512-bit value where the first 256 bits are used as the one-time key. Instead of generating one-time keys with the ChaCha20 block function, one can generate keys with PRFs or other ways that require smaller keys as long as every Poly1305 computation receives a unique 32-byte one-time key. The K_m is the same during a session, and only the K_{otk} must be unique per MAC computation.

Authenticated encryption with K_m and K_c

If one split the key in ChaCha20-Poly1305 into two keys, K_m and K_c, the authenticated encryption would look like this:

$$\textit{ChaCha20-Poly1305}(K_m, K_c, N, P, AD) \rightarrow C, T$$

The algorithm performs encryption by generating a keystream using the ChaCha20 block function with the confidentiality key and the nonce. The keystream is XOR-ed with the plaintext and outputs a ciphertext. The expression below illustrates the encryption:

$$\textit{ChaCha20-block}(K_c, N) \rightarrow KS$$

$$KS \oplus P \rightarrow C$$

Further, the algorithm must generate an authentication tag using the MAC key K_m. The MAC generation consists of two steps. First, K_m is used to generate a one-time key. The one-time key is divided into two parts, K_r and K_s. Further, the one-time key is used with the ciphertext and AD to generate the MAC tag. The Poly1305 authenticator divides messages into blocks of 16 bytes. Therefore there may be a need to add padding to the last blocks of the ciphertext and AD. The operations of the MAC generation are illustrated below:

$$\textit{Poly1305-KeyGen}(K_m, N) \rightarrow \textit{ChaCha20-block}(K_m, N) \rightarrow K_r, K_s$$

$$\textit{Poly1305}(K_r, K_s, C, AD) \rightarrow$$

$$\textit{Poly1305}\Big(K_r, K_s, C, pad_C, AD, pad_{AD}, len(C), len(AD)\Big) \rightarrow T$$

Figure 5.7 illustrates how the authenticated encryption in ChaCha20-Poly1305 would look if one split the key K into a MAC key K_m and confidentiality key K_c. Although the nonce (N) is mentioned twice in the illustration, it is the same value used in the MAC generation and encryption. However, the nonce must be unique for every invocation with the same key.

Verifying the MAC with K_m

The intermediate party should be able to verify the authentication tag using the integrity key K_m. The cloud party already possesses the key K_m, assuming it has been agreed upon before the session. When the IoT sensor sends data to the cloud party, the cloud party receives N, T, AD, and C. In order to compute the new tag (T'), the cloud party performs the following operations:

$$\textit{ChaCha20-block}(K_m, N) \rightarrow K_r, K_s$$

$$\textit{Poly1305}(K_r, K_s, C, pad_C, AD, pad_{AD}, len(C), len(AD)) \rightarrow T'$$

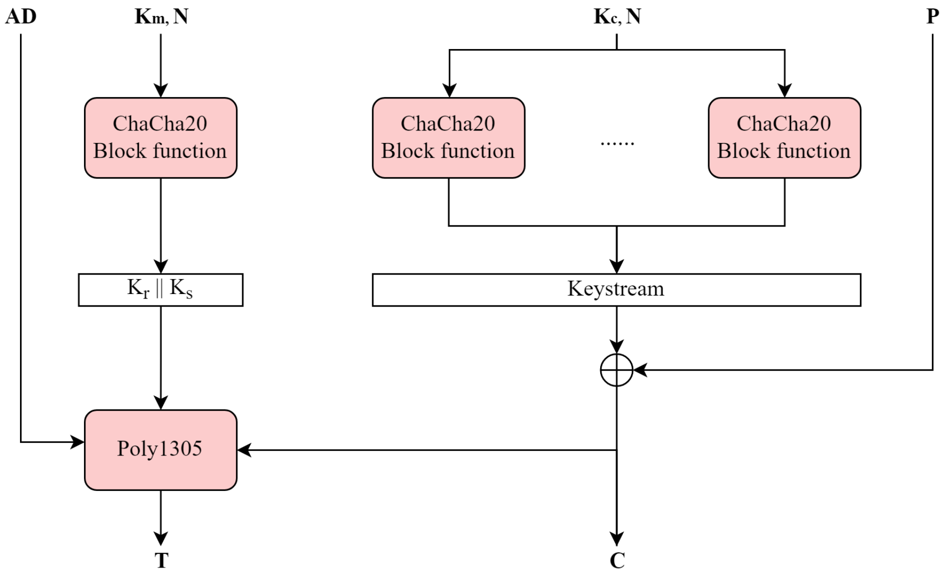


Figure 5.7: Authenticated encryption in the modified ChaCha20-Poly1305 algorithm using two keys, K_m and K_c.

The cloud compares the output of the MAC computation with the tag received. If $T = T'$, the entity knows that the message received is the same as the message sent by the sender. Figure 5.8 shows how the middle party can recompute the MAC tag using only the MAC key K_m.

Authenticated decryption with K_m and K_c

As the sensor data have been authenticated and encrypted with K_m and K_c, it should be possible to verify and decrypt the data with the same keys. The end-user possesses both K_m and K_c. The authenticated decryption would look like this:

$$\textit{ChaCha20-Poly1305}(K_m, K_c, N, AD, C, T) \rightarrow P$$

The end-user can verify the integrity by performing the same computation as the cloud party. Figure 5.8 illustrates integrity verification. If the tag is validated, the decryption occurs with the following computation:

$$\textit{ChaCha20-block}(K_c, N, C) \rightarrow KS$$

$$KS \oplus C \rightarrow P$$

Figure 5.9 illustrates the decryption at the end-user in ChaCha20-Poly1305 with the keys K_m and K_c.

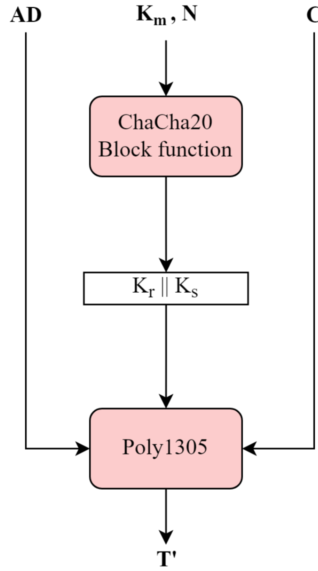


Figure 5.8: MAC verification in the modified ChaCha20-Poly1305.

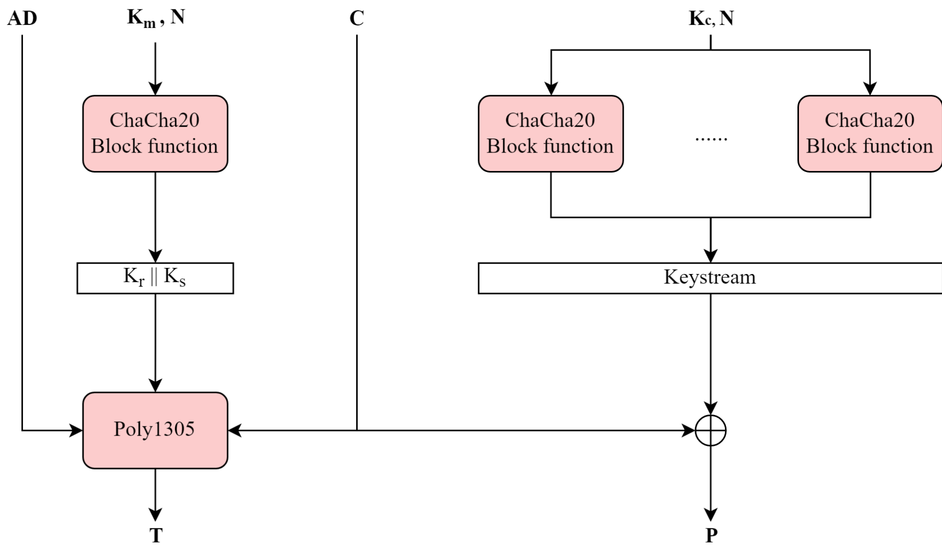


Figure 5.9: The modified ChaCha20-Poly1305 decryption algorithm using two keys.

5.2.4 EAX with split keys

Although EAX is not well established compared to the other AEAD algorithms, it can be used in the cloud-IoT architecture with split keys. Let us assume that a key is split into K_c and K_m, with the same definition as ChaCha20-Poly1305 with split keys discussed in Section 5.2.3.

Authenticated encryption with K_m and K_c

The sensor would have to generate the ciphertext and MAC using the two key parts. The ciphertext is computed by calculating $C = \mathrm{CTR}\Big(K_c, \mathrm{OMAC}^0_{K_m}(N), P\Big)$ and the authentication tag generated by:

$$T = MSB_t\Big(\mathrm{OMAC}^0_{K_m}(N) \oplus \mathrm{OMAC}^1_{K_m}(AD) \oplus \mathrm{OMAC}^2_{K_m}(C)\Big)$$

Verifying the MAC with K_m

The cloud party would verify the integrity using only the MAC key K_m as only the three OMAC^s computations are necessary to re-generate the MAC:

$$T = MSB_t\Big(\mathrm{OMAC}^0_{K_m}(N) \oplus \mathrm{OMAC}^1_{K_m}(AD) \oplus \mathrm{OMAC}^2_{K_m}(C)\Big)$$

Authenticated decryption with K_m and K_c

The end-user would perform authenticated decryption by verifying the integrity with the same computation as the cloud party using K_m and perform decryption using K_c:

$$P = \mathrm{CTR}\Big(K_c, \mathrm{OMAC}^0_{K_m}(N), C\Big)$$

Figure 5.10 illustrates the authenticated decryption with two keys, K_m and K_c. The cloud can perform the operations to the left in the illustration that use K_m to verify the integrity. The advantage of EAX is its flexibility. The block size in the algorithm can be of arbitrary length, but this thesis considers the mode of operation based on the 128-bit block cipher AES. The scheme supports plaintexts, ADs, and nonces of arbitrary length. The length of the nonce and the tag are optional, depending on the desired security from the integrity guarantees [BRW04]. Table 5.2 summarizes the inputs of this EAX version with split keys and their corresponding lengths.

This approach would allow the sensor and the trusted key management party to exchange one key of 256-bits that can be split into two parts. Also, it allows for shorter nonces and MAC tags, as they can be specified in the desired length. Shorter input values are positive for constrained devices, but they must be of an appropriate

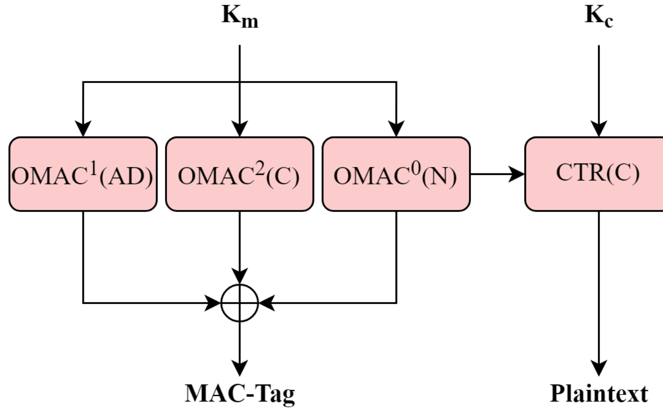


Figure 5.10: The modified EAX decryption algorithm using two keys.

Table 5.2: The inputs of the modified EAX algorithm

Input values	**Length**
Integrity key K_m	128-bit
Confidentiality key K_c	128-bit
Nonce	Arbitrary length
Plaintext	Arbitrary length
AD	Arbitrary length

security level. EAX with split keys is an appropriate solution that could solve this thesis's confidentiality and integrity requirements.

Today, the use of EAX is limited. There is another version of EAX, called EAX-prime, used in the smart grid standard ANSI C12.22, but it was broken in 2013 [MLMI13]. EAX is provable secure, but not NIST-approved, and many choose GCM over EAX because of its parallelization and performance. Although it is not widely established today, it could be a solution to the problems in this thesis.

5.3 Discussion of the solutions

There are a lot of different solutions for authenticated encryption in the cloud-IoT architecture. This chapter has discussed double AEAD encryption, AEAD with additional MAC, and splitting keys in ChaCha20-Poly1305 and EAX. Choosing the best solution depends on different factors; use cases, capabilities of the IoT sensors, security threats, priorities, performance, and cost are some of them. This section will compare the proposed approaches and discuss their advantages and disadvantages.

Deciding on the algorithms is greatly affected by the sensor's capabilities. Many different microcontrollers with different CPUs and resources support various calculations and algorithms. As mentioned in Chapter 4, the AES algorithms are fast on dedicated hardware, but if sensors lack such accelerations, the performance is considerably lower. For instance, instruction sets such as AES-NI and PCLMULQDQ are designed to optimize hardware implementations of AES. However, some constrained microcontrollers for IoT sensors do not have the possibility for such hardware acceleration. IoT sensors may consist of everything from 8-bit to newer ARM 32-bit microcontrollers. Common for all is that they have a low clock speed, little RAM, little flash, and constrained battery life. Therefore, the involved parties must consider the solutions carefully and discuss performance and security tradeoffs.

Memory requirements of the IoT sensors are one of many performance metrics that play a part in choosing a suitable solution. When looking at ROM, the memory required to store algorithms, static data, and keys, different solutions have different requirements. Since ChaCha20-Poly1305 requires a software implementation, sensors must store it in the ROM. The same goes for parts of AES in some cases; if there is a lack of hardware support, an S-box lookup table can be stored in memory. Additionally, sensors must store the cryptographic keys in the ROM. All discussed solutions require two key parts, but the key sizes vary. AES is defined for key sizes of 128, 192, and 256 bits, and one can choose two keys of 128-bit to minimize the memory consumption. Splitting keys in EAX is the best approach for memory consumption as the key size, nonce length, and MAC tag can be chosen based on the implementation because of the algorithm's flexibility. This allows shortening the values to the minimum security level if desired to reduce the cost of the constrained devices as much as possible.

The ChaCha20-Poly1305, on the other hand, is specified with a 256-bit key in RFC 8439 [NL18]. If two 256-bit key parts are needed, it is more costly than the AES approaches for ROM. Although the ChaCha20 cipher requires a 256-bit key, one could use the Poly1305 authenticator with a 128-bit key if one can derive the one-time key differently than deriving the key with the ChaCha20 block function. Although ChaCha-Poly requires more ROM than the AES approaches, another debate worth discussing is whether 128-bit keys are sufficient. AES with 128-bit keys is considered secure today [Bar20], but if Moore's law continues to apply, it is uncertain how many years 128-bits of security will be sufficient. As this paragraph states, the best solution depends on the algorithms, ROM and RAM capabilities, and desired security strength when it comes to memory metrics.

When it comes to attacks, one must discuss different security concerns. AES and ChaCha20-Poly1305 have various security advantages in preventing side-channel attacks [NJJ+18]. As mentioned in Section 4.2.4, the stream cipher ChaCha20 is

resistant to timing side-channel attacks. On the other hand, AES can be vulnerable to such attacks. When microcontrollers do not have hardware acceleration, AES often implements lookup tables obtaining values from array indexing instead of runtime computations. Implementations of lookup tables are sometimes vulnerable to cache timing attacks. When sensors use lookup tables, the CPU's memory cache leaks memory access patterns and information that discloses knowledge about the cryptographic primitives [OST06]. Specific implementations can mitigate the vulnerability on behalf of the performance. However, if hardware accelerators are in place, there is no need to implement lookup tables. Then, computations run in constant time, which mitigates the vulnerabilities. Hardware accelerators, however, introduce other vulnerabilities such as power side-channel attacks [OGOP04]. A power side-channel attack involves analyzing the electrical activity on cryptographic hardware devices. Although AES is the most exposed, both AES and ChaCha20 can be vulnerable to power side-channel attacks [NJJ+18]. To mitigate this, one must mask operations used in the algorithms. Implementing resistance to power side-channel attacks for both algorithms is possible, but protecting AES against power side-channel attacks requires less overhead than for ChaCha20.

Several studies have shown that ChaCha20-Poly1305 is faster than AES-128 in GCM or CCM mode on microprocessors such as ARM Cortex-M4, which does not have AES hardware acceleration [DSS17] [SGTW20] [TLSIV]. The performance of EAX mode is not stated as studies choose CCM or GCM over it, but the performance is close to CCM but often faster because of its performance attributes [BRW04]. However, $OMAC^s$ can not be parallelized, so it can not compete with GCM in performance. The ChaCha20 stream cipher is fast because it uses ARX operations: addition, rotation, and XOR. As such operations are very CPU-friendly, the stream cipher is faster than AES on software platforms. A study evaluating the performance of symmetric key algorithms on IoT devices showed how much faster ChaCha20-Poly1305 is than AES-GCM on devices without hardware acceleration [SLdP+19b]. The test was performed using smartphones with different ARM CPUs deployed in IoT devices. The two CPUs tested were ARMv7-a Cortex-A7 and ARMv8-a Cortex-A53. The first one was used without AES hardware acceleration, while the latter was tested both with and without. Table 5.3 illustrates some of the results of the study. The table shows the average throughput of the different algorithms and packet sizes. Packet sizes are given in mebibytes (MiB) and the throughput in mebibytes per second (MiB/s). For reference, 1 MiB = 1024^2 bytes = 1 048 576 bytes.

As the table illustrates, the ChaCha20-Poly1305 is the best approach if no hardware acceleration is supported. When looking at the results for ARMv7-a, one can see that the throughput of ChaCha20-Poly1305 is approximately three times faster than AES-128-GCM. On ARMv8-a, the ChaCha20-Poly1305 is about twice as quick as AES. However, when including the hardware acceleration, the AES is

Algorithm/	**ARMv7-a**			**ARMv8-a**		
Packet size	**1MiB**	**5MiB**	**10MiB**	**1MiB**	**5MiB**	**10MiB**
AES-128-GCM	12,528	12,935	12,989	77,539	78,058	77,586
AES-256-GCM	11,073	11,286	11,313	59,882	61,793	61,67
ChaCha20-Poly1305	36,805	38,777	38,951	134,081	137,656	138,336
AES-128-GCM-HW	X	X	X	325,789	414,087	426,964
AES-256-GCM-HW	X	X	X	299,368	399,009	391,087

Table 5.3: Average throughput (MiB/s) for AEAD encryption of different packet sizes [SLdP+19b].

significantly faster. In this case, the AES with optimized instructions is three times faster than ChaCha20-Poly1305. The study measured battery drain as well, and as expected, ChaCha20-Poly1305 drains less battery than AES, but AES with hardware optimization gives the best result.

Choosing the right design depends on the main priorities of the system. It depends on the devices, use cases, security threats, and preferences. If the sensors have hardware accelerations, it is natural to look at AES-CCM, AES-EAX, or AES-GCM. If the cloud needs to receive sensitive metadata that should be confidential, double encryption is appropriate. However, although hardware acceleration is in place, double encryption will affect factors such as battery life and performance in a negative manner. Otherwise, GCM mode with GMAC or CCM mode with CMAC are good alternatives. If implementation cost and battery life are important, CCM mode with CMAC are the cheapest of the two solutions. Although EAX is not widely established or NIST-approved, it could be implemented using the approach with splitting keys. It is a flexible algorithm that can shorten input values, making it more suitable for constrained devices.

If a sensor does not have hardware support for AES, the ChaCha20-Poly1305 is faster than the block cipher algorithms. It is resistant to timing attacks, gives sufficient security strength, and solves all confidentiality and integrity requirements if split keys are possible. Overall, ChaCha20-Poly1305 is a decent solution for all scenarios. Although AES with hardware accelerations performs better than others, the difference is not that big if one compares ChaCha20-Poly1305 with double AES encryption or AES-GCM mode with additional GMAC. ChaCha20-Poly1305 with split keys would only need to run once per data packet and only need one key exchange per session. The algorithm can be implemented and performs well on most devices. Implementations on constrained devices such as ATmega328 AVR microcontrollers, ESP-WROOM-32, ESP8266, ARM Cortex-M4, ARMv7-a, and ARMv8-a, have been proven to work [DSS17] [SGTW20] [SLdP+19b]. The first two are based on an 8-bit

microcontroller, while the others are on 32-bit microcontrollers. Although the 32-bit outperformed the resource-limited 8-bit ones, a ChaCha20-Poly1305 implementation is suitable for all the devices.

Table 5.4 summarizes some important factors when comparing the solutions. The table includes key sizes, block-cipher calls, and ciphertext expansions. In double AEAD and AEAD with an additional MAC, both GCM and CCM are considered. EAX is specified with 128 bits, as AES defines this block size. The table contemplates that the AD is added in the first encryption process when looking at the block-cipher calls in double AEAD. Otherwise, adding AD in the second encryption process reduces the number of block-cipher authentication calls. One should also be aware that the algorithms specify different MAC and nonce lengths. EAX and GCM specify MAC tags between 0 and 128 bits, ChaCha20-Poly1305 considers tags of 128-bits, and CCM a value from 32 to 128 bits. According to standards, GCM and ChaCha20-Poly1305 use nonces of 96 bits, CCM 56-104 bits, and EAX a value between 0 and 128 bits. The length of these values is also something to consider when deciding on the most suitable approach.

This chapter has discussed different approaches to fulfill the confidentiality and integrity requirements for the cloud-IoT architecture. The chapter has discussed splitting keys and how the parties could use the keys. Section 5.2 discussed techniques such as double encryption, AEAD with additional MAC, and splitting keys in EAX and ChaCha20-Poly1305. Section 5.3 has discussed the approaches and their advantages and disadvantages. To choose the best solution, one must consider many tradeoffs, and it is clear that different techniques are suitable for various use cases. Overall, the EAX and ChaCha20-Poly1305 approaches seem like the most obvious choices. As EAX is not that established today, and CCM and GCM are the preferred block cipher modes, it is interesting to continue focusing on ChaCha20-Poly1305. The algorithm is specified for TLS 1.3, which means it is more future-oriented than EAX. The ChaCha20-Poly1305 fits most use cases because of advantages such as overall performance, implementation on constrained devices, and the possibility for split keys. The proposal of the modified ChaCha20-Poly1305 has until now only been explained in theory. In order to confirm that this solution meets the integrity and confidentiality requirements defined in Section 1.2, the next chapter will examine the implementation and simulation of the ChaCha20-Poly1305 approach with split keys.

Double AEAD:		
Key size	AEAD key 1	128, 192, or 256 bits
	AEAD key 2	128, 192, or 256 bits
Block-cipher calls	Double GCM	Encryption: $2 \times (\lceil P/128 \rceil + 1)$
		GHASH: $2 \times (\lceil (AD + P)/128 \rceil + 1)$
	Double CCM	$2 \times (2 \times \lceil P/128 \rceil + \lceil AD/128 \rceil)$
Ciphertext expansion		MAC tag $\times 2$
AEAD with additional MAC:		
Key size	GCM/CCM key	128, 192, or 256 bits
	GMAC/CMAC key	128, 192, or 256 bits
Block-cipher calls	GCM + GMAC	Encryption: $\lceil P/128 \rceil + 2$
		GHASH: $2 \times \lceil (AD + P)/128 \rceil + 2$
	CCM + CMAC	$2 \times \lceil P/128 \rceil + \lceil AD/128 \rceil$ $+ \lceil (AD + P)/128 \rceil$
Ciphertext expansion		MAC tag $\times 2$
ChaCha20-Poly1305 with split keys:		
Key size	Confidentiality key	256 bits
	Integrity key	256 bits (smaller keys possible)
Block-cipher calls		N/A
Ciphertext expansion		MAC tag
EAX with split keys:		
Key size	Confidentiality key	128, 192, or 256 bits
	Integrity key	128, 192, or 256 bits
Block-cipher calls		$2 \times \lceil P/128 \rceil + \lceil AD/128 \rceil + \lceil N/128 \rceil$
Ciphertext expansion		MAC tag

Table 5.4: Comparison of key sizes, block-cipher calls, and ciphertext expansions of the design solutions from Chapter 5.

Chapter 6
Implementation and Validation

This thesis has looked at authenticated encryption approaches that solve the cloud-IoT architecture's confidentiality and integrity requirements. The main focus has been to use an AEAD algorithm where one can separate the integrity and confidentiality functions. Section 5.2.3 looked at ChaCha20-Poly1305 with split keys as a suitable technique. In order to validate if the suggested design works, this chapter will focus on the implementation of ChaCha20-Poly1305 with split keys. The chapter will describe how it was implemented and verified with test vectors and simulations. The final part of the chapter will discuss the performance differences between this and the original implementation. This chapter confirms that the implementation makes end-users trust the security of IoT sensor data sent through different communication channels, systems, and services in the cloud-IoT architecture.

6.1 Implementation

The modified ChaCha20-Poly1305 algorithm explained in Section 5.2.3 was implemented at a high level in Python as proof of concept. The implementation builds on the pseudocode in RFC 8439 [NL18] and Bernstein's documentation [ChaCha] [Poly1305]. Some code snippets of the modified version developed in this thesis will be discussed in this chapter. Additionally, this thesis's full implementation and simulation are available on Github [Github]. The implementation shows that the modified algorithm works as expected, allowing for authenticated encryption, authenticated decryption, and separate integrity validation.

6.1.1 ChaCha20 and Poly1305

This section will briefely explain the implementation of ChaCha20 and Poly1305 algorithms. The full implementation in this thesis can be found on Github [Github].

The *chacha20_block()* function generates the pseudorandom keystream. It handles the ChaCha state as a list and performs addition, XOR, and left rotation in quarter

rounds. After 20 rounds, the initial state is added to the current state for each state. Lastly, the final ChaCha state is serialized, and the function outputs a 512-bit pseudorandom string. The function *chacha20_encrypt()* defines the ChaCha20 encryption and decryption. Although the function's name indicates that it handles encryption, it operates both. The reason for this is that the operations are the same. The encryption performs $P_i \oplus KS_i = C_i$ and decryption $C_i \oplus KS_i = P_i$. With the same nonce and counter, the functions generate the same keystream.

The function *poly1305_key_generation()* generates a one-time-key by using the *chacha20_block()* function. Further, *poly1305_mac()* performs the polynomial operations described in Section 2.5.3. The output of the function is the MAC tag.

6.1.2 ChaCha20-Poly1305 with split keys

The function *chacha20_aead_encrypt()* is defined with two keys, K_m and K_c. K_m generates one-time keys for the MAC computation in *poly1305_mac()*, and K_c is used to encrypt the plaintext. This code assumes that the nonce is 96 bits, specified in RFC 8439. If the nonce is smaller than 96 bits, the specification recommends adding a constant value. However, this has not been considered in this code as it defines the nonce as 96 bits. The *chacha20_encrypt()* function uses the initial value 1 as the counter parameter and increases it for each block in the *chacha20_encrypt()* function. The output of *chacha20_aead_encrypt()* is a ciphertext generated by K_c and a MAC tag computed with K_m. A code snippet of the authenticated encryption in AEAD ChaCha20-Poly1305 is given below:

```
def chacha20_aead_encrypt(ad, k_m, k_c, nonce, plaintext):
    one_time_key = poly1305_key_generation(k_m, nonce)
    ciphertext = chacha20_encrypt(k_c, 1, nonce, plaintext)
    mac_data = concatenate_mac_data(ad, ciphertext)
    tag = poly1305_mac(mac_data, one_time_key)
    return (ciphertext, tag)

def chaCha20_poly1305_authenticated_encryption(k_m, k_c, nonce,
     plaintext, ad):
    return chacha20_aead_encrypt(k_m = k_m, k_c = k_c, nonce=nonce,
         plaintext=plaintext, ad=ad)
```

The ChaCha20-Poly1305 decryption function is the same as the encryption function described above, except that the plaintext and ciphertext are interchanged. The algorithm performs the same operations, but the function outputs the plaintext and not the ciphertext. The only significant difference is that the decryption process compares the received MAC tag with the MAC tag computed to verify the integrity. The MAC validation occurs before the plaintext is returned.

The function *poly1305_verify_message()* is designed to verify the MAC tag without being able to decrypt the ciphertext. The function uses K_m to recompute the MAC. It compares the generated tag with the received tag, and if they are the same, the integrity is verified. This function is designed for the intermediate part of the cloud-IoT architecture that should only verify the integrity. The code is illustrated below:

```
def poly1305_verify_message(ad, k_m, nonce, ciphertext):
    one_time_key = poly1305_key_generation(k_m, nonce)
    mac_data = concatenate_mac_data(ad, ciphertext)
    tag = poly1305_mac(mac_data, one_time_key)
    return tag

def compare_mac(a, b):
    if len(a) == len(b):
        if a == b:
            return True
    else:
        return False

def chaCha20_poly1305_authenticate_only(k_m, nonce, ciphertext, mac
     , ad):
    tag = poly1305_verify_message(k_m=k_m, nonce=nonce, ciphertext=
        ciphertext, ad=ad)
    return compare_mac(tag, mac)
```

This implementation has been an attempt to realize split keys in the ChaCha20-Poly1305 algorithm with a separate function verifying integrity. The code runs as expected, and the input and output are tested with test vectors and discrete-event simulations, which Sections 6.2 and 6.3 will describe.

Although this code is a valid implementation of the ChaCha20-Poly1305 algorithm, there are several things with the code that implementers should assess regarding the implementation in IoT sensors. The most important thing is that the code should perform operations in constant time. The reason for this is to avoid side-channel attacks such as timing attacks. The ChaCha20 stream cipher is already in constant time as the implementation relies on fixed operations such as XOR, addition, and rotation. The IoT sensors should implement the Poly1305 algorithm and its arithmetic operations in constant time. For instance, one must handle the carry propagation carefully to achieve constant time. There exist many ways of implementing the arithmetic operations in Poly1305. Bernstein has published a NaCl library that allows for an efficient implementation that runs in constant time [NaCl]. There also exist several other constant time implementations considered efficient

[poly1305-donna] [BearSSL]. One should also implement the comparison of MAC tags in constant time. The validate function should perform a bitwise comparison of the received tag and the calculated tag without revealing information about how long the prefixes of the two tags are identical. An attacker can send many identical messages with different MAC tags and try to "brute force" the tag by sending it into the comparison function. If the timing of the comparison reveals how much of a tag is correct, the attacker can avoid testing all 2^{128} tags and only try the suffix that seems to be wrong. Because of this, the comparison function should be a constant-time function so the attacker cannot reveal how many bits of the tags are identical.

6.2 Running the code with test vectors

This section describes the process of running the code with test vectors to verify that it performs as expected. The code works on and processes bytes, but for simplicity, the output of the computations is converted from bytes to hexadecimal to make it more readable for human eyes. In this scenario, the plaintext is the confidential sensor data, and the AD is some metadata sent in the clear. The sensor possesses two keys: the MAC key K_m and the confidentiality key K_c. In addition, the sensor generates a nonce that must be unique for each invocation with the same key K_c. The parameters that the sensor possesses are illustrated below:

$\boldsymbol{K_m}$ = `cb8ea58af34fc7e9ddce0f41894e2a024c1c1f2eec1c3c0efedadf4052a2280d`
$\boldsymbol{K_c}$ = `a902883257c889d2043ac6a0a7cd1d6bcf63c500e4f57600aa12e50b17a6343f`
Nonce = `d6acb6d43a2696732f05cbf2`
Plaintext = "Secret message" = `536563726574206d657373616765`
AD = "This is AD" = `54686973206973204144`

The plaintext and AD are illustrated in both cleartext and hexadecimal. The algorithm processes them as bytes, but the computations illustrated in this section perform on the text's hexadecimal values. The result is still the same. The keys, K_m and K_c, are 256 bits each. Note that the length of the hexadecimal strings is 64, which in bits are $64 \times 4 = 256$. The nonce is 96 bits, which is a requirement for the algorithm. The plaintext and AD can be of arbitrary length. In this example, they are 112 and 80 bits.

6.2.1 Authenticated encryption

Authenticated encryption involves generating a MAC tag and a ciphertext. One can divide the process into two parts: encrypting the plaintext using ChaCha20 and

generating a ciphertext from Poly1305.

Encrypting the plaintext using ChaCha20:

The ChaCha20 block function generates a keystream XOR-ed with the plaintext, outputting an unreadable ciphertext. The ChaCha20 block function generates the keystream KS_{enc} on the following parameters: K_c, N, and P. The ChaCha20 block function always outputs keystream blocks of 512 bits, regardless of whether the algorithm uses all bits or not. The following keystream is generated with the parameters stated above:

KS_{enc} = `d601b21a00dcfab6a011857ebd67` `91e489ebb6840702fa8ee9553668f95f06e28eab91e386719c06a130ddff2c916f340f70c314bf0c3c9e7fe3ce9c5be454c2`

Since ChaCha20 is a stream cipher, the keystream must be equally long as the plaintext. Therefore, only the first 112 bits of the keystream are used in this example. The following computation takes place to generate the ciphertext:

Ciphertext = Plaintext $\oplus$ KS_{enc} = `536563726574206d657373616765` $\oplus$ `d601b21a00dcfab6a011857ebd67` = `8564d16865a8dadbc562f61fda02`

The following ciphertext: `8564d16865a8dadbc562f61fda02` is the result of the encryption function.

Generating a MAC using Poly1305:

After generating a ciphertext, the Poly1305 generates the MAC tag, using K_m. One can divide this process into two operations: generating the one-time key and generating the authentication tag using the one-time key. In the first operation, the ChaCha20 block function generates a 512-bit block of keystream KS_{poly}. The first 256 bits of the keystream are the one-time key. It is divided into two 128-bit values, K_r and K_s. With the values in this example, the algorithm generates the following keystream:

KS_{poly} = `c0dd7e8ccbb0bab9697c00e5f24c60e0` `78265b1fed44e13c37d8c12ac0d3ed47` `6c16a8249d1c0f3b283173be54cd33ff03d905cdd61f03f155c0aff3761bdc04`

The one-time key is the first half of K_{poly}, $K_{otk} = K_{poly}[0{:}256 \text{ bits}]$:

K_{otk} = `c0dd7e8ccbb0bab9697c00e5f24c60e0` `78265b1fed44e13c37d8c12ac0d3ed47`

The one-time key is divided into two parts, K_r and K_s:

K_r = c0dd7e8ccbb0bab9697c00e5f24c60e0

K_s = 78265b1fed44e13c37d8c12ac0d3ed47

The two key values are used in the Poly1305 computation. In addition to K_r and K_s, MAC data is the input of the Poly1305 computation. MAC data is defined as follows: MAC data = AD || pad(AD) || C || pad(C) || len(AD) || len(C)

The MAC data is: 5468697320697320414400000000000 08564d16865 a8dadbc562f61fda02 0000 0a00000000000000 0e00000000000000

AD || pad(AD) = 54686973206973204144 000000000000
C || pad(C) = 8564d16865a8dadbc562f61fda02 0000
len(AD) = 0a00000000000000
len(C) = 0e00000000000000

The algorithm always states the length-values in little-endian integers of 64 bits. Little-endian means that one sorts the least significant bytes first. In this example, the ciphertext length is 000000000000000e in hexadecimal, which can be converted to the number 14. As 14 bytes is 112 bits ($14 \times 8 = 112$), the same as the plaintext length, it can be verified as correct. The same goes for the AD length since 000000000000000a = 80 bits. The following Poly1305 computation takes place, where the output is the MAC tag:

Poly1305(K_r,K_s,MAC data) = 26cd34ef1201da2beb8a1b8f89815ee3

The result of the authenticated encryption is the ciphertext and MAC tag. When the two values are generated, the IoT sensor sends the following to the receiver:

- Ciphertext = 8564d16865a8dadbc562f61fda02
- MAC tag = 26cd34ef1201da2beb8a1b8f89815ee3
- AD = 54686973206973204144
- Nonce = d6acb6d43a2696732f05cbf2

6.2.2 Authenticated decryption

The receiver receives the AD, nonce, ciphertext, and MAC tag. The end-user possesses the keys K_m and K_c, and the cloud holds K_m. Verifying the MAC, the only operation that the cloud can perform is the first process described in this section.

Verifying the MAC using Poly1305:

When receiving the message, it must first be validated to ensure the consistency of the data. If the data is different from the data sent by the receiver, it is discarded. The receiver recomputes the tag with K_m as described in Section 6.2.1 and compares it with the received one. The MAC data, K_r, and K_s are the same values as the generation. The recomputation of the MAC tag is, therefore:

Poly1305(K_r,K_s,MAC data) = `26cd34ef1201da2beb8a1b8f89815ee3`

The generated tag is compared to the received tag:
`26cd34ef1201da2beb8a1b8f89815ee3` = `26cd34ef1201da2beb8a1b8f 89815ee3`

As the tags are the same, the receiver knows that the data is legitimate. The verification makes the cloud trust the information it receives and lets it store and manage the sensor data. The end-user performs the same verification and additionally performs the decryption.

Decrypting the ciphertext using ChaCha20:

The confidentiality key K_c and nonce are used with the ChaCha20 block function to generate the keystream KS_{enc}. As long as the parameters are the same, the KS_{enc} should be the same keystream as the one generated in the encryption. Therefore, the decryption takes place by XOR-ing the KS_{enc} with the ciphertext:

Plaintext = Ciphertext $\oplus$ KS_{enc} = `8564d16865a8dadbc562f61fda02` $\oplus$ `d601b21a00dcfab6a011857ebd67` = `5365637265742066d657373616765`

The following plaintext: `5365637265742066d657373616765` is the result of the decryption function. If the plaintext is converted back from hexadecimal to text, one can see that the plaintext is the same as the one sent from the sensor:

`53 65 63 72 65 74 20 6d 65 73 73 61 67 65` → "Secret message"

This example is taken from the implementation of the ChaCha20-Poly1305 with split keys. It shows that it is possible to encrypt and decrypt messages and verify the integrity, both as the intermediate part and the receiver.

6.3 Simulation and validation

In order to validate that the code runs as expected and fulfills the confidentiality and integrity requirements, this work has simulated a small cloud-IoT architecture in Python. The simulation was done using SimPy, a process-based discrete-event simulation framework based on Python [SimPy]. The simulation consists of a sensor,

```
Process communication:
SENSOR sends the following message at time 10: {"nonce": "c86589f37e66154d57ebd104", "header": "54686973206973204144",
"ciphertext": "7a14e37f206f3bbeeb68783e25836d7c209c1f279547684395", "tag": "ab8e7eafc8fa7e98df1cddd98be171e2"}

CLOUD receives the following message at time 20: {"nonce": "c86589f37e66154d57ebd104", "header": "54686973206973204144"
, "ciphertext": "7a14e37f206f3bbeeb68783e25836d7c209c1f279547684395", "tag": "ab8e7eafc8fa7e98df1cddd98be171e2"}
Message is verified by cloud
CLOUD sends the following message at time 40: {"nonce": "c86589f37e66154d57ebd104", "header": "54686973206973204144", "
ciphertext": "7a14e37f206f3bbeeb68783e25836d7c209c1f279547684395", "tag": "ab8e7eafc8fa7e98df1cddd98be171e2"}

END-USER receives the following message at time 60: {"nonce": "c86589f37e66154d57ebd104", "header": "546869732069732041
44", "ciphertext": "7a14e37f206f3bbeeb68783e25836d7c209c1f279547684395", "tag": "ab8e7eafc8fa7e98df1cddd98be171e2"}
Message verified! Plaintext is:
This is a secret message!
```

Figure 6.1: Screenshot of the output after the simulation of scenario **S1** is run.

an intermediate cloud, and an end-user. The sensor and end-user possess K_m and K_c in the simulation, while the cloud possesses only K_m. The simulation models the communication between the sensor, cloud, and end-user. The sensor performs authenticated encryption and sends it to the cloud. Further, the cloud verifies the integrity and sends the encrypted data to the end-user. The latter performs authenticated decryption. However, the described communication flow is the desired flow. In order to test that all functionality works as desired, the thesis specified three scenarios:

S1: Data is sent from the sensor to the end-user via the cloud. The sensor performs authenticated encryption and sends the packet to the cloud. The cloud verifies the message with K_m and sends it to the end-user, performing authenticated decryption. This is the desired flow.

S2: Data is sent from the sensor to the end-user via the cloud. An attacker performs a MITM attack between the sensor and the cloud and modifies the ciphertext. The cloud is not able to verify the integrity and discards the message.

S3: Data is sent from the sensor to the end-user via the cloud. The authentication tag is changed between the cloud and the end-user. The end-user cannot verify the integrity when decrypting and discards the message.

All simulations worked as expected and show that the implementation meets the integrity and confidentiality requirements specified in Section 1.2. All the described scenarios worked as expected. Figure 6.1 shows a screenshot of the process communication of scenario **S1** in one of the simulations. The simulation illustrates data packets as JSON strings, where the data is converted to hexadecimal to make it more readable.

Authenticated encryption		Authenticated decryption	
Modified	$5,74 \cdot 10^{-4}$ s	Modified	$5,80 \cdot 10^{-4}$ s
Normal	$5,73 \cdot 10^{-4}$ s	Normal	$5,77 \cdot 10^{-4}$ s
Difference	$1,00 \cdot 10^{-6}$ s	Difference	$3,00 \cdot 10^{-6}$ s

Table 6.1: Performance time of authenticated encryption and authenticated decryption in seconds.

6.4 Performance

Timing measurements were carried out to verify that the performance of the modified ChaCha20-Poly1305 does not differ much from the original ChaCha20-Poly1305 specified in RFC 8439. A Python module called *timeit* was used [Timeit] to measure the execution time. The thesis performed many timing measurements and calculated the average. The following functions were measured:

- *chacha20_poly1305_authenticated_encryption()*
- *chacha20_poly1305_authenticated_decryption()*

The authenticated encryption and the authenticated decryption were measured. The measurements executed the following code to measure the authenticated encryption function:

```
timeit.timeit(lambda: chacha20_poly1305_authenticated_encryption
    (key, nonce, plaintext, ad), number=10000)
```

The functions were tested five times each to make enough measurements, where the timeit module performed 10 000 executions every time. After the executions, the average time was calculated. Table 6.1 shows the performance of authenticated encryption and authenticated decryption. The times are stated in seconds and are the average time from the 50 0000 executions. As the table illustrates, there is almost no difference in performance between the normal and the modified implementation. The authenticated encryption with split keys uses, on average, $1,00 \cdot 10^{-6}$ seconds more time than in the normal function. Additionally, the authenticated decryption takes $3,00 \cdot 10^{-6}$ seconds more than the original implementation. Such numbers are so small that one considers them negligible.

The performance was calculated on an Intel Core i5-8250U processor with 8 GB RAM. The result from the performance measurements is not directly transferable to IoT sensors. The resources on IoT sensors are more limited than this processor. Additionally, the software implementation of the algorithm would be a more low-

level approach than in Python. The performance time should be different on IoT processors, but the relative differences between the normal implementation and the one with split keys should not deviate much. Table 6.1 illustrates the performance of authenticated encryption and decryption in the two implementations.

The function *chaCha20_poly1305_authenticate_only()* was also tested so it could be compared to the authenticated decryption function. On average, the function takes $3,01 \cdot 10^{-4}$ seconds to execute. This time is $2,79 \cdot 10^{-4}$ seconds faster than the authenticated decryption, which seems reasonable as the function only verifies the integrity and does not handle decryption.

This thesis has not considered an implementation of the algorithm in constant time. However, the actual implementation should realize this. Implementers should implement operations in constant time to avoid side-channel vulnerabilities. Since this implementation does not consider constant time and is at a high level, the performance measurements do not provide more usefulness than showing that the performance of the modified version of ChaCha20-Poly1305 is approximately the same as the original one.

This chapter has covered the implementation and validation of ChaCha20-Poly1305 with split keys. The implementation works as expected, and one can use the AEAD technique while separating the integrity and confidentiality. This approach allows intermediate parties to only verify the integrity by possessing the integrity key. The implementation has been validated in simulations and therefore stands as a good alternative to solving this cloud-IoT architecture's confidentiality and integrity requirements.

Chapter 7
Discussion and Conclusion

This thesis has considered authenticated encryption in the cloud-IoT architecture. The overall goal was to determine how end-users can trust the security of IoT sensor data sent through different communications channels, systems, and services. As specified in the introduction, protecting the data from attackers using integrity and confidentiality mechanisms was the main focus. Since all sensor data goes through an intermediate cloud party, the motivation for this project was to meet the requirements mentioned in Section 1.2. This chapter includes a discussion and conclusion of the work performed in this thesis. The discussion reflects on the work's process, findings, and limitations, and the conclusion explains how the result has answered the research questions. As a continuation of this work, the chapter ends by discussing the future work of this thesis.

7.1 Discussion

The work in this thesis has followed the design cycle to find a solution to the challenges of securing data in the cloud-IoT architecture described in Chapter 1. Starting with defining the scope by identifying the stakeholders and their goals, then specifying all requirements, the thesis worked on designing a solution. The design focused on building on and using known algorithms and standards. This step required research and discussion of the various tradeoffs and approaches. The design solution was validated through implementation and simulation to predict how it will interact in the real-world cloud-IoT architecture and validate that the requirements were satisfied.

The cloud-IoT architecture is complex since it involves numerous parties, constraints, systems, security threats, and elements. The thesis has limited the scope to authenticated encryption of sensor data. Different techniques were explored, and various advantages and disadvantages were considered in designing a solution. The rationale is that the best solution depends on the most important priorities decided

by the stakeholders. As a result, different tradeoffs in cost, security threats, devices, and performance were discussed.

The chosen solution is one of many possible approaches to solving the research questions. Despite discussing solutions involving CBC mode and HMAC, further research only included AEAD algorithms. Different algorithms could have led to other alternative solutions. This thesis focused on AEAD since TLS 1.3 has cut out algorithms considered legacy and only includes AEAD algorithms, as described in Section 4.1. Another alternative would have been to focus on lightweight algorithms such as PRESENT, Simon, or Speck, but as discussed in Section 2.3.2, they introduce some uncertainty and are not widely used. Therefore, the focus was on lightweight implementations of widespread algorithms that are standardized and proven secure by trusted authorities.

The thesis has discussed approaches such as double AEAD encryption, AEAD with additional MAC, and splitting keys in EAX and ChaCha20-Poly1305. Different tradeoffs and use cases appropriate to different solutions have been described to give a reflected discussion. ChaCha20-Poly1305 with split keys was implemented and simulated in python as it generally was the best approach to answer the research question. The thesis introduced splitting keys in the algorithm to reduce the number of keys and key exchanges for the IoT sensors. However, this requires a trusted key management entity. The implementation and simulation in python were performed to predict how the algorithm would behave in the cloud-IoT context. The simulation involved scenarios such as MITM and packet sniffing attacks to test that the design solves the problems stated. The utilization of the algorithm depends on use cases, but the thesis shows that it offers benefits in many aspects.

Although ChaCha20-Poly1305 with split keys answers the problems stated in this thesis, there are challenges to this solution. How the system should perform key management and splitting keys must be specified. Different ways of doing so were explained, and the appropriate solution depends on stakeholders and their priorities. Another challenge is that the algorithm requires quite large key sizes, which is a downside to the solution. The algorithm performs better when encrypting a larger amount of data simultaneously rather than frequently encrypting small amounts of data as it always generates keystream blocks of 512-bits. Depending on the size of the sensor data, the algorithm might generate a lot of unused keystream bits in bulk encryption which results in unnecessary computations.

This thesis has conducted a more theoretical analysis with a high-level implementation as a proof of concept. The implementation and validation of ChaCha20-Poly1305 show that the algorithm works in a simulated context. The work in this project has not been conducted on IoT sensors, but the output and result should be transfer-

able to the IoT sensor context. The thesis has discussed in theory the possibility of ChaCha20-Poly1305 implementations on sensors. The high-level implementation gives guidance on converting it to a more low-level implementation on sensors. However, to validate the solution in a more real-world context, there is a need to implement and test the solution on IoT sensors in the cloud-IoT architecture. The algorithm should be implemented and tested on IoT sensors before one can draw great conclusions.

7.2 Conclusion

This master thesis has worked on the following research question: *How can end-users trust the security of IoT sensor data sent through different communication channels, systems, and services?*

This thesis has focused on the design problem of designing an algorithm that fulfills integrity and confidentiality for end-users while achieving integrity for a third party in the cloud to make the parties in the cloud-IoT architecture trust the security of the sensor data. The project has worked on answering research questions **RQ1-RQ4**, defined in Section 1.3, to find a solution that would meet these requirements.

In order to find answers to the research questions, authenticated encryption in symmetric cryptography has been the focus. Because of the limited resources of IoT sensors, lightweight cryptography is the best approach to ensure the integrity and confidentiality of sensor data. As discussed in Chapter 2, researchers and organizations work towards standardizing more lightweight algorithms. Today, constrained devices often use regular standardized algorithms that support lightweight implementations. As an answer to **RQ1:** *What type of cryptography and which algorithms should be used?*, this thesis focused on such algorithms. It concluded that stream ciphers, block ciphers, and hashing algorithms were the best cryptography for constrained devices because of their low-cost operations. Because of this, AEAD algorithms are the best approach as they are forward-oriented and use such cryptographic primitives to combine integrity and confidentiality securely.

This thesis researched splitting keys in AEAD modes and detaching and checking the MAC separately in such modes to solve research question **RQ2:** *How can the chosen solution fulfill integrity for a third party in the cloud and achieve integrity and confidentiality for end-users?*. The conclusion is that splitting keys in EAX and ChaCha20-Poly1305 fulfills integrity for a third party in the cloud and achieves integrity and confidentiality for end-users. The modified ChaCha20-Poly1305 algorithm supports software implementations, uses CPU-friendly operations, and is resistant to timing side-channel attacks. As discussed in the thesis, ChaCha20-Poly1305 works well in performance and can be implemented on constrained devices. ChaCha20-

Poly1305 with split keys was implemented and simulated at a high level in python to predict how it would interact in the cloud-IoT context. The implementation and simulation worked as expected, and it was validated that the solution fulfills the confidentiality and integrity requirements. Additionally, this work has shown that the performance of the modified algorithm does not deviate from the original one.

Splitting keys and key management has been discussed to answer **RQ3:** *How does key management affect possible solutions?*. If the sensor and the party responsible for key management exchange one key where the parties split it into confidentiality and integrity parts, the solution only requires one key exchange for the sensor per session. The party responsible for key management must further distribute the necessary key parts to the involved parties. As discussed in Chapter 3, sensors can securely agree upon keys with either ECDH or key evolution schemes. These approaches are the best key exchange protocols when looking at constrained devices. Independent of the key exchange algorithm the system uses, the algorithms should be based on pre-shared keys generated and distributed during production to reduce the number of expensive operations.

The ChaCha20-Poly1305 solution gives both advantages and disadvantages in contrast to other algorithms. Section 5.3 compared the solution with different schemes and discussed their pros and cons to answer the research question **RQ4:** *What advantages and disadvantages does the solution introduce compared to other schemes?*. ChaCha20-Poly1305 is acceptable for constrained devices because of its low-cost operations. The algorithm performs well in performance and battery consumption, only out-competed by AES on devices with hardware accelerators. The ChaCha20-Poly1305 and AES solutions have various security advantages in preventing side-channel attacks depending on the implementation. Common to them is that the implementations must consider side-channel attacks carefully. The ChaCha20-Poly1305 solution is not perfect and introduces challenges in terms of memory requirements because of potentially large key sizes. However, on the other hand, it ensures better security strength than AES with 128-bit or 192-bit keys which is an important security aspect considering Moore's law. If key sizes are too large in ChaCha20-Poly1305, EAX stands out as a good solution. It allows for splitting keys and is the best approach for memory consumption as the key size, nonce length, and MAC tag can be chosen based on the implementation because of the algorithm's flexibility. The comparison showed that ChaCha20-Poly1305 with splitting keys solution is a good solution in most use cases because of advantages such as overall performance, implementation on constrained devices, and the possibility for split keys.

The ChaCha20-Poly1305 with split keys solution described and implemented in this work lets the cloud party verify the integrity and makes end-users trust the

security of IoT sensor data sent through different communication channels, systems, and services. This work concludes that the solution answers the research questions in this thesis.

7.3 Further work

The next reasonable step in future work is implementing the ChaCha20-Poly1305 proposal on IoT sensors and simulating them in the cloud-IoT architecture in the real-world context. It would be beneficial to implement the solution on IoT sensors to measure metrics and observe the algorithm in practice. This approach would allow going into more detail and discovering potential challenges. Further, simulations could construct attacks such as MITM, packet sniffing, and side-channel attacks to evaluate the solution. Also, it would be interesting to implement the proposed EAX solution from Section 5.2.4 on IoT sensors and compare it with the ChaCha20-Poly1305 proposal.

This thesis researched approaches to splitting keys in AEAD modes, where one could detach and check the MAC separately. The focus ended up on ChaCha20-Poly1305, but using GCM would also be an interesting approach. GCM follows the EtM approach, although it includes some small modifications. As mentioned in Section 4.2.2, it would be interesting to perform some small changes to the algorithm to perform integrity verification without obtaining the confidentiality key. It should be researched if it breaks the standard, and if not, it could be implemented in the cloud-IoT architecture to compare it with the ChaCha20-Poly1305 approach.

Another approach is to investigate key management in ChaCha20-Poly1305 or EAX in more detail. As key evolution schemes are relatively new, it would be interesting to implement such schemes and compare them with ECDH to find out if or how much better key evolution would be for constrained devices. Additionally, a sensor can have two encryption streams with key evolution: one for the end-user and another for the cloud service. It would be interesting to research if a sensor could maintain both these sessions simultaneously so that the cloud can decode one stream and the end-user the other. This approach would allow for other solutions that could be useful. If this is feasible or too expensive for IoT sensors would be an interesting question to answer in future work.

References

[ACF19] G. Avoine, S. Canard, and L. Ferreira, *Symmetric-key Authenticated Key Exchange (SAKE) with Perfect Forward Secrecy*, Cryptology ePrint Archive, Report 2019/444, 2019.

[AP13] N. J. AlFardan and K. G. Paterson, «Lucky Thirteen: Breaking the TLS and DTLS Record Protocols», in *2013 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2013, pp. 526–540.

[Bar20] E. Barker, «NIST special publication 800-57 part 1, revision 5», *NIST, Tech. Rep*, vol. 16, May 2020.

[BBB+06] E. Barker, E. Barker, *et al.*, *Recommendation for key management: Part 1: General.* National Institute of Standards and Technology, 2006.

[BCK+17] E. Barker, L. Chen, *et al.*, «Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography», National Institute of Standards and Technology, Tech. Rep., 2017.

[BD15] E. Barker and Q. Dang, «Recommendation for Key Management Part 3: Application-Specific Key Management Guidance», *NIST Special Publication 800-57 Part 3 Revision 1*, pp. 1–102, Jan. 2015.

[BDdK+21] C. Boyd, G. T. Davies, *et al.*, *Symmetric Key Exchange with Full Forward Security and Robust Synchronization*, Cryptology ePrint Archive, Report 2021/702, 2021.

[BearSSL] Constant-Time in BearSSL. [Online]. Available: https://www.bearssl.org/constanttime.html (last visited: May 3, 2022).

[Ber05] D. J. Bernstein, «The Poly1305-AES Message-Authentication Code», in *Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds., ser. Lecture Notes in Computer Science, vol. 3557, Springer, 2005, pp. 32–49.

[BH09] M. Badra and I. Hajjeh, «ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)», *Internet Eng. Task Force, RFC5489, Fremont, CA, USA*, 2009.

[BK+07] E. B. Barker, J. M. Kelsey, *et al.*, *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, NIST, 2007.

[BKL+07] A. Bogdanov, L. R. Knudsen, *et al.*, «PRESENT: An Ultra-Lightweight Block Cipher», P. Paillier and I. Verbauwhede, Eds., vol. 4727, Springer, 2007, pp. 450–466.

[BN00] M. Bellare and C. Namprempre, «Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm», T. Okamoto, Ed., ser. Lecture Notes in Computer Science, vol. 1976, Springer, 2000, pp. 531–545.

[BN14] C. Blondeau and K. Nyberg, «Links between Truncated Differential and Multidimensional Linear Properties of Block Ciphers and Underlying Attack Complexities», in *Advances in Cryptology - EUROCRYPT 2014*, P. Q. Nguyen and E. Oswald, Eds., ser. Lecture Notes in Computer Science, vol. 8441, Springer, 2014, pp. 165–182.

[BP17] A. Biryukov and L. Perrin, «State of the Art in Lightweight Symmetric Cryptography», *IACR Cryptol. ePrint Arch.*, p. 511, 2017.

[BR18] E. Barker and A. Roginsky, «Transitioning the use of cryptographic algorithms and key lengths», National Institute of Standards and Technology, Tech. Rep., 2018.

[BRW03] M. Bellare, P. Rogaway, and D. A. Wagner, «EAX: A Conventional Authenticated-Encryption Mode», *IACR Cryptol. ePrint Arch.*, p. 69, 2003.

[BRW04] M. Bellare, P. Rogaway, and D. Wagner, «The EAX mode of operation», in *International Workshop on Fast Software Encryption*, Springer, 2004, pp. 389–407.

[BSS+13] R. Beaulieu, D. Shors, *et al.*, *The SIMON and SPECK Families of Lightweight Block Ciphers*, Cryptology ePrint Archive, Report 2013/404, 2013.

[ChaCha] The ChaCha family of stream ciphers. [Online]. Available: https://cr.yp.to/chacha.html (last visited: Apr. 6, 2022).

[ChaPol] Wikipedia - ChaCha20-Poly1305. [Online]. Available: https://en.wikipedia.org/wiki/ChaCha20-Poly1305 (last visited: Mar. 28, 2022).

[Che+08] L. Chen *et al.*, «Recommendation for key derivation using pseudorandom functions», *NIST special publication*, vol. 800, p. 108, 2008.

[CSRC17] Update to Current Use and Deprecation of TDEA. [Online]. Available: https://csrc.nist.gov/news/2017/update-to-current-use-and-deprecation-of-tdea (last visited: Mar. 8, 2022).

[CWE-323] CWE-323: Reusing a Nonce, Key Pair in Encryption. [Online]. Available: https://cwe.mitre.org/data/definitions/323.html (last visited: May 5, 2022).

[DR08] T. Dierks and E. Rescorla, «The Transport Layer Security (TLS) Protocol Version 1.2», *RFC*, vol. 5246, pp. 1–104, 2008.

[DSS17] F. De Santis, A. Schauer, and G. Sigl, «ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications», 2017, pp. 692–697.

[Dwo04] M. Dworkin, «Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality», *NIST Special Publication 800-38C*, pp. 1–27, May 2004.

[Dwo12] M. J. Dworkin, «Recommendation for block cipher modes of operation: methods for key wrapping», 2012.

[Dwo16] M. Dworkin, «Recommendation for block cipher modes of operation: The CMAC mode for authentication», 2016.

[Eri18] R. Eric, «The Transport Layer Security (TLS) Protocol Version 1.3», *RFC*, vol. 8446, pp. 1–160, 2018.

[ET05] P. Eronen and H. Tschofenig, «Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)», *RFC*, vol. 4279, pp. 1–15, 2005.

[Github] Github repository - haakotj/master-thesis. [Online]. Available: https://github.com/haakotj/master-thesis.git (last visited: May 18, 2022).

[GK10] S. Gueron and M. E. Kounavis, «Intel® carry-less multiplication instruction and its usage for computing the GCM mode», *White Paper*, p. 76, 2010.

[Gro+17] C. L. C. W. Group *et al.*, «CRYPTREC Cryptographic Technology Guideline (Lightweight Cryptography)», *CRYPTREC Report March*, 2017.

[Gut14] P. Gutmann, «Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)», *RFC*, vol. 7366, pp. 1–7, 2014.

[HH18] C. Hung and W. Hsu, «Power Consumption and Calculation Requirement Analysis of AES for WSN IoT», *Sensors*, vol. 18, no. 6, p. 1675, 2018.

[Hou07] R. Housley, «Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS)», *RFC*, vol. 5084, pp. 1–11, 2007.

[IEE20] IEEE, «IEEE Standard for Low-Rate Wireless Networks», *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pp. 1–800, 2020.

[ISO19772] ISO/IEC 19772 - Authenticated Encryption. [Online]. Available: https://www.iso.org/standard/81550.html (last visited: Jan. 31, 2022).

[ISO29167-21] ISO/IEC 29167-21:2018 Information technology - Part 21: Crypto suite SIMON security services for air interface communications. [Online]. Available: https://www.iso.org/standard/70388.html (last visited: Mar. 4, 2022).

[ISO29167-22] ISO/IEC 29167-22:2018 Information technology - Part 22: Crypto suite SPECK security services for air interface communications. [Online]. Available: https://www.iso.org/standard/70389.html (last visited: Mar. 4, 2022).

[ISO29192] ISO/IEC 29192-2:2019 Information security - Lightweight cryptography - Part 2: Block ciphers. [Online]. Available: https://www.iso.org/standard/78477.html (last visited: Mar. 6, 2022).

[JdOB+11] M. A. S. Jr., B. T. de Oliveira, *et al.*, «Comparison of Authenticated-Encryption schemes in Wireless Sensor Networks», in *IEEE 36th Conference on Local Computer Networks*, C. T. Chou, T. Pfeifer, and A. P. Jayasumana, Eds., IEEE Computer Society, 2011, pp. 450–457.

[Jea16] J. Jean, *TikZ for Cryptographers*, https://www.iacr.org/authors/tikz/, 2016.

[Jon02] J. Jonsson, «On the security of CTR + CBC-MAC», K. Nyberg and H. M. Heys, Eds., ser. Lecture Notes in Computer Science, vol. 2595, Springer, 2002, pp. 76–93.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti, «HMAC: Keyed-Hashing for Message Authentication», *RFC*, vol. 2104, pp. 1–11, 1997.

[Kra01] H. Krawczyk, «The order of encryption and authentication for protecting communications (Or: how secure is SSL?)», *IACR Cryptol. ePrint Arch.*, p. 45, 2001.

[LCM+16] A. Langley, W. Chang, *et al.*, «ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)», *RFC*, vol. 7905, pp. 1–8, 2016.

[LDS09] A. H. Lashkari, M. M. S. Danesh, and B. Samadi, «A survey on wireless security protocols (WEP, WPA and WPA2/802.11i)», in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, 2009, pp. 48–52.

[Lee14] C. Lee, «Biclique cryptanalysis of PRESENT-80 and PRESENT-128», *J. Supercomput.*, vol. 70, no. 1, pp. 95–103, 2014.

[LL15] I. Lee and K. Lee, «The Internet of Things (IoT): Applications, investments, and challenges for enterprises», *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[MA20] I. Mohiuddin and A. Almogren, «Security Challenges and Strategies for the IoT in Cloud Computing», in *2020 11th International Conference on Information and Communication Systems*, Apr. 2020, pp. 367–372.

[MBSM16] K. McKay, L. Bassham, *et al.*, «Report on lightweight cryptography», National Institute of Standards and Technology, Tech. Rep., 2016.

[MBTM17] K. McKay, L. Bassham, *et al.*, *Report on Lightweight Cryptography*, Mar. 2017.

[McG05] D. A. McGrew, «Efficient Authentication of Large, Dynamic Data Sets Using Galois/Counter Mode (GCM)», in *3rd International IEEE Security in Storage Workshop (SISW 2005)*, IEEE Computer Society, 2005, pp. 89–94.

[ML16] The internet of things is in your future - the law says so! [Online]. Available: https://www.techtarget.com/iotagenda/blog/IoT-Agenda/The-internet-of-things-is-in-your-future-the-law-says-so (last visited: May 10, 2022).

[MLMI13] K. Minematsu, S. Lucks, *et al.*, «Attacks and Security Proofs of EAX-Prime», S. Moriai, Ed., ser. Lecture Notes in Computer Science, vol. 8424, Springer, 2013, pp. 327–347.

[Mor07] D. Morris, «Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC», *NIST Special Publication 800-38D*, pp. 1–39, Nov. 2007.

[MV04] D. McGrew and J. Viega, «The Galois/counter mode of operation (GCM)», *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[NaCl] NaCl: Networking and Cryptography library. [Online]. Available: https://nacl.cr.yp.to/ (last visited: May 3, 2022).

[Nan09] M. Nandi, «Fast and Secure CBC-Type MAC Algorithms», in *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, O. Dunkelman, Ed., ser. Lecture Notes in Computer Science, vol. 5665, Springer, 2009, pp. 375–393.

[NBT20] N. Nguyen, D. Bui, and X. Tran, «A Lightweight AEAD encryption core to secure IoT applications», IEEE, 2020, pp. 35–38.

[NJJ+18] Z. Najm, D. Jap, *et al.*, «On Comparing Side-channel Properties of AES and ChaCha20 on Microcontrollers», in *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2018, pp. 552–555.

[NL18] Y. Nir and A. Langley, «ChaCha20 and Poly1305 for IETF Protocols», *RFC*, vol. 8439, pp. 1–46, 2018.

[NSAcyber] Simon and Speck bibliography. [Online]. Available: https://nsacyber.github.io/simon-speck/bibliography/ (last visited: Mar. 6, 2022).

[OGOP04] S. Ors, F. Gurkaynak, *et al.*, «Power-analysis attack on an ASIC AES implementation», in *International Conference on Information Technology: Coding and Computing*, vol. 2, 2004, 546–552 Vol.2.

[OST06] D. A. Osvik, A. Shamir, and E. Tromer, «Cache Attacks and Countermeasures: The Case of AES», in *Topics in Cryptology - CT-RSA*, D. Pointcheval, Ed., ser. Lecture Notes in Computer Science, vol. 3860, Springer, 2006, pp. 1–20.

[PBM00] D. Park, C. Boyd, and S.-J. Moon, «Forward secrecy and its application to future mobile communications security», in *International Workshop on Public Key Cryptography*, Springer, 2000, pp. 433–445.

[PH05] E. Pasi and T. Hannes, «Pre-shared key ciphersuites for transport layer security (TLS)», RFC 4279, December, Tech. Rep., 2005.

[PJW10] S. Paquette, P. T. Jaeger, and S. C. Wilson, «Identifying the security risks associated with governmental use of cloud computing», *Gov. Inf. Q.*, vol. 27, no. 3, pp. 245–253, 2010.

[Poly1305] A state-of-the-art message-authentication code. [Online]. Available: https://cr.yp.to/mac.html (last visited: Apr. 6, 2022).

[poly1305-donna] Poly1305-Donna. [Online]. Available: https://github.com/floodyberry/poly1305-donna (last visited: May 3, 2022).

[PQC] Post-Quantum Cryptography. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography (last visited: Apr. 25, 2022).

[PQC-R3] Post-Quantum Cryptography. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions (last visited: May 13, 2022).

[Pro14] G. Procter, «A Security Analysis of the Composition of ChaCha20 and Poly1305», *IACR Cryptol. ePrint Arch.*, p. 613, 2014.

[RDJ+17] B. Ray, S. Douglas, *et al.*, *Notes on the design and analysis of SIMON and SPECK*, Cryptology ePrint Archive, Report 2017/560, 2017.

[Res18] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018.

[ROC+20] S. S. Rizvi, R. J. Orr, *et al.*, «Identifying the attack surface for IoT network», *Internet Things*, vol. 9, p. 100 162, 2020.

[SBK+17] M. Stevens, E. Bursztein, *et al.*, «The first collision for full SHA-1», in *Annual international cryptology conference*, Springer, 2017, pp. 570–596.

[SBSD16] S. Salami, J. Baek, *et al.*, «Lightweight Encryption for Smart Home», in *11th International Conference on Availability, Reliability and Security*, IEEE Computer Society, Sep. 2016, pp. 382–388.

[SGTW20] V. K. Sarker, T. N. Gia, *et al.*, «Lightweight Security Algorithms for Resource-constrained IoT-based Sensor Nodes», in *2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

[SH02] J. Schaad and R. Housley, «Advanced Encryption Standard (AES) Key Wrap Algorithm», *RFC*, vol. 3394, pp. 1–41, 2002.

[SHS15] Y. Sheffer, R. Holz, and P. Saint-Andre, «Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)», *RFC*, vol. 7457, pp. 1–13, 2015.

[SimPy] Discrete event simulation for Python. [Online]. Available: https://simpy.readthedocs.io/en/latest/ (last visited: May 2, 2022).

[SKK10] P. Szalachowski, B. Ksiezopolski, and Z. Kotulski, «CMAC, CCM and GCM/GMAC: Advanced modes of operation of symmetric block ciphers in wireless sensor networks», *Inf. Process. Lett.*, vol. 110, no. 7, pp. 247–251, 2010.

[SKP20] Y. Sovyn, V. Khoma, and M. Podpora, «Comparison of Three CPU-Core Families for IoT Applications in Terms of Security and Performance of AES-GCM», *IEEE Internet Things J.*, vol. 7, no. 1, pp. 339–348, 2020.

[SLdP+19a] D. A. F. Saraiva, V. R. Q. Leithardt, *et al.*, «PRISEC: Comparison of Symmetric Key Algorithms for IoT Devices», *Sensors*, vol. 19, no. 19, p. 4312, 2019.

[SLdP+19b] D. A. F. Saraiva, V. R. Q. Leithardt, *et al.*, «PRISEC: Comparison of Symmetric Key Algorithms for IoT Devices», *Sensors*, vol. 19, no. 19, 2019.

[Sma16] N. P. Smart, «Historical Stream Ciphers», in *Cryptography Made Simple*, Springer, 2016, pp. 179–194.

[Sony] Sony Develops "CLEFIA" - New Block Cipher Algorithm Based on State-of-the-art Design Technologies. [Online]. Available: https://www.sony.com/en/SonyInfo/News/Press/200703/07-028E/ (last visited: Mar. 6, 2022).

[SSM+17] S. Singh, P. Sharma, *et al.*, «Advanced lightweight encryption algorithms for IoT devices: Survey, challenges and solutions», *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–18, May 2017.

[SSS17] F. D. Santis, A. Schauer, and G. Sigl, «ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications», in *Design, Automation & Test in Europe Conference & Exhibition*, D. Atienza and G. D. Natale, Eds., IEEE, 2017, pp. 692–697.

[Sta10] F. Standaert, «Introduction to Side-Channel Attacks», in *Secure Integrated Circuits and Systems*, ser. Integrated Circuits and Systems, I. M. R. Verbauwhede, Ed., Springer, 2010, pp. 27–42.

[Šve16] P. Švenda, «Basic comparison of Modes for Authenticated-Encryption (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS)», vol. 35, 2016.

[Timeit] timeit - Measure execution time of small code snippets. [Online]. Available: https://docs.python.org/3/library/timeit.html (last visited: May 2, 2022).

[Tjo21] H. Tjomsland, «Flexible Security for Sensor Data in Heteregenous Networks», vol. 1, no. 1, pp. 00–15, Nov. 2021.

[TLSIV] TLS Symmetric Crypto. [Online]. Available: https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html (last visited: May 3, 2022).

[TMC+21] M. Turan, K. McKay, *et al.*, «Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process», *NISTIR 8369*, pp. 1–92, Jun. 2021.

[TRK20] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, «Lightweight Cryptography for IoT: A State-of-the-Art», *CoRR*, vol. abs/2006.13813, 2020.

[Tur11] S. Turner, *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*, RFC 6151, Mar. 2011.

[Vau02] S. Vaudenay, «Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...», L. R. Knudsen, Ed., ser. Lecture Notes in Computer Science, vol. 2332, Springer, 2002, pp. 534–546.

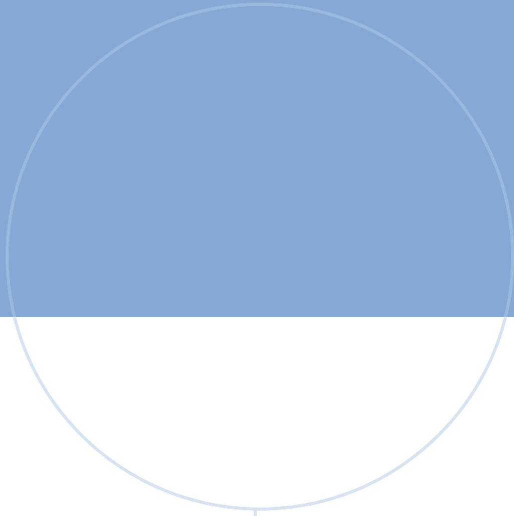
[VP17] M. Vanhoef and F. Piessens, «Key reinstallation attacks: Forcing nonce reuse in WPA2», in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1313–1328.

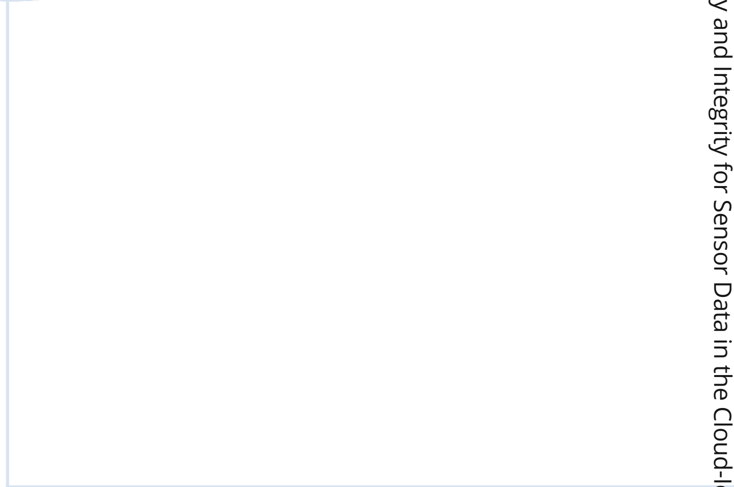
[WCS+18] D. Wang, D. Chen, *et al.*, «From IoT to 5G I-IoT: The Next Generation IoT-Based Intelligent Algorithms and 5G Technologies», *IEEE Communications Magazine*, vol. 56, no. 10, pp. 114–120, 2018.

[WDH+19] M. Wazid, A. K. Das, *et al.*, «Authentication in cloud-driven IoT-based big data environment: Survey and outlook», *J. Syst. Archit.*, vol. 97, pp. 185–196, 2019.

[WHF03] D. Whiting, R. Housley, and N. Ferguson, «Counter with CBC-MAC (CCM)», 2003.

[YCT15] X. Yao, Z. Chen, and Y. Tian, «A lightweight attribute-based encryption scheme for the Internet of Things», *Future Gener. Comput. Syst.*, vol. 49, pp. 104–112, 2015.





Håkon Tjomsland

Confidentiality and Integrity for Sensor Data in the Cloud-IoT Architecture

NTNU
Norwegian University of
Science and Technology