Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Petter Solnør

Authenticated Encryption Methods for Feedback Control Systems

Master's thesis in Cybernetics and Robotics

Supervisor: Thor I. Fossen

December 2020

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Master's Thesis

Authenticated Encryption Methods for Feedback Control Systems

Petter Solnør

Submission date: December 22nd, 2020
Supervisor: Thor I. Fossen
Co-supervisor: Slobodan Petrovic

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Master's Thesis Description

Introduction

This is a master's thesis written at the Department of Engineering Cybernetics, NTNU. The work is a continuation of the work described in the TTK4550 project report completed in the spring of 2020.

Main Objective

The work described in this thesis is motivated by the need to enhance the cybersecurity of feedback control systems. The signals transmitted in feedback control systems must be made resistant against unauthorized eavesdropping, and spoofed signals must not be accepted by the feedback control system. The system must be cryptographically strong while not inducing intolerable latencies or synchronization problems that would be detrimental to the overall performance of the system, possibly resulting in a loss of control.

Tasks

- Conduct a literature study on how researchers have attempted to secure feedback control systems using cryptographic methods.
- Demonstrate the cyber-physical vulnerability of feedback control systems.
- Demonstrate that previously proposed schemes are vulnerable to both passive and active attacks.
- Propose an alternative, cryptographically strong scheme.
- Expand the collection of cryptographic algorithms that were implemented in TTK4550.
- Standardize the interfaces of the algorithm implementations to create a proper toolbox.
- Provide enhanced implementations of AES and AEGIS that take advantage of `ARMv8` architecture and `x86` architecture hardware acceleration features.
- Assess the performance of the proposed scheme with the algorithm implementations, in the encryption laboratory that was built in TTK4550.
- Demonstrate that the proposed scheme is robust against attacks.
- Outline some suggestions for future work based on the results and the research.

Abstract

Feedback control systems require that signals are transmitted between sensors, state estimators, controllers, and actuators. These signal transmissions enable adversaries to eavesdrop on the signals that are transmitted to conduct system identification, potentially revealing system parameters and control parameters that may be considered confidential. Worse yet, an adversary could potentially alter messages or inject spoofed messages to manipulate the behavior of the controller, state estimator, or actuator to gain control of the system. The hijacking of a dynamical system, for example, an Unmanned Surface Vehicle (USV), can result in catastrophic material and economic consequences if used to inflict damage as part of a terrorist attack or as an act of war. Therefore, securing the signal transmission in feedback control systems is of great importance.

We may accomplish this through the use of cryptographic tools. While some research has been conducted in this area in the past, the research that has been published has consisted of ciphers that are not optimal by modern standards, both concerning security and performance. The cryptographic algorithms have also been used in insecure ways, thus enabling attacks. In this thesis, the previously proposed schemes are analyzed, and attacks are implemented. The attacks demonstrate that the previously proposed schemes leak information and even enable the injection of forged messages, contrary to the claims of the authors. This thesis attempts to improve on the work that has been published by bringing in modern, cryptographically sound techniques. A scheme that is cryptographically strong, and not vulnerable to the demonstrated attacks, is then proposed.

To implement the proposed scheme, a toolbox containing high-performance implementations of state-of-the-art cryptographic algorithms is developed. The algorithm implementations are benchmarked for considerations such as the latency that is induced by processing various amounts of data. Other important aspects, such as synchronization mechanisms and traffic expansion, are also treated. The experiments are conducted on industrial Raspberry Pis in an encryption laboratory setup that was built as part of previous project work. The Raspberry Pis are used to ensure that the results obtained are relevant in an industrial setting in which embedded devices are more likely to be present than powerful desktop machines. The results show that the state-of-the-art stream cipher implementations outperform traditional block ciphers. Furthermore, the algorithms induce very little latency, less than a millisecond on small amounts of data, and are therefore well-suited for real-time applications.

Keywords: Applied Cryptography, Cryptanalysis, Feedback Control Systems, Networked Control Systems, Cyber-Physical Systems

Sammendrag

Tilbakekoblingssystemer krever at signaler sendes mellom sensorer, tilstandsestimatorer, regulatorer og aktuatorer. Disse signalene er sårbare for tyvlytting, som gjør at uautoriserte aktører kan få kjennskap til potensielt konfidensielle system- og regulatorparametre gjennom systemidentifikasjon. Uautoriserte aktører kan også manipulere komponenter i tilbakekoblingssystemet ved å manipulere signaler og sende inn falske signaler. Dersom en uautorisert aktør klarer å kapre et slikt system, for eksempel et ubemannet fartøy, kan det få katastrofale materielle og økonomiske følger dersom det kaprede fartøyet benyttes som et verktøy i et terrorangrep eller i en krigshandling. Sikring av signalene i et tilbakekoblingssystem er derfor svært viktig.

For å sikre signalene i et tilbakekoblingssystem kan vi benytte oss av kryptografiske verktøy. Selv om noe forskning er publisert på dette fagområdet tidligere, har forskere brukt eldre, og tidvis utdaterte, algoritmer. Måtene de kryptografiske algoritmene har vært benyttet på har også vært mangelfulle, noe som har gjort at mange av forslagene er svært sårbare for angrep. I denne avhandlingen analyserer vi tidligere forslag og implementerer angrep som viser sårbarhetene. Angrepene demonstrerer at de tidligere forslagene gjør at systemene lekker informasjon og fører til tap av konfidensialitet, og også gjør det mulig å forfalske signaler og dermed manipulere systemet. Som et alternativ, foreslås derfor kryptografisk sterke metoder som ikke er sårbare for disse angrepene.

For å implementere de foreslåtte kryptografiske metodene blir en verktøykasse med moderne kryptografiske algoritmer utviklet. Ytelsen til algoritmeimplementasjonene blir deretter målt ved å måle tidsforsinkelsen til ulike mengder data på industrielle Raspberry Pi maskiner. I tillegg diskuteres synkroniseringsmetoder og trafikkekspansjon som er viktig. Resultatene viser at implementasjonene av moderne flytchiffer har bedre ytelse og gir mindre tidsforsinkelser enn tradisjonelle blokkchiffer. Implementasjonene gir svært små tidsforsinkelser, godt under 1 millisekund for mindre datamengder, og egner seg derfor svært godt for sanntidssystemer.

Nøkkelord: Anvendt Kryptografi, Kryptoanalyse, Tilbakekoblede Systemer, Nettverkskoblede Reguleringssystemer, Cyber-fysiske Systemer

Preface

The work described in this master's thesis was conducted at the Department of Engineering Cybernetics, NTNU, in Trondheim during the fall of 2020. The thesis is submitted as a requirement for the degree of Master of Technology in Engineering Cybernetics.

I would like to thank my main supervisor, Professor Thor I. Fossen, at the Department of Engineering Cybernetics, for inspiring discussions and valuable guidance during the work described in this thesis. Additionally, I would also like to thank my co-supervisor, Professor Slobodan Petrovic, at the Department of Information Security and Communication Technology, who contributed with valuable insights on the cryptographic aspects of the work. Finally, I would like to thank my colleague, Øystein Volden, at the Department of Engineering Cybernetics, who, through his work, used the proposed scheme and the algorithms that were implemented in the Robot Operating System (ROS) environment. He brought valuable insight and identified implementation-specific details that made improvements to the algorithm implementations possible.

It was quickly determined that the work described in this thesis would be a continuation of the work that was described in the TTK4550 project report, submitted in the spring semester of 2020. In addition to improving and standardizing the algorithm interfaces from the project work, additional algorithms have been included to form a comprehensive toolbox of cryptographic algorithms. This has resulted in a journal paper describing the toolbox, which is currently under peer review. Furthermore, attacks against previously proposed communication schemes used for feedback control systems that were briefly drafted in the project work have been fully described and implemented, supporting the claim that these communication schemes contain several catastrophic weaknesses. The proposed communication scheme in this thesis resists these attacks and is also robust to disturbances if attacks are attempted. This is shown towards the end of the thesis.

Petter Solnør
Trondheim, December 2020

Table of Contents

List of Tables

List of Figures

Listings

List of Abbreviations

USV	Unmanned Surface Vehicle
ROS	Robot Operating System
MSD	Mass-Spring-Damper
PI	Proportional-Integral
UDP	User Datagram Protocol
IP	Internet Protocol
DES	Data Encryption Standard
3DES	Triple DES
AES	Advanced Encryption Standard
ECB	Electronic Code Book
MD5	Message Digest 5
STM	Secure Transmission Mechanism
HMAC	Keyed-Hash Message Authentication Code
LiDAR	Light Detection and Ranging
TCP	Transmission Control Protocol
DoS	Denial of Service
CIA	Confidentiality, Integrity and Availability
COA	Ciphertext-Only Attack
KPA	Known-Plaintext Attack
CPA	Chosen-Plaintext Attack
CCA	Chosen-Ciphertext Attack
SPN	Substitution-Permutation Network
ARX	Add-Rotate-XOR
SHA	Secure Hash Algorithm
LUT	Lookup Table
NIST	National Institute of Standards and Technology
CFB	Cipher Feedback
OFB	Output Feedback
CBC	Cipher Block Chaining
CTR	Counter

PRF	Pseudo-Random Function
OTP	One-Time Pad
PRNG	Pseudorandom Number Generator
LFSR	Linear Feedback Shift Register
S-box	Substitution box
IV	Initialization Vector
SSSC	Self-Synchronizing Stream Cipher
MAC	Message Authentication Code
CBC-MAC	Cipher Block Chaining Message Authentication Code
CMAC	Cipher-based Message Authentication Code
nonce	number-used-only-once
CMA	Chosen Message Attack
CRC	Cyclic Redundancy Check
SSH	Secure Shell
E&M	Encrypt-and-MAC
MtE	MAC-then-Encrypt
TLS	Transport Layer Security
GCM	Galois/Counter Mode
EtM	Encrypt-then-MAC
CCM	Counter with CBC-MAC
OCB	Offset Codebook
RTT	Round-trip Time
IETF	Internet Engineering Task Force
RFC	Request For Comment
AES-NI	Advanced Encryption Standard New Instructions
CAESAR	Competition for Authenticated Encryption: Security, Applicability and Robustness

Chapter 1

Introduction

This master's thesis was written during the fall semester of 2020 at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology in Trondheim. This chapter is based on Chapter 1 of the project thesis from TTK4550, which has been adapted and extended to fit this thesis.

The work described in this thesis is part of an effort to enhance the cyber-physical security of feedback control systems. The main goal is to provide security against eavesdropping attacks, data manipulation attacks, and spoofing attacks. In particular, the thesis focuses on how the signals transmitted between the components of a feedback control system can be kept confidential and how the authenticity and integrity of these signals can be ensured. Of great importance is the observation that these properties should be obtained without inducing adverse effects such as large latencies and problems with synchronization between transmitters and receivers. Such effects would be detrimental to the overall performance of the feedback control system, thus rendering the scheme useless. We seek to achieve this goal through the use of modern cryptographic methods. By applying encryption, confidential signal transmission is achieved, and by applying cryptographic message authentication codes, the authenticity and integrity of the transmitted signals are ensured. However, these cryptographic methods must be applied in a cryptographically strong way to obtain the desired properties.

To address the main goal of enhancing the cyber-physical security of feedback control systems, this thesis treats the three following main topics; The first topic of this thesis is to evaluate communication schemes that have previously been presented for feedback control systems, and then propose a new, cryptographically strong communication scheme. The second topic is to present a toolbox containing high-performance implementations of state-of-the-art cryptographic algorithms that may be used to implement the proposed scheme. In the final part, we show how the proposed scheme may be implemented using the algorithms from the tool-

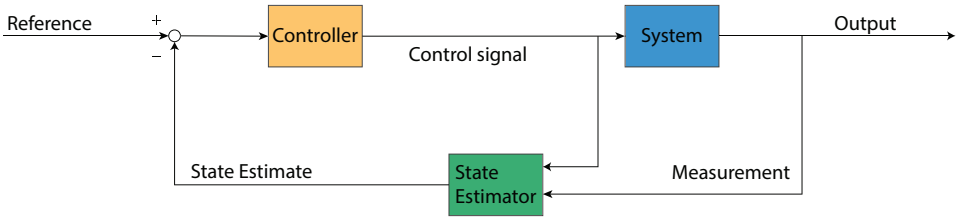

Figure 1.1: A generic feedback control system.

box, and the performance of the proposed scheme implemented with the algorithm implementations is assessed.

1.1 Background

A generic feedback control system can be seen in Figure 1.1. Feedback control systems consist of sensory systems, state estimators, controllers, and actuators. These components need to communicate by transmitting measurements, state estimates, and control inputs. Often, these components are connected over a local network spanning the plant or the vehicle, and the signals are transmitted over this network. These signals are vulnerable to cyber-physical attacks.

Some systems may have components with confidential system parameters or confidential control parameters. If an unauthorized adversary gains access to the network over which these signals are transmitted, system identification may be performed, and confidential parameters may thus be leaked. This could, for example, be an attack vector for industrial espionage. With access to the network, an adversary could also inject spoofed data to alter the behavior of a system component and thus alter the behavior of the system as a whole. Such an attack could cause a loss of control. Worse yet, combined with knowledge of the system parameters or control parameters, the adversary could effectively hijack the system. The system could then be used as part of a terrorist attack. Such attacks have been contemplated in the past. For example, in 2019, it was shown in a white paper by Kiley (2019) that an adversary with access to the transmission lines on board a small aircraft could effectively manipulate the behavior of the craft by injecting spoofed data. It was argued that small airplanes are rarely closely guarded. Thus, an adversary could gain physical access to these transmission lines without the owners or the operators noticing.

We refer to feedback control systems that 'close the loop' through a network as networked control systems, and the cyber-physical vulnerability of such systems has been investigated in the past. The former attack, in which an adversary eavesdrops and identifies confidential parameters, is called a *system identification attack* and was investigated by de Sá et al. (2017). The latter attack, in which the adversary attempts to manipulate the behavior of the system by injecting spoofed data, is called a deception attack and has been discussed in detail by Teixeira et al. (2013).

1.1.1 A motivating example

To illustrate the vulnerability of a feedback control system, a Mass-Spring-Damper (MSD) system, a Proportional-Integral (PI) controller, and an adversary were implemented (in C++). A *server* runs the PI controller while a *client* simulates the MSD system. The client measures[1] the position of the mass and transmits the measurement to the server running the PI controller. Based on the received measurement and the PI control law, the controller computes a control signal that is transmitted back to the server. All signals are transmitted over User Datagram Protocol (UDP)/Internet Protocol (IP) *unencrypted* and *unauthenticated* with static IP addresses.

The MSD system is modeled according to

$$m\ddot{x} = -kx - d\dot{x} + F \tag{1.1}$$

$$\ddot{x} = -\frac{k}{m}x - \frac{d}{m}\dot{x} + \frac{F}{m} \tag{1.2}$$

in which x denotes the position of the mass, k denotes the spring constant, d denotes the damping constant, m denotes the mass and F denotes the driving force.

The controller is modeled according to

$$F = K_p \cdot (x_r - x) + K_i \int_0^t (x_r - x)\,\mathrm{dt} \tag{1.3}$$

in which x denotes the measurement, x_r denotes the reference, and K_p and K_i are the proportional and integral gain parameters, respectively. Both the controller and the simulator run at a frequency of 50 Hz. Note that if an adversary is capable of eavesdropping on the transmitted signals, and if the system is in a non-zero, stationary state ($\ddot{x} = 0$, $\dot{x} = 0$, $x \neq 0$), the adversary can easily identify the spring constant by

$$k = \frac{F}{x} \tag{1.4}$$

During the simulation, the coefficients of the MSD system and the PI controller were set according to Tables 1.1 and 1.2, respectively. The reference signal to the controller is switched every 20 seconds between $x_r = 0$ and $x_r = 1$. Suppose the adversary now wants to force the system to the state $x = 2$. Having identified the spring constant k by eavesdropping on the transmitted signals and using (1.4), the adversary can now compute a spoofed control signal given by

[1] Since it is a simulation, we consider the exact position. No measurement noise of any kind was added.

Table 1.1: System parameters used in the hijacking experiment.

k [N/m]	d [N s/m]	m [kg]
25	20	1

Table 1.2: Controller parameters used in the hijacking experiment.

K_p	K_i
200	70

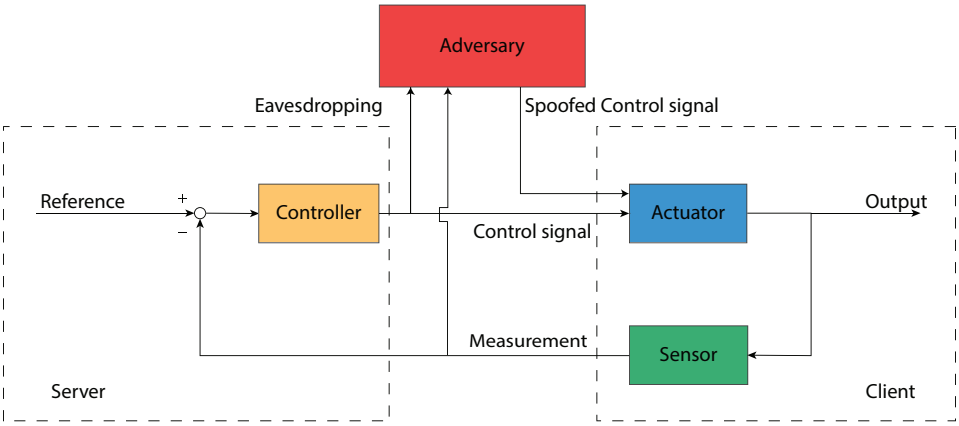

Figure 1.2: An illustration of how an adversary can launch attacks against a feedback control system.

$$\begin{aligned} F &= kx \\ &= 25\ \mathrm{N/m} \cdot 2\ \mathrm{m} = 50\ \mathrm{N} \end{aligned} \tag{1.5}$$

At some point during the simulation, the adversary initiates a deception attack by transmitting the spoofed control signal to the client at a frequency of 100 Hz, twice the frequency of the controller, to hijack the system. An illustration of the setup can be seen in Figure 1.2. The client logged the state of the simulated system for each iteration, and the state was later plotted (using MATLAB®). The successful hijacking can be seen in Figure 1.3.

To provide security against such attacks, cryptographic techniques can be used to make *system identification* difficult through *encryption* and to make *deception attacks* difficult by *authenticating the origin* of the received control and measurement signals. An illustration of a secured system can be seen in Figure 1.4.

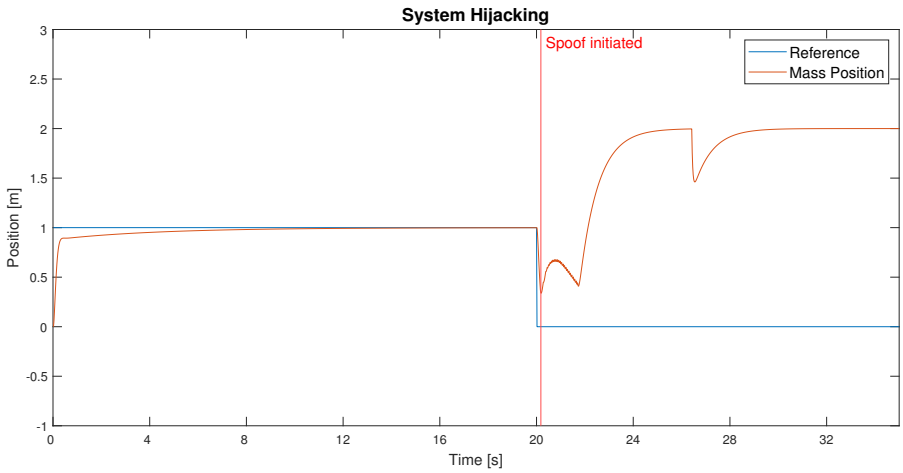

Figure 1.3: An illustration of how a mass-spring-damper system is hijacked and forced to a state determined by the adversary. Irregularities are caused by original control signals passing through to the actuator during the attack.

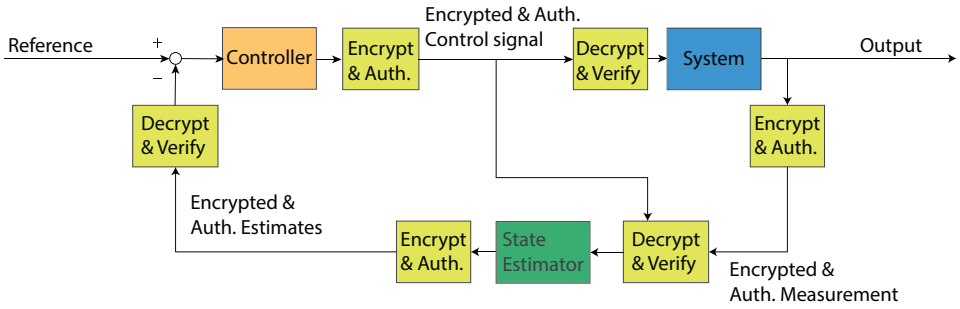

Figure 1.4: A feedback control system in which the measurement signals and control signals are encrypted and authenticated before transmission.

1.2 Related Work

In recent years, many researchers have sought to incorporate cryptographic algorithms into feedback control systems. Gupta & Chow (2008) assessed the performance of the Data Encryption Standard (DES), Triple DES (3DES), and Advanced Encryption Standard (AES) applied in the Electronic Code Book (ECB) mode in a feedback control system. Pang & Liu (2010) extended the scheme from Gupta & Chow (2008) by applying the Message Digest 5 (MD5) (Rivest 1992) cryptographic hash algorithm to provide message integrity and authenticity. The same authors (Pang et al. 2011, Pang & Liu 2012) further enhanced the scheme by including a *timestamp* to prevent replay attacks. The latter scheme was named the Secure Transmission Mechanism (STM). While several papers have questioned the secu-

rity of the STM (for example Ulz et al. (2017) and Sparrow et al. (2015)), many authors have cited the STM as a possible solution, providing confidentiality, integrity, and message authenticity for the signals that are transmitted in feedback control systems (for example de Sá et al. (2017), Yang et al. (2017), Liu (2017), Chen et al. (2017), Wu et al. (2016), Sun et al. (2017), de Sá et al. (2018), d. Sa et al. (2017), Chen et al. (2019), de Sá et al. (2019), and Yaseen & Bayart (2016)). Finally, Jithish & Sankaran (2017) proposed an alternative scheme in which the 3DES block cipher operated in ECB mode was combined with the MD5 algorithm used in conjunction with the Keyed-Hash Message Authentication Code (HMAC) (Dang 2008). Unfortunately, the security of these schemes is rarely if ever analyzed properly.

On the applications-side, Matellán et al. (2016) and Rodríguez-Lera et al. (2018) incorporated the 3DES, Blowfish, and AES encryption algorithms into the Robot Operating System (ROS) environment, a popular middleware used for robot systems. While cryptographic algorithms are available through open-source libraries such as PyCrypto (Litzenberger 2020) (used by Matellán et al. (2016) and Rodríguez-Lera et al. (2018)), OpenSSL (OpenSSL Software Foundation 2020), Crypto++ (Dai 2020), and wolfCrypt (wolfSSL Inc. 2020), these libraries may be hard to navigate and do not provide access to modern stream ciphers such as AEGIS or the stream ciphers from the eSTREAM portfolio. Therefore, researchers have used cryptographic algorithms that do not typically provide the best performance. Notably, the DES encryption algorithm is not even considered secure anymore (Biham & Shamir 1992).

Furthermore, the use of *homomorphic encryption* in control systems has been proposed, for example, by Kogiso & Fujita (2015) in which both RSA (Rivest et al. 1978) and Elgamal (Elgamal 1985) encryption is considered for which the homomorphisms hold for multiplication. The idea is that the controller partially computes the control signal by computing the multiplications required for the control signal based on *encrypted* control parameters, reference signals and measurements as seen by Figure 1.5, after which other operations (such as additions) must be performed by the plant. The motivation is that in the event that the *controller* is compromised by an adversary, only encrypted information can be extracted from the controller. Additionally, since the controller does not need to perform any decryption or encryption operations, the controller does not need to store any keys, thus simplifying the key management. While the former argument is appealing, a homomorphic encryption scheme is malleable *by design*. Without a scheme to provide message authenticity, such a scheme is entirely insecure in the face of an active adversary because the adversary will be capable of passing *meaningful*, spoofed measurements and control signals to the controller and actuator, respectively. The topic of message authenticity and integrity was *not* treated by Kogiso & Fujita (2015), nor in any of the proof-of-concept papers that followed by Kim et al. (2016) and Kogiso et al. (2018).

In their most recent paper, Teranishi et al. (2020) consider how the quantization required for the Elgamal encryption affects the stability of the system. This is im-

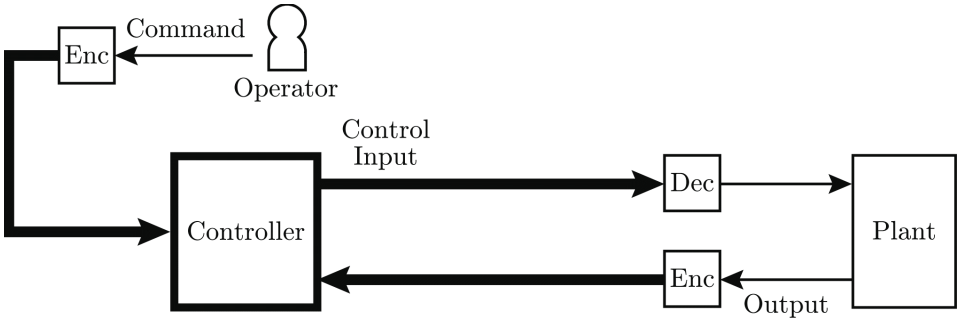

Figure 1.5: The homomorphic encryption scheme proposed by Kogiso & Fujita (2015). Image courtesy of (Kogiso & Fujita 2015).

portant in the Elgamal cryptosystem, because the plaintext space in the Elgamal cryptosystem is limited to a subset of integers, thus inducing significant quantization errors. This problem does *not* occur when symmetric encryption schemes are used. The use of the Elgamal cryptosystem also suffers from the fact that the ciphertext is twice the size of the plaintext, thus significantly increasing the amount of data that must be transmitted. This is particularly significant for systems that use vision-based signals such as camera data and Light Detection and Ranging (LiDAR) data. Not to mention that the Elgamal cryptosystem is computationally expensive compared to symmetric encryption schemes. In their paper, Teranashi et al. limits the keysize to 33 bits to ensure real-time operation, a keysize that is easily brute-forced by modern computers. Finally, the dynamic Elgamal scheme proposed requires perfect synchronization, an assumption the authors fulfill by using the Transmission Control Protocol (TCP). However, the TCP protocol is unsuitable for real-time operation.

An attempt to incorporate homomorphic authenticated encryption in a drone system was done by Cheon et al. (2018). In theory, this avoids the malleability of previously proposed homomorphic encryption schemes. However, the encryption algorithm proposed has not been critically scrutinized by the cryptographic community, and the security proofs are perceived as questionable.

1.3 Previous Work

The work described in this thesis is a continuation of the work documented in the project report submitted as part of the course TTK4550 in the spring of 2020. In particular, Chapter 2 and Chapter 3 are based on Chapters 2 and 3 from the project report, respectively, with certain changes and additions. Certain ideas from Chapter 4 were briefly touched upon in the project report. Also, certain segments in Chapter 5 are based on the project report. If a segment is based on previous work, this will be specified at the introduction of each segment. Otherwise, the work was completed as part of this thesis.

1.4 Problem Definition

As illustrated by the motivating example in Section 1.1.1, feedback control systems are vulnerable to cyber-physical attacks in which an adversary can perform system identification and disclose potentially confidential information regarding a system or a controller by eavesdropping on the communication between sensors, estimators, controllers, and actuators. Worse yet, if parameters are known, the system can be hijacked by passing spoofed control signals to the actuator of the system or by passing spoofed measurements to the controller.

To prevent such attacks, it is desirable to ensure that the confidentiality, integrity, and authenticity of the transmitted signals are ensured. This may be accomplished through the use of cryptographic tools. However, cryptographically strong algorithms must be used in a cryptographically strong construction. The schemes proposed by the authors in Section 1.2 are questionable, and the security of several of these schemes will be investigated in Sections 4.1.1 and 4.1.2.

While the strategies involving homomorphic encryption are intriguing, the motivation for such a scheme is debatable and, at the very least, reserved for a small subset of feedback control systems. Indeed, for most feedback control systems, the controller would be considered *secure*. Furthermore, the majority of the proposed homomorphic encryption schemes rely on *asymmetric* encryption techniques such as the Elgamal cryptosystem. For such schemes, problems related to the quantization of signals and parameters arise when dealing with these cryptosystems, as they only deal with subsets of integers and may not be used on the data directly. This is not the case for symmetric encryption schemes. Additionally, with the rise of quantum computing, the future security of such systems is debatable due to Shor's Algorithm (Shor 1994), whereas symmetric encryption schemes would only require a doubling of the current key size due to Grover's Algorithm (Grover 1996). Thus, for a system with a long time horizon, the use of symmetric cryptography is likely to be the most secure option.

Finally, we note that many encryption schemes are *stateful*. This is the case both for symmetric encryption schemes such as stream ciphers and for the homomorphic encryption scheme proposed by Teranishi et al. (2020). This requires both the transmitter and the receiver to be synchronized. In (Teranishi et al. 2020), the synchronization problem was solved by using the TCP protocol. This is unfortunate because the TCP protocol can induce significant latencies if packets are lost and completely break down if an active adversary filters out certain packets. Preferably, a different synchronization mechanism not requiring the reception of all the previous messages in the correct order should be used.

As we proceed in this thesis, we seek to answer the following research questions.

RQ1 How do 'secure' communication schemes for feedback control systems proposed by researchers from the control community fare against cyber-physical attacks?

RQ2 May cryptographic techniques be used to enhance the security of feedback

control systems through a new, more secure communication scheme?

RQ3 How may the transmitter and receiver of a feedback control system achieve synchronous communication if stateful cryptographic methods are used over non-reliable communication protocols?

RQ4 To what extent do open-source cryptographic libraries provide access to modern cryptographic algorithms, and how does their performance compare to direct implementations of the algorithms?

RQ5 Which cryptographic algorithms provide the best performance, and should be used to obtain authenticated encryption, in feedback control systems?

1.5 Main Contributions

The following are considered the main contributions of the work described in this thesis:

- Previously proposed schemes that were designed to provide confidential and authenticated transmission of signals in feedback control systems have been shown to fail catastrophically against known-plaintext attacks.
- A new cryptographically strong transmission mechanism for feedback control systems has been proposed.
- A toolbox containing multiple high-performance cryptographic algorithms for feedback control systems has been developed. The toolbox contains both portable software implementations and implementations that take advantage of enhanced instruction sets on `x86` and `ARMv8` processors.
- Modern, high-performance cryptographic algorithms are shown to induce little latency when used to protect typical signals transmitted in feedback control systems on embedded devices.
- The work has resulted in a journal paper currently under peer review.

1.6 Organization of the Thesis

As we proceed, the thesis is organized as follows.

In Chapter 2, terms and concepts from cryptology are introduced and defined. In addition to providing a brief overview of key concepts from cryptology, the research question regarding synchronization between the transmitter and the receiver (**RQ3**) is touched upon in this chapter.

In Chapter 3, the hardware and the software of the encryption laboratory are described. The encryption laboratory was used in the motivating example in Section 1.1.1, to demonstrate attacks in Section 4.1.1 and Section 4.1.2, and in the benchmarks and verification experiments of the algorithms in Chapter 7.

In Chapter 4, the security of previously proposed transmission mechanisms for feedback control systems is analyzed, treating **RQ1**. Furthermore, a new scheme is proposed, treating **RQ2**.

In Chapter 5, the CryptoToolbox is introduced. The CryptoToolbox is a toolbox containing new implementations of a selection of modern high-performance cryptographic methods.

In Chapter 6, code examples of how the proposed scheme may be implemented using the CryptoToolbox algorithms is given, to aid practitioners implementing the proposed scheme.

In Chapter 7, the cryptographic algorithms from the CryptoToolbox, described in Chapter 5, are benchmarked in the scheme proposed in Section 4.2. It is also demonstrated that the scheme proposed in Section 4.2 is robust against disturbances, thus providing some security against Denial of Service (DoS) attacks. This chapter focuses on answering **RQ4** and **RQ5**.

Finally, Chapter 8 concludes the thesis by summarizing the findings and relating the findings to the research questions posed in Section 1.4.

All figures have been created using Adobe Illustrator®unless stated otherwise, while all plots have been created using MATLAB®. If a figure is based on an existing figure, the source is cited in the figure description.

Chapter 2

Cryptographic Methods

The goal of this chapter is to give a brief theoretical introduction to the cryptographic methods that are used in Chapters 4 – 7. This chapter is based on Chapter 2 from the TTK4550 project report, with some additions and changes.

There exist different cryptographic primitives designed to achieve different security goals. In the literature, these security goals are often summarized as Confidentiality, Integrity and Availability (CIA), as illustrated by Figure 2.1. In this chapter, these terms will be elaborated upon, and the cryptographic primitives set to achieve them are explained. In the following chapter, a notion known as *Kerckhoff's Principle* is important. Kerckhoff's Principle states that *the security of a cryptosystem should only rely on the secrecy of the key, not on the secrecy of other parts of the system.* In particular, this statement is orthogonal to that of *security through obscurity.* That is, the security of the system should *not* rely on keeping the inner workings of the system secret. Thus, in the following discussions, is it assumed that an adversary has full knowledge of the algorithms that are used, and *only the key, and material that is directly derived from the key, are unknown to an adversary.* Examples of material that is directly derived from the key and considered *secret* are, for instance, the *key schedule* of a block cipher or the *state* of a stream cipher. These concepts will be described in a later section.

2.1 Algebra

When treating the cryptographic methods in the following sections, certain mathematical constructs and properties are of particular interest and importance. The goal of this section is to define and get acquainted with the notion of a *finite field.* For a more detailed treatment on this topic, the reader is referred to a book by McEliece (1986).

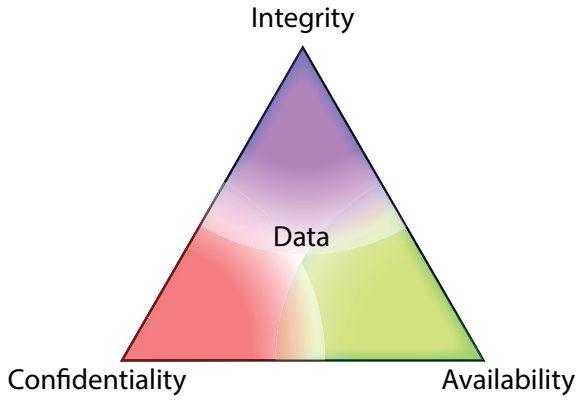

Figure 2.1: The core concepts of information security.

To define a *field*, an algebraic *group* must be defined. We adopt the definition from Menezes et al. (1996).

Definition 1 (Group)**.** A *group* $(G, *)$ consist of a set G with a binary operation $*$ on G satisfying the following three axioms.

1. The group operation is *associative*. That is, $a*(b*c) = (a*b)*c \; \forall \; a, b, c \in G$.
2. There is an element $1 \in G$, called the *identity element*, such that $a * 1 = 1 * a = a \; \forall \; a \in G$.
3. For each $a \in G$ there exists an element $a^{-1} \in G$, called the *inverse* of a, such that $a * a^{-1} = a^{-1} * a = 1$.

Additionally, a group is *abelian*[1] if the following criteria is satisfied.

4. $a * b = b * a \; \forall \; a, b \in G$.

Further, if there exists an element $g \in G$ such that for each $b \in G \; \exists \; c \in \mathbb{Z}^+ \mid g^c = b$ we call the group *cyclic*, and g a *generator*. Using the definition of an abelian group, defining a *field* is easy.

Definition 2 (Field)**.** A *field* $(F, +, \cdot)$ consists of a set F with two binary operators, $+$ and $\cdot$ called *addition* and *multiplication*, respectively, such that the following three properties are satisfied.

1. $(F, +)$ is an abelian group where 0 is the identity element.
2. $(F \setminus \{0\}, \cdot)$ is an abelian group where 1 is the identity element.
3. Multiplication distributes over addition, i.e., $a \cdot (b + c) = a \cdot b + a \cdot c$.

Finally, a *finite field* can be defined.

[1] Named after Niels Henrik Abel, a Norwegian mathematician.

Definition 3 (Finite field)**.** A *finite field* is a field $(F, +, \cdot)$ in which the set F contains a finite number of elements.

The number of elements in the set of a finite field is *always* p^n in which p is a prime, and n is a strictly positive integer. Furthermore, *there only exists one field* with p^n elements. The finite field with p^n elements is said to be of order p^n and to be of *characteristic p*. In the digital era, the finite fields of characteristic 2 are particularly interesting. In the literature, a finite field is often referred to as a *Galois field*[2], and the finite field with p^n elements is often referred to as $GF(p^n)$. This notation will be adopted in this thesis.

Lemma 1. *The multiplicative group of a finite field,* $(F \setminus \{0\}, \cdot)$*, is cyclic.*

Lemma 1 ensures that there always exists at least one generator in the multiplicative group of the field. This element is called a *primitive element*.

The elements of a finite field $GF(p^n)$ may be represented by *polynomials* of degree strictly less than n with coefficients from the set $\{0, ..., p-1\}$. If the polynomial $a(x)$ representing an element in the finite field may be factored into two polynomials $b(x) \neq 1, c(x) \neq 1$, that is, $a(x) = b(x)c(x)$ then $a(x)$ is called a *reducible polynomial*. If the polynomial $a(x)$ may *not* be factored, then $a(x)$ is called an *irreducible polynomial*. The order of the polynomial $a(x)$ is denoted by the smallest integer e such that $a(x)$ divides $x^e - 1$. Finally, if the polynomial $a(x)$ is irreducible *and* has order $p^n - 1$ we say that $a(x)$ is a *primitive polynomial*.

Given a primitive polynomial $\alpha(x)$ and an irreducible polynomial $\beta(x)$ any element of the finite field $GF(p^n)$ may be represented by the polynomial given by $\alpha(x)^c \bmod \beta(x)$ for some c $\in \{0, \ldots, p^n - 1\}$. Thus, the irreducible polynomials play the role of prime numbers[3] and the primitive polynomials play the role of the generators. Note that the choice of the primitive polynomial and the irreducible polynomial merely determines the order of the elements that are generated.

2.2 Confidentiality

The task of keeping information private and inaccessible to unathorized eavesdroppers is known as *confidentiality*, and as such, this private information is often referred to as *confidential* information. Cryptographic confidentiality is achieved through the use of *encryption*.

The goal of an encryption scheme is to ensure that confidential data can be transmitted securely through an insecure channel, as illustrated by Figure 2.2. An encryption scheme can be *symmetric* or *asymmetric*. In a symmetric, or *secret key*, encryption scheme, the encryption and decryption keys are easily deduced from one another and may thus be considered *the same*. As illustrated by Figure 2.2, a

[2] Named after Évariste Galois, a French mathematician.

[3] While a practical analogy in this context, one ought to be careful. It is, in fact, quite easy to factor a reducible polynomial or to verify that a polynomial is irreducible. Factoring integers or verifying that an integer is a prime can be quite cumbersome!

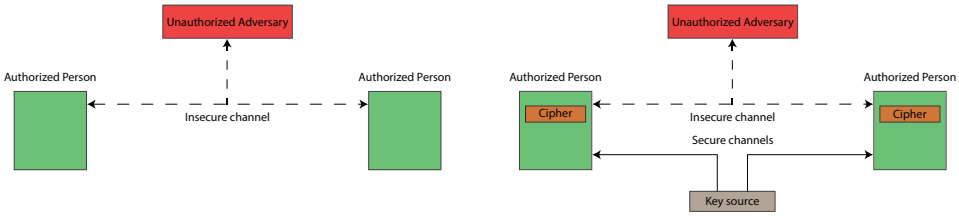

Figure 2.2: Two actors seeking to communicate over an insecure channel with and without encryption.

symmetric encryption scheme requires a *secure key distribution system* as well as a *key generator* that is capable of generating highly randomized keys. This is a problem of its own and is not treated in this report. That is, *it is assumed that the keys that are used are, in fact, highly randomized and pre-distributed.*

Asymmetric encryption schemes also exist, in which the decryption key is *not* easy to deduce from the encryption key. These schemes typically rely on number-theoretical problems that are *assumed* to be *computationally hard*. Asymmetric encryption schemes are typically used to *exchange a secret key*, for example, using RSA, or to *deduce a common secret from which a secret key can be derived*, for example, using the *Diffie-Hellman key exchange*[4]. This is because asymmetric encryption schemes are *computationally expensive* compared to symmetric encryption schemes, and for this reason, they will not be treated in this report.

Ultimately an adversary may have two goals in mind; uncovering the plaintext of a particular ciphertext or recovering the key used in the encryption scheme. Thus it is important that information from the plaintext, nor the key, leaks through to the ciphertext. The terms confusion and diffusion were used by Shannon (1945) to describe these goals. Specifically, the confusion component describes the goal of making the relationship between the ciphertext and the key as complicated as possible, while the diffusion component describes the goal of making the relationship between the plaintext and the ciphertext as complicated as possible. The confusion component is often achieved through the use of nonlinear operations, for example, multiplications, while the diffusion component is often achieved through rearrangements to spread and hide any plaintext redundancy.

2.2.1 Attack models

When discussing cryptographic algorithms, it is useful to keep *attack models* in mind, as certain types of algorithms grant more power to the adversary. The goal of a particular cryptographic design is to ensure that any *reasonable attack* against the particular design is *no more efficient than an exhaustive search*. That is, if a symmetric key cipher system claims to provide n-bit security, the most efficient attack should require an exhaustive search over the keyspace of size 2^n. In

[4]Named after Whitfield Diffie and Martin Hellman, whom with their paper (Diffie & Hellman 1976) practically invented asymmetric cryptography.

computational complexity terms, an exhaustive search is, on average, of complexity 2^{n-1}.[5]

Ciphertext-only attack

The most basic attack an adversary can perform is a Ciphertext-Only Attack (COA). In a COA, the adversary only has access to the transmitted ciphertext. While being the most basic attack model, an adversary will always be capable of performing a COA attack. If a COA attack is not feasible, the messages are presumably transmitted through a secure channel, at which point encryption is pointless.

Known-plaintext attack

A slightly more advanced attack is a Known-Plaintext Attack (KPA). In a KPA, the adversary knows some plaintext-ciphertext pairs.

Chosen-plaintext attack

Whereas the plaintext-ciphertext pairs available to the adversary were arbitrary in the KPA-model, the adversary is capable of controlling the plaintext in a Chosen-Plaintext Attack (CPA) while observing the resulting ciphertext.

Chosen-ciphertext attack

Finally, in a Chosen-Ciphertext Attack (CCA) attack, the adversary is capable of controlling the ciphertext while observing the corresponding plaintext. In practical terms, the CCA-model corresponds to a scenario in which the adversary is in possession of a decryption device.

Other models

In addition to the attack models mentioned above, there exists a whole host of other attack models. An exhaustive list will not be given here. However, attack models such as *related-key attacks* and *side-channel attacks* are all important to consider, depending on the application.

2.2.2 Block ciphers

In modern cryptography, block ciphers have been dominating. Not only do they provide confidentiality through encryption, but they have also served as the backbone of cryptographic hash functions and message authentication algorithms. Modern block ciphers were introduced with the cipher Lucifer designed by Horst Feistel in 1971 (Feistel 1971). This was followed by the standardization of the DES in the

[5] On average, one would have to attempt one half of the total number of keys before finding the correct key.

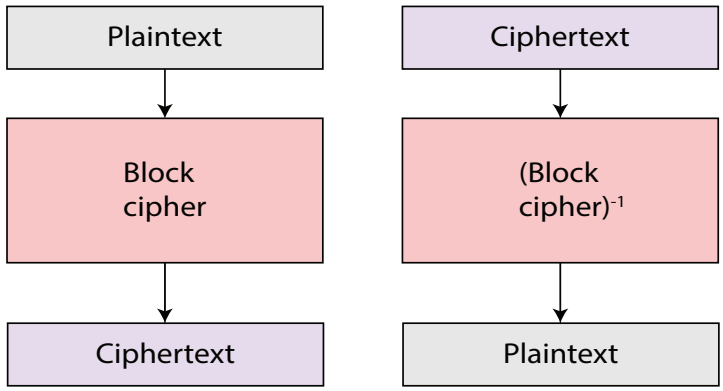

Figure 2.3: A high-level illustration of how a block cipher acts as a substitution.

mid-1970s (National Bureau of Standards 1977) and the AES process around the year 2000, resulting in the AES cipher (NIST 2001).

Block ciphers are permutations $F : \{0,1\}^B \times \{0,1\}^K \mapsto \{0,1\}^B$ where K denotes the key size, and B denotes the block size. The block cipher takes a B-bit input, performs a series of operations, and outputs a B-bit output. Externally the block cipher can be perceived as a black box, acting merely as a substitution as illustrated by Figure 2.3. Thus, for a block size B with a total of 2^B distinct blocks, there exists a total of $2^B!$ block ciphers. For a specific block cipher design, the key acts as a parameterization, thus limiting the number of block ciphers to 2^K. The job of constructing a block cipher is therefore equivalent to the selection of 2^K substitutions from a total of $2^B!$ available substitutions. As B becomes large, the 2^K substitutions of a block cipher are dwarfed by the total number of $2^B!$ possible substitutions.

Modern block ciphers have largely followed one of three design methodologies, all of which consist of multiple, iterated rounds of nonlinear operations and permutations, a design known as a *product cipher*. By iterating through multiple rounds, increased diffusion and confusion are achieved, effectively increasing the effort required to perform certain statistical attacks such as linear cryptanalysis and differential cryptanalysis. Note that each round must be keyed. Otherwise, an adversary could backtrack through the cipher since the block cipher is a permutation. Thus, block ciphers usually contain a *key schedule* that is designed to extract multiple *round keys* from the secret *master key*. Having a complex relationship between the round keys is necessary. Otherwise, the cipher is vulnerable to cryptanalytical attacks such as *slide attacks*, as demonstrated by Biryukov & Wagner (1999). The round keys may be computed during the encryption and decryption procedures, or they may be precomputed and stored. The latter is usually preferred, except on severely constrained devices, or on certain devices with hardware acceleration support, in which the former is used.

Internally. a block cipher may be constructed in various ways.

Feistel cipher The Feistel cipher design methodology is motivated by the original Lucifer cipher invented by Horst Feistel, and is used in DES. A Feistel cipher is characterized by splitting the input of each round in two, performing a keyed nonlinear operation on one half, and then $\oplus$-ing the output with the other half. The two halves are then swapped for the next round. In this fashion, at least two rounds are required for the entire block to be affected by the nonlinear operation. A beautiful aspect of the Feistel cipher is the observation that decryption merely consists of the same operations with a reversed key schedule.

Substitution-permutation networks The Substitution-Permutation Network (SPN) design methodology seeks to separate Shannon's concepts of confusion and diffusion (Shannon 1945) in distinct layers and optimize the two separately. This design methodology is used in the *Rijndael* cipher, which was later adopted as the AES. The cipher that came in second in the AES-process, Serpent, is also an SPN-cipher.

Add-rotate-XOR Finally, the Add-Rotate-XOR (ARX) design methodology has been used in ciphers such as the SPECK cipher and the block cipher used in the Secure Hash Algorithm (SHA)-2. The operations of the aforementioned Feistel and SPN designs can be rather complex, for example, involving substitution boxes requiring multiplications over some finite field $GF(2^n)$, and permutations achieved through matrix multiplications. These operations are *computationally expensive*, especially in simple architectures in which multiplications are solved using repeated additions. On the other hand, pre-computing these operations and storing them in Lookup Tables (LUTs) may leave the cipher vulnerable to timing attacks[6]. The motivation of the ARX design methodology is to enhance performance and resistance against timing attacks by only resorting to operations that are *efficient in hardware*, for example, by performing additions over some field $GF(2^w)$ where w is the *word size* of the architecture, rotations of words, and bitwise addition over $GF(2^w)$, that is, the $\oplus$-operation. The ARX design methodology has also been used in the design of stream ciphers, such as the popular ChaCha stream cipher. Because neither the confusion nor the diffusion component of a round in an ARX cipher can be optimized with such operations, ARX designs are often characterized by a high number of rounds compared to SPN and Feistel designs.

An illustration of the round function of a Feistel cipher, an SPN cipher and an ARX cipher can be seen in Figure 2.4.

Modes of operation

Operating a block cipher directly is known as the ECB mode and provides a form of *stateless* encryption in the sense that encrypting the same plaintext always results in the same ciphertext. This is unfortunate because information about the plaintext

[6] If the LUT does not fit in the CPU cache, this can result in varying processing time that can be detected. Timing attacks are an example of side-channel attacks. Notice that it is the *cipher implementation* that is attacked, not the cipher itself.

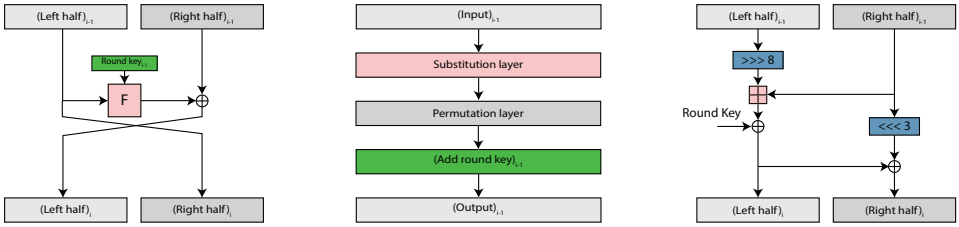

Figure 2.4: An illustration of a typical Feistel cipher round on the left, a typical substitution-permutation network round in the middle, and a typical add-rotate-xor round on the right. The add-rotate-xor round is based on the structure of the SPECK cipher (Beaulieu et al. 2015).

is leaking through to the ciphertext. To prevent information about the plaintext from leaking through to the ciphertext, block ciphers may be operated in other *modes of operation* that provide a randomization element.

Originally the National Institute of Standards and Technology (NIST) certified four randomized modes of operation (Dworkin 2001); Cipher Feedback (CFB), Output Feedback (OFB), Cipher Block Chaining (CBC), and Counter (CTR) mode. While the details of each of the modes are out of scope for this thesis, the general idea is that a block cipher by design functions as a Pseudo-Random Function (PRF). Thus, by ensuring that the input does not entirely depend on the plaintext, the output is randomized. While a block cipher operating in CBC mode is still considered a block cipher by most, the CFB, OFB, and CTR modes effectively convert the block cipher to a *stream cipher* by introducing a *state*. Furthermore, the ECB and CBC modes of operation require that the input is a multiple of the block size B. This is ensured by using padding schemes, resulting in the transmission of larger amounts of data. This increase can be significant, especially if each message is small.

In addition to the original modes of operation proposed, additional modes of operation that provide a service known as authenticated encryption exists. These modes of operation provide both message authenticity and confidentiality. These will briefly be discussed in Section 2.4.

2.2.3 Stream ciphers

Historically, stream ciphers have been the cornerstone of encryption. In particular, historical ciphers such as those obtained from the rotary-machines during the second world war, for example, the German *Enigma* machine, were stream ciphers. Stream ciphers are often referred to as *stateful* ciphers and consist of a *state transition function* and an *output function*. Depending on whether the state transition function takes the previous state or the previous ciphertexts as argument, the stream cipher is referred to as a *synchronous* or a *self-synchronizing* stream cipher.

The motivation of modern stream ciphers is to *approximate* the One-Time Pad

(OTP). The OTP is a stream cipher in which a *truly random string* of the same length as the message that is to be encrypted serves as the *key* and is only used once, hence the name. The Vernam Cipher, patented by Gilbert Vernam in 1919, is an example of the OTP (Vernam 1919). In the Vernam Cipher, the key is merely added to the plaintext to encrypt. While Gilbert Vernam could not prove that his cipher was, in fact, *unbreakable*, this was later proven by Shannon (1949) using information theory, by showing that the *ciphertext* in such a scheme is *statistically independent* of the plaintext, thus obtaining what Shannon referred to as *perfect secrecy*[7].

Unfortunately, as described, the OTP requires a *truly random key with equal or greater length than the message that is to be encrypted.* This is problematic because the generation of truly random numbers is a difficult problem on its own, not to mention if a high bandwidth is required, for example, if many bits must be generated in a short amount of time. Furthermore, these keys must be pre-distributed *in advance* from the source of the key to the encryption and decryption devices. If large amounts of data are to be transmitted, this results in the transfer of equally large amounts of keying material, thus becoming somewhat of a logistical nightmare. The key distribution could, for example, be solved with a trusted courier, but with the number of devices that are connected in modern telecommunications, this is not a feasible option.

It is for these reasons that the OTP has largely been reserved for a select few communication channels in which only the highest possible level of security has been required. Rumors have it that the *red line* between Moscow and Washington, D.C. used the OTP during the Cold War Era. However, the analytical results obtained for the OTP make it an attractive target to *approximate.* In particular, if one could *extend* a short truly random string to a *much longer* pseudo-random string using a *deterministic algorithm*, the OTP could be approximated.

While a novice might be tempted to extend the truly random string to a pseudo-random string by *repeating* the truly random string (the 'N-Time Pad'), this results in a catastrophic failure, as shown by *Kasiskis Attack.* Suppose two ciphertexts c_1, c_2 are encrypted using *the same* key k ('Two-Time Pad'). Then an adversary is capable of obtaining a relationship between the two ciphertexts and the corresponding plaintexts p_1, p_2 *by eliminating the key from the equation.* Thus, if at some point the adversary gains knowledge of one of the plaintexts, for example, p_1 in a KPA as described in Section 2.2.1, the second plaintext p_2 is easily recovered as shown by

[7] It is emphasized that even an adversary with infinite time and resources will not be capable of breaking the OTP provided that the key is truly random.

$$\begin{aligned} c_1 &= p_1 \oplus k \\ c_2 &= p_2 \oplus k \\ c_1 \oplus c_2 &= p_1 \oplus k \oplus k \oplus p_2 = p_1 \oplus p_2 \\ p_2 &= c_1 \oplus c_2 \oplus p_1 \end{aligned} \tag{2.1}$$

The key itself is easily recovered, and any message encrypted with the same key is therefore compromised, leaving the cipher completely broken. If the adversary does *not* have access to any plaintext, the adversary may resort to a COA attack known as the *crib-dragging* attack. The idea of the crib-dragging attack is that the adversary 'guesses' one plaintext encrypted with the key, for example, by going through common phrases, and if the *other* plaintext also yields a meaningful result with the guessed plaintext, then it is assumed that the guess is correct. This is much more efficient than a brute force attack because plaintext is not at all random.

Thus, it should be clear that a better way of expanding a truly random string to a much longer pseudo-random string is required. It is noted that if the stream cipher uses a pseudo-random string expanded from some truly random string to perform encryption rather than the truly random string itself, as was the case in the OTP), then the truly random string is referred to as the *key* while the expanded, pseudo-random string is referred to as the *keystream*, sometimes called the *running key*.

The goal of a modern stream cipher is, therefore, to *efficiently* expand a truly random string to a pseudo-random string such that deducing the key from the keystream is *hard* while ensuring that the probability of ever repeating the keystream is negligible[8].

Synchronous stream ciphers

Most stream ciphers are what we refer to as synchronous stream ciphers. A synchronous stream cipher may be described by

$$\sigma_{i+1} = g(\sigma_i) \tag{2.2}$$

$$z_i = h(\sigma_i) \tag{2.3}$$

in which (2.2) describes the state transition function and (2.3) describes the output function. Encryption and decryption then consist of combining the keystream with the plaintext and ciphertext, respectively. In digital communications, this is predominantly achieved by addition over GF(2), that is, the exclusive-or operator ($\oplus$). Because addition over GF(2) is an involution, decryption merely consists of adding the same keystream to the ciphertext.

[8] Thus, providing security against attacks such as Kasiski's attack or the crib-dragging attack.

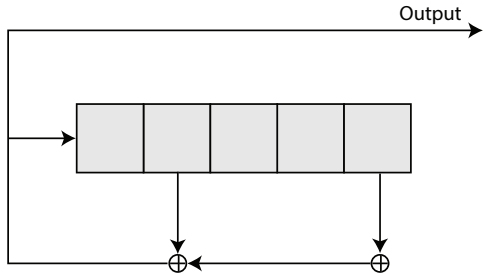

Figure 2.5: An illustration of a linear feedback shift register with a *reducible* feedback polynomial $p(x) = 1 + x^2 + x^4 + x^5 = (x + 1)(x^4 + x + 1)$. Such a polynomial does *not* produce a max-length sequence of period $2^5 - 1$. A sequence produced by a linear feedback shift register with a reducible feedback polynomial also carries other statistical properties that are undesirable.

The development of a synchronous stream cipher is then reduced to the development of a Pseudorandom Number Generator (PRNG) that can expand the key, for example, the initial state, and an output function with *desirable properties*.

To prevent Kasiski's attack and the crib-dragging attack, it is paramount that one can ensure that the period of the PRNG is at least as long as the message that is to be encrypted. In practice, the period of a cryptographically strong PRNG is *much longer* on the scale of 10^{50} bits. Furthermore, it is observed that in the event of a KPA, the adversary has access to both the plaintext and the ciphertext, and therefore access to parts of the keystream. Thus, it must be infeasible for an adversary to recover the internal state of the PRNG given its output. By recovering the internal state of the PRNG, the adversary with knowledge of the cipher will be capable of producing all future keystream bits, thereby breaking the cipher. Hence the relationship between the internal state and the output keystream must be highly nonlinear.

Period The construction of PRNGs with long periods has received much attention and is rich in theoretical results. Much of the theory was developed in the 1980s with figures such as *James L. Massey*, *Jovan Golic*, and *Rainer Rueppel*. Their work was centered around the Linear Feedback Shift Registers (LFSRs), and a thorough treatment on LFSR-based stream ciphers is given in a book by Rueppel (1986). An LFSR is illustrated in Figure 2.5. For a specific LFSR, it is possible to associate with it a *feedback polynomial*, and it was shown that *the properties of the LFSR depend on the properties of the feedback polynomial*. In particular, it was shown that an LFSR with a *primitive feedback polynomial* would generate a so-called *m-sequence*, which is a sequence of maximal length. Thus, if the state was held in an n-bit register, the sequence produced by the LFSR would have a period of $2^n - 1$.[9]

[9]The all-zero state is degenerate.

While some modern stream ciphers such as *SNOW*, SOSEMANUK, and *Grain* still rely on the theory of LFSRs, other design approaches also exist. In particular, design approaches more akin to those of *block ciphers* have emerged because they are easily translated to efficient software implementations. Furthermore, the RC4 stream cipher originally invented by Ron Rivest consists of a dynamic substitution table in which each entry consists of a byte. Following the initialization, the RC4 cipher is remarkably simple, in which each round consists of updating one byte of the table while outputting another byte of the table, thus producing one byte of keystream each iteration.

While weaknesses in the RC4 stream cipher were discovered, and the cipher is now regarded as *insecure*, the design approach of large substitution tables is still very relevant. In particular, this design approach is used in the modern software-oriented stream cipher HC-128 designed by Hongjun Wu. Because the HC-128 cipher operates on 32-bit words rather than 8-bit bytes, the performance of the HC-128 stream cipher is also significantly better than the RC4 cipher on architectures with a word-size greater than or equal to 32-bits, which includes most modern architectures. While the PRNGs of such stream ciphers *do not* produce m-sequences, an estimate of the periods of the cycles produced can be obtained.

The state transition function of the HC-128 cipher is *invertible*, i.e., a permutation. If it is also assumed that the state transition function is *random*, one can apply theory from iterated permutations stating that the *expected cycle length of a random iterated permutation* is given by

$$\frac{n+1}{2} \tag{2.4}$$

in which n is the total number of elements, that is, states[10]. Now, for HC-128 the state, that is, the substitution tables, consists of

$$2 \cdot 512 \cdot 32 = 32778$$

bits. Thus, there are 2^{32778} total states, that is, by (2.4), the expected cycle length in this scenario is

$$\frac{2^{32778}-1}{2} \approx 2^{32777}$$

Now of course, the *expected* cycle length is *not* a lower bound. Furthermore, the state transition is *not* a random function. However, this goes to illustrate how astronomically large the periods *can* be for such ciphers. Concerning HC-128, the author did not provide such figures but stated that the *average* period of

[10]Source here: http://pnp.mathematik.uni-stuttgart.de/iadm/Riedel/papers/RPS.pdf page 12. (Accessed May 23rd)

the keystream is expected to be *much more than* 2^{256}. Of course, none of this guarantees that you won't find a very short period for certain initial states.

Nonlinearity Because the determination of the internal state of the PRNG results in a complete break of the cipher, the use of, for example, an LFSR directly to obtain a long period is cryptographically weak. The theory of LFSRs encapsulates the difficulty in retrieving the internal state given the output in a term known as *linear complexity*. The linear complexity of a sequence is defined as *the shortest LFSR that is required to produce a given sequence*. In particular, for an LFSR, the maximum linear complexity $\mathcal{L}$ is equal to the length of the register holding the internal state. Because of the linear relationship between the internal state and the output, algorithms such as the *Massey-Berlekamp*-algorithm can synthesize an LFSR and an internal state producing the same output sequence given $2\mathcal{L}$ bits of output. Thus, if an LFSR were to be used directly, an infeasibly large register would be required.

Therefore, rather than outputting the *state* directly, it is passed through a nonlinear function. This can be accomplished in various ways, for example, by nonlinearly combining the output from multiple PRNGs such as LFSRs, by using the output sequence from one PRNG to decimate the output sequence of another PRNG, or by a nonlinear filter in which multiple components of the PRNG state is combined in a nonlinear fashion. As an example, the SOSEMANUK cipher, which keeps the state in an LFSR, uses a nonlinear filter, which is constructed using certain Substitution Boxes (S-boxes) from the Serpent block cipher. On the other hand, the ciphers consisting of large substitution tables usually have very simple output functions. For example, HC-128 has a very simple nonlinear output function consisting of addition over $GF(2^{32})$[11].

Initialization vectors Modern synchronous stream ciphers usually permit the use of a public Initialization Vector (IV) as a parameter to an initialization phase. The initialization phase is designed to deduce an initial state of the keystream generator from the secret key and the public IV. This process is often referred to as a *key mixing phase* and effectively provides a synchronization mechanism. An illustration may be seen in Figure 2.6. Because the IV is a *public* parameter that an adversary potentially could manipulate, this opens a new attack vector in which an adversary can attempt to recover the key by passing IVs to initialize the cipher and observing the resulting output. Therefore, the key mixing phase must also be a highly nonlinear operation. Practically, this is solved in multiple ways, for example, by passing the IV through a block cipher in which the key serves as the key to the block cipher. This is done in the SOSEMANUK cipher.

[11] This might seem weird, but while addition over, for example, the real numbers is a linear operation, it is *not* a linear operation in $GF(2^n)$ for $n \geq 2$.

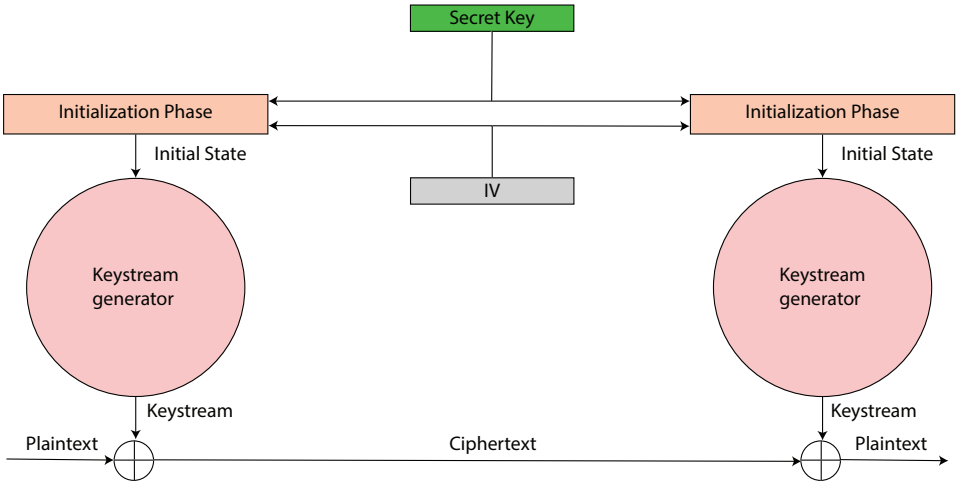

Figure 2.6: An illustration of a synchronous stream cipher permitting the use of an initialization vector.

Self-synchronizing stream ciphers

At first sight, a Self-Synchronizing Stream Cipher (SSSC) is seemingly quite similar to a synchronous stream cipher. However, as we shall see, there are profound differences between the two. Like a synchronous stream cipher, an SSSC can be described by a state transition function and an output function as seen by

$$\sigma_{i+1} = g_k(c_i, \ldots, c_{i-L}) \tag{2.5}$$
$$z_i = h_k(\sigma_i) \tag{2.6}$$

where σ_{i+1} denotes the next state, and z_i denotes the keystream output. Notice that while the state transition function of a synchronous stream cipher (2.2) is uniquely determined by the previous state, the state transition function of an SSSC (2.5) is determined by the L previous ciphertext bits, in which L is a finite number. Because of this, the SSSC can *resynchronize* in the event of a synchronization loss. It also means that the SSSC can jump right into an ongoing transmission and achieve correct decryption after L bits have been correctly received. Also, both the state transition function (2.5) and the output function (2.6) *may* be keyed in the SSSC. In any event, at least one of the two must be keyed. Otherwise, any adversary in possession of the device could eavesdrop merely by turning on the device.

While the SSSC does achieve resynchronization easily, the cryptographic properties of the SSSC have become fundamentally different from that of the synchronous stream cipher. Because the ciphertext now enters the state of the cipher, an adversary in possession of a device is capable of launching powerful CCAs such as

differential cryptanalysis, commonly used to analyze block ciphers. This seemingly minor difference has led many cryptographers to argue that the SSSCs are more closely related to the block ciphers than the synchronous stream ciphers.

Throughout recent history, attempts of modifying existing synchronous stream ciphers to construct SSSCs have been made. For example, the Self-Synchronizing Sober (SSS) based on the SOBER-family (Rose et al. 2005) on the software side, all of which have been broken, many to CCAs. On the hardware side, the KNOT-family of ciphers, in which MOUSTIQUE (Daemen & Kitsos 2008) was last in line, produced a series of new SSSC designs, all of which were also broken. Thus, to date, the only publicly available SSSCs are those obtained from a block cipher in the CFB mode of operation. Unfortunately, a block cipher in CFB mode is inefficient and does not inherit the benefits of the synchronous stream ciphers.

2.3 Integrity and Message Authenticity

While confidentiality provides security against passive eavesdroppers, a dedicated service ensuring *message authenticity*[12] is required to fend off an active adversary. Often message authenticity and integrity[13] are regarded as even more important than that of confidentiality, perhaps even more so in a control system. More often than not, a particular signal, for example, a measurement from an accelerometer or a control signal to an actuator, need not be kept secret from a passive adversary. As was shown by the example in Section 1.1.1, while perhaps the system parameters may not always be confidential, it is almost universally undesirable if an adversary is capable of manipulating the contents of a signal in a feedback control system, or even injecting spoofed signals.

Essentially, there are two cryptographic techniques capable of providing message integrity and authenticity; A Message Authentication Code (MAC) and a *digital signature*. A MAC is a *symmetric* cryptographic technique in which the parties must share a secret key, akin to a symmetric cipher. The operations used in a MAC are often quite similar to those seen in symmetric ciphers, and some MACs are even constructed using block ciphers. On the other hand, digital signatures are universally constructed using *asymmetric* cryptographic techniques relying on number-theoretical problems that are assumed to be computationally hard, much like asymmetric ciphers. Just as with ciphers, asymmetric digital signatures are computationally expensive compared to symmetric MACs. However, in addition to message authenticity, digital signatures also provide a service known as *non-repudiation* that is *not* provided with a MAC. Non-repudiation is a service that is of most interest in an environment in which the entities cannot be trusted, such as in a financial transaction. In a control system in which the entities are trusted,

[12] Message authenticity implies that the recipient of a message can authenticate the origin of the message. Not to be confused with entity authentication.

[13] It should be clear that message authenticity also provides message integrity. If an adversary alters a message in transit, the origin of the altered message is the adversary, not the transmitter of the original message.

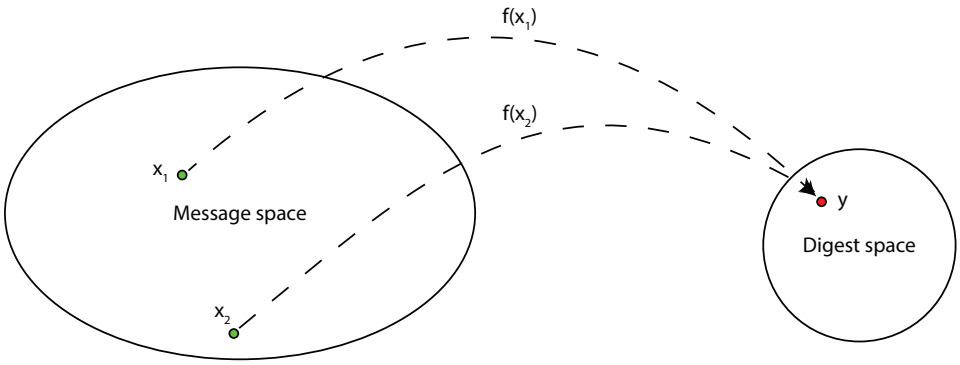

Figure 2.7: An illustration of how two messages map to the same digest via a cryptographic hash function.

that is, there is no rogue controller involved, for instance, non-repudiation is not particularly interesting. Thus, message authenticity through MAC is the most attractive option in this context.

2.3.1 Cryptographic hash functions

Before describing MACs, we introduce the notion of a cryptographic hash function. A cryptographic hash function $f : \{0,1\}^* \times \{0,1\}^K \mapsto \{0,1\}^B$ maps a message of arbitrary length to a B-bit, fixed-length output called the *digest*. The second K-bit input is a *public* parameter, meaning that the mapping is *unkeyed*.

Cryptographic hash functions are required to satisfy at least the following three properties; *preimage resistance*, *second preimage resistance*, and *collision resistance*. The first property, preimage resistance, means that for a cryptographic hash function $f_K(x) = y$, determining x given f_K, y should be computationally infeasible. The second property and the third property are both linked to the observation that the domain of the cryptographic hash function is an infinite set while the range of the cryptographic hash function is a finite set. Thus, by Dirichlet's pigeonhole principle, there must exist elements in the range with multiple preimages, that is, there must exist collisions as illustrated by Figure 2.7. The second property, second preimage resistance, states that given $f_K, x_1, y \mid y = f_K(x_1)$, finding $x_2 \neq x_1 \mid y = f_K(x_2)$ is computationally infeasible. The third property, collision resistance, states that finding *any* pair $x_1, x_2 \mid f_K(x_1) = f_K(x_2)$ should be computationally infeasible.

Collision resistance is generally considered the toughest requirement to satisfy. Assuming the cryptographic hash function is balanced, the probability of obtaining an output y given a random input x is $\frac{1}{2^B}$. The probability of finding an arbitrary collision, that is, finding *any two inputs that map to the same digest*, if k elements are tested is then given by

$$\begin{aligned} P(\text{collision}) &= 1 - P(\text{no collision}) \\ &= 1 - \left[\left(\frac{2^B - 1}{2^B}\right) \cdot \left(\frac{2^B - 2}{2^B}\right) \cdot \ldots \cdot \left(\frac{2^B - (k-1)}{2^B}\right)\right] \\ &= 1 - \left[\left(1 - \frac{1}{2^B}\right) \cdot \left(1 - \frac{2}{2^B}\right) \cdot \ldots \cdot \left(1 - \frac{(k-1)}{2^B}\right)\right] \end{aligned} \tag{2.7}$$

Assuming $k \ll 2^B$ then $\frac{k}{2^B} \ll 1$. Using the first-order Maclaurin approximation for the exponential function given by

$$e^x \approx 1 + x \tag{2.8}$$

we may rewrite (2.7) as

$$\begin{aligned} P(\text{collision}) &\approx 1 - \left[e^{\frac{-1}{2^B}} \cdot e^{\frac{-2}{2^B}} \cdot \ldots \cdot e^{\frac{-(k-1)}{2^B}}\right] \\ &= 1 - e^{\frac{-\sum_{i=1}^{k-1} i}{2^B}} \\ &= 1 - e^{\frac{-k(k-1)}{2 \cdot 2^B}} \end{aligned} \tag{2.9}$$

in which (2.9) is obtained by exploting that

$$\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2} \tag{2.10}$$

By applying (2.8) again, we obtain

$$\begin{aligned} P(\text{collision}) &\approx -\left(\frac{-k(k-1)}{2 \cdot 2^B}\right) \\ &\approx \frac{k^2}{2 \cdot 2^B} \end{aligned} \tag{2.11}$$

when k is large. Thus, we expect to find a collision after approximately

$$\begin{aligned} k &\approx \sqrt{2^B} \\ &= 2^{\frac{B}{2}} \end{aligned} \tag{2.12}$$

attempts. So, if a cryptographic hash function produces a 256-bit tag, i.e., B = 256, we would expect to find a collison, that is, break the cryptographic hash function, after approximately $2^{\frac{256}{2}} = 2^{128}$ attempts. This is called the *birthday problem* (sometimes referred to as the birthday paradox, although it is no paradox ...).

Since cryptographic hash functions are unkeyed, they may not be applied directly to ensure the integrity of a transmitted message. Instead, they are commonly used *before* a digital signature is computed, that is, the digital signature is computed over the digest of the original message, or they are used as a component of a keyed MAC.

2.3.2 Message authentication codes

A MAC $f : \{0,1\}^* \times \{0,1\}^K \mapsto \{0,1\}^B$ maps a message of arbitrary length to a B-bit output called the *tag*. A MAC is a symmetric-key construct in which the authorized parties must share a secret K-bit key. The transmitter computes a tag, before the tag is then appended to the message, after which the message and tag are sent to the receiver. Upon reception, the receiver recomputes the tag and compares the received tag with the recomputed tag. If there is a match, the message is accepted. If the two tags do not match, the message is invalidated and discarded.

Just as with a cryptographic hash function, the domain of the MAC is an infinite set while the range is a finite set. Therefore, given a tag size of B bits, finding a collision, on average, requires approximately $2^{\frac{B}{2}}$ attempts due to the birthday problem described above. The tag size varies depending on the assumed capabilities of an adversary, and the underlying technical capabilities of the system, such as the bandwidth, and usually tag sizes range from 32 bits to 512 bits. It should be noted that a MAC is an abstract construct, which may be designed in various ways. Among common design methodologies we find MACs based on cryptographic hash functions, such as the HMAC, and MACs based on block ciphers such as the Cipher Block Chaining Message Authentication Code (CBC-MAC) (Dang 2008) and the Cipher-based Message Authentication Code (CMAC) (Dworkin 2005). Other designs are quite different, such as the Poly1305, which is number-used-only-once (nonce) based (Bernstein 2005). We shall see how a MAC can be constructed from a cryptographic hash function in Section 5.1.7, when we describe an implementation of HMAC that uses the SHA-256 cryptographic hash function.

2.3.3 Attack models

In the context of message authenticity, the attack models are somewhat different from those discussed in the context of confidentiality. In particular, the most efficient attack an adversary may be capable of is that of a Chosen Message Attack (CMA). That is, the adversary is free to attempt an attack on a message of their choice. The goal of an attacker is typically categorized as one of the following:

Total break

A total break is described as an attack in which the adversary has recovered the secret key of the message authentication scheme.

Universal forgery

A universal forgery is described as an attack in which the adversary is capable of forging a valid tag on *any* message. Note that this is different from that of a total break in that the adversary has not actually recovered the key.

Selective forgery

A selective forgery is described as an attack in which the adversary has the capability of forging a valid tag on *selective* messages.

Existential forgery

An existential forgery is described as an attack in which the adversary has the capability of forging a valid tag on *at least one message*.

In the context of a MAC, the scheme should resist existential forgery under CMA. If it does not, for example, in the context of a control system, the adversary is capable of forging valid inputs to the control system or the actuators, potentially resulting in a catastrophic loss of control.

2.3.4 A word of caution

It is important to emphasize that encryption does *not* provide message integrity nor message authenticity. This is easily demonstrated by the classical *bit-flipping attack* on stream ciphers. As seen in Section 2.2.3, the encryption operation of a stream cipher often consists of combining the plaintext with the keystream with the $\oplus$-operator. Flipping a bit in the ciphertext is, therefore, equivalent to flipping a bit in the plaintext. Thus an adversary can effectively *alter the plaintext in a meaningful way* merely by altering the ciphertext. Such a stream cipher is said to be *malleable*. Block ciphers operating in CBC mode and CFB mode have different error propagation properties. Thus, altering the plaintext in a meaningful way is not always feasible. However, an adversary can still alter the ciphertext, albeit in a non-meaningful way. Previous attempts of avoiding the use of dedicated integrity schemes, for example, by relying on redundancies in the plaintext or by checking for legal padding in block ciphers, have resulted in efficient attacks.

While the literature on SSSCs is scarce, it is unfortunately littered with dubious advice. In particular, an often-cited benefit of SSSCs is that the error-propagation properties provide message integrity and authenticity. Such statements can, for instance, be found in papers by Maurer (1991), Anderson (1991), and Millerioux & Guillot (2010), as well as in a book specifically treating stream ciphers by Klein (2013) and in the Ph.D. thesis of Dravie (2017). It is paramount to realize that *this*

is not the case. Some of these assumptions rely on redundancies in the plaintext, and the dangers of using such schemes was stressed in a paper by Joan Daemen et al. already in 1992 (Daemen et al. 1992). Relying on the redundancy of the plaintext to detect altered ciphertext is dangerous and has been demonstrated to be prone to attacks when used in conjunction with block ciphers. Anderson claims that the use of an unkeyed Cyclic Redundancy Check (CRC) of the plaintext combined with the error propagation properties of SSSCs provides message integrity, quite similar to the integrity scheme used in Secure Shell (SSH) version 1. This is bad practice at best and comparable to a Hash-then-Encrypt scheme. Such schemes are known to be vulnerable. An example of an attack against a scheme in which the CRC of the plaintext serves as an integrity check for an SSSC is found in a white paper by Futoransky et al. (October, 1998) in which an attack against a block cipher in CFB mode is described in the context of the SSH version 1.5 protocol.

2.4 Authenticated Encryption

In Section 2.3.4, it was emphasized that encryption alone does not provide message authenticity and integrity. The bit-flipping attack was given as an example of how an active attacker can perform successful attacks even if encryption is used, in the lack of proper authentication methods. Thus, to achieve secure transmission of signals through an insecure channel in which both passive and active attackers are present, both encryption and message authentication techniques should be imposed. This is known as authenticated encryption. Also, in some cases, it may be possible to authenticate additional data that is transmitted unencrypted along with the ciphertext. This is known as authenticated encryption with associated data and can be useful to authenticate the IV. Historically, authenticated encryption and authenticated encryption with associated data have been achieved through the use of ad-hoc schemes in which encryption and authentication were applied in some way. Some of these ad-hoc schemes were vulnerable to attacks, warranting the need for some standardization.

2.4.1 Generic compositions

In a great effort by Bellare & Namprempre (2008), three commonly used generic compositions of encryption ciphers and MACs were addressed. Each of these schemes has been used in real applications, yet the security levels are not similar. The generic compositions investigated are illustrated in Figure 2.8.

Encrypt-and-MAC The Encrypt-and-MAC (E&M) composition provides authenticated encryption by computing the MAC tag over the plaintext, encrypting the plaintext, and then appending the tag to the ciphertext before transmission. Because the MAC is computed over the plaintext and is readily available to eavesdroppers, this scheme leaks information about the plaintext as the transmission of the same message twice is set to produce the *same* tag. The E&M-scheme is, therefore, the least secure generic composition. Nevertheless, the E&M composition has

been used in the SSH protocol and provides sufficient security in some scenarios.

MAC-then-Encrypt The MAC-then-Encrypt (MtE) composition provides authenticated encryption by computing the MAC tag over the plaintext and then encrypting the plaintext together with the tag. Thus the MtE composition provides integrity of plaintexts. However, as shown by Bellare & Namprempre (2008), this does not imply the integrity of ciphertexts. The MtE composition has historically been used in Transport Layer Security (TLS). As of the most recent version, TLS-1.3, the generic composition has been replaced by dedicated modes such as the Galois/Counter Mode (GCM).

Encrypt-then-MAC In the Encrypt-then-MAC (EtM) composition, the plaintext is first encrypted before the MAC tag is computed over the ciphertext. In addition to authenticated encryption, the EtM-scheme permits authenticated encryption with associated data by including the associated data in the computation of the tag. It was shown by Bellare & Namprempre (2008) that the EtM-scheme achieves the strongest security by providing integrity of ciphertexts, which also implies the integrity of plaintexts. Also, the EtM-scheme is computationally efficient by verifying the integrity of the message before any decryption is performed. If the tag validation fails, the message is just discarded and never decrypted. The EtM composition is, for these reasons, the most common generic composition and has historically been used in the IPsec protocol.

2.4.2 Authenticated encryption modes

In addition to the generic compositions, the use of dedicated modes for block ciphers is available. Indeed, today most authenticated encryption and authenticated encryption with associated data is achieved with dedicated modes of operation. In particular, AES in GCM mode (Dworkin 2007) is frequently used, along with AES in Counter with CBC-MAC (CCM) mode (Dworkin 2004). Unfortunately, all publicly available authenticated encryption and authenticated encryption with associated data modes of operation are so-called “two-pass” algorithms. That is, two passes through the data are required to both authenticate and encrypt the data, thus affecting the overall efficiency. Note that both CCM and GCM internally are constructed as a generic composition, although the user only acts on one interface. For example, the CCM mode is just a block cipher operating in CTR mode combined with the CBC-MAC from the same block cipher in a MtE composition.

Whether or not generic compositions or dedicated modes are preferred have been a subject open to debate. While some claim that dedicated modes are harder to misuse, others claim that separation of services providing confidentiality and integrity yields a “cleaner” design and is easier to analyze. An interesting aspect supporting the first statement is the fact that many security breaches result from circumvention of the cryptography, for example, due to implementation error, as

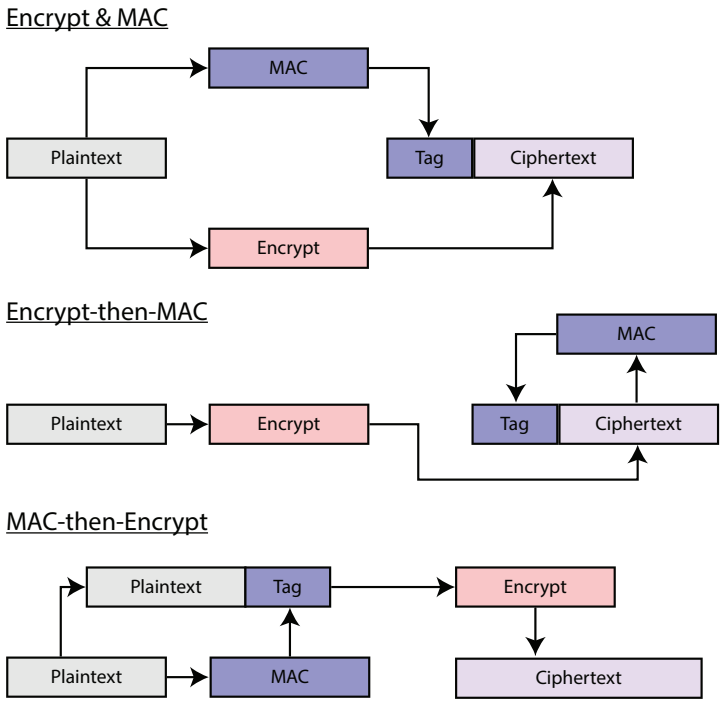

Figure 2.8: An illustration of the three generic compositions discussed in Bellare & Namprempre (2008).

was the case for *Heartbleed*[14], rather than breaking it.

2.4.3 Dedicated authenticated encryption algorithms

Finally, in addition to the generic compositions and the block cipher modes of operation described above, there exist dedicated authenticated encryption algorithms. Examples of such dedicated ciphers are the new AEGIS stream cipher, which will be discussed in Chapter 5, as well as the AES Offset Codebook (OCB) mode. The latter could be interpreted as a block cipher mode of operation akin to the CCM and GCM modes. However, it is standardized uniquely with the AES as the underlying block cipher and is therefore considered a dedicated authenticated encryption algorithm. Note that the AES OCB algorithm will not be treated further in this thesis, as this algorithm is currently patented and encumbered by licenses[15].

[14] Heartbleed was an implementation error in the TLS implementation of the OpenSSL cryptographic library. More information about the error can be found here: https://heartbleed.com/ (Accessed May 7th, 2020)

[15] It should be stated that the inventor and patent assignee states on his website that he freely licenses the OCB for most settings. More information can be found here: https://www.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm#patent:phil (Accessed May 19th 2020)

2.5 Availability

The introduction of MACs and encryption can also affect the system in other ways. In particular, the issue of synchronizing the transmitter and the receiver will be addressed. Furthermore, addressing the resulting traffic expansion is important, particularly for systems transmitting small packets at a high frequency.

2.5.1 Synchronization

As discussed in Section 2.2.3, stream ciphers are *stateful* ciphers, requiring synchronization between the transmitter and the receiver. Relying on reliable transmission of signals through protocols such as TCP, as was done by Teranishi et al. (2020), is not a satisfactory solution. In a real-time control system, data is likely to be transmitted in an inherently unreliable fashion, for example, using the UDP protocol. If a packet is lost or injected into the transmission, synchronization is lost between the transmitter and receiver, resulting in incorrect decryptions. Without a timely resynchronization, such a loss of synchronization can result in a loss of control.

In this chapter, we see that symmetric cryptographic methods offer multiple solutions to the synchronization problem. A way of circumventing the problem is by switching to an SSSC. However, as described in Section 2.2.3, this leaves inefficient block ciphers in CFB mode as the only option. An alternative solution used in synchronous stream ciphers, for example, A5/1 (used in GSM), is to transmit a frame number that is used to indicate an offset from some known position of the keystream generator. Finally, as seen in Section 2.2.3, modern stream ciphers permit the use of public IVs that are used to explicitly synchronize the cipher by reinitializing the cipher for each message.

2.5.2 Traffic expansion

The choice of the encryption algorithm and the choice of tag size used in the MAC affects the size of the transmitted data. Traffic expansion is especially important to consider in applications where small messages are transmitted at a very high frequency, as it will significantly increase the required bandwidth to avoid network congestion and packet loss.

Choice of cipher

The use of padding for the CBC mode of operation in block ciphers invariably results in a ciphertext that is larger than the original plaintext. For small amounts of data, the difference can be substantial and have a significant effect on the network if packets are transmitted at high frequencies. Also, most encryption algorithms require IVs to be transmitted along with each message, either for safe operation, for example, CBC mode, as illustrated by the BEAST attack (Duong & Rizzo 2011), or to achieve synchronization for synchronous stream ciphers.

Consider the following example: A 4-byte floating-point measurement is to be

transmitted. Encrypting the measurement using AES in CBC mode first requires the 4-byte measurement to be padded to a multiple of the block size, that is, 16 bytes. Additionally, each message requires a 16 byte unpredictable IV. Thus the initial 4-byte measurement results in a 32-byte message, an 800% expansion, and this does not even include a MAC tag.

Choice of tag size

As mentioned in Section 2.3.2, the tag size determines the effort required to forge a message. However, the tag size also has a direct impact on the amount of data transmitted. If the message authenticity scheme is to provide a similar level of security as an N-bit encryption scheme, the tag should be of size $2N$ due to the birthday problem. Thus, if the traffic is encrypted using a cipher providing 128-bit security, the MAC tag should be 256 bits.

Chapter 3

The Encryption Laboratory

This chapter describes the laboratory setup that was built to assess the cryptographic algorithms and to implement the attacks in the coming chapters, in addition to the software setup that was used during tests and attacks. The laboratory setup was built as part of the TTK4550 project report, and this chapter is based on Chapter 3 from the TTK4550 report.

3.1 Hardware Setup

A laboratory setup was built to conduct experiments. The setup consists of two RevPi Connect+[1] industrial Raspberry Pis from KUNBUS with two ethernet interfaces each, powered by a 24V DC power supply. The hardware specification of the RevPi Connect+ can be seen in Table 3.1. Note that the version of the ARM Cortex A53 used in the RevPi Connect+, the Broadcomm BCM2837B0, does not include the ARM Crypto Extension. The available instruction set features on the RevPi Connect+ can be seen in Figure 3.1. A modified Raspbian Stretch 32-bit operating system, the RevPi-Stretch-2019-03-14, serves as the operating system.

3.1.1 Latency measurements

For the latency measurements, a direct connection between the two RevPi Connect+ machines was set up with an ethernet crossover cable, and static IP addresses were set on ethernet interface 1 on both RevPi Connect+ machines. Furthermore, the two RevPi Connect+ machines were connected to the internet and remotely accessed via SSH through ethernet interface 0. An illustration of the setup can be seen in Figure 3.2, while a schematic view of the setup can be seen in Figure 3.3.

[1] Datasheet available here: https://revolution.kunbus.com/revpi-connect/ (Accessed May 10th, 2020)

Table 3.1: KUNBUS RevPi Connect+ hardware specification.

Attribute	Value
Processor Type	ARM Cortex A53 (ARMv8) BCM2837B0
Number of cores	4
Clock speed	1.2 GHz
Memory (RAM)	1 GB

```
pi@RevPi121492:~ $ cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 76.80
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 1
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 76.80
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 2
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 76.80
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

processor       : 3
model name      : ARMv7 Processor rev 4 (v7l)
BogoMIPS        : 76.80
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd03
CPU revision    : 4

Hardware        : BCM2835
Revision        : a02100
Serial          : 00000000607bdf2e
```

Figure 3.1: The available instruction set features in the RevPi Connect+. Note that it claims that the processor is an ARMv7 BCM2835 processor. However, this is incorrect.

3.1.2 System simulation

For the hijacking experiments, the hardware setup was slightly modified. Rather than transmitting the signals across the ethernet crossover cable, the signals were transmitted via the router, through ethernet interface 0. The 'adversary' consisted of a laptop connected to the same router through Wi-Fi. The hardware setup for the hijacking experiments can be seen in Figure 3.4.

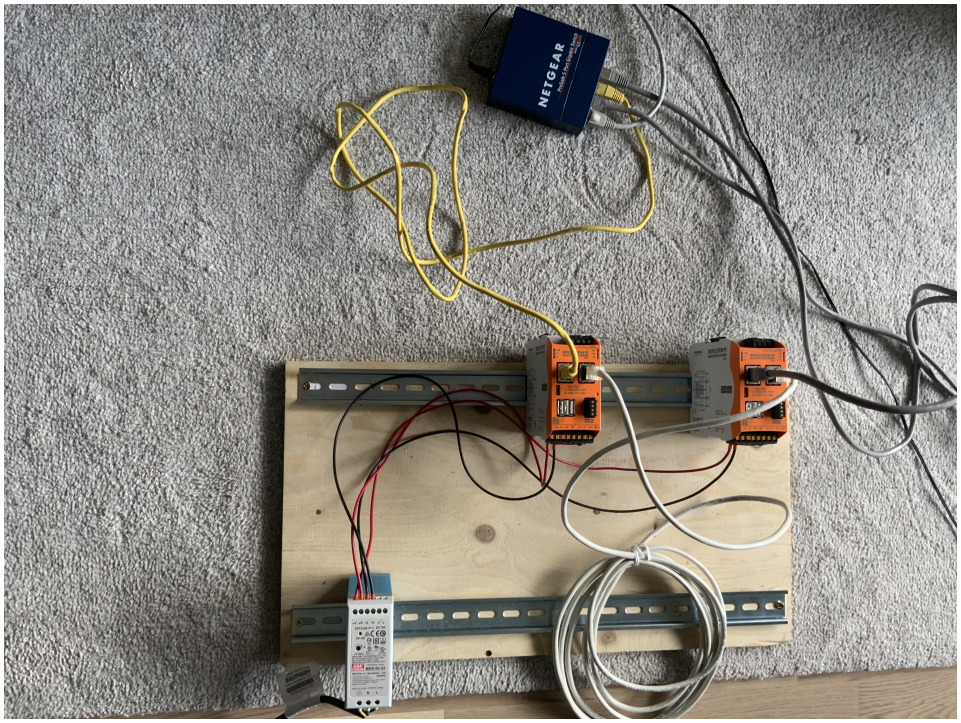

Figure 3.2: An overview of the encryption laboratory setup that was used to assess the performance of the cryptographic algorithms.

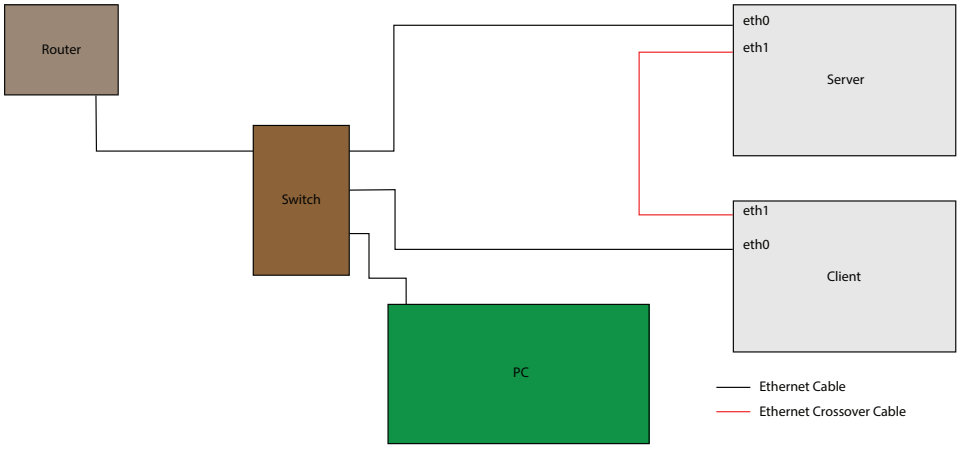

Figure 3.3: A schematic view of the encryption laboratory setup that was used to assess the performance of the cryptographic algorithms.

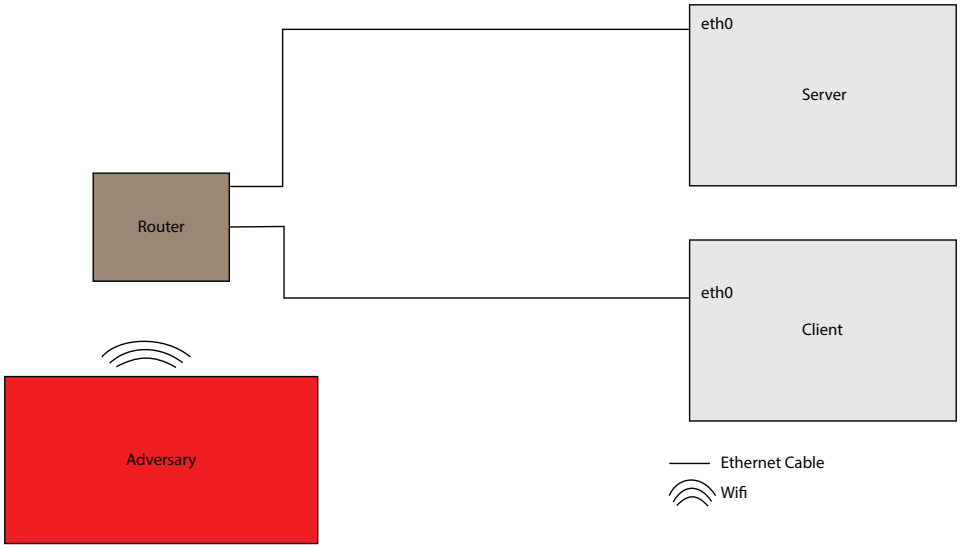

Figure 3.4: A schematic view of the hardware setup that was used to launch the hijacking experiment, both with and without encryption.

3.2 Software Setup

In this section, the software setup that was used is described. First we describe some auxiliary functions that were used, then we proceed by describing two software lab setups that were used extensively during the experiments. The first lab setup described was used to measure the latency induced by the cryptographic algorithms. The second lab setup described was used to perform hijacking experiments and to attack the communication schemes proposed by other researchers.

3.2.1 Endianness

Because cryptographic algorithms often operate on words rather than bytes, the way byte arrays are mapped to words is important to achieve the correct output from algorithms in which the input enters the algorithms and are operated on, e.g., block ciphers and cryptographic hashing algorithms. Specifically, bytes are commonly mapped to words in two ways; the *big-endian* order in which the bytes are mapped from the least-significant byte at the highest memory address to the most significant byte at the smallest memory address, hence the name **big**-endian. Conversely, the bytes can be mapped from the most-significant byte at the highest memory address to the least-significant byte at the smallest memory address, a convention known as **little**-endian. The difference between the two mappings can be seen in Figure 3.5.

Most processor architectures today, such as `x86` and `ARMv8` processors, utilize the little-endian convention. However, some older algorithms, such as AES and SHA-2,

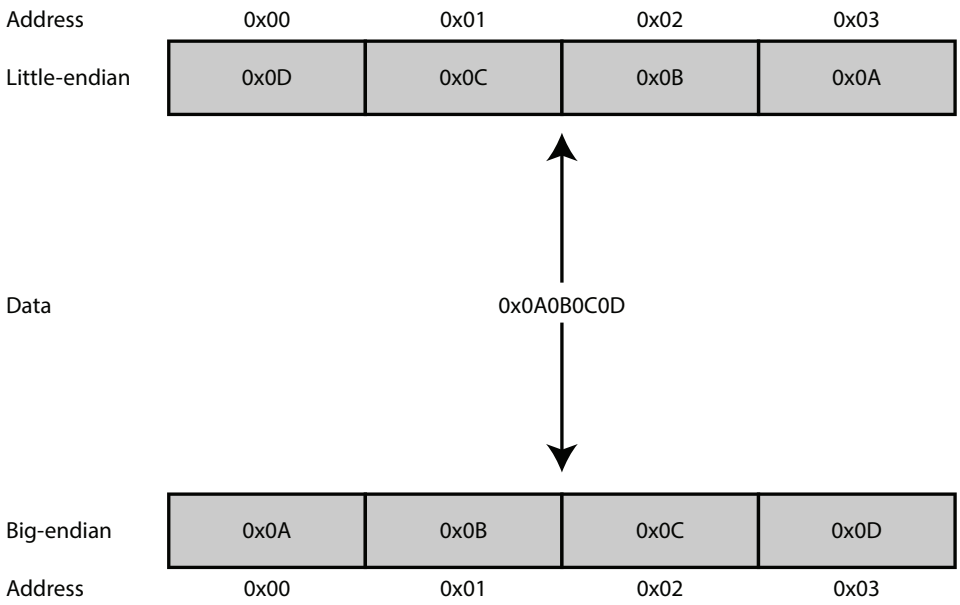

Figure 3.5: An illustration of how a 32-bit integer is mapped to memory on a little-endian and big-endian architecture, respectively.

are defined using the big-endian convention. Therefore, to verify that algorithm implementations are correct with the official test vectors, we must be able to change the endianness of the input to the algorithms. Specifically, for the RevPi Connect+, running a 32-bit system, we must be *change the byte order of the 32-bit words*. We achieve this through the function seen in Listing 3.1.

Listing 3.1: Endianness Swap

```
// Byte swap changes the endianness of
// words on a 32-bit system.
void byte_swap(uint8_t *output, uint8_t *input, int size)
{
    for (int i = 0; i < size/4; i++)
    {
        uint32_t num = ((uint32_t*)input)[i];
        uint32_t swapped = ((num >> 24) & 0xff) |
            ((num << 8) & 0xff0000) |
            ((num >> 8) & 0xff00) |
            ((num << 24) & 0xff000000);
        ((uint32_t*)input)[i] = swapped;
    }
}
```

Notice that endianness is also often discussed in networking, since in networks byte arrays are transmitted and therefore, upon reception, one must agree upon how

these byte arrays are to be interpreted at the receiving end. This is critical for the networks in which different systems are connected, since a message could be transmitted from little-endian to big-endian systems and vice versa. For example, in the Internet Protocol all messages should be transmitted in big-endian order according to RFC1700 (Reynolds & Postel 1994). However, since we are transmitting data between little-endian devices only, we do not treat this topic in more detail.

3.2.2 Serialization and deserialization

The data is kept in C structs, as seen by Listing 3.2. While the entries in C structs *may* be contiguous in memory, this is not guaranteed due to data alignment in memory, i.e., the structs may contain padding. The padding is added, for example, to ensure that the data is aligned with the word size of the system. Notice that for the particular struct illustrated in Listing 3.2, the struct is *very likely* to be contiguous in memory on *most* systems because a `double` and a `std::chrono::system_clock::time_point` is usually a multiple of the underlying word size. Nevertheless, to guarantee that the data that is to be processed by the cryptographic algorithms or sent as a UDP packet are contiguous, the data must be serialized. After a packet is received the packet must be deserialized to recover the original data format. Serialization and deserialization is done using the functions described in Listing 3.3, in which the data is kept in the C struct `data_struct`. The constant `LOAD_SIZE` describes how much data the struct contains, and can vary, for example, when assessing the latency induced on data of varying size in Chapter 7.

Listing 3.2: Data struct

```
struct data_struct {
    double load[LOAD_SIZE];
    std::chrono::system_clock::time_point time_stamp;
};
```

Listing 3.3: Serialization and Deserialization

```
// Serialization function
void serialize(data_struct* data, uint8_t* serialized_data)
{

    // Map the data to the buffer
    double *x = (double*) serialized_data;
    for (int i = 0; i < LOAD_SIZE; i++)
    {
        *x = data->load[i]; *x++;
    }

    // Map the timepoint to the buffer
    std::chrono::system_clock::time_point *p = (std::chrono::system_clock::
        time_point *) x;
    *p = data->time_stamp; *p++;
}
```

```
// Deserialization function
void deserialize(const uint8_t* serialized_data, data_struct* data)
{

    // Map the data to the struct
    double *x = (double*) serialized_data;
    for (int i = 0; i < LOAD_SIZE; i++)
    {
        data->load[i] = *x; *x++;
    }

    // Map the timepoint to the struct
    std::chrono::system_clock::time_point *p = (std::chrono::system_clock::
        time_point *) x;
    data->time_stamp = *p; p++;
}
```

3.2.3 Latency measurements

During latency testing, one RevPi Connect+ served as a client, for example an actuator with a sensor, whereas the other RevPi Connect+ served the role of a server, for example a controller. In a dynamical control system, the client would play the role of a measurement instrument and an actuator in this setup, whereas the server serves the role of the control unit where the control signal is computed. Confidentiality and integrity of the signal transmission in both directions must be ensured. Therefore, encryption, decryption, MAC tag generation, and MAC tag validation occur twice.

By transmitting the signals back and forth, the Round-trip Time (RTT), or latency, could accurately be measured using the clock on the client-side only. The first timestamp, t_1, was taken at the time the struct was generated at the client, and the second timestamp, t_2, was taken at the time the struct was successfully decrypted at the client. The RTT was then computed according to

$$RTT = t_2 - t_1 \tag{3.1}$$

The software was written in C++. As minimizing latency is emphasized over the possible loss of a packet, the UDP protocol was preferred over the TCP protocol, as is common in real-time applications. Sockets were opened on ports 4322 and 4321 for the server, and the client, respectively. The data was transmitted over the ethernet crossover cable connected to ethernet interface 1 on both devices.

To take advantage of the quad-core processor, the tasks on the client and the server were split into two threads. On the client-side, one thread handled the sampling, the encryption, the tag generation, and the transmission of the packets

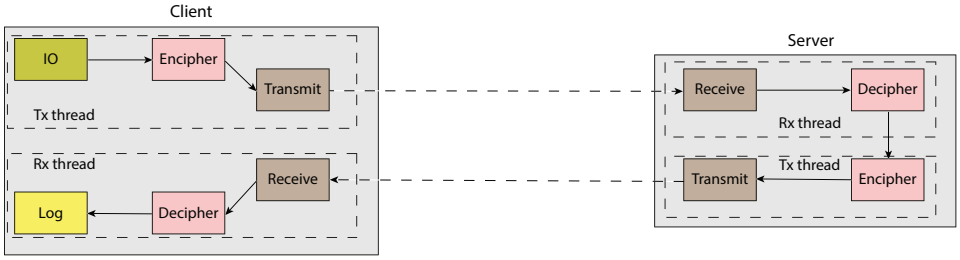

Figure 3.6: An overview of the program flow that was used in the encryption laboratory lab when the latency of different cryptographic algorithms was measured.

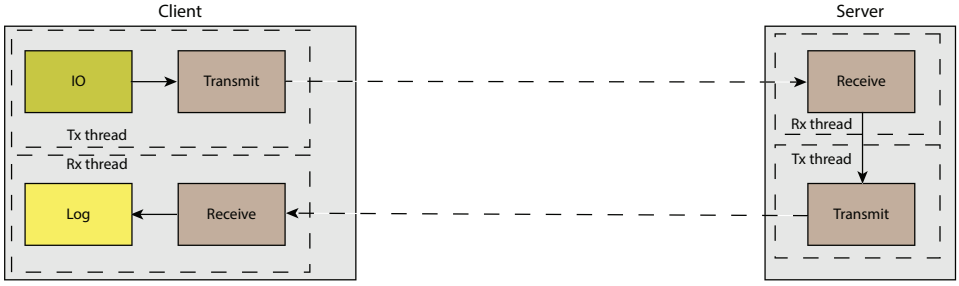

Figure 3.7: An overview of the program flow used in the encryption laboratory when the latency without encryption in the feedback loop was measured.

to the server, while the other thread handled the reception, the tag verification, the decryption, and the logging of the packets coming from the server. On the server-side, one thread handled the reception, the tag verification, and the decryption of the packet coming from the client, while the other thread handled the encryption, the tag generation, and the transmission to the client. On the server-side, data was transferred between the two threads using a shared object guarded by mutexes. An illustration of the signal flow can be seen in Figure 3.6. To measure a *base-line* latency *without* cryptographic techniques involved, the encryption, the decryption, the tag generation, and the tag verification procedures were simply removed. An illustration of the setup used when the base-line was measured can be seen in Figure 3.7.

The data structure, given in Listing 3.2, that was passed between the client and the server contained a timestamp to prevent replay attacks and to measure latency, and an array to hold the data. The size of the array was changed via the `DATA_SIZE` constant when the performance on different data sizes was measured.

3.2.4 System simulation

In Section 1.1.1, an example of how a system without encryption and message authentication could be hijacked was demonstrated. The demonstration was carried

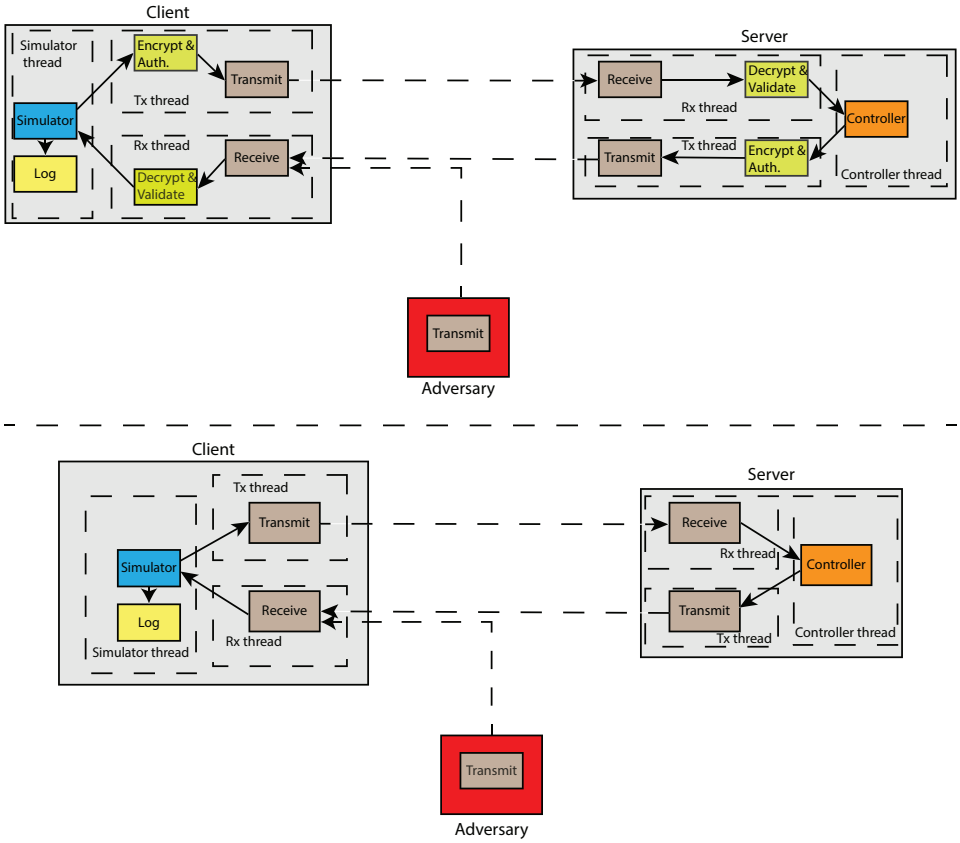

Figure 3.8: The software setup used in the hijacking experiment with and without encryption and authentication.

out in the hardware setup described in section 3.1.2.

The software setup merely consisted of establishing a third thread on the server and client, running a PI controller, and an MSD simulation, respectively. As before, data on the server-side was passed between the threads through shared objects, that is, the received state measurement and the computed control signal. On the client-side, shared objects were set up to pass the received control signal to the MSD simulation and the position state from the MSD simulation to the transmission thread. All shared objects were guarded by mutexes. The adversary software merely consisted of a brief C++ program that opened a socket and transmitted the spoofed control signal to the client at a frequency of 100 Hz. The software setup, with and without encryption, can be seen in Figure 3.8.

Note that, in addition to being used in the motivating example in Section 1.1.1, this setup was used to implement the attacks described in Chapter 4 and in Chapter 7.

Chapter 4

Applied Cryptography in Feedback Control Systems

In this chapter, we begin by analyzing how previously proposed schemes fail to provide confidential signal transmission or integrity and authenticity of the transmitted signals. Attacks are implemented, demonstrating how they fail in real scenarios in experiments conducted in the Encryption Laboratory described in Chapter 3. Finally, we show how proper authenticated encryption can be used to provide confidentiality, integrity, and authenticity of the signals transmitted in feedback control systems.

4.1 Analysis of Previous Proposals

In Section 1.2, several previously proposed schemes were mentioned. However, these schemes, while commonly cited as possible solutions, are rarely, if ever analyzed from a critical perspective. In this section, we investigate the security of these proposals.

4.1.1 Electronic codebook encryption in feedback control systems

As noted in Section 1.2, many of the proposed schemes apply block ciphers in ECB mode. Gupta & Chow (2008) argue; *As ECB (Electronic Codebook) is considered the fastest mode of operation, it is used commonly in real time systems.* Unfortunately, a block cipher operating in ECB mode merely acts as a fixed substitution, thus repeatedly encrypting the same plaintext block will *always* result in the same ciphertext block. This property causes non-trivial structural information about the plaintext to leak through to the ciphertext.

To illustrate how such a scheme fails to provide confidential signal transmission, consider the STM proposed by Pang et al. (2011) seen in Figure 4.1. The STM may be implemented (in C++) using the `data_struct` described in Section 3.2.2, using the *time_point* struct from the standard library as timestamp and holding the data in a `double`. Notice that the data field and the timestamp are (very likely) of size eight bytes, the same as the block size of DES. DES was implemented and operated in ECB mode. Finally, the MD5 cryptographic hash function was implemented, producing a 16-byte digest. Because the size of the timestamp, data field, and MD5 digest all are of a size that is a multiple of the DES block size, they are processed *independently* by DES in ECB mode. An illustration of this scenario is shown in Figure 4.2.

Furthermore, consider a scenario in which the STM is used to transmit signals from a bang-bang controller to a system. A bang-bang controller only produces two outputs, i.e., plaintexts; an 'on' and an 'off' signal. Because the data field only contains two valid plaintext blocks, we expect the corresponding ciphertext block, C_2, to only contain two distinct blocks, as seen by Figure 4.3. A bang-bang controller was implemented and used to transmit messages, using the STM, to an eavesdropper not in possession of the secret key. The eavesdropper then logged the messages in hexadecimal encoding. A selection of the received packets can be seen in Table 4.1. Notice that, as expected, the C_2 ciphertext block corresponding to the data field of the message always takes on two values only. Thus, it is trivial to extract information such as whether a control signal has changed or not. Furthermore, if a valid plaintext-ciphertext pair is ever available to the eavesdropper, that is, a KPA, all confidentiality is lost for past and future control signals. It is important to note that such an attack applies to *any* scheme in which a block cipher is used in ECB mode. Thus, using a 'more secure' block cipher such as 3DES or AES would not make a difference, although with AES the block size would be different. Note that such a scenario with a bang-bang controller and data sizes that exactly fit the block size were chosen for illustrative purposes and similar attacks do apply even if the data field has a large number of valid plaintexts; breaking all confidentiality would only require a greater number of known plaintext-ciphertext pairs. In addition, operating a block cipher in ECB mode fails catastrophically if the data that is to be transmitted contains a lot of structure as this structure leaks through to the ciphertext. Such problems would be encountered, for example, if an image stream was being encrypted.

Finally, while Gupta & Chow (2008) claim that ECB offers the highest performance and should therefore be used in real-time systems, this benefit if true is negligible. As an example, the CTR block cipher mode of operation is parallelizable just as the ECB mode, yet CTR mode does not suffer from the problem described above. Such a negligible difference in performance is in no way justified when it enables attacks as can be seen above. As a rule of thumb, block ciphers should *always* be operated in a mode of operation for which repeated encryption of the same plaintext blocks is highly unlikely to result in the same ciphertext blocks. Examples of such modes of operation are the CBC, CFB, OFB, and CTR modes, as discussed in Section

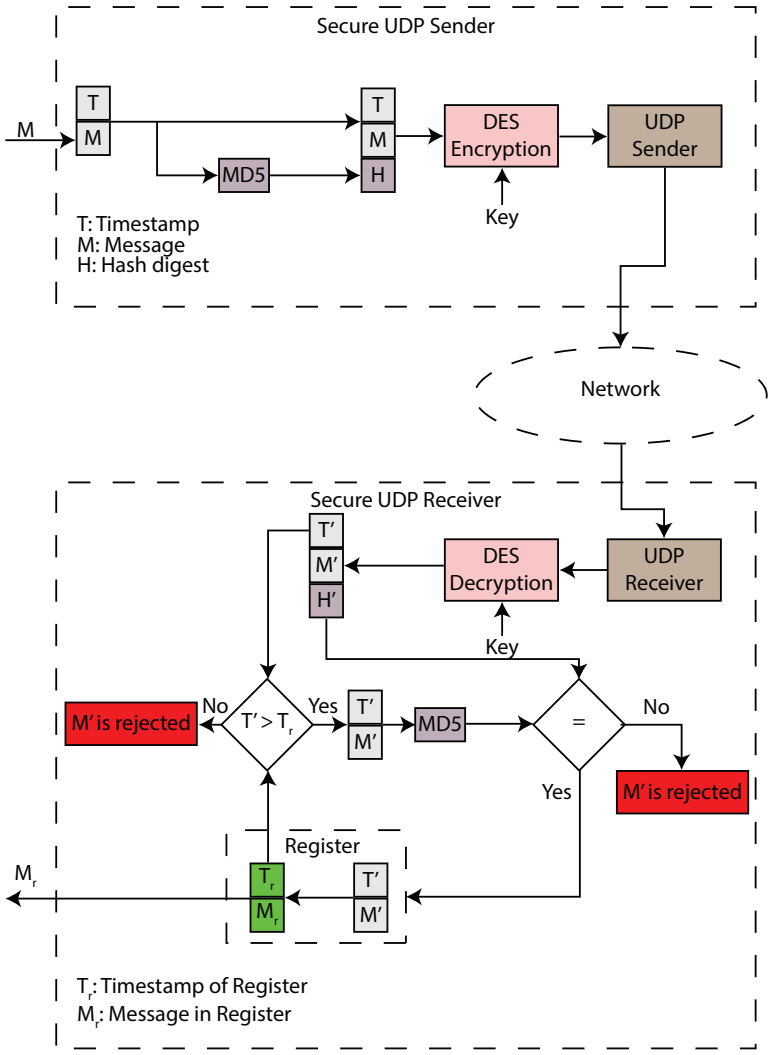

Figure 4.1: The secure signal transmission proposed by Pang et al. (2011), utilizing DES, MD5, and timestamps. Even though DES is considered very outdated and was broken in 1992 by Biham & Shamir (1992), we argue that the scheme is fundamentally flawed independently of the block cipher used.

2.2.2.

4.1.2 The secure transmission mechanism

As illustrated in the preceding section, operating a block cipher in ECB mode is not a viable option. Thus, one might be tempted to operate the block cipher in the STM proposed by Pang et al. in a different mode of operation. Unfortunately, the

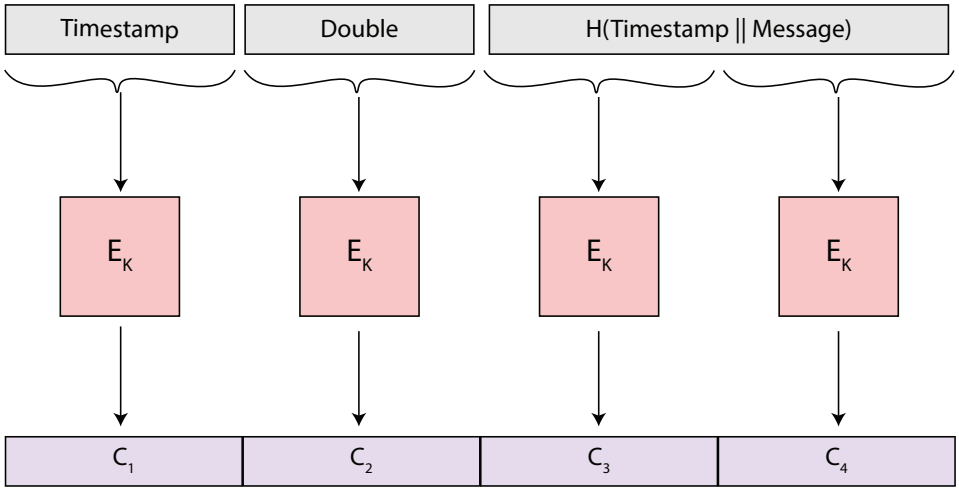

Figure 4.2: Encrypting a message directly using a block cipher in ECB mode.

Table 4.1: Hexadecimal encoding of packets transmitted using the STM proposed by Pang et al. in (Pang et al. 2011, Pang & Liu 2012).

Timestamp	Data	MD5 digest	
550BDEBE34C8AD07	E94DA68CA723B1C1	D155562D239AF118	AFF6CB7C57BEA580
FB37DBFF53E398FF	E94DA68CA723B1C1	8F03E61289A402EF	CC4BDF300CBA0C66
F2E79339D9D763EF	E94DA68CA723B1C1	D8E10C0DA8DAAE42	0A812E517FA9C8BD
3DB452729D6566FD	21C986E4D3AC8173	DB036300D42C1057	32BF91542D9AD2A0
2B147599C412420A	21C986E4D3AC8173	A4E5285923490AD1	B76F1478AE1FB748
E2B865023B1DE2E0	21C986E4D3AC8173	6A6DEB55D859CE76	4A4AAB359F2C123D
7579EB89F3150BF8	21C986E4D3AC8173	274F3DA7B4D2BC54	099BA03FD0E184D7
2E57268A139EBB08	21C986E4D3AC8173	AC2BD63A2CC93245	179815E9339BA5D7
BCBBBB8AF9217BF1	21C986E4D3AC8173	E983900302D2E899	9A68344B96C49FD1

'Hash-then-Encrypt'-like STM has a fatal flaw that the enables injection of spoofed messages if this is done, as is illustrated by the following example.

Consider a scenario in which the STM is augmented with AES operating in CTR mode. Note that CTR mode is parallelizable just like ECB mode and is likely to perform as well as the ECB mode in most scenarios thus countering the 'efficiency-argument' of Gupta & Chow (2008). Recall from Section 2.2.2, that by operating a block cipher in CTR mode we essentially transform the block cipher to a synchronous stream cipher by introducing a *state* determined by a nonce and a counter value. The block cipher produces a keystream by using an IV consisting of the nonce and counter value as input. Recall that the IV acts as a synchronization mechanism between the transmitter and the receiver, and *must not be repeated*. Each iteration produces 128 bits of keystream, after which the counter is incremented. The plaintext is then encrypted by combining the keystream with the plaintext

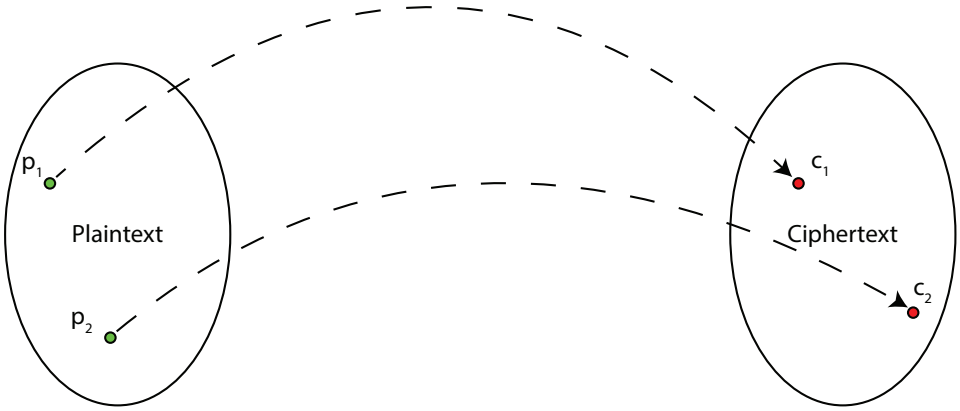

Figure 4.3: ECB encryption mapping two plaintexts to two ciphertexts.

through the exclusive-or ($\oplus$) operator, according to

$$\begin{aligned} C &= \mathrm{Enc}_{K,IV}(P) \\ &= KS \oplus (T||M||H) \end{aligned} \tag{4.1}$$

Remember that the IV is a *public* parameter that is usually transmitted along with the ciphertext in the plaintext.

The recipient, in possession of the secret key, can then generate the same keystream by feeding the IV into the block cipher and recover the plaintext by combining the keystream with the ciphertext through the $\oplus$-operator as seen by

$$\begin{aligned} P &= \mathrm{Dec}_{K,IV}(C) \\ &= KS \oplus C \\ &= KS \oplus KS \oplus (T||M||H) \\ &= (T||M||H) \end{aligned} \tag{4.2}$$

Notice, however, that by this scheme, an adversary in possession of a plaintext-ciphertext pair, that is, a KPA, is *also* capable of recovering a valid keystream by combining the ciphertext with the corresponding plaintext according to

$$\begin{aligned} C \oplus P &= KS \oplus (T||M||H) \oplus (T||M||H) \\ &= KS \end{aligned} \tag{4.3}$$

Suppose that an adversary is in possession of a plaintext-ciphertext pair. Furthermore, assume that the adversary would like to manipulate the behavior of the

system, the controller, or both by injecting a forged message $\widetilde{M}$. Because the STM uses an *unkeyed* cryptographic hash function, the adversary can easily compute $\widetilde{H} = H(\widetilde{T}||\widetilde{M})$ in which $\widetilde{T}$ is a valid timestamp. The adversary is then free to encrypt *any* forged message $\widetilde{P} = (\widetilde{T}||\widetilde{M})$ by combining the keystream recovered by (4.3) with the forged message using the $\oplus$-operator according to

$$\begin{aligned} \widetilde{C} &= \widetilde{P} \oplus KS \\ &= (\widetilde{T}||\widetilde{M}||\widetilde{H}) \oplus KS \end{aligned} \tag{4.4}$$

The adversary then prepends the IV that was associated with the plaintext-ciphertext pair from which the keystream was recovered, and transmits the message.

Upon reception of the forged message the receiver extracts the IV from the message and generates a keystream to decrypt the message. After decrypting the message by (4.2), the receiver verifies the validity of the timestamp and the MD5 digest of the $(\widetilde{T}||\widetilde{M})$-pair. Since the MD5 is unkeyed, the recomputed digest matches the digest that was transmitted and the receiver accepts the message as authentic as shown in Figure 4.1. We observe that the adversary is in fact capable of forging *any* correctly formatted message, even though the adversary is not in possession of the secret key. This corresponds to a universal forgery on the STM, as defined in Section 2.3.3, given a single plaintext-ciphertext pair. We note that this *contradicts the claim of the authors who state that the use of MD5 is secure as long as DES is not broken!*

To illustrate how vulnerable such a scheme is, the STM was augmented with AES operating in CTR mode. The MSD system and PI controller from Section 1.1.1 were used. This time, a timestamp was associated with each message that is transmitted between the system and the controller, and the messages are hashed with MD5 and encrypted with AES in CTR mode. Assume that the adversary already knows the parameters of the system, and wants to force the system to the state $x = 2$ as in the motivating example in Section 1.1.1. The adversary computes a spoofed control signal using (1.5). The adversary then logs all messages transmitted between the controller and the system and at some point gains access to a plaintext that corresponds to a logged ciphertext. The adversary recovers the keystream from the known plaintext-ciphertext pair using (4.3) and logs the valid keystream and the corresponding IV. The adversary then uses a valid timestamp, computes the MD5 digest of the spoofed control signal and timestamp, encrypts the message using the recovered keystream according to (4.4), and repeatedly transmits the forged message with the associated IV as described above. The result of this attack can be seen in Figure 4.4. Since the forged messages are accepted by the recipient, the system is successfully hijacked and forced to a state chosen by the adversary.

At this point, it should be clear that breaking AES (or DES) to successfully forge messages using the STM is not necessary. Again, the choice of block cipher is irrelevant to the attack, as the cipher is circumvented. Thus, using DES, 3DES

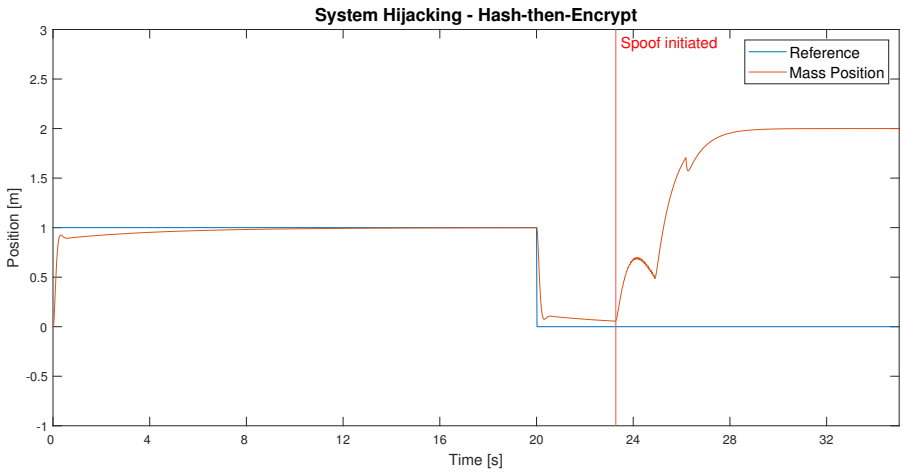

Figure 4.4: A successful known-plaintext attack against the STM, resulting in a system hijacking.

or any other block cipher would not offer greater resistance to a KPA. It is worth noting that the same attack applies if the OFB mode of operation is used. If the CBC or CFB mode of operation is used, it is harder to see how such an attack would apply but these modes are less efficient than CTR mode and are not parallelizable. However, it should be clear from these examples that the STM is very prone to attacks. To prevent this attack, one could use a keyed MAC such as the HMAC algorithm, for example, by augmenting the scheme proposed by Jithish & Sankaran (2017) by operating the block cipher in another mode of operation than ECB.

Furthermore, the reader might wonder whether it is reasonable to assume that an adversary is capable of executing a KPA. We argue that in most instances it is; plaintext is *not* random and the adversary is often capable of making educated guesses. Perhaps more importantly, successfully breaking the system would be no harder than guessing a valid plaintext, which is entirely independent of the key. Finally, even in a scenario where the plaintext *is* highly randomized, a leak of logs would be catastrophic.

4.2 Authenticated Encryption for Feedback Control Systems

As illustrated in the preceding section, the previously proposed schemes possess weaknesses that make them vulnerable to attacks, resulting in loss of confidentiality or even successful injection of spoofed data. Instead of using *ad-hoc* schemes such as the STM, we urge that *cryptographically sound* constructions that provide authenticated encryption should be used. The STM could be enhanced as seen in Figure 4.5, to accommodate this change. As observed in Section 4.1.1, it is im-

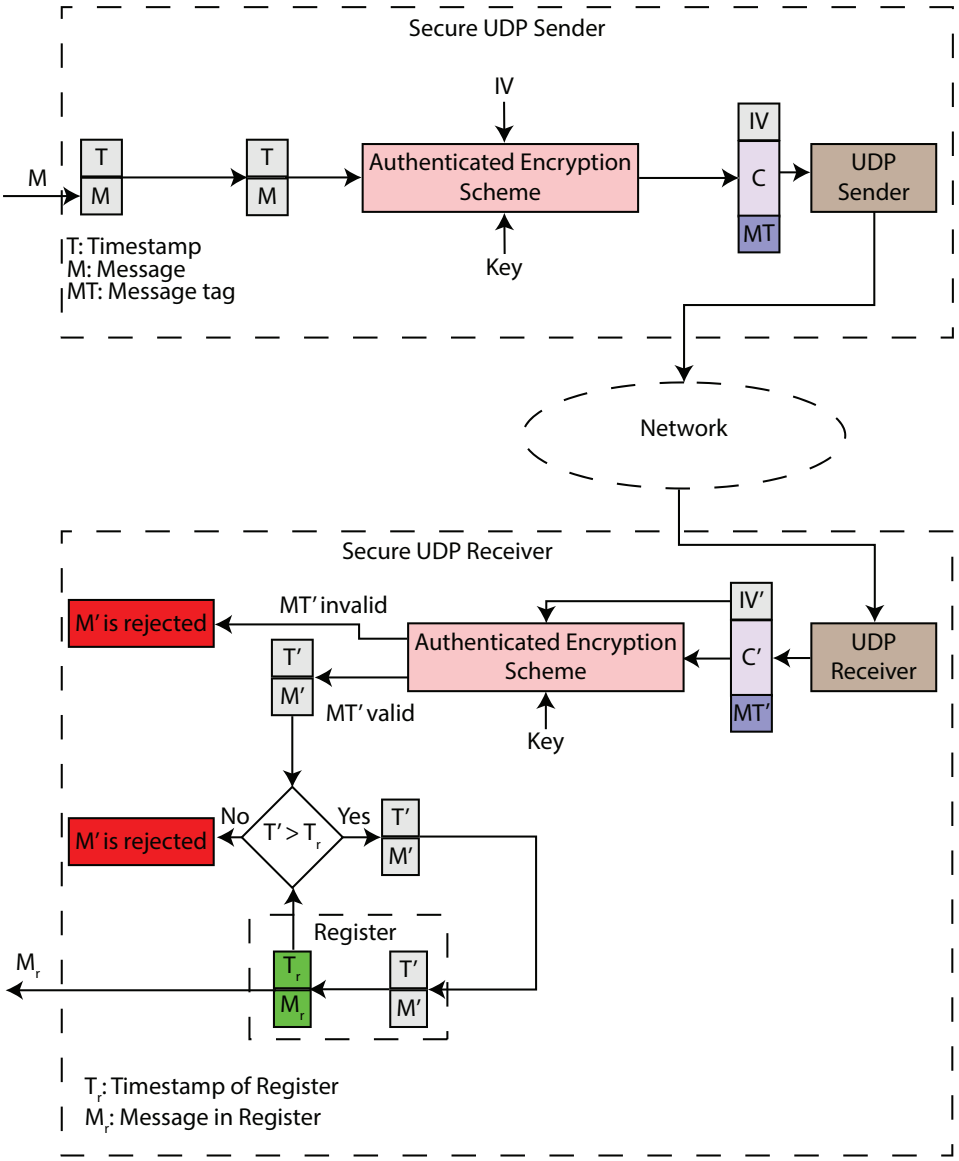

Figure 4.5: An enhanced STM, providing proper authenticated encryption.

portant to avoid operating the block ciphers in ECB mode to prevent information from leaking through to the ciphertext. Furthermore, combining ciphers with *keyed* MACs rather than unkeyed cryptographic hash functions is important to prevent the injection of forged messages.

Remember from Section 2.4 that authenticated encryption may be achieved through

generic compositions, dedicated block cipher modes of operation, or dedicated authenticated encryption algorithms. Regarding generic compositions, we note that the E&M composition suffers from the same problem as the ECB mode of encryption in the sense that the message tag is computed over the plaintext and is visible to the eavesdroppers. Thus, the same plaintext results in the same tag, and the eavesdropper can identify if the same message is sent twice. Although it should be noted that the tag is computed over the *entire* plaintext, whereas the ECB mode of encryption treated each block independently, thus the E&M composition is not quite as vulnerable. In brief, we choose to combine MACs and encryption algorithms in the EtM composition as this ensures that we are capable of discarding invalid (message, tag)-pairs without needing to decrypt them first. The pseudocode to implement the transmitter and the receiver of the enhanced STM, using authenticated encryption achieved through EtM compositions, can be seen in Algorithms 1 and 2, respectively.

Algorithm 1 Enhanced STM Transmitter

```
1: Initialize MAC_K, E_{K,IV}
2: while true do
3:    Load data.
4:    Load fresh timestamp.
5:    Plaintext ← Serialize(data and timestamp)
6:    Ciphertext ← E_{K,IV}(Plaintext)
7:    Tag ← MAC_K(Ciphertext, IV)
8:    Transmit (IV||Ciphertext||Tag)
9:    Update IV
10: end while
```

Algorithm 2 Enhanced STM Receiver

```
1: Initialize MAC_K, D_{K,IV'}
2: while true do
3:    Receive (IV'||Ciphertext'||Tag')
4:    Tag ← MAC_K(Ciphertext', IV')
5:    if Tag != Tag' then
6:       Reject message.
7:    end if
8:    Plaintext' ← D_{K,IV'}(Ciphertext')
9:    Data, Timestamp ← Deserialize(Plaintext')
10:   if Timestamp invalid then
11:      Reject message.
12:   end if
13:   Update time and pass on the data.
14: end while
```

Notice that if an authenticated encryption algorithm is used directly, the plaintext would be authenticated and encrypted in one function call in the transmitter, and the authenticity of the ciphertext would be verified and the ciphertext decrypted in one function call in the receiver.

Consider, for a moment, the proposed scheme implemented with EtM. We find that even if the adversary recovers a valid keystream through, for example, a KPA attack as described in Section 4.1.2, the adversary will *not* be capable of universal forgery as with the orignal STM. Without knowledge of the MAC key, the adversary has no way of obtaining a valid tag for the desired ciphertext, except for a brute force attack of complexity 2^{B-1} where B is the keysize. An existential forgery is possible with complexity $2^{\frac{B}{2}}$ due to the birthday problem described in Section 2.3.1, however, this gives the adversary no control over the resulting IV nor ciphertext, thus rendering the keystream obtained from the KPA useless. A replay attack is not possible because the timestamp will be deprecated upon reception.

Note that while the MD5 algorithm is still regarded as secure when used in HMAC (Bellare 2015), the Internet Engineering Task Force (IETF) issued a Request For Comment (RFC) in 2011 (Turner & Chen 2011) stating that the use of MD5 as part of HMAC is not recommended for future applications, listing SHA-2 as an alternative. Thus, while augmenting the scheme proposed by Jithish & Sankaran (2017) by not operating the block cipher in ECB mode would be quite secure, we argue that one should consider replacing MD5 with SHA-2 (for example SHA-256 or SHA-512). Additionally, the 3DES algorithm used by Jithish and Sankaran is very inefficient in software regardless of the platform it is implemented on by modern standards and should not be used.

In addition to these generic compositions, we recall from Section 2.4.2 that a block cipher may be operated in GCM and CCM mode to achieve authenticated encryption directly. Unfortunately, the polynomial multiplication over the finite field $GF(2^{128})$, required in the GCM mode, is very inefficient unless special instructions are available (for example CLMUL on $x86$), and the CCM mode requires two passes through the block cipher, providing poor performance unless hardware support for the block cipher is available (for example AES-NI on $x86$). Thus, it is of interest to see whether one can achieve increased performance on embedded devices *without* such hardware support through generic compositions of high-performance stream ciphers and MACs, or through the state-of-the-art stream cipher AEGIS (Wu & Preneel 2014) which provides authenticated encryption directly, as mentioned in Section 2.4.3.

Chapter 5

Cryptographic Algorithms and the CryptoToolbox

This chapter gives an introduction to the CryptoToolbox. This work has been presented in a paper currently under peer review. The paper can be found in Appendix A.

As seen in Section 1.2, researchers have predominantly resorted to open-source cryptographic libraries when investigating how cryptographic algorithms can be used in feedback control systems. Unfortunately, popular cryptographic libraries such as OpenSSL and pyCrypto primarily support block ciphers and asymmetric cryptographic algorithms and do not support state-of-the-art symmetric cryptographic algorithms and, in particular, high-performance stream ciphers. The wolfCrypt cryptographic library does offer some high-performance stream ciphers, notably Rabbit, HC-128, and ChaCha20, while the Crypto++ cryptographic library is the only open-source cryptographic library to support all of the eSTREAM stream ciphers, that is, Rabbit, HC-128, ChaCha20/12, and Sosemanuk. No open-source cryptographic library supports the state-of-the-art authenticated encryption algorithm AEGIS. Furthermore, the Crypto++ library, providing the widest range of algorithms, is quite difficult to navigate. Because of this, the CryptoToolbox was developed.

The CryptoToolbox contains a wide range of algorithms. Figure 5.1 illustrates the structure of the CryptoToolbox contents. In addition to the cryptographic algorithms, the CryptoToolbox provides a hexadecimal encoder. Each algorithm operates on memory buffers, and it is assumed that the data that is to be processed is contiguous in memory. This can be ensured through serialization and deserialization functions, such as described in Section 3.2.2. The algorithms operate on buffers of type `uint8_t`.

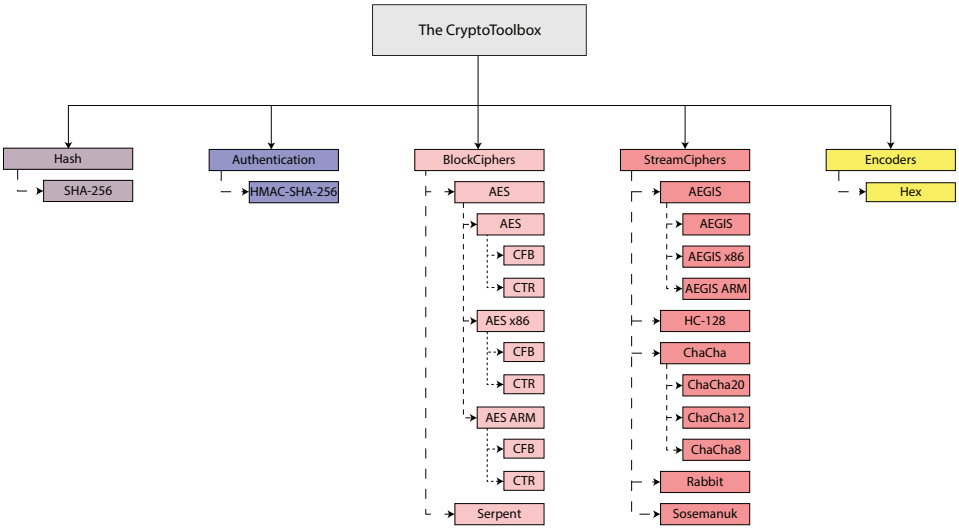

Figure 5.1: An overview of the algorithms available through the CryptoToolbox.

Note that it is the user's responsibility to supply highly randomized keys to the algorithms. For algorithms that utilize IVs and nonces, it is the user's responsibility to ensure that repeated IVs and nonces do not occur for a fixed key. Furthermore, it is assumed that the keys are pre-distributed. Users who do not adhere to these guidelines are vulnerable to security breaches.

5.1 Algorithm Implementations

In this section, we describe the algorithms that were implemented primarily from an implementation and user perspective. All the algorithms described have been made publicly available through the CryptoToolbox Github repository (Solnør 2020).

5.1.1 Advanced encryption standard

As introduced in Section 2.2.2, the AES was the result of an international effort to develop a new block cipher around the year 2000. The winner, the Rijndael cipher, was designed by Vincent Rijmen and Joan Daemen and is an SPN. Figure 5.2 illustrates the structure of the AES cipher. Note that like all block ciphers, AES is stateless. The AES cipher operates on blocks of 128 bits, thus resulting in a fixed $\{0,1\}^{128} \times \{0,1\}^{K} \mapsto \{0,1\}^{128}$ substitution parametrized by the K-bit key if operated directly. The official AES standard accepts three key sizes; 128, 192, and 256 bits, respectively. The CryptoToolbox implementations support 128-bit keys. As seen in Section 4.1.1, the direct operation of a block cipher in the ECB mode leaks structural information from the plaintext to the ciphertext. This leak is an unfortunate characteristic as shown in Section 4.1.1, and block ciphers are

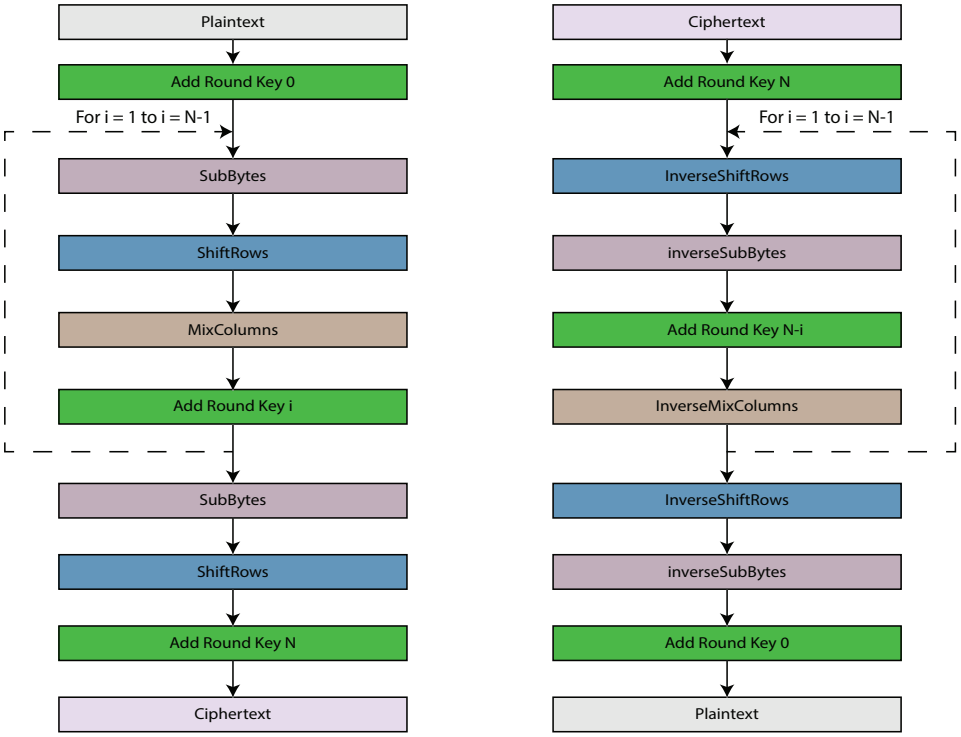

Figure 5.2: The high-level structure of AES. Left hand side illustrate the encryption mode, while the right hand side illustrates the decryption mode.

therefore primarily operated in other modes of operation such as the CBC, CFB, OFB, and CTR modes as described in Section 2.2.2. The CryptoToolbox contains implementations of AES operating in the CTR mode of operation and a modified CFB mode of operation.

Counter Mode

The CTR mode transforms the block cipher into a synchronous stream cipher by introducing a state determined by a nonce and a counter value. The nonce combined with the counter value is often referred to as the IV and serves as input to the block cipher. The output of the block cipher is called the keystream, and after each iteration, the cipher increments the counter value. The keystream is then mixed with the plaintext or ciphertext through the $\oplus$-operator to form the ciphertext or plaintext, respectively. If packets arrive out of order, or if a message is lost or injected, the transmitter and the receiver of a transmission encrypted with a synchronous stream cipher lose synchronization. The IV acts as a synchronization mechanism to provide robustness against such events. Because the IV is a public parameter it may be transmitted along with the ciphertext in the plaintext. Note that only the nonce needs to be transmitted, as the counter value can be agreed

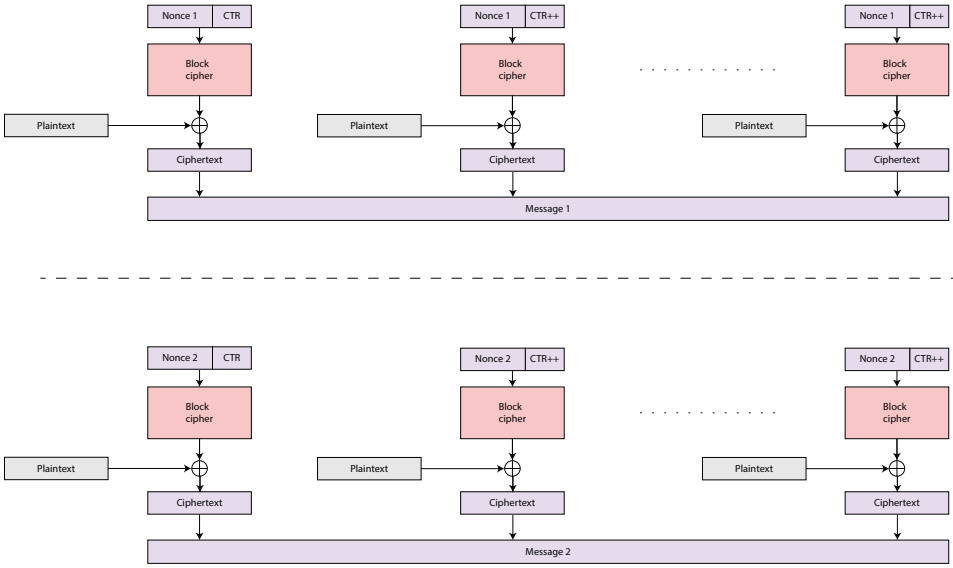

Figure 5.3: A block cipher operating in CTR mode. Notice that the initialization vector consists of a nonce and a counter. The counter is incremented each time the block cipher is iterated, and is usually initialized to a pre-determined value for each message. The nonce must be shared between the transmitter and the receiver.

upon beforehand (for example by always initializing the counter value to zero for each message). The size of the nonce and the counter value depends on the application; if small packets are transmitted at a high frequency, the nonce value is chosen to be large (for example 96 bits for AES). If large packets are transmitted less frequently, more bits can be reserved to the counter value. A common configuration for AES consists of 96 bits reserved to the nonce value and 32 bits reserved to the counter value. This is the configuration used by the CryptoToolbox implementation, and the counter value is always initialized to 1 for a new nonce. An illustration of a block cipher operating in CTR mode can be seen in Figure 5.3.

The AES CTR cipher is accessed through the interface seen in Listing 5.1.

Listing 5.1: AES CTR Interface

```
void aes_load_key(aes_state *cs, uint8_t key[16]);
void aes_load_iv(aes_state *cs, uint8_t nonce[12]);
void aes_ctr_process_packet(aes_state *cs, uint8_t *out, uint8_t *in, int
    size);
```

Note that the `aes_load_key()` function is only called once per key to derive the round keys, while the `aes_load_iv()` function is called to resynchronize the transmitter and the receiver using the public nonce, usually on a per-message basis. Both encryption and decryption is achieved through the `aes_ctr_process_packet()` function.

Cipher Feedback Mode

The AES CFB implementation was part of the TTK4550 Project Report.

As discussed in Section 2.5.2, for some applications, it may be desirable to minimize the amount of data that is to be transmitted. Because stateful ciphers often require IVs to guarantee synchronous behavior between the transmitter and the receiver, each message must carry a (unique) IV in addition to the ciphertext.

The CFB mode converts the block cipher to an SSSC by making the state uniquely determined by a finite number of ciphertext bits. By modifying the CFB mode slightly, the need for IVs can be removed by using the final ciphertext block of the previous message as the IV for the next message, thus reducing the amount of data that must be transmitted. This is referred to this as a carry-over IV design. It may be tempting to apply a similar modification to the CBC mode, but due to the nature of the CBC decryption mode, such an implementation is vulnerable to attacks as shown by the BEAST attack by (Duong & Rizzo 2011). For this reason, NIST recommends that the IVs for both CFB and CBC mode should be *unpredictable* in addition to being unique. Thus, even though no attacks are (publicly) known against this modified CFB mode, we warn that this implementation defies best-practice as defined by NIST.

Because the state is uniquely determined by a finite number of ciphertext bits, a transmission error propagates and results in burst errors. This can happen, for example, if packets are received out-of-order, if packets are injected, or if packets are lost in transmission.

An illustration of the carry-over IV CFB mode can be seen in Figure 5.4. Unlike a block cipher operating in CTR mode, a block cipher operating in CFB mode must be aware of whether is it used to encrypt or decrypt data. This is done by passing either of the pre-defined macros `ENCRYPT` and `DECRYPT` in the final function argument.

The AES CFB cipher is accessed through the interface seen in Listing 5.2.

Listing 5.2: AES CFB Interface

```
void aes_cfb_initialize(aes_state *cs, uint8_t key[16], uint8_t iv[16]);
void aes_cfb_process_packet(aes_state *cs, uint8_t *out, uint8_t *in, int
    size, int mode);
```

The cipher is only initialized once per fixed key using `aes_cfb_initialize()`, after which the `aes_cfb_process_packet()` function is used to encrypt and decrypt.

Table-Driven Round Function

The table-driven AES round function described in this subsection was implemented as part of the TTK4550 project report.

As seen in Figure 5.2, the AES cipher iterates multiple rounds of the same four operations. A byte substitution element, commonly referred to as an S-box, provides

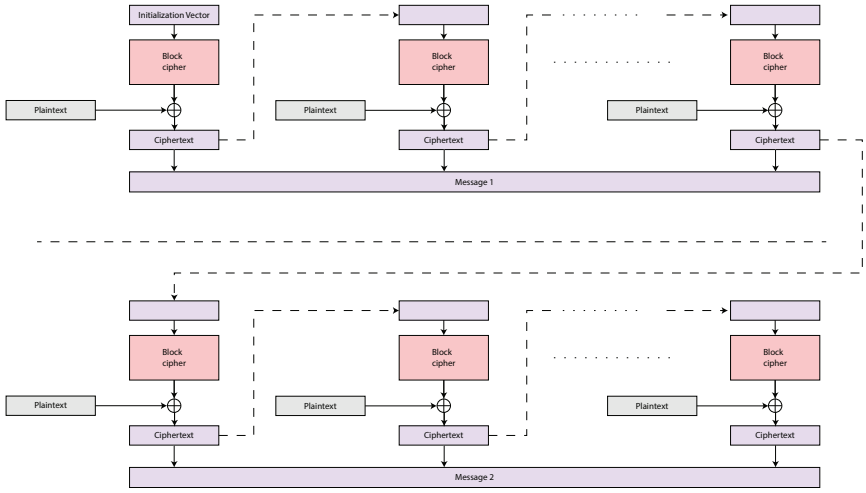

Figure 5.4: A block cipher operated in CFB mode, with a carry-over IV.

the nonlinearity. A shift row and a mix column operation provide the diffusion. Finally, a round key is added to prevent slide attacks (Biryukov & Wagner 1999). This is called the AES round function. The round keys are derived from the secret key through a key schedule. Because the byte substitution operates on bytes and consists of computationally expensive operations such as exponentiations, and because the mix column operation consists of matrix multiplications, the round function is very inefficient if implemented directly. At the very least, the byte substitution should be pre-computed and implemented as a LUT. However, because most systems today have 32 or 64-bit word sizes and because we still have to deal with the matrix multiplication step, such an implementation is not very efficient. Therefore, the CryptoToolbox implementation of the AES round function uses a time-memory trade-off in which the byte substitution, shift row, and mix column operations are pre-computed and stored in four 1 KB LUTs. As such, an iteration of the AES round function requires only 16 table lookups and 16 bitwise $\oplus$-operations. The precomputed LUTs suggested in the AES specification are used. The steps to construct these LUTs is shown below.

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} Sub[a_{0,j}] \\ Sub[a_{1,j}] \\ Sub[a_{2,j}] \\ Sub[a_{3,j}] \end{bmatrix} \tag{5.1}$$

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix} \tag{5.2}$$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} \tag{5.3}$$

Equation (5.1) shows a column-wise byte substitution, (5.2) shows how a column is changed following the shift row transformation and (5.3) shows the matrix multiplication of the mix column transformation. By substituting (5.1) into (5.2) and then substituting the resulting equation into (5.3) we get

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} Sub[a_{0,j}] \\ Sub[a_{1,j-1}] \\ Sub[a_{2,j-2}] \\ Sub[a_{2,j-3}] \end{bmatrix} \tag{5.4}$$

Equation (5.4) is then rewritten as a linear combination of the columns of the mix column matrix, yielding

$$\begin{aligned} \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = Sub[a_{0,j}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus Sub[a_{0,j-1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \\ \oplus Sub[a_{0,j-2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus Sub[a_{0,j-3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \end{aligned} \tag{5.5}$$

Equation (5.5) may now be reduced to four LUTs, T_0-T_3, containing 256 32-bit words each, that is, a total size of 4096 bytes, and three $\oplus$ operations as illustrated by

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \tag{5.6}$$

This implementation translates well to the 32-bit platform on the RevPi Connect+.

AES on x86 and ARMv8

Because of the wide adoption of AES, microprocessor manufacturers have included enhanced instruction sets that provide hardware-acceleration of the AES operations. In 2010 Intel included the Advanced Encryption Standard New Instructions (AES-NI) on their x86-processors. Advanced Micro Devices followed shortly after, and included AES-NI on their x86-processors. Later, ARM provided an optional cryptographic extension to their ARMv8-processors, the ARMv8 Crypto Extension. Therefore, rather than resorting to the table-driven variant of the AES round function described , we may instead take advantage of these enhanced instruction sets. These instructions may easily be accessed through intrinsic functions, as seen in Listing 5.3, in which the encryption mode of AES is implemented using the AES-NI intrinsics for the x86 architecture accessed through the <x86intrin.h> header.

Listing 5.3: AES-NI Intrinsics.

```
void aes_encrypt(aes_state *cs, uint32_t keystream[])
{
        __m128i B_S = _mm_loadu_si128 ((__m128i*)&cs->reg1);
        __m128i B_K0 = _mm_loadu_si128 ((__m128i*)&cs->rk);
        __m128i B_K1 = _mm_loadu_si128 ((__m128i*)&cs->rk[4]);
        __m128i B_K2 = _mm_loadu_si128 ((__m128i*)&cs->rk[8]);
        __m128i B_K3 = _mm_loadu_si128 ((__m128i*)&cs->rk[12]);
        __m128i B_K4 = _mm_loadu_si128 ((__m128i*)&cs->rk[16]);
        __m128i B_K5 = _mm_loadu_si128 ((__m128i*)&cs->rk[20]);
        __m128i B_K6 = _mm_loadu_si128 ((__m128i*)&cs->rk[24]);
        __m128i B_K7 = _mm_loadu_si128 ((__m128i*)&cs->rk[28]);
        __m128i B_K8 = _mm_loadu_si128 ((__m128i*)&cs->rk[32]);
        __m128i B_K9 = _mm_loadu_si128 ((__m128i*)&cs->rk[36]);
        __m128i B_K10 = _mm_loadu_si128 ((__m128i*)&cs->rk[40]);

        B_S = _mm_xor_si128 (B_S, B_K0);
        B_S = _mm_aesenc_si128 (B_S, B_K1);
        B_S = _mm_aesenc_si128 (B_S, B_K2);
        B_S = _mm_aesenc_si128 (B_S, B_K3);
        B_S = _mm_aesenc_si128 (B_S, B_K4);
        B_S = _mm_aesenc_si128 (B_S, B_K5);
        B_S = _mm_aesenc_si128 (B_S, B_K6);
        B_S = _mm_aesenc_si128 (B_S, B_K7);
        B_S = _mm_aesenc_si128 (B_S, B_K8);
        B_S = _mm_aesenc_si128 (B_S, B_K9);
        B_S = _mm_aesenclast_si128 (B_S, B_K10);

        _mm_storeu_si128 ((__m128i*)keystream, B_S);
}
```

On systems with a modern x86 processor with the AES-NI instruction set available, the user may compile AES CTR and AES CFB using the g++ commands seen in Listing 5.4 to take advantage of the AES-NI instructions.

Listing 5.4: AES x86 AES-NI Compilation.

```
g++ test_vectors.cpp aes_ctr.cpp ../../../../Encoders/Hex/encoder.cpp -o
    test_vectors -D x86_INTRINSICS -march=native
g++ main.cpp aes_cfb.cpp -o main -D x86_INTRINSICS -march=native
```

On systems running an `ARMv8` processor with the `ARMv8` Crypto Extension instruction set available, the user may compile AES CTR and AES CFB using the `g++` commands seen in Listing 5.5 to take advantage of the `ARMv8` Crypto Extension instructions.

Listing 5.5: AES ARMv8 Crypto Extension Compilation.

```
g++ test_vectors.cpp aes_ctr.cpp ../../../../Encoders/Hex/encoder.cpp -o
    test_vectors -D ARM_INTRINSICS -march=armv8-a+crypto
g++ main.cpp aes_cfb.cpp -o main -march=armv8-a+crypto -D ARM_INTRINSICS
```

If compiled using `CMAKE`, the preprocessor flags can be set using `add_definitions()`. Note that these hardware-accelerated variants are, in addition to being faster, less prone to side-channel attacks, that is, attacks that target the algorithm implementations rather than the algorithms themselves. An example of such a side-channel attack is the timing attack in which an adversary attempts to extract information based on the time certain operations take. For example, there may be variations in the time required to compute multiplication operations depending on the inputs, and the time needed to access lookup tables depends on where the lookup tables are stored, such as the level-1 cache or level-2 cache.

5.1.2 HC-128

The HC-128 cipher was implemented as part of the TTK4550 Project Report.

The HC-128 stream cipher was designed by Wu (2008) and rely on large permutation tables. The HC-128 cipher offers excellent performance on bulk-encryption, at the cost of a large initialization overhead. The cipher, therefore, suffers from poor performance if small packets are encrypted frequently.

The HC-128 stream cipher is accessed through the interface seen in Listing 5.6:

Listing 5.6: The HC-128 Interface.

```
void hc128_initialize(hc128_state *cs, uint8_t key[16], uint8_t iv[16]);
void hc128_process_packet(hc128_state *cs, uint8_t *output, uint8_t *input,
     uint64_t size);
```

The `hc128_initialize()` function derives an initial state from the secret key and IV by mapping the key and the IV to the tables containing the state, and iterating the cipher 1024 times. Once initialized, the `hc128_process_packet()` function is used to encrypt and decrypt.

The remarkably efficient keystream generator function of the HC-128 stream cipher can be seen in Algorithm 3. The CryptoToolbox implementation of the keystream generator function can be seen in Listing 5.7. Note that $g_{1,2}$ and $h_{1,2}$ are functions consisting only of 32-bit rotations, modular additions, and bitwise $\oplus$-operations, while P and Q denote the tables that make up the state of the cipher.

Listing 5.7: The HC-128 Keystream Generator Function.

```
void hc128_generate_keystream(hc128_state *cs, uint32_t *keystream,
    uint64_t size)
{
  // Generate keystream
  for (int i = 0; i <= (size-1)/4; i++)
  {
    int j = (i&0x1FF);
    if ( (i&0x3FF) < 512 )
    {
      // Operate on P
      cs->P[j] = cs->P[j] + g1(cs->P[(j-3)&0x1FF],
          cs->P[(j-10)&0x1FF],
          cs->P[(j-511)&0x1FF]);
      *keystream = h1(cs, cs->P[(j-12)&0x1FF]) ^ (cs->P[j]);
      keystream++;
    } else {
      // Operate on Q
      cs->Q[j] = cs->Q[j] + g2(cs->Q[(j-3)&0x1FF],
          cs->Q[(j-10)&0x1FF],
          cs->Q[(j-511)&0x1FF]);
      *keystream = h2(cs, cs->Q[(j-12)&0x1FF]) ^ (cs->Q[j]);
      keystream++;
    }
  }
}
```

5.1.3 Sosemanuk

The Sosemanuk cipher was implemented as part of the TTK4550 Project Report.

The Sosemanuk stream cipher was the result of a cooperative effort between multiple French cryptologists and was submitted by Berbain et al. (2008) to the eSTREAM competition. The Sosemanuk stream cipher consists of a linear feedback shift register composed with a nonlinear output function. The nonlinear output function is constructed using components from the Serpent block cipher designed by Anderson et al. (2000), which was the runner-up submission to the AES-process. An overview of the Sosemanuk cipher can be seen in Figure 5.5. The Sosemanuk cipher is accessed through the interface seen in Listing 5.8.

Listing 5.8: The Sosemanuk Interface.

```
void sosemanuk_load_key(sosemanuk_state *cs, uint8_t *key, int keysize);
```

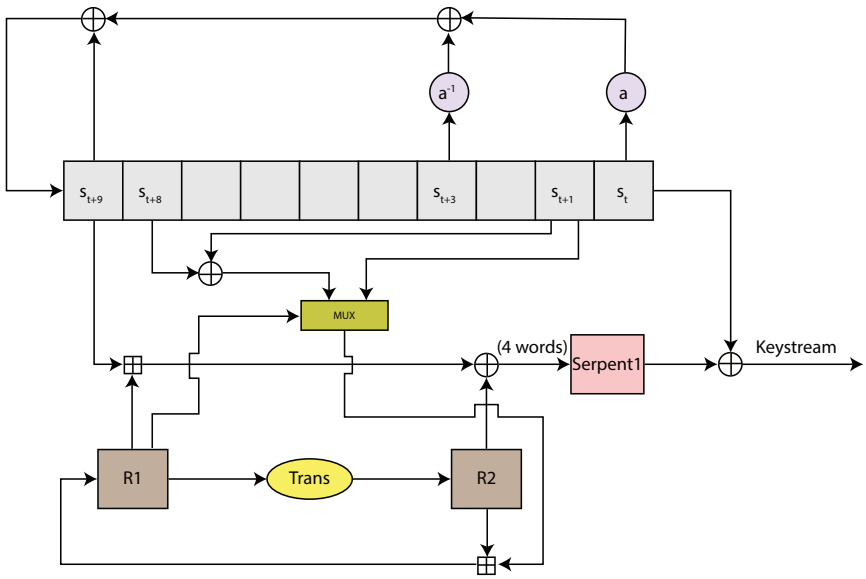

Figure 5.5: An overview of the Sosemanuk stream cipher.

```
void sosemanuk_load_iv(sosemanuk_state *cs, uint8_t iv[16]);
void sosemanuk_process_packet(sosemanuk_state *cs, uint8_t *out, uint8_t *
    in, uint64_t size);
```

The `sosemanuk_load_key()` function is called once per key, while the `sosemanuk_load_iv()` function is called to resynchronize the transmitter and the receiver by deducing an initial state of the cipher from the pre-loaded key and an IV. This is usually done on a per-message basis. Encryption and decryption is achieved through the `sosemanuk_process_packet()` function.

Algorithm 3 HC-128 keystream generation algorithm

1: $i = 0$
2: **while** more keystream bits are required **do**
3: $j = i \bmod 512$
4: **if** $(i \bmod 1024) < 512$ **then**
5: $P[j] = P[j] + g_1(P[j \boxminus 3], P[j \boxminus 10], P[j \boxminus 511])$
6: $s_i = h_1(P[j \boxminus 12]) \oplus P[j]$
7: **else**
8: $Q[j] = Q[j] + g_2(Q[j \boxminus 3], Q[j \boxminus 10], Q[j \boxminus 511])$
9: $s_i = h_2(Q[j \boxminus 12]) \oplus Q[j]$
10: **end if**
11: $i = i + 1$
12: **end while**

Serpent

The Serpent block cipher is constructed as a SPN like AES. As in AES, the non-linear component of the cipher consists of S-boxes. However, because the Serpent S-boxes are $\{0,1\}^4 \mapsto \{0,1\}^4$ mappings, they do not lend themselves well to LUT implementations. Instead, a bit-slicing technique may be applied. In the CryptoToolbox implementation, the bit-slicing techniques proposed by Osvik (2000) are used to implement the Serpent S-boxes. The Serpent block cipher is accessed indirectly through the Sosemanuk function calls, and it is noted that only the parts used in the Sosemanuk cipher are implemented. The Serpent block cipher is therefore not available as a stand-alone cipher. An implementation of a bit-sliced Osvik S-box used in the Serpent cipher can be seen in Listing 5.9.

Listing 5.9: A Bitsliced Osvik S-Box for the Serpent Block Cipher.

```
inline void S4(uint32_t *r0, uint32_t *r1, uint32_t *r2, uint32_t *r3,
    uint32_t *r4)
{
       *r1 ^= *r3; *r3 = ~(*r3);
       *r2 ^= *r3; *r3 ^= *r0;
       *r4 = *r1; *r1 &= *r3;
       *r1 ^= *r2; *r4 ^= *r3;
       *r0 ^= *r4; *r2 &= *r4;
       *r2 ^= *r0; *r0 &= *r1;
       *r3 ^= *r0; *r4 |= *r1;
       *r4 ^= *r0; *r0 |= *r3;
       *r0 ^= *r2; *r2 &= *r3;
       *r0 = ~(*r0); *r4 ^= *r2;
}
```

5.1.4 Rabbit

The Rabbit stream cipher is a cipher designed by Boesgaard et al. (2008) that was a successful entrant to the eSTREAM competition. The theoretical foundation of the Rabbit cipher comes from the theory of chaotic systems. The cipher deduces a secret *master state* from the key, and each IV is mixed with the master state to produce an initial state of the cipher. The master state can also be used directly, that is, without an IV. However, if an IV is not used, any packet losses or packet injections will result in a loss of synchronization.

The Rabbit stream cipher is accessed through the interface seen in Listing 5.10.

Listing 5.10: The Rabbit Interface.

```
void rabbit_load_key(rabbit_state *cs, uint8_t key[16]);
void rabbit_load_iv(rabbit_state *cs, uint8_t iv[8]);
void rabbit_process_packet(rabbit_state *cs, uint8_t *output, uint8_t *
    input, uint64_t size);
```

The `rabbit_load_key()` function deduces the master state, and is called once per key. The `rabbit_load_iv()` function derives an initial state from the master state and a public IV. This is usually done on a per-message basis. The `rabbit_process_packet()` function is used to encrypt and decrypt.

5.1.5 ChaCha

The ChaCha stream cipher is a variant of the Salsa-family of stream ciphers and was designed by Bernstein (2008). The ChaCha stream cipher follows an ARX-design, and is generally used in three configurations; the full cipher consisting of twenty rounds (ChaCha20), a round reduced variant consisting of twelve rounds (ChaCha20/12), and a further round reduced variant consisting of eight rounds (ChaCha20/8). The round reduced variants offer increased performance at the cost of reduced security. The CryptoToolbox provides the full ChaCha20 cipher as default, however, the round reduced variants may be accessed by passing the `-D TWELVE_ROUNDS` and `-D EIGHT_ROUNDS` preprocessor flags for the twelve and eight round variants, respectively, as seen in Listing 5.11:

Listing 5.11: The ChaCha Compilation Options.

```
g++ main.cpp chacha.cpp ../../Encoders/Hex/encoder.cpp -o main
g++ main.cpp chacha.cpp ../../Encoders/Hex/encoder.cpp -o main -D TWELVE_
    ROUNDS
g++ main.cpp chacha.cpp ../../Encoders/Hex/encoder.cpp -o main -D EIGHT_
    ROUNDS
```

All variants of the ChaCha stream cipher are accessed through the interface seen in Listing 5.12.

Listing 5.12: The ChaCha Interface.

```
void chacha_initialize(chacha_state *cs, uint8_t key[32], uint8_t nonce
    [12]);
void chacha_process_packet(chacha_state *cs, uint8_t *output, uint8_t *
    input, uint64_t size);
```

The `chacha_initialize()` function is used to derive an initial state from the secret key and the public IV, normally on a per-message basis, after which `chacha_process_packet()` is used to encrypt and decrypt.

The core of the ChaCha stream cipher revolves around the quarter-round function shown in Listing 5.13. Notice that only modular additions, 32-bit rotations and bitwise $\oplus$-operations are used.

Listing 5.13: The ChaCha Quarter-Round Function.

```
inline void q_round(chacha_state *cs, int a, int b, int c, int d){

  cs->state[a] += cs->state[b];
  cs->state[d] ^= cs->state[a];
```

```
  cs->state[d] = ROTL_32((cs->state[d]), 16);

  cs->state[c] += cs->state[d];
  cs->state[b] ^= cs->state[c];
  cs->state[b] = ROTL_32((cs->state[b]), 12);

  cs->state[a] += cs->state[b];
  cs->state[d] ^= cs->state[a];
  cs->state[d] = ROTL_32((cs->state[d]), 8);

  cs->state[c] += cs->state[d];
  cs->state[b] ^= cs->state[c];
  cs->state[b] = ROTL_32((cs->state[b]), 7);
}
```

5.1.6 AEGIS

Briefly mentioned in Section 2.4.3 and Section 4.2, the AEGIS stream cipher was designed by Wu & Preneel (2014) and submitted to the Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR). The AEGIS stream cipher is a synchronous stream cipher that is heavily based on the AES round function and provides authenticated encryption directly. Note that AEGIS also can be used to provide message authenticity without encryption or to authenticate additional data that is not encrypted. The latter is commonly used to authenticate the IV in plaintext, in addition to the ciphertext. The AEGIS stream cipher is accessed through the interface displayed in Listing 5.14.

Listing 5.14: The AEGIS Interface.

```
void aegis_load_key(aegis_state *cs, uint8_t key[16]);
void aegis_encrypt_packet(aegis_state *cs, uint8_t *ct, uint8_t tag[16],
    uint8_t *pt, uint8_t *ad, uint8_t iv[16], uint64_t adlen, uint64_t
    msglen);
int aegis_decrypt_packet(aegis_state *cs, uint8_t *pt, uint8_t *ct, uint8_t
     *ad, uint8_t iv[16], uint8_t tag[16], uint64_t adlen, uint64_t msglen)
    ;
```

The `aegis_load_key()` function is called once per key, while the `aegis_encrypt_packet()` and `aegis_decrypt_packet()` functions are used to encrypt and decrypt, respectively. Note that the `aegis_decrypt_packet()`-function returns 1 if the (message, tag)-pair is valid and 0 otherwise. If the (message, tag)-pair is invalid, the pt-buffer and the tag-buffer are zeroized. This is done to prevent CCA attacks.

AEGIS on `x86` and `ARMv8`

Because the AEGIS stream cipher utilizes AES operations, the cipher can take advantage of the enhanced instruction sets provided by some modern microproces-

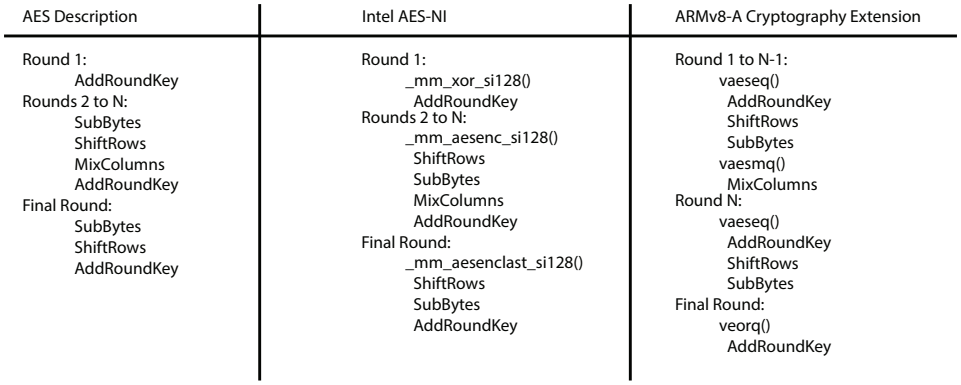

Figure 5.6: An illustration of the AES description, the AES-NI operations and the ARMv8 cryptography extension operations. The difference between the AES-NI and ARMv8 cryptography extension round function means that extra operations are required when using ARM hardware-acceleration to implement AEGIS. This figure is based on a figure from Crutchfield (2014).

sors. The CryptoToolbox provides enhanced implementations for `x86` and `ARMv8` processors featuring AES-NI and `ARMv8` Crypto Extension, respectively.

On systems running an `x86` processor with the AES-NI instruction set available, AEGIS is compiled using the `g++` command seen in Listing 5.15:

Listing 5.15: AEGIS x86 AES-NI Compilation.

```
g++ test_vectors.cpp aegis_128.cpp ../../../Encoders/Hex/encoder.cpp -o test
    _vectors -D x86_INTRINSICS -march=native
```

On systems running an `ARMv8` processor with the `ARMv8` Crypto Extension instruction set available, AEGIS is compiled using the `g++` command seen in Listing 5.16:

Listing 5.16: AEGIS ARMv8 Crypto Extension Compilation.

```
g++ test_vectors.cpp aegis_128.cpp ../../../Encoders/Hex/encoder.cpp -o test
    _vectors -march=armv8-a+crypto -D ARM_INTRINSICS
```

Notice in Figure 5.6, however, that the ARMv8 Cryptography Extension intrinsic functions are not perfectly aligned with the 'true' AES round function. Since AEGIS only utilizes the 'true' AES round function and not the first and last rounds, the round keys must be pre- and post-added. An excerpt from the ARMv8 AEGIS implementation in the CryptoToolbox illustrates this in Listing 5.17.

Listing 5.17: Reconstruction of AES Round using ARMv8 Intrinsics.

```
#ifdef ARM_INTRINSICS
  // ARM INTRINSICS
  B_S3 = veorq_u8(B_S3, B_KEY);
  B_S3 = vaesmcq_u8(vaeseq_u8(B_S3, B_KEY));
```

```
  B_S3 = veorq_u8(B_S3, B_KEY);
  B_TMP = B_KEY;
  vst1q_u8((uint8_t*)cs->s3, B_S3);
#else
```

5.1.7 Keyed-hash message authentication code

The HMAC algorithm was implemented as part of the TTK4550 Project Report.

In addition to the cipher implementations, the HMAC message authentication algorithm was implemented as defined by NIST(Dang 2008). The HMAC algorithm constructs a *keyed* MAC from an *unkeyed* cryptographic hash function. To do this, the HMAC algorithm requires a secret key of variable length. If the secret key is *shorter* than 512 bits, the key is expanded by appending 0-bits until the key length is 512 bits. If the secret key is *longer* than 512 bits the key is hashed with the associated hashing function, and the output, the digest, is appended with 0-bits until it is of length 512 bits and used as the key.

From the 512-bit key K, an *inner key* K_i and an *outer key* K_o are computed according to

$$K_i = K \oplus \text{ipad} \tag{5.7}$$
$$K_o = K \oplus \text{opad} \tag{5.8}$$

where ipad and opad are 512-bit patterns obtained by repeating the bit patterns given by the hexadecimal numbers $0x36$ and $0x5C$, respectively. These keys may be pre-computed and stored, or computed on-the-fly when required.

An inner digest is obtained by computing the hash of the inner key prepended to the message. The outer key is then prepended to the inner digest and fed back into the hashing algorithm, and the output constitutes the MAC tag. An illustration of the HMAC algorithm is given by Figure 5.7. Notice that only the digest and the outer key are processed in the final hashing, thus the HMAC algorithm only passes through the data once. In the implementation, the key, the inner key and the outer key are held in a `hmac_state`-struct. The implementation is accessed through the interface seen in Listing 5.18.

Listing 5.18: The HMAC Interface

```
void hmac_load_key(hmac_state *cs, uint8_t *key, int keysize);
void hmac_tag_generation(hmac_state *cs, uint8_t* tag, uint8_t *message,
    uint64_t dataLength, int tagSize);
int hmac_tag_validation(hmac_state *cs, uint8_t *tag, uint8_t *message,
    uint64_t dataLength, int tagSize);
```

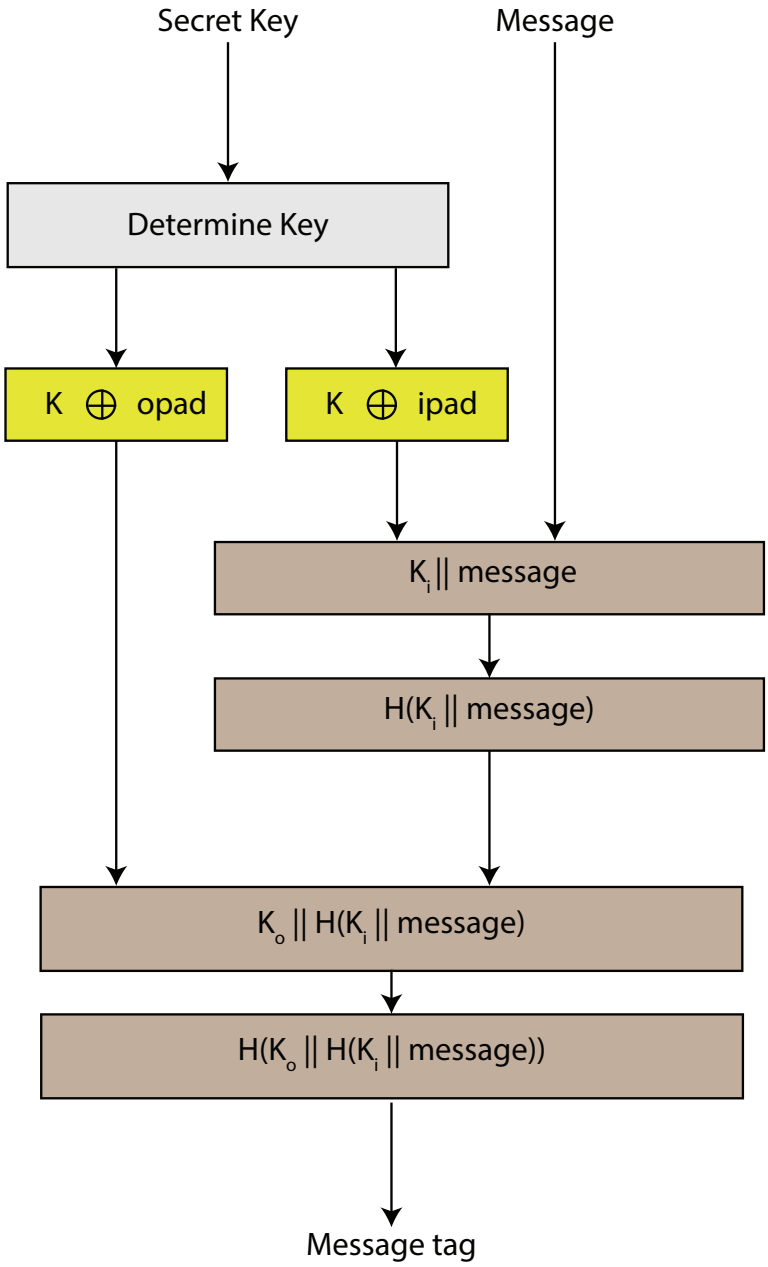

Figure 5.7: The HMAC tag generation algorithm. Based on figure from (Dang 2008).

The `hmac_load_key()`-function stores the key in the struct and computes the inner key and the outer key according to (5.7) and (5.8), respectively. The `hmac_tag`

_generation()-function computes a tag for a message, and the hmac_tag_validation()-function validates or invalidates a (message, tag)-pair. A full example is also available in the CryptoToolbox repository. The HMAC implementation was verified in conjunction with SHA-256 using test vectors from IETF[1]. The SHA-256 algorithm is described in the following section.

SHA-256

The cryptographic hash function used in the HMAC algorithm described above was the SHA-256. The decision to go with the SHA-256 algorithm over the SHA-3 algorithm (National Institute of Standards and Technology 2015) was due to the reported higher efficiency of the SHA-256 algorithm.

The SHA-256 algorithm is a NIST-certified cryptographic hashing algorithm, specified in (Dang 2015). The SHA-256 algorithm is a *Davies-Meyer* construction consisting of repeated iterations of a block cipher with a block size of 256 bits and a key size of 512 bits, i.e., $F : \{0,1\}^{256} \times \{0,1\}^{512} \mapsto \{0,1\}^{256}$.

During the pre-processing stage the message is padded to a form a multiple of the 512-bit key size of the block cipher, after which the message is parsed into 512-bit message blocks, held in sixteen 32-bit words. Finally, the 256-bit initial hash value is set.

Following the pre-processing stage, the hash computation takes place. The SHA-256 hash function is based on an ARX-like block cipher, where the ARX-structure is as described by Section 2.2.2. Each iteration consists of four stages, and may be described by Algorithm 4. A round of the SHA-256 block cipher is illustrated in Figure 5.8 where Ch, Maj, Σ_0 and Σ_1 are given by

$$\mathrm{Ch}(x,y,z) = (x \,\&\, y) \oplus (\neg x \,\&\, z) \tag{5.9}$$

$$\mathrm{Maj}(x,y,z) = (x \,\&\, y) \oplus (x \,\&\, z) \oplus (y \,\&\, z) \tag{5.10}$$

$$\Sigma_0(x) = (x >>> 2) \oplus (x >>> 13) \oplus (x >>> 22) \tag{5.11}$$

$$\Sigma_1(x) = (x >>> 6) \oplus (x >>> 11) \oplus (x >>> 25) \tag{5.12}$$

, respectively. K_i denotes a round constant, and additions are performed in $GF(2^{32})$.

Note that the *message schedule* plays the role of the *key schedule* of a typical block cipher, and the working variables play the role of the input and output blocks of the block cipher. Thus, the message is processed by consuming key material at a rate of 512 bits per iteration. The unkeyed SHA-256 algorithm is also accessible through the interface displayed in Listing 5.19. Note that an unkeyed cryptographic hash function *should not* be used to provide message authenticity and integrity directly, as it is easy to run into the problems that made the STM vulnerable, as shown in Section 4.1.2.

[1] HMAC-SHA-256 test vectors can be found here: https://tools.ietf.org/html/rfc4231

Listing 5.19: SHA-256 Interface

```
void sha256_process_message(uint8_t *digest, uint8_t *message, uint64_t
    size);
```

Algorithm 4 SHA-256 Iteration Steps

1: Prepare the message schedule.
2: Initialize the eight working variables, (**a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**), with the previous hash.
3: Iterate through 64 rounds of the ARX network.
4: Compute the hash value.

5.2 Hexadecimal Encoding

In addition to the cryptographic algorithms, the CryptoToolbox contains a hexadecimal encoder. The hexadecimal encoder is useful for converting the output of the cryptographic algorithms into a printable format. Because the algorithms operate on buffers of type `uint8_t`, each byte represents a number in the interval [0, 255]. However, only numbers in the interval [32, 255] represent printable characters, some of which are unintelligible. The hexadecimal encoder abates this problem by interpreting each byte as a hexadecimal number. The CryptoToolbox also provides a decoder that interprets a buffer of hexadecimal numbers as `uint8_t`. The decoder is generally used in scenarios in which correctly formatted input is needed to confirm the correct operation of an algorithm with official test vectors.

The hexadecimal encoder from the CryptoToolbox was used in Table 4.1 to show how a block cipher in ECB mode fails to provide confidentiality. The interfaces for the hexadecimal encoder and decoder are displayed in Listing 5.20.

Listing 5.20: The Hexadecimal Encoder and Decoder Interfaces.

```
void hex_encode(char* output, const uint8_t* input, int size);
void hex_decode(uint8_t* output, const char* input, int size);
```

5.3 Applications

We proceed by showing how the cryptographic algorithms may be used to perform encryption, authentication, and authenticated encryption. Pseudocode is listed for the data loading, data transmission, data reception, and the acceptance interface.

5.3.1 Encryption using Rabbit

We begin by showing how data may be encrypted using the Rabbit stream cipher in Listing 5.21. We assume that the data that is to be encrypted is contiguous in

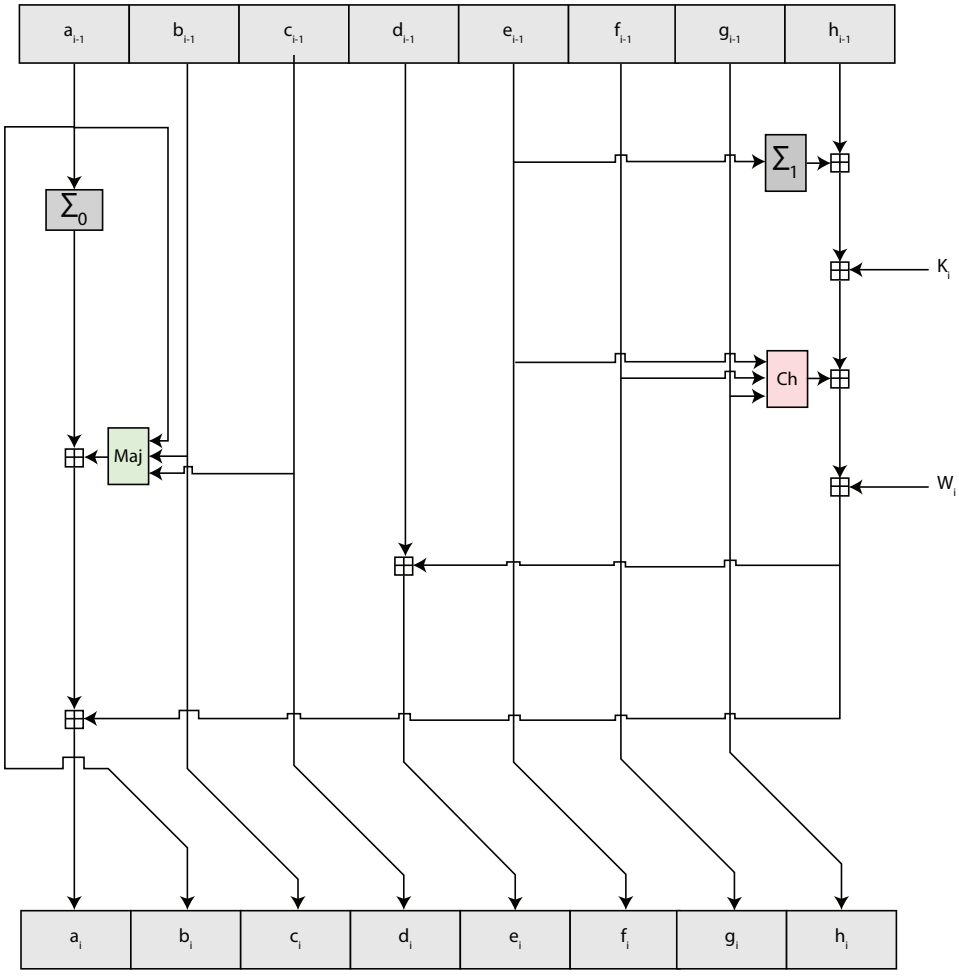

Figure 5.8: An illustration of the ARX-like structure of the SHA-256 block cipher round. Based on figure from (Sanadhya & Sarkar 2008).

memory.

Listing 5.21: Rabbit encryption example

```
#include "rabbit.h"
#include <cstring> // for memcpy

int main()
{

  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
```

```
  uint8_t iv[8] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */


  /*One buffer for plaintext and one
    buffer for the ciphertext and IV*/
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE];

  while(1)
  {
    /* Get new data */
    plaintext <- LoadData();

    /* RABBIT ENCRYPT */
    std::memcpy(message, iv, 8);
    rabbit_load_iv(&cs, iv);
    rabbit_process_packet(&cs, &message[8], plaintext, DATA_SIZE);
    (*(uint64_t*)iv)++;
    /* ENCRYPT FINISHED */

    /* Transmit (IV || Ciphertext) */
    Transmit(message);
  }
}
```

Notice that from the user perspective, the `rabbit.h` header file must be included to gain access to the Rabbit cipher. In addition, the `rabbit.cpp` file must be included during compilation. Notice that some setup is required initially to instantiate a cipher struct, the key, and the initial IV. A message-specific initial state is then deduced using a **unique** IV, which is transmitted with the message. The estimates may then be encrypted, after which the IV must be incremented. The 64-bit IV ensures that a total of 2^{64} messages may be sent per key if a counter is used. If chosen at random, less than 2^{32} messages should be sent per key due to the birthday paradox.

On the receiving end, the operations are very similar. The key must be the same as on the transmitting end, while the message-specific IV is extracted from the message to ensure synchronous behavior between the transmitter and the receiver. The corresponding decryption operations can be seen in Listing 5.22, assuming we receive the message buffer from Listing 5.21.

Listing 5.22: Rabbit decryption example

```
#include "rabbit.h"

int main()
{
```

```
  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */

  /*One buffer for plaintext and one
    buffer for the ciphertext and IV*/
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE];

  while(1)
  {
    /* Receive message (IV || Ciphertext) */
    message <- Receiver();

    /* RABBIT DECRYPT */
    rabbit_load_iv(&cs, message);
    rabbit_process_packet(&cs, plaintext, &message[8], DATA_SIZE);
    /* DECRYPT FINISHED */

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

The procedure is very similar to all the other encryption algorithms. A notable exception is the AES CFB implementation, which is self-synchronizing and does not require IVs. If the AES CFB algorithm is used, the message buffer would be of size `DATA_SIZE`.

5.3.2 Authentication and verification using HMAC-SHA-256

We proceed by showing how HMAC-SHA-256 may be used to ensure cryptographic message authenticity in Listing 5.23.

Listing 5.23: HMAC-SHA-256 authentication

```
#include "hmac.h"
#include <cstring> // for memcpy

int main()
{

  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
```

```
  /* SETUP FINISHED */

  /* One buffer holds the plaintext,
     and the other holds the plaintext
     and the 32-byte tag. */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[DATA_SIZE+32];

  while(1)
  {
    /* Get new data */
    plaintext <- LoadData();

    /* COMPUTE TAG */
    std::memcpy(message, plaintext, DATA_SIZE);
    hmac_tag_generation(&as, &message[DATA_SIZE], plaintext, DATA_SIZE, 32);
    /* TAG GENERATION FINISHED */

    /* Transmit (Plaintext || Tag) */
    Transmit(message);
  }
}
```

The authenticated message generated by the function calls above may then be validated using the function calls shown in Listing 5.24.

Listing 5.24: HMAC-SHA-256 validation

```
#include "hmac.h"

int main()
{
  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
  /* SETUP FINISHED */

  /* One buffer holds the plaintext,
     and the other holds the plaintext
     and the 32-byte tag. */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[DATA_SIZE+32];

  while(1)
  {
    /* Receive message (Plaintext || Tag) */
    message <- Receiver();

    /* HMAC-SHA-256 VALIDATE MESSAGE */
```

```
    if (!hmac_tag_validation(&as, &message[DATA_SIZE], message, DATA_SIZE,
        32)){
      /* TAG IS INVALID */
      continue;
    } else {
      std::memcpy(plaintext, message, DATA_SIZE);
    }
    /* MESSAGE VALIDATION FINISHED */

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

Note that you also have to compile add the `sha-256.cpp` file to the compilation when using HMAC-SHA-256.

5.3.3 Authenticated encryption using AEGIS

Finally, we show how authenticated encryption may be obtained. An example of using the authenticated encryption algorithm AEGIS is presented, although the generic EtM composition of any of the other encryption algorithms and the HMAC-SHA-256 MAC may also be used. In Listing 5.25, the code to encrypt and authenticate a message is shown.

Listing 5.25: AEGIS encryption and authentication

```
#include "aegis_128.h"
#include <cstring> // for memcpy

int main()
{
  /* AEGIS SETUP */
  aegis_state cs;
  uint8_t key[16] = {0};
  uint8_t iv[16] = {0};
  aegis_load_key(&cs, key);
  /* SETUP FINISHED */

  /* Buffer for plaintext and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[16+DATA_SIZE+16];

  while(1)
  {
    /* Get new data*/
    plaintext <- LoadData();
```

```
    /* AEGIS ENCRYPT & AUTHENTICATE */
    std::memcpy(message, iv, 16);
    aegis_encrypt_packet(&cs, &message[16], &message[16+DATA_SIZE],
        plaintext, iv, iv, 16, DATA_SIZE);
    (*(uint64_t*)iv)++;

    /*Transmit (IV || Ciphertext || Tag) */
    Transmit(message);
  }
}
```

The same message can be validated and decrypted by the code seen in Listing 5.26. If the message is invalidated, the corresponding plaintext and tag are zeroized, and a '0' is returned.

Listing 5.26: AEGIS validation and decryption

```
#include "aegis_128.h"

int main()
{
  /* AEGIS SETUP */
  aegis_state cs;
  uint8_t key[16] = {0};
  aegis_load_key(&cs, key);
  /* SETUP FINISHED */

  /* Buffer for plaintext and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[16+DATA_SIZE+16];

  while(1)
  {
    /* Receive message */
    message <- Receiver();

    /* AEGIS VALIDATE & DECRYPT */
    if (!aegis_decrypt_packet(&cs, plaintext, &message[16], message, message
        , &message[16+DATA_SIZE], 16, DATA_SIZE))
    {
      // Invalid msg
      continue;
    }
    /*COMPLETED*/

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

We may now use these as components to establish secure transmission of signals between the components of a feedback control system.

Chapter 6

Implementing Secure Signal Transmission in Feedback Control Systems

After implementing the cryptographic algorithms and the CryptoToolbox, we may now achieve secure signal transmission in feedback control systems by implementing the enhanced STM transmitter proposed in Algorithm 1 and the enhanced STM receiver proposed in Algorithm 2. We adopt the serialization tool we introduced in Section 3.2.2. We will show how the the schemes may be implemented both through the generic EtM composition, and through the authenticated encryption algorithm AEGIS.

6.1 Secure transmission using Encrypt-then-MAC

We first illustrate how the proposed transmitter algorithm, Algorithm 1, may be implemented using an EtM composition of the HC-128 cipher and the HMAC-SHA-256 MAC.

6.1.1 Transmitter

We first implement the transmitter described in Algorithm 1. Since we now have all the tools we need, we merely replace the pseudocode with proper function calls. The code to implement the secure transmitter can be seen in Listing 6.2.

Listing 6.1: The enhanced STM transmitter using an EtM composition.

```
#include "hc128.h"
#include "hmac.h"
```

```
#include <cstring>
#include <chrono>

int main()
{
    /* PREPARE HC-128 */
    hc128_state cs;
    uint8_t e_key[16] = {0};
    uint8_t iv[16] = {0};
    /* HC-128 READY */

    /* PREPARE HMAC */
    hmac_state as;
    uint8_t a_key[32] = {0};
    hmac_load_key(&as, a_key, 32);
    /* HMAC READY */

    /* Struct to hold data */
    data_struct data;

    /* Buffers to hold serialized data and the
       final message (IV || Ciphertext || Tag). */
    uint8_t serialized_data[DATA_SIZE];
    uint8_t message[16+DATA_SIZE+32];

    /* Operational loop */
    while(1)
    {
        /* New data to transmit */
        data.load <- DataAcquisition();

        /* Fresh timestamp */
        data.timestamp = std::chrono::system_clock::now();

        /* Serialize */
        serialize(&data, serialized_data);

        /* Encrypt */
        std::memcpy(message, iv, 16);
        hc128_initialize(&cs, e_key, iv);
        hc128_process_packet(&cs, &message[16], serialized_data, DATA_SIZE);
        (*(uint64_t*)iv)++;

        /* Authenticate - generate 256-bit tag */
        hmac_tag_generation(&as, &message[16+DATA_SIZE], message, 16+
            DATA_SIZE, 32);

        /* Transmission */
        Transmit(message);
    }
```

```
}
```

Note that only pseudocode is given for the data acquisition step and the transmission step. This scheme is agnostic to what data is being transmitted, and how it is transmitted, whether it be directly through sockets or through some abstract interface.

6.1.2 Receiver

We proceed by showing how the corresponding secure receiver may be implemented. The code to implement the secure receiver can be seen in Listing 6.2. Pseudocode is given for the reception and acceptance interfaces.

Listing 6.2: The enhanced STM transmitter using an EtM composition.

```
#include "hc128.h"
#include "hmac.h"
#include <cstring>
#include <chrono>

int main()
{
    /* PREPARE HC-128 */
    hc128_state cs;
    uint8_t e_key[16] = {0};
    uint8_t iv[16] = {0};
    /* HC-128 READY */

    /* PREPARE HMAC */
    hmac_state as;
    uint8_t a_key[32] = {0};
    hmac_load_key(&as, a_key, 32);
    /* HMAC READY */

    /* Struct to hold data */
    data_struct data;

    /* Buffers to hold serialized data
       and the final message. */
    uint8_t serialized_data[DATA_SIZE];
    uint8_t message[16+DATA_SIZE+32];

    /* Time point */
    std::chrono::system_clock::time_point prev_timestamp = std::chrono::
        system_clock::from_time_t(0);

    /* Operational loop */
    while(1)
    {
```

```
        /* Receive data */
        message <- Receive();

        /* Verify that authenticity of data */
        if (!hmac_tag_validation(&as, &message[16+DATA_SIZE], message, 16+
            DATA_SIZE, 32))
        {
            /* Invalid message */
            continue;
        }

        /* Decrypt */
        hc128_initialize(&cs, e_key, message);
        hc128_process_packet(&cs, serialized_data, &message[IV_SIZE],
            DATA_SIZE);

        /* Deserialize */
        deserialize(serialized_data, &data);

        /* Check timestamp */
        if(!(data.timestamp > prev_timestamp))
        {
            /* Invalid message */
            continue;
        }
        prev_timestamp = data.timestamp;

        /* Pass on the data */
        Accept(data.load);
    }
}
```

6.2 Secure transmission using AEGIS

Instead of using an EtM composition, we may replace HC-128 and HMAC-SHA-256 with an authenticated encryption algorithm directly.

6.2.1 Transmitter

The transmitter would look slightly different when an authenticated encryption algorithm is used directly. An implementation of the transmitter implemented with AEGIS can be seen in Listing 6.3.

Listing 6.3: The enhanced STM transmitter implemented using AEGIS.

```
#include "aegis_128.h"
#include <cstring>
#include <chrono>
```

```
int main()
{
    /* PREPARE AEGIS */
    aegis_state cs;
    uint8_t key[16] = {0};
    uint8_t iv[16] = {0};
    aegis_load_key(&cs, key);

    /* Struct to hold data */
    data_struct data;

    /* Buffers to hold serialized data
       and the final message. */
    uint8_t serialized_data[DATA_SIZE];
    uint8_t message[16+DATA_SIZE+16];

    /* Operational loop */
    while(1)
    {
        /* New data to transmit */
        data.load <- DataAcquisition();

        /* Fresh timestamp */
        data.timestamp = std::chrono::system_clock::now();

        /* Serialize */
        serialize(&data, serialized_data);

        /* Encrypt-and-Authenticate */
        std::memcpy(message, iv, 16);
        aegis_encrypt_packet(&cs, &message[16],
          &message[16+DATA_SIZE], serialized_data,
          iv, iv, 16, DATA_SIZE);
        (*(uint64_t*)iv)++;

        /* Transmission */
        Transmit(data);
    }
}
```

6.2.2 Receiver

Similarly the receiver would be adapted, as seen in Listing 6.4.

Listing 6.4: The enhanced STM transmitter implemented using AEGIS.

```
#include "aegis_128.h"
#include <cstring>
#include <chrono>
```

```
int main()
{
    /* PREPARE AEGIS */
    aegis_state cs;
    uint8_t key[16] = {0};
    uint8_t iv[16] = {0};
    aegis_load_key(&cs, key);

    /* Struct to hold data */
    data_struct data;

    /* Buffers to hold serialized data
       and the final message. */
    uint8_t serialized_data[DATA_SIZE];
    uint8_t message[16+DATA_SIZE+16];

    /* Time point */
    std::chrono::system_clock::time_point prev_timestamp = std::chrono::
        system_clock::from_time_t(0);

    /* Operational loop */
    while(1)
    {
        /* Receive data */
        message <- Receive();

        /* Decrypt and verify */
        if(!aegis_decrypt_packet(&cs, serialized_data, &message[16], message
            , message, &message[16+DATA_SIZE], 16, DATA_SIZE))
        {
            // Invalid message
            continue;
        }

        /* Deserialize */
        deserialize(serialized_data, &data);

        /* Check timestamp */
        if(!(data.timestamp > prev_timestamp))
        {
            /* Invalid message */
            continue;
        }
        prev_timestamp = data.timestamp;

        /* Pass on the data */
        Accept(data.load);
    }
}
```

Chapter 7

Practical Experiments and Verification

In this section, the algorithm implementations described in Chapter 5 are benchmarked in proper authenticated encryption schemes as proposed in Section 4.2. The experiments are conducted in the encryption laboratory described in Chapter 3.

7.1 Performance Tests of the CryptoToolbox Implementations

The eSTREAM ciphers and AES, both open-source implementations from Crypto++ and the implementations from the CryptoToolbox described in Chapter 5, were benchmarked in an EtM-composition with HMAC-SHA-256. The AEGIS stream cipher implementation from the CryptoToolbox described in Chapter 5 was also benchmarked, using the version implemented with the 32-bit table-driven variant of the AES round function. The implementation of AES GCM from the Crypto++ library was also assessed.

To measure the latency induced by the cryptographic algorithms, the RTT was measured in the encryption laboratory setup introduced in Section 3.1.1 and Section 3.2.3. The client tags the data with a timestamp t_1 at which time it was instantiated to form the plaintext. The plaintext is encrypted, and then authenticated along with the IV before the ciphertext, tag and IV are transmitted to the controller. Upon reception, the controller verifies the authenticity of the ciphertext and IV, decrypts the ciphertext, re-encrypts the recovered plaintext with a new IV, and authenticates the new ciphertext and IV before transmitting the new ciphertext, tag and IV back to the client. The client verifies the authenticity of the ciphertext

and IV and decrypts the ciphertext to recover the plaintext. The time at which the plaintext is successfully recovered t_2 is recorded, and the RTT is computed according to (3.1). Each RTT is then logged, after which a mean RTT and a standard deviation are computed according to

$$\mu = \frac{\sum_{i=1}^{N} RTT_i}{N} \tag{7.1}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N} (RTT_i - \mu)^2}{N}} \tag{7.2}$$

over 10 000 sampled RTTs, i.e., N = 10 000. The mean RTT and standard deviation of the above scheme is also measured *without* the use of cryptographic algorithms, to capture a baseline latency.

7.2 Quantitative Results and Discussion

The results seen in Table 7.1 show the latencies induced by the authenticated encryption schemes using algorithm implementations from the Crypto++ open-source cryptographic library, while the results seen in Table 7.2 show the latencies induced by the authenticated encryption schemes using algorithm implementations from the CryptoToolbox. The results show that while AES provides good performance on small data, AEGIS and the stream ciphers from the eSTREAM portfolio significantly outperform AES on larger data. The best performance was seen by the implementation of the AEGIS stream cipher from the CryptoToolbox, which is virtually tied with the EtM-composition of Sosemanuk and HMAC-SHA-256 implementation using algorithms from the Crypto++ library, as seen from Figure 7.1. We also observe that, as expected, AES-GCM provides poor performance on the 32-bit system without carryless multiplication support and should therefore be avoided in favor of generic EtM-compositions or AEGIS on such systems. Note that while AES would be expected to perform significantly better in systems with instruction sets that provide hardware acceleration of AES, these instruction sets would also increase the performance of AEGIS. The choice of MAC also affects the performance of the EtM-composition, and choosing a more efficient MAC based on universal hashing (for example Poly1305 or Blake2b), rather than HMAC utilizing a cryptographic hash such as SHA-256, could result in increased performance.

Figure 7.2 and Figure 7.3 show the mean latencies induced by the various cryptographic algorithms using implementations from the Crypto++ library and the CryptoToolbox respectively. Notice that the latency induced grows linearly with the size of the data that is processed. Also notice the constant initialization time required for each of the algorithms.

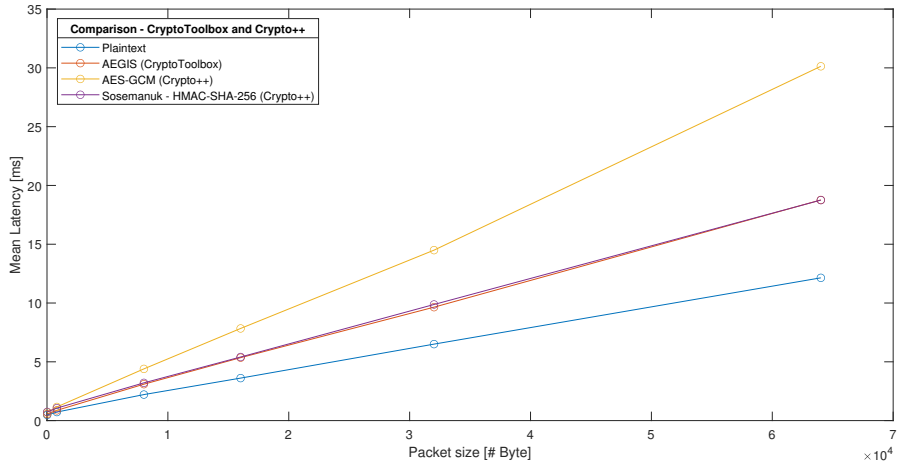

Figure 7.1: A comparison between the top performing algorithms from the CryptoToolbox and the Crypto++ library, in addition to the popular AES GCM algorithm.

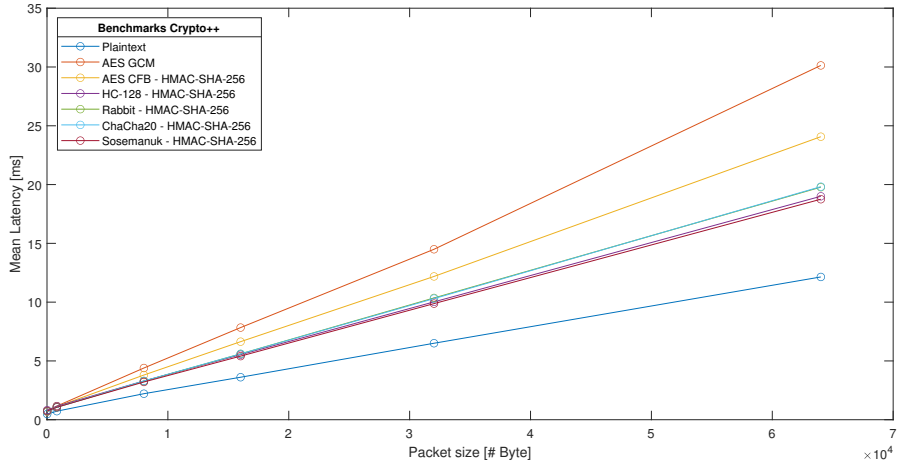

Figure 7.2: The mean latency induced by transmitting packets of varying data sizes, processed by the various cryptographic algorithms from the Crypto++ library to obtain authenticated encryption.

7.3 Qualitative Experiments

In addition to the quantitative results shown in Section 7.2, some qualitative results are shown, illustrating how efficient the scheme proposed in Section 4.2 is at resisting attacks when applied in conjunction with algorithms from the CryptoToolbox, described in Chapter 5. Finally, a successful application of the CryptoToolbox algorithms in the ROS environment is shown.

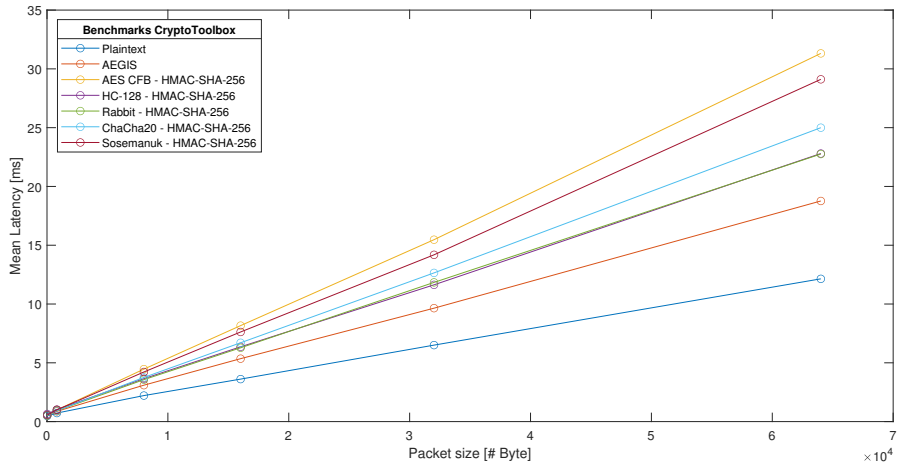

Figure 7.3: The mean latency induced by transmitting packets of varying data sizes, processed by the various cryptographic algorithms from the CryptoToolbox to obtain authenticated encryption.

7.3.1 Back to the motivating example

The MSD system and the PI controller from the motivating example in Section 1.1.1 were implemented, except this time the communication scheme proposed in Section 4.2 was applied. The AEGIS authenticated encryption stream cipher from the CryptoToolbox was applied to provide authenticated encryption.

Initially, the same adversary as in the motivating example was used, transmitting a 50N spoofed control signal at a frequency of 100 Hz to the system simulation. The result can be seen in Figure 7.4. Notice that the system resists the attack by dismissing the spoofed packets and is unaffected by the attack.

Later, a replay attack was attempted. That is, *a valid ciphertext* was recorded and later replayed. The valid ciphertext was replayed at a frequency of 100 Hz to the system simulation. Because of the invalid timestamp, the replayed control signal was dismissed by the system, even though the (message, tag)-pair was valid. The result can be seen in Figure 7.5. Observe that the replay attack is also successfully resisted.

7.3.2 Application in the ROS environment

ROS is an open-source middleware and framework commonly used in robotics and works by having *nodes* exchange information over *topics*. A node *publishes* information to a topic that other nodes may *subscribe* to. However, ROS does not provide any form of cryptographic security, and an adversary is, therefore, able to both eavesdrop on the information that is transmitted between the nodes and to inject spoofed data. While some researchers in the past have tried to integrate crypto-

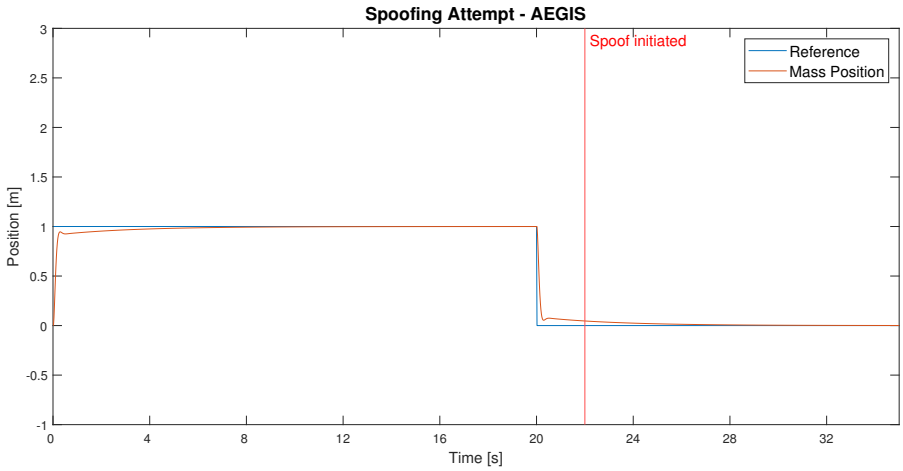

Figure 7.4: An illustration of the system simulation resisting the spoofed control signal when using the scheme proposed in Section 4.2.

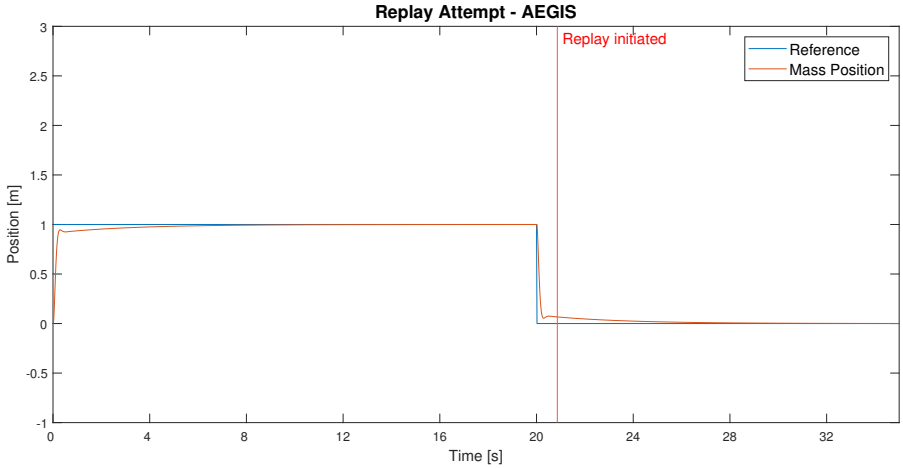

Figure 7.5: An illustration of the system simulation resisting the replay attack when using the scheme proposed in Section 4.2.

graphic methods into the ROS environment previously as described in Section 1.2, they have largely applied confidentiality-only cryptographic methods such as 3DES and AES operating in ECB mode or CBC mode. However, as described in Section 4.1.2, a block cipher operating in ECB mode *leaks structural information about the plaintext*, of which there is much in an image. No modern stream ciphers were considered in these works, and the data was never authenticated, thus permitting injection and manipulation of the data that is transmitted.

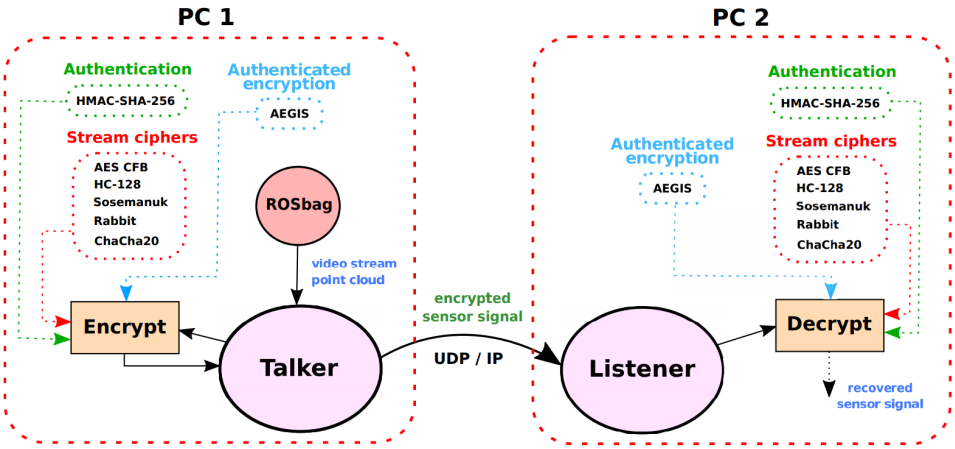

Figure 7.6: The CryptoToolbox integrated into the ROS environment. Image courtesy of Volden & Solnør (2020).

Therefore, the CryptoToolbox algorithms were integrated into the ROS environment in collaboration with colleague Øystein Volden (Volden & Solnør 2020). Only cryptographic methods that *do not* leak structural information were applied, and proper authenticated encryption methods were used, such as the authenticated encryption algorithm AEGIS, and encryption algorithms and message authentication algorithms in the EtM construction. The pipeline that was proposed can be seen in Figure 7.6. Both video and LiDAR data were transmitted between the machines. Experiments conducted on Nvidia Jetson Xavier machines featuring the ARMv8 Cryptography Extension, permitting the use of the hardware accelerated versions of AES and AEGIS described in Section 5.1.1 and Section 5.1.6, respectively, showed that the hardware-accelerated versions were up to 65% faster than the portable software implementation and managed to encrypt & authenticate and decrypt & verify a 3.15 MB LiDAR point-cloud in as little as 6.5 milliseconds. AEGIS also showed the best performance when the hardware accelerated verions were used. A screenshot of an encrypted video stream and the corresponding recovered video stream in the ROS environment can be seen in Figure 7.7.

7.4 Summary

By using the cryptographic algorithms as described in Section 4.2, the feedback control system resists deception attacks and eavesdropping attacks. Also, it is clear from Section 7.3 that the proposed scheme is robust against disturbances and guarantees synchronous behavior between the transmitter and the receiver when under an attack. Thus, the scheme provides some resistance against DoS attacks. Finally, the versatility of the CryptoToolbox implementations described in Chapter 5 was shown in Section 7.3.2, where the algorithms were applied in the ROS environment.

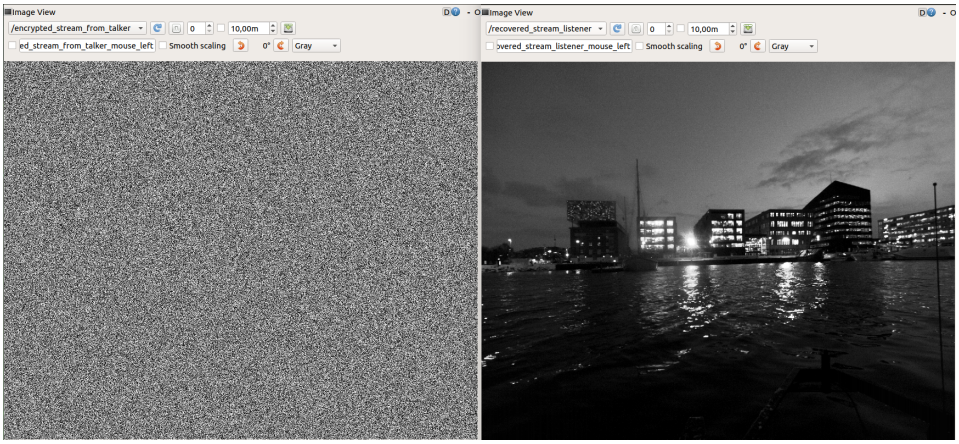

Figure 7.7: An encrypted video stream in the ROS environment on the left, with the corresponding recovered video stream on the right. The video stream was encrypted using the AES CFB algorithm from the CryptoToolbox. Image courtesy of Volden & Solnør (2020).

Table 7.1: A comparison of authenticated encryption performance using EtM compositions of the eSTREAM portfolio stream ciphers and HMAC-SHA-256, an EtM composition of AES CFB and HMAC-SHA-256, and the AES GCM authenticated encryption mode from the Crypto++ open-source cryptographic library.

Cryptographic Algorithm		**Crypto++**	
Encryption Algorithm	**Authentication Algorithm**	**Mean RTT [ms]**	**Std.Dev [ms]**
Data size: 16 bytes		μ	σ
No encryption	No authentication	0.481	0.066
AES GCM		0.695	0.021
AES CFB	HMAC (SHA-256)	0.737	0.075
HC-128	HMAC (SHA-256)	0.813	0.028
Rabbit	HMAC (SHA-256)	0.711	0.035
ChaCha20	HMAC (SHA-256)	0.712	0.079
Sosemanuk	HMAC (SHA-256)	0.736	0.062
Data size: 816 bytes		μ	σ
No encryption	No authentication	0.726	0.104
AES GCM		1.161	0.032
AES CFB	HMAC (SHA-256)	1.116	0.104
HC-128	HMAC (SHA-256)	1.150	0.148
Rabbit	HMAC (SHA-256)	1.057	0.040
ChaCha20	HMAC (SHA-256)	1.062	0.483
Sosemanuk	HMAC (SHA-256)	1.063	0.039
Data size: 8016 bytes		μ	σ
No encryption	No authentication	2.213	0.018
AES GCM		4.398	0.065
AES CFB	HMAC (SHA-256)	3.801	0.081
HC-128	HMAC (SHA-256)	3.296	0.040
Rabbit	HMAC (SHA-256)	3.274	0.083
ChaCha20	HMAC (SHA-256)	3.295	0.158
Sosemanuk	HMAC (SHA-256)	3.209	0.141
Data size: 16016 bytes		μ	σ
No encryption	No authentication	3.613	0.057
AES GCM		7.836	0.107
AES CFB	HMAC (SHA-256)	6.638	0.133
HC-128	HMAC (SHA-256)	5.531	0.096
Rabbit	HMAC (SHA-256)	5.601	0.120
ChaCha20	HMAC (SHA-256)	5.599	0.044
Sosemanuk	HMAC (SHA-256)	5.408	0.046
Data size: 32016 bytes		μ	σ
No encryption	No authentication	6.504	0.112
AES GCM		14.496	0.121
AES CFB	HMAC (SHA-256)	12.186	0.129
HC-128	HMAC (SHA-256)	10.023	0.105
Rabbit	HMAC (SHA-256)	10.356	0.093
ChaCha20	HMAC (SHA-256)	10.300	0.293
Sosemanuk	HMAC (SHA-256)	9.875	0.151
Data size: 64016 bytes		μ	σ
No encryption	No authentication	12.142	0.214
AES GCM		30.133	0.816
AES CFB	HMAC (SHA-256)	24.072	0.823
HC-128	HMAC (SHA-256)	19.002	0.094
Rabbit	HMAC (SHA-256)	19.778	0.233
ChaCha20	HMAC (SHA-256)	19.822	0.652
Sosemanuk	HMAC (SHA-256)	18.753	0.326

Table 7.2: A comparison of authenticated encryption performance using EtM compositions of the eSTREAM portfolio stream ciphers and HMAC-SHA-256, an EtM composition of AES CFB and HMAC-SHA-256, and the AEGIS authenticated encryption cipher from the CryptoToolbox.

Cryptographic Algorithm		**Direct Implementation**	
Encryption Algorithm	**Authentication Algorithm**	**Mean RTT [ms]**	**Std.Dev [ms]**
Data size: 16 bytes		μ	σ
No encryption	No authentication	0.481	0.066
AEGIS		0.548	0.019
AES CFB	HMAC (SHA-256)	0.533	0.069
HC-128	HMAC (SHA-256)	0.644	0.039
Rabbit	HMAC (SHA-256)	0.531	0.025
ChaCha20	HMAC (SHA-256)	0.544	0.093
Sosemanuk	HMAC (SHA-256)	0.554	0.016
Data size: 816 bytes		μ	σ
No encryption	No authentication	0.726	0.104
AEGIS		0.872	0.066
AES CFB	HMAC (SHA-256)	0.990	0.020
HC-128	HMAC (SHA-256)	1.023	0.088
Rabbit	HMAC (SHA-256)	0.926	0.185
ChaCha20	HMAC (SHA-256)	0.930	0.048
Sosemanuk	HMAC (SHA-256)	0.996	0.049
Data size: 8016 bytes		μ	σ
No encryption	No authentication	2.213	0.018
AEGIS		3.097	0.063
AES CFB	HMAC (SHA-256)	4.463	0.043
HC-128	HMAC (SHA-256)	3.651	0.085
Rabbit	HMAC (SHA-256)	3.566	0.232
ChaCha20	HMAC (SHA-256)	3.755	0.109
Sosemanuk	HMAC (SHA-256)	4.219	0.049
Data size: 16016 bytes		μ	σ
No encryption	No authentication	3.613	0.057
AEGIS		5.352	0.231
AES CFB	HMAC (SHA-256)	8.163	0.399
HC-128	HMAC (SHA-256)	6.360	0.111
Rabbit	HMAC (SHA-256)	6.276	0.319
ChaCha20	HMAC (SHA-256)	6.704	0.480
Sosemanuk	HMAC (SHA-256)	7.622	0.129
Data size: 32016 bytes		μ	σ
No encryption	No authentication	6.504	0.112
AEGIS		9.646	0.164
AES CFB	HMAC (SHA-256)	15.471	0.193
HC-128	HMAC (SHA-256)	11.632	0.282
Rabbit	HMAC (SHA-256)	11.838	0.242
ChaCha20	HMAC (SHA-256)	12.647	0.598
Sosemanuk	HMAC (SHA-256)	14.185	0.223
Data size: 64016 bytes		μ	σ
No encryption	No authentication	12.142	0.214
AEGIS		18.764	0.231
AES CFB	HMAC (SHA-256)	31.315	0.962
HC-128	HMAC (SHA-256)	22.800	0.712
Rabbit	HMAC (SHA-256)	22.755	0.607
ChaCha20	HMAC (SHA-256)	24.997	0.669
Sosemanuk	HMAC (SHA-256)	29.110	1.013

Chapter 8

Conclusion

This thesis has emphasized the importance of protecting the signals transmitted in feedback control systems. While numerous authors have previously proposed communication schemes for feedback control systems, these schemes have rarely, if ever, been properly analyzed. In this thesis, it has been demonstrated that these communication schemes do not provide the security that was promised. The deficiencies of previously proposed schemes were not only shown analytically, but attacks were also demonstrated. As an alternative, a cryptographically strong communication scheme has been proposed.

A software toolbox containing high-performance implementations of state-of-the-art cryptographic algorithms has also been developed and made publicly available in a Git repository. The cryptographic algorithms have been benchmarked and shown to induce a very low latency on the signals transmitted inside a feedback control system, making them suitable for real-time systems.

8.1 Summary of Findings

Throughout this thesis, the following key findings have been observed:

- It has been demonstrated that previously proposed schemes using ECB encryption fail to provide confidentiality.
- It has been demonstrated that the STM proposed by Pang & Liu (2012) either fails to provide confidentiality or is prone to deception attacks.
- A scheme incorporating cryptographically strong authenticated encryption methods has been presented.
- A toolbox was built, providing easy access to state-of-the-art high-performance cryptographic algorithms.

- The performance of the implementations in the toolbox was assessed and benchmarked against an open-source cryptographic library.
- The algorithm implementations are well-suited to be incorporated in software commonly used in feedback control systems, such as ROS.

Using this information, the research questions posed in Section 1.4 may now be answered.

8.1.1 Research question 1

In **RQ1** we asked: *How do 'secure' communication schemes for feedback control systems proposed by researchers from the control community fare against cyber-physical attacks?*

From the analysis in Section 4.1, it is clear that there has been a lack of analysis of previously proposed schemes. Attacks against the schemes have been proposed and implemented, demonstrating that the schemes are insecure.

8.1.2 Research question 2

In **RQ2** we asked: *May cryptographic techniques be used to enhance the security of feedback control systems through a new, secure communication scheme?*

This research question was largely answered in Section 4.2. It was shown that using proper authenticated encryption rather than the ad-hoc schemes analyzed in Section 4.1 would significantly enhance the security of feedback control systems.

8.1.3 Research question 3

In **RQ3** we asked: *How may the transmitter and receiver of a feedback control system achieve synchronous communication if stateful cryptographic methods are used over non-reliable communication protocols?*

In Section 2.5.1, several ways of synchronizing stateful algorithms were discussed. In practical terms, two alternatives were used. Using AES in the self-synchronizing CFB mode with a carry-over IV would automatically ensure synchronous behavior while minimizing the amount of data that needs to be transmitted. The minimization of traffic does come at the cost of having to resort to a less efficient cipher, as observed in Chapter 7. The other alternative is to explicitly synchronize the transmitter and the receiver through the use of (public) IVs (or nonces) that must be transmitted with each message. For small and frequent messages, this would result in a significant increase in the amount of data that must be transmitted.

8.1.4 Research question 4

In **RQ4** we asked: *To what extent do open-source cryptographic libraries provide access to modern cryptographic algorithms, and how does their performance compare to direct implementations of the algorithms?*

Unfortunately, open-source cryptographic libraries commonly provide access to a variety of block ciphers, public-key algorithms, and deprecated stream ciphers. The availability of modern high-performance stream ciphers is comparably low and largely confined to the Crypto++ library. This resulted in the development of the CryptoToolbox described in Chapter 5. Through the CryptoToolbox, the user gains access to a selection of high-performance state-of-the-art stream ciphers. The results obtained in Chapter 7 reveal that in addition to providing access to cryptographic algorithms that are not available through open-source libraries (AEGIS), the CryptoToolbox implementations perform very well.

8.1.5 Research question 5

In **RQ5** we asked: *Which cryptographic algorithms provide the best performance, and should be used to obtain authenticated encryption, in feedback control systems?*

From the results in Section 7.2, it is clear that the AEGIS stream cipher, which provides authenticated encryption directly, provides the best performance among the algorithms.

It should be noted that while latency is very important in feedback control systems, the AES CFB algorithm with a carry-over IV can be used if it is desirable to minimize the amount of data that must be transmitted. While less efficient, the difference is negligible for small data packets.

8.2 Future Work

In this thesis, the focus has been on symmetric cryptographic methods and how these may be applied to provide secure signal transmission in feedback control systems. It has been shown and demonstrated that symmetric cryptographic methods achieve this without inducing significant latency and other detrimental effects on the systems. However, the use of homomorphic encryption schemes in feedback control systems has also been explored, as mentioned in Section 1.2. It would be of interest to investigate if a homomorphic authenticated encryption scheme with sufficient security also could be used in feedback control systems.

Homomorphic cryptography is known to be computationally expensive compared to symmetric cryptography, and it is therefore expected that the induced latencies would be significantly higher. In their demonstrations, Teranishi et al. (2020) had to use a keysize of 33 bits in order to maintain real-time operation when using their homomorphic encryption method on a state feedback controller. Such a low key size is not sufficient, and yet they did not apply authentication methods, nor is their system resilient against synchronization loss. In addition, the homomorphic encryption schemes using asymmetric cryptographic methods, for example Elgamal as was done by Teranishi et al. (2020), places constraints on the plaintext spaces, thus resulting in quantization errors that must be accounted for when considering the stability of the overall control system, in addition to the latencies that are

induced. A closer look at these aspects would be of interest, and if found suitable, the CryptoToolbox could be extended with homomorphic cryptographic methods.

While the cryptographic algorithms described in Chapter 5 have been applied successfully in laboratory setups, including in ROS, full-scale experiments on unmanned surface vehicles in which weaknesses could first be demonstrated, and secure operation could later be achieved through application of the algorithms, would be of great interest.

Reference List

Anderson, R., Biham, E. & Knudsen, L. (2000), The case for serpent.

Anderson, R. J. (1991), 'Tree functions and cipher systems', *Cryptologia* **15**(3), 194–202.

Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B. & Wingers, L. (2015), The simon and speck lightweight block ciphers, *in* 'Proceedings of the 52nd Annual Design Automation Conference', DAC '15, Association for Computing Machinery, New York, NY, USA.

Bellare, M. (2015), 'New proofs for nmac and hmac: Security without collision resistance', *J. Cryptol.* **28**(4), 844–878.

Bellare, M. & Namprempre, C. (2008), 'Authenticated encryption: Relations among notions and analysis of the generic composition paradigm', *J. Cryptol.* **21**(4), 469–491.

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T. & Sibert, H. (2008), *Sosemanuk, a fast software-oriented stream cipher*, Springer-Verlag, Berlin, Heidelberg, p. 98–118.

Bernstein, D. (2008), 'Chacha, a variant of salsa20'.

Bernstein, D. J. (2005), The poly1305-aes message-authentication code, *in* 'Proceedings of the 12th International Conference on Fast Software Encryption', FSE'05, Springer-Verlag, Berlin, Heidelberg, p. 32–49.

Biham, E. & Shamir, A. (1992), Differential cryptanalysis of the full 16-round des, *in* 'Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology', CRYPTO '92, Springer-Verlag, Berlin, Heidelberg, p. 487–496.

Biryukov, A. & Wagner, D. (1999), Slide attacks, *in* L. Knudsen, ed., 'Fast Software Encryption', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 245–259.

Boesgaard, M., Vesterager, M. & Zenner, E. (2008), *The Rabbit Stream Cipher*, Springer-Verlag, Berlin, Heidelberg, p. 69–83.

Chen, H., Meng, C., Shan, Z., Fu, Z. & Bhargava, B. K. (2019), 'A novel low-rate denial of service attack detection approach in zigbee wireless sensor network by combining hilbert-huang transformation and trust evaluation', *IEEE Access* **7**, 32853–32866.

Chen, J., Zhang, F. & Sun, J. (2017), 'Analysis of security in cyber-physical systems', *Science China Technological Sciences* **60**(12), 1975–1977.

Cheon, J. H., Han, K., Hong, S., Kim, H. J., Kim, J., Kim, S., Seo, H., Shim, H. & Song, Y. (2018), 'Toward a secure drone system: Flying with real-time homomorphic authenticated encryption', *IEEE Access* **6**, 24325–24339.

Crutchfield, C. (2014), Implementing and optimizing encryption algorithms for the armv8-a architecture, Master's thesis, California State University - Sacramento, 6000 J St, Sacramento, CA 95819, USA.

d. Sa, A. O., d. C. Carmo, L. F. R. & Machado, R. C. S. (2017), Use of switching controllers for mitigation of active identification attacks in networked control systems, *in* '2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)', pp. 257–262.

Daemen, J., Govaerts, R. & Vandewalle, J. (1992), On the design of high speed self-synchronizing stream ciphers, *in* '[Proceedings] Singapore ICCS/ISITA '92', pp. 279–283 vol.1.

Daemen, J. & Kitsos, P. (2008), *The Self-synchronizing Stream Cipher Moustique*, Vol. 4986, pp. 210–223.

Dai, W. (2020), 'Crypto++'. Accessed: 2020-12-16.
URL: *https://www.cryptopp.com/*

Dang, Q. H. (2008), The keyed-hash message authentication code (hmac) - fips 198-1, Technical report, Gaithersburg, MD, USA.

Dang, Q. H. (2015), Secure hash standard - fips 180-4, Technical report, Gaithersburg, MD, USA.

de Sá, A. O., d. C. Carmo, L. F. R. & Machado, R. C. S. (2017), 'Covert attacks in cyber-physical control systems', *IEEE Transactions on Industrial Informatics* **13**(4), 1641–1651.

de Sá, A. O., d. C. Carmo, L. F. R. & Santos Machado, R. C. (2019), Countermeasure for identification of controlled data injection attacks in networked control systems, *in* '2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT)', pp. 455–459.

de Sá, A. O., da Costa Carmo, L. F. & Machado, R. C. (2018), 'A controller design for mitigation of passive system identification attacks in networked control systems', *Journal of Internet Services and Applications* **9**(1), 1–19.

Diffie, W. & Hellman, M. (1976), 'New directions in cryptography', *IEEE Transactions on Information Theory* **22**, 644–654.

Dravie, B. (2017), Synchronization and dynamical systems: application to cryptography, PhD thesis.

Duong, T. & Rizzo, J. (2011), Here come the ⊕ ninjas. Unpublished.

Dworkin, M. (2005), The cmac mode for authentication, Technical report, Gaithersburg, MD, USA.

Dworkin, M. J. (2001), Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques, Technical report, Gaithersburg, MD, USA.

Dworkin, M. J. (2004), Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality, Technical report, Gaithersburg, MD, USA.

Dworkin, M. J. (2007), Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac, Technical report, Gaithersburg, MD, USA.

Elgamal, T. (1985), A public key cryptosystem and a signature scheme based on discrete logarithms, *in* 'Proceedings of CRYPTO 84 on Advances in Cryptology', Springer-Verlag, Berlin, Heidelberg, p. 10–18.

Feistel, H. (1971), 'Block cipher cryptographic system'.

Futoransky, A., Kargieman, E. & Pacetti, A. M. (October, 1998), An attack on crc-32 integrity checks of encrypted channels using cbc or cfb modes, Technical report.

Grover, L. K. (1996), A fast quantum mechanical algorithm for database search, *in* 'Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing', STOC '96, Association for Computing Machinery, New York, NY, USA, p. 212–219.

Gupta, R. A. & Chow, M. (2008), Performance assessment and compensation for secure networked control systems, *in* '2008 34th Annual Conference of IEEE Industrial Electronics', pp. 2929–2934.

Jithish, J. & Sankaran, S. (2017), Securing networked control systems: Modeling attacks and defenses, *in* '2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)', pp. 7–11.

Kiley, P. (2019), 'Investigating can bus network integrity in avionics systems', https://www.rapid7.com/research/report/investigating-can-bus-network-integrity-in-avionics-systems/.

Kim, J., Lee, C., Shim, H., Cheon, J. H., Kim, A., Kim, M. & Song, Y. (2016), 'Encrypting controller using fully homomorphic encryption for security of cyber-physical systems', *IFAC-PapersOnLine* **49**(22), 175 – 180. 6th IFAC Workshop on Distributed Estimation and Control in Networked Systems NECSYS 2016.

Klein, A. (2013), *Stream Ciphers*, Springer Publishing Company, Incorporated.

Kogiso, K., Baba, R. & Kusaka, M. (2018), 'Development and examination of encrypted control systems'.

Kogiso, K. & Fujita, T. (2015), Cyber-security enhancement of networked control systems using homomorphic encryption, *in* '2015 54th IEEE Conference on Decision and Control (CDC)', pp. 6836–6843.

Litzenberger, D. C. (2020), 'Python cryptography toolkit (pycrypto)'. Accessed: 2020-12-20.
URL: *https://www.dlitz.net/software/pycrypto/*

Liu, G. (2017), 'Predictive control of networked multiagent systems via cloud computing', *IEEE Transactions on Cybernetics* **47**(8), 1852–1859.

Matellán, V., Balsa, J., Casado García, F., Fernández, C., Martín, F. & Rodríguez Lera, F. (2016), Cybersecurity in autonomous systems: Evaluating the performance of hardening ros.

Maurer, U. M. (1991), New approaches to the design of self-synchronizing stream ciphers, *in* D. W. Davies, ed., 'Advances in Cryptology — EUROCRYPT '91', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 458–471.

McEliece, R. (1986), *Finite Fields for Computer Scientists and Engineers*, The Springer International Series in Engineering and Computer Science, Springer US.

Menezes, A. J., Vanstone, S. A. & Oorschot, P. C. V. (1996), *Handbook of Applied Cryptography*, 1st edn, CRC Press, Inc., USA.

Millerioux, G. & Guillot, P. (2010), 'Self-synchronizing stream ciphers and dynamical systems: State of the art and open issues', *International Journal of Bifurcation and Chaos* **20**, 2979–2991.

National Bureau of Standards (1977), 'Data encryption standard (des)', Federal Information Processing Standards Publication 46.

National Institute of Standards and Technology (2015), Sha-3 standard: Permutation-based hash and extendable-output functions - fips 202, Technical report, Gaithersburg, MD, USA.

NIST (2001), 'Specification for the advanced encryption standard (aes)', Federal Information Processing Standards Publication 197.

OpenSSL Software Foundation (2020), 'OpenSSL'. Accessed: 2020-12-20.
URL: *https://www.openssl.org/*

Osvik, D. A. (2000), Speeding up serpent, *in* 'AES Candidate Conference'.

Pang, Z. & Liu, G. (2010), Secure networked control systems under data integrity attacks, *in* 'Proceedings of the 29th Chinese Control Conference', IEEE, pp. 5765–5771.

Pang, Z. & Liu, G. (2012), 'Design and implementation of secure networked predictive control systems under deception attacks', *IEEE Transactions on Control Systems Technology* **20**(5), 1334–1342.

Pang, Z., Zheng, G., Liu, G. & Luo, C. (2011), Secure transmission mechanism for networked control systems under deception attacks, *in* '2011 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems', pp. 27–32.

Reynolds, J. & Postel, J. (1994), Assigned numbers, RFC 1700, RFC Editor.

Rivest, R. (1992), The md5 message-digest algorithm, RFC 1321, RFC Editor.

Rivest, R. L., Shamir, A. & Adleman, L. (1978), 'A method for obtaining digital signatures and public-key cryptosystems', *Commun. ACM* **21**(2), 120–126.
URL: *https://doi.org/10.1145/359340.359342*

Rodríguez-Lera, F. J., Matellán-Olivera, V., Balsa-Comerón, J., Guerrero-Higueras, n. M. & Fernández-Llamas, C. (2018), 'Message encryption in robot operating system: Collateral effects of hardening mobile robots', *Frontiers in ICT* **5**, 11.

Rose, G., Hawkes, P., Paddon, M. & de Vries, M. W. (2005), 'Primitive specification for sss', **28**.

Rueppel, R. A. (1986), *Analysis and Design of Stream Ciphers*, Springer-Verlag, Berlin, Heidelberg.

Sanadhya, S. K. & Sarkar, P. (2008), Non-linear reduced round attacks against sha-2 hash family, *in* 'Proceedings of the 13th Australasian Conference on Information Security and Privacy', ACISP '08, Springer-Verlag, Berlin, Heidelberg, p. 254–266.

Shannon, C. (1945), A mathematical theory of cryptography, Classified report, Bell Laboratories, Murray Hill, NJ, USA.

Shannon, C. E. (1949), 'Communication theory of secrecy systems'.

Shor, P. W. (1994), Algorithms for quantum computation: discrete logarithms and factoring, *in* 'Proceedings 35th Annual Symposium on Foundations of Computer Science', pp. 124–134.

Solnør, P. (2020), 'CryptoToolbox', https://github.com/pettsol/CryptoToolbox.

Sparrow, R. D., Adekunle, A. A., Berry, R. J. & Farnish, R. J. (2015), Simulating and modelling the impact of secure communication latency for closed loop control, *in* '2015 Internet Technologies and Applications (ITA)', pp. 138–143.

Sun, H.-T., Peng, C., Zhou, P. & Wang, Z.-W. (2017), 'A brief overview on secure control of networked systems', *Advances in Manufacturing* **5**(3), 243–250.

Teixeira, A., Sou, K. C., Sandberg, H. & Johansson, K. H. (2013), *Quantifying Cyber-Security for Networked Control Systems*, Springer International Publishing, Heidelberg, pp. 123–142.

Teranishi, K., Shimada, N. & Kogiso, K. (2020), 'Stability-guaranteed dynamic elgamal cryptosystem for encrypted control systems', *IET Control Theory Applications* **14**(16), 2242–2252.

Turner, S. & Chen, L. (2011), Updated security considerations for the md5 message-digest and the hmac-md5 algorithms, RFC 6151, RFC Editor.

Ulz, T., Pieber, T., Steger, C., Matischek, R. & Bock, H. (2017), Towards trustworthy data in networked control systems: A hardware-based approach, *in* '2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)', pp. 1–8.

Vernam, G. (1919), 'Secret signaling system'.

Volden, Ø. & Solnør, P. (2020), 'Crypto ROS: Real-time authenticated encryption of vision-based sensor signals in ROS', https://github.com/oysteinvolden/Real-time-sensor-encryption.

wolfSSL Inc. (2020), 'wolfCrypt'. Accessed: 2020-12-20.
URL: *https://www.wolfssl.com/products/wolfcrypt-2/*

Wu, G., Sun, J. & Chen, J. (2016), 'A survey on the security of cyber-physical systems', *Control Theory and Technology* **14**(1), 2–10.

Wu, H. (2008), The stream cipher hc-128, *in* 'The eSTREAM Finalists'.

Wu, H. & Preneel, B. (2014), Aegis: A fast authenticated encryption algorithm, *in* T. Lange, K. Lauter & P. Lisoněk, eds, 'Selected Areas in Cryptography – SAC 2013', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 185–201.

Yang, H., Xu, Y., Xia, Y. & Zhang, J. (2017), 'Networked predictive control for nonlinear systems with arbitrary region quantizers', *IEEE Transactions on Cybernetics* **47**(8), 2244–2255.

Yaseen, A. A. & Bayart, M. (2016), Attack-tolerant networked control system in presence of the controller hijacking attack, *in* '2016 International Conference on Military Communications and Information Systems (ICMCIS)', pp. 1–8.

Appendix A

A Cryptographic Toolbox for Feedback Control Systems

Modeling, Identification and Control, Vol. 41, No. 4, 2020, pp. 1–20, ISSN 1890–1328

A Cryptographic Toolbox for Feedback Control Systems

Petter Solnør

Department of Engineering Cybernetics, Norwegian University of Science and Technology, 7491 Trondheim, Norway. E-mail: petter.solnor@ntnu.no

Abstract

Feedback control systems consist of components such as sensory systems, state estimators, controllers, and actuators. By transmitting signals between these components across insecure transmission channels, feedback control systems become vulnerable to cyber-physical attacks. For example, passive eavesdropping attacks may result in a leak of confidential system and control parameters. Active deception attacks may manipulate the behavior of the state estimators, controllers, and actuators through the injection of spoofed data. To prevent such attacks, we must ensure that the transmitted signals remain confidential across the transmission channels, and that spoofed data is not allowed to enter the feedback control system. We can achieve both these goals by using cryptographic tools. By encrypting the signals, we achieve confidential signal transmission. By applying message authentication codes (MACs), we assert the authenticity of the data before allowing it to enter the components of the feedback control system. In this paper, a toolbox containing implementations of state-of-the-art high-performance algorithms such as the Advanced Encryption Standard (AES), the AEGIS stream cipher, the Keyed-Hash Message Authentication Code (HMAC), and the stream ciphers from the eSTREAM portfolio, is introduced. It is shown how the algorithm implementations can be used to ensure secure signal transmission between the components of the feedback control system, and general guidelines that the users must adhere to for safe operation are provided.

Keywords: Cryptography, Feedback Control System, Networked Control System, Authenticated Encryption

1. Introduction

By consisting of components such as sensory systems, state estimators, controllers, and actuators, feedback control systems are inherently modular. These components need to communicate by transmitting measurements, state estimates, and control inputs. Since these components may be spatially distributed, they are often connected through a network or a field bus spanning the vehicle or plant.

We refer to a feedback control system, in which the components are connected through a network, as a networked control system (Hespanha et al., 2007). Unfortunately, these signal transmissions also make the feedback control systems vulnerable to cyber-physical attacks such as eavesdropping and deception attacks. An adversary that gains access to the network may eavesdrop on the transmitted signals and perform unauthorized system identification, thus gaining knowledge of system parameters or control parameters that may be considered confidential, as discussed by de Sá et al. (2017). Furthermore, an adversary with access to the network can also perform deception attacks by injecting spoofed data, thus manipulating the behavior of the system. Combined with system knowledge, such a deception attack could even result in a successful sys-

doi:10.4173/mic.2020.4.1

tem hijacking, as discussed by Teixeira et al. (2013).

The leak of system and controller parameters, and the hijacking of a dynamical system, poses a significant risk to the system and its surroundings. Exploring methods that enhance the resilience of feedback control systems against such cyber-physical attacks is therefore important.

1.1. Cryptographic methods and feedback control signals

To prevent system identification attacks, the confidentiality of the transmitted signals must be ensured across the insecure transmission channels, while the origin of the transmitted signals must be authenticated before they are allowed to enter the feedback control system in order to prevent deception attacks. Both of these goals may be achieved by using cryptographic tools. Confidential signal transmission is ensured by encrypting the signals before transmission, and the origin of the transmitted signals may be authenticated by using message authentication codes (MACs).

In recent years, many researchers have investigated the use of cryptographic algorithms in feedback control systems, such as Gupta and Chow (2008), Pang et al. (2011), Jithish and Sankaran (2017), Lera et al. (2016), and Rodríguez-Lera et al. (2018). While cryptographic algorithms are available through open-source libraries such as OpenSSL (OpenSSL Software Foundation, 2020), Crypto++ (Dai, 2020), and wolfCrypt (wolfSSL Inc., 2020), these libraries may be hard to navigate and do not provide access to modern stream ciphers such as AEGIS or the stream ciphers from the eSTREAM portfolio. Therefore, researchers have used cryptographic algorithms that do not typically provide the best performance, such as the Data Encryption Standard (DES), 3DES, Blowfish, and the Advanced Encryption Standard (AES). Notably, the DES encryption algorithm is not even considered secure anymore. Worse yet, the algorithms have been used in insecure configurations, such as the Electronic Codebook (ECB) mode for block ciphers.

This paper presents a toolbox with implementations of state-of-the-art high-performance cryptographic algorithms that are ready to use in feedback control systems. The algorithms have been implemented both in portable software implementations (in `C++`) and in platform-specific implementations that take advantage of hardware acceleration features available on most modern `x86` processors and a subset of `ARMv8` processors through intrinsic functions. This provides control engineers with a set of accessible high-performance cryptographic algorithms on most popular platforms with full source code available. Examples are given to show how the algorithms may be used to secure feedback control systems against adversaries, and only secure configurations are provided limiting the possibility for misuse.

1.2. Organization of the article

The article is organized as follows. In Section 2, a scenario in which the CryptoToolbox can be used is presented. Weaknesses in the control architecture motivating the need for cryptographic methods are identified. A brief introduction to cryptographic terminology is also given. Then, in Section 3, a brief overview of the CryptoToolbox is presented. The algorithms that the users may access through the CryptoToolbox are presented. Key propeties of the respective algorithms are explained, in addition to important user guidelines. In Section 4, focus is shifted to how the cryptographic algorithms may be applied in a concrete example to secure the guidance, navigation, and control (GNC) system of a vehicle. Finally, Section 5 concludes the article.

2. Motivation and terminology

We begin by motivating the need for the cryptographic algorithms in the CryptoToolbox by describing how they may be used to enhance the security of feedback control systems. We introduce a use case for the CryptoToolbox, which will be treated more in detail in Section 4 after the cryptographic algorithms contained in the CryptoToolbox have been introduced.

2.1. Security issues of guidance, navigation, and control systems

Figure 1: The Otter USV. Image courtesy by Maritime Robotics (2020).

Throughout the paper, we will illustrate how the CryptoToolbox algorithms can be used in GNC architectures prevalent in many autonomous and unmanned systems. Such systems are becoming more and more

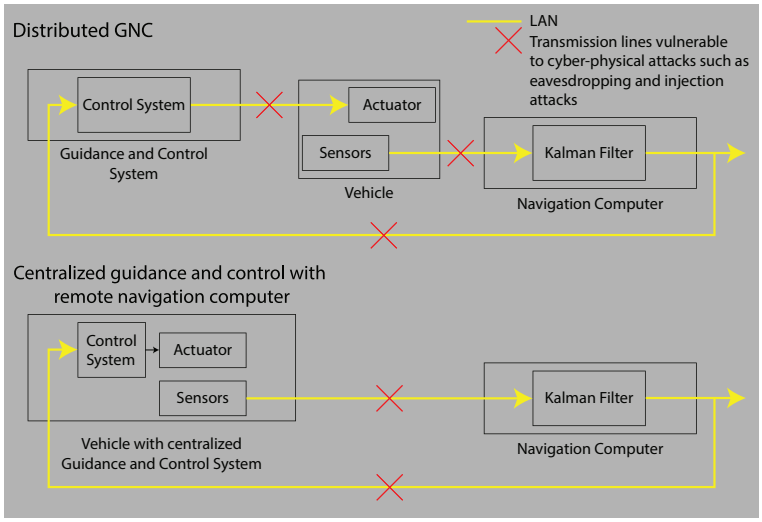

Figure 2: A generic schematic of the signal flow in vehicles with a distributed GNC architecture and a centralized guidance and control computer with a remote navigation computer. Surfaces that are vulnerable to attacks are marked, and the goal is to make the system resistant against attacks on these surfaces.

common, and securing them against cyber-physical threats is important. An example would be the growing industry of autonomous and unmanned surface vessels, some designed to transport people, others to collect possibly sensitive information for industrial purposes, such as the Otter USV seen in Figure 1. While the use case described in this article is focused on GNC systems, we emphasize that the algorithms may be used similarly for other control applications.

An illustration of the typical signal flow in vehicles with a distributed GNC architecture and a centralized guidance and control computer with a remote navigation computer may be seen in Figure 2. In these examples, we assume that the signals are transmitted across a network spanning the vehicle, for example, using the UDP/IP or TCP/IP protocols over ethernet, or a field bus such as CAN. Since none of these protocols provide cryptographic protection by default, the signals transmitted between the components may be eavesdropped upon, and spoofed signals may be injected into the transmission to manipulate the behavior of the vehicle.

Such attacks are very serious threats. System and controller parameters may be trade secrets that are very valuable to businesses and developers, and industrial espionage is a serious concern in high tech industries. On the other hand, if an adversary is capable of manipulating the behavior of the vehicle through the injection of spoofed signals to the GNC components, the vehicle may be used as a tool in a terrorist attack or an act of war to inflict great damage. Therefore, the signals must be secured by other means. At the same time, it is important that the security measures do not deteriorate the performance of the GNC system.

2.2. Cryptographic preliminaries

An attack in which an adversary eavesdrops on the transmitted signals to conduct system identification is referred to as a *passive* attack and does not directly affect the system. On the other hand, an attack in which an adversary manipulates transmitted signals and injects spoofed signals is called an *active* attack. To provide protection against passive eavesdropping attacks, we may apply *encryption*, and to provide protection against the active attacks, we may apply MACs.

Encryption

Encryption provides *confidential transmission of data over insecure transmission channels*. We refer to unencrypted information as *plaintext* and encrypted information as *ciphertext*. While techniques that provide perfect secrecy, that is, encryption algorithms that *cannot be broken*, exist, these are practically infeasible to implement. Instead, encryption algorithms that are *practically secure* are used. The goal of these encryption algorithms is to ensure that it is infeasible to break the encryption algorithm in a *computational sense*. That is, we assume that potential adversaries

have limited time and computational resources. Encryption algorithms are typically categorized as *asymmetric* or *symmetric* depending on whether it is *easy to deduce the decryption key from the encryption key or not*. For asymmetric ciphers, this is *believed to be computationally infeasible*, typically under the assumption that a particular number-theoretic problem is hard to solve. For symmetric ciphers, the encryption and decryption keys are easy to deduce from one another and are typically described as *the same*.

Asymmetric encryption algorithms are computationally expensive compared to symmetric encryption algorithms, and for this reason, symmetric encryption algorithms are the main focus in this article. Because the symmetric encryption algorithms act directly on memory buffers, the *only* impact of the symmetric encryption algorithms on the overall stability of a feedback control system is the *latency* that is induced, provided that the encryption and decryption devices achieve synchronous behavior.

Since the encryption algorithms are often *stateful*, a lost or injected packet will cause the encryption and decryption devices to lose synchronization. Most modern encryption algorithms solve this by deducing an initial state *for each packet* through a *public* parameter called an *initialization vector* (IV) or a *nonce*[1]. These are called *synchronous* ciphers. Other encryption algorithms solve this by feeding the ciphertext back into the cipher. These are called *self-synchronizing* ciphers because the decryption device automatically synchronizes to the encryption device after a finite number of ciphertext bits have been received in the correct order.

Cryptographic integrity and authenticity

While encryption algorithms may provide confidential transmission of signals across insecure channels, they do *not* ensure that the data that is received originate from a trusted source and contains the content of the data that was originally transmitted. We refer to the former as *data origin authenticity* and the latter as *data integrity*. Data origin authenticity is a stronger notion and implies data integrity, and data origin authenticity is typically achieved through MACs. Note that MACs are different from non-cryptographic integrity checks, such as cyclic redundancy checks (CRCs). While CRCs may be used by the respective protocols, for example, ethernet and CAN, they are *unkeyed* and are only suitable to detect inadvertent transmission errors. Because CRCs are unkeyed, an active adversary can easily compute a valid CRC for a spoofed packet. We emphasize that this is often also true *even if the output of the CRC is encrypted*. Encrypting the output of an unkeyed integrity check in order to provide data origin authenticity is bad practice and should be avoided.

[1] Number used only once.

Authenticated encryption

If both data origin authenticity and data confidentiality are required, a concept known as authenticated encryption may be used. While authenticated encryption may be obtained through use of dedicated algorithms such as AEGIS, it may also be obtained through generic compositions of encryption algorithms and MACs although one ought to be careful. The *recommended* generic composition is known as *Encrypt-then-MAC*, in which the MAC is computed over the ciphertext. The flow in an Encrypt-then-MAC scheme would be encrypt → authenticate → validate → decrypt. In addition to being the most secure composition, it is also efficient in the sense that invalid messages are discarded before they are decrypted (Bellare and Namprempre, 2008).

3. The CryptoToolbox

The CryptoToolbox (Solnør, 2020) was developed to give easy-access to state-of-the-art high-performance cryptographic algorithms and contains a range of cryptographic algorithms that provide either encryption, MACs, or authenticated encryption. Figure 3 illustrates the structure of the CryptoToolbox contents, while a brief summary explaning the contents of the CryptoToolbox is found in Table 1. Each algorithm operates on memory buffers, and it is assumed that the data that is to be processed is contiguous in memory.

Note that the direct operation of AES, called the Electronic Codebook (ECB) mode, has been deliberately excluded from the CryptoToolbox. The reason for this is that the use of ECB mode results in the same plaintexts consistently being mapped to the same ciphertexts. Because of this, the structure of the plaintext leaks through to the ciphertext, and data confidentiality is lost under very real circumstances. The ECB mode has been misused in multiple previous publications (for example Gupta and Chow (2008), Pang et al. (2011), and Jithish and Sankaran (2017)), and because there is no scenario in which the ECB mode should be used in a feedback control system, the ECB mode has been excluded from the CryptoToolbox to limit the possibility of user errors.

The properties of the cryptographic algorithms in the CryptoToolbox are summed up in Table 2. As we proceed, we will show how the cryptographic algorithms from the CryptoToolbox may be used to ensure that the feedback control signals remain secure across the insecure transmission channels shown in Fig-

Table 1: An explanation of the contents of the CryptoToolbox.

Hash		Cryptographic hash functions are unkeyed, accept inputs of arbitrary length and produces a fixed-length output called a *digest*.
	SHA-256	A variant of the Secure Hash Algorithm 2 (SHA-2) producing a 256-bit digest (Dang, 2015).
Authentication		Contains keyed message authentication codes that accept inputs of arbitrary length and produces a fixed-length output called a tag.
	HMAC-SHA-256	A variant of the Keyed-Hash Message Authentication Code (HMAC) using SHA-256 as the underlying cryptographic hash function (Dang, 2008).
BlockCiphers		Contains stateless encryption algorithms and algorithms deduced from these stateless encryption algorithms.
	AES	The Advanced Encryption Standard, a NIST certified block cipher (NIST, 2001).
	AES CFB	A way of operating AES as a self-synchronizing stream cipher, called the cipher feedback (CFB) mode (Dworkin, 2001).
	AES CTR	A way of operating AES as a synchronous stream cipher, called the counter (CTR) mode (Dworkin, 2001).
	AES x86	An implementation of AES taking advantage of an enhanced instruction set on (most) x86 processors called Advanced Encryption Standard New Instructions (AES-NI), which provides hardware support for AES.
	AES ARM	An implementation of AES taking advantage of an enhanced instruction set on (some) ARMv8 processors called the ARMv8 Cryptographic Extension, which provides hardware support for AES.
	Serpent	A block cipher that was the runner up submission to AES (Anderson et al., 2000). Used as part of the Sosemanuk stream cipher.
StreamCiphers		Contains stateful encryption algorithms.
	AEGIS	A stream cipher that provides authenticated encryption directly. Based on the AES block cipher. Part of the CAESAR portfolio (Wu and Preneel, 2014).
	AEGIS x86	An implementation of AEGIS that takes advantage of AES-NI.
	AEGIS ARM	An implementation of AEGIS that takes advantage of the ARMv8 Cryptographic Extension.
	HC-128	A synchronous stream cipher. Part of the eSTREAM portfolio (Wu, 2008).
	ChaCha	A synchronous stream cipher. Part of the eSTREAM portfolio. May be operated as the full cipher (ChaCha20), or in round reduced variants (ChaCha12, ChaCha8) for increased performance at the cost of reduced security (Bernstein, 2008).
	Rabbit	A synchronous stream cipher. Part of the eSTREAM portfolio (Boesgaard et al., 2008).
	Sosemanuk	A synchronous stream cipher. Part of the eSTREAM portfolio (Berbain et al., 2008).
Encoders		Components that convert data to/from specific formats.
	Hex	Converts data to/from hexadecimal encoding.

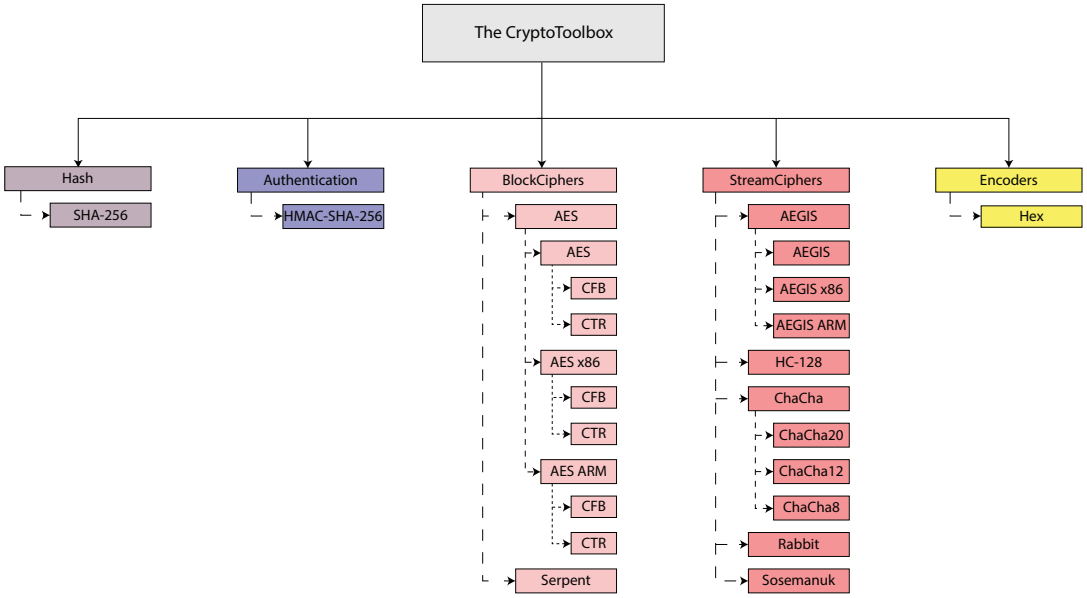

Figure 3: An overview of the algorithms available through the CryptoToolbox.

ure 2. More details regarding the cryptographic algorithms, compilation options, and implementation-related specifics are found in Appendix A for the interested reader.

3.1. Important remarks

Note that it is the users' responsibility to supply keys to the algorithms. These should be highly randomized and preferably drawn from a uniform distribution. For algorithms that utilize IVs and nonces, it is the users' responsibility to ensure that repeated IVs and nonces do not occur for a fixed key. This can easily be solved by incrementing the initialization vectors and nonces after each message on the encryption device. Furthermore, it is assumed that the keys are pre-distributed. If these guidelines are not followed, the resulting system will be vulnerable to attacks.

4. Case study: Securing the GNC system of an autonomous vehicle

We proceed by showing how the algorithms from the CryptoToolbox may be used to provide secure signal transmission in the use case described in Section 2. The cryptographic algorithms should be applied immediately before transmission and upon reception, as shown in Figure 4. Notice that the content of the **E** and **D** blocks would depend on whether data confidentiality, data origin authenticity, or both is required. In Algorithms 1 and 2, the general flow of the **E** and **D** blocks is outlined in pseudocode if authenticated encryption is required. If only data confidentiality or data origin authenticity is required, the excessive lines of code, for the encryption or MAC, are removed. Now the question regarding which algorithms the practitioner should choose to implement the **E** and **D** blocks remain.

Algorithm 1 E block outline.

1: Initialize $E_{K,IV}, MAC_K$
2: **while** true **do**
3: Plaintext ← Load Data
4: Ciphertext ← $\mathrm{E}_{K,IV}$(Plaintext)
5: Tag ← MAC_K(IV, Ciphertext)
6: Message ← (IV||Ciphertext||Tag)
7: Transmit Message
8: Update IV
9: **end while**

4.1. When to use which cryptographic primitive?

As described in Section 2.1, it is important to understand that while encryption provides data confidential-

Table 2: An overview of the properties of the cryptographic algorithms in the CryptoToolbox.

Algorithm	Data confidentiality	Data origin authenticity	Additional information
HMAC-SHA-256		X	Used to obtain data origin authenticity to prevent active deception attacks. May also be used in an Encrypt-then-MAC composition with an encryption algorithm to obtain authenticated encryption.
AES CFB	X		Converts the AES block cipher to a self-synchronizing stream cipher. Eliminates the need for IVs. Performs well on small data, subpar performance as the amount of data increases.
AES CTR	X		Converts the AES block cipher to a synchronous stream cipher. Requires IVs, but offers slightly better performance compared to the CFB mode. Subpar performance as the amount of data increases.
AEGIS	X	X	An authenticated encryption algorithm with excellent performance, particularly as the amount of data increases, e.g. on images and point-clouds.
HC-128	X		A synchronous stream cipher with a significant initialization overhead. Should be avoided for small data, but provides excellent performance on bulk encryption.
ChaCha20/12/8	X		A synchronous stream cipher with no initialization overhead. Provides decent performance on small data, worse as the amount of data increases. Better performance achieved for the round reduced variants, at the cost of reduced security.
Rabbit	X		A synchronous stream cipher with a small initialization overhead and excellent performance as the amount of data increases.
Sosemanuk	X		A synchronous stream cipher with a small initialization overhead. Subpar performance for large data.

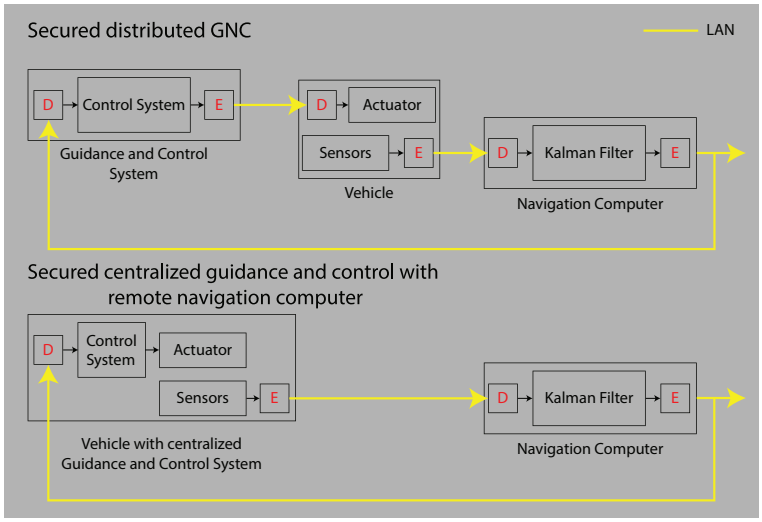

Figure 4: An overview of how a vehicle may be enhanced with secure signal transmission. The cryptographic algorithms are applied immediately before transmission and upon reception. Whether encryption, authentication, or authenticated encryption is applied would depend on which cryptographic properties are of interest.

Algorithm 2 D block outline.

```
Initialize E_{K,IV}, MAC_K
while true do
    Receive (IV'||Ciphertext'||Tag')
    Tag ← MAC_K(IV', Ciphertext')
    if Tag != Tag' then
        Reject message.
    end if
    Plaintext' ← D_{K,IV'}(Ciphertext')
    Accept Plaintext'
end while
```

ity, it does *not* provide data integrity nor data origin authenticity. For this, MACs must be used. Therefore, the block ciphers and stream ciphers described should *only* be used if data confidentiality is required, with the notable exception of AEGIS, which can be used to provide data origin authenticity only, by passing all the data in as authenticated data and none as plaintext, or to provide authenticated encryption. The HMAC-SHA-256 MAC should be used if data origin authenticity is required, but it does *not* provide data confidentiality. We emphasize that using the unkeyed SHA-256 to generate a digest and then encrypting the message and the digest is insecure. After the algorithms have been initialized, the run-time of the algorithms increases linearly with the size of the input.

Without AES-NI and ARM Cryptographic Extension

Without hardware acceleration support, the AES block cipher provides decent performance on small packets ($<$ 1 KB) with no initialization overhead for each packet. If traffic expansion and network congestion is a concern, the self-synchronizing CFB mode may be used to eliminate the need to transmit IVs. Otherwise, CTR mode may be used. The HC-128 stream cipher should be avoided for small packets due to the significant initialization overhead. When used in conjunction with HMAC-SHA-256, the AES, ChaCha, Rabbit, and Sosemanuk ciphers achieve authenticated encryption in an Encrypt-then-MAC composition with the cryptographic algorithms inducing less than 1 ms latency for the encryption & authentication and verification & decryption processes combined on modern computers ($<$ 300 μs on a Raspberry Pi 3+).

For mid-range data (1 KB - 64 KB) the ChaCha, Rabbit, and HC-128 stream ciphers offer the strongest encryption performance, while the EtM composition of HMAC-SHA-256 with ChaCha, Rabbit, and HC-128 offer nearly the same authenticated encryption performance as the AEGIS stream cipher, with AEGIS gaining the upper-hand as the data size increases.

For large quantities of data, e.g. vision-based signals such as video streams and point-clouds, the Rabbit and HC-128 stream ciphers offer the best encryption performance with encryption & decryption combined of a 1.3 MB image taking less than 4 ms and a 3.2 MB

point-cloud taking less than 10 ms on an Nvidia Jetson Xavier (Volden and Solnør, 2020). For authenticated encryption on large data AEGIS should be used, with encryption & authentication and decryption & verification combined inducing approximately 8 ms and 18 ms latency on an image and a point-cloud on an Nvidia Jetson Xavier, respectively (Volden and Solnør, 2020). Note that these numbers should be used as guidelines, and will vary depending on the system specification. However, the relative performance between the algorithms are expected to be similar between different systems.

With AES-NI or ARM Cryptographic Extension With hardware support, AES and AEGIS offer the by far best performance. The AES CTR and CFB implementations may be considered for encryption-only operations on small quantities of data, with the latter being preferred if traffic expansion and network congestion is a concern. For larger quantities of data and authenticated encryption, AEGIS should be used. The hardware-accelerated implementation of AEGIS reduces the induced latency by approximately 65% compared to the portable software implementation when processing a 1.3 MB image and a 3.2 MB point-cloud on an Nvidia Jetson Xavier, performing encryption & authentication and verification & decryption combined of a 1.3 MB image in 2.9 ms and a 3.2 MB point-cloud in 6.5 ms on an Nvidia Jetson Xavier (Volden and Solnør, 2020).

4.2. Implementing E and D

To assist the reader in implementing the scheme proposed in Figure 4 to obtain secure signal transmission, code is provided to obtain data confidentiality, data origin authenticity, or both in Appendix B. The code is generic, with pseudocode for the transmitter, receiver, data loading, and acceptance interface. The `DATA_SIZE` parameter is a parameter to denote the number of bytes that are to be processed. To obtain data confidentiality, the reader may use the Rabbit cipher as shown in B.1. To obtain data origin authenticity, the reader may use the HMAC-SHA-256 MAC as shown in B.2. To obtain data confidentiality and data origin authenticity the reader may use the Rabbit cipher and the HMAC-SHA-256 in an 'Encrypt-then-MAC' composition as shown in B.3, or the authenticated encryption algorithm AEGIS directly as described in B.4.

In the data confidentiality example, and in the 'Encrypt-then-MAC' composition, the Rabbit cipher can be changed with any of the other encryption algorithms that provide data confidentiality. Note that while the interfaces are quite similar, there may be some minor differences. If interested, the reader should consult Appendix A, or look at the sample programs in their respective CryptoToolbox folders.

Notice that neither encryption nor MACs provide direct protection against *replay attacks*. Replay attacks are active attacks in which an adversary has logged valid messages and inject them into the transmission at a later stage to disrupt the system. However, protection against replay attacks is easy to achieve by combining MACs with some additional logic, such as timestamps or sequence numbers, to ensure that old packets are dismissed. The MAC should then be computed over the timestamp or sequence number in addition to the data. Encryption may or may not be applied.

5. Conclusion

In this article, the CryptoToolbox for control applications has been presented. The toolbox contains implementations of several high-performance cryptographic algorithms that provide data confidentiality, data origin authenticity, or both. Examples illustrating how the cryptographic algorithms may be used to obtain data confidentiality and data origin authenticity across insecure transmission channels in feedback control systems have been shown, and an example with a GNC system has been presented. The latency induced by the algorithms is very low and well-suited for real-time applications, and synchronous behavior is guaranteed when the algorithms are operated correctly.

Future work

The CryptoToolbox is planned to undergo further development, with the addition of additional cryptographic algorithms in the future.

Acknowledgments

This work was supported by the Norwegian Research Council (project no. 223254) through the NTNU Center of Autonomous Marine Operations and Systems (AMOS) at the Norwegian University of Science and Technology.

References

Anderson, R., Biham, E., and Knudsen, L. The case for serpent. 2000.

Bellare, M. and Namprempre, C. Authenticated encryption: Relations among notions and analysis of

the generic composition paradigm. *J. Cryptol.*, 2008. 21(4):469–491. doi:10.1007/s00145-008-9026-x.

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., and Sibert, H. *Sosemanuk, a fast software-oriented stream cipher*, page 98–118. Springer-Verlag, Berlin, Heidelberg, 2008.

Bernstein, D. Chacha, a variant of salsa20. 2008.

Biryukov, A. and Wagner, D. Slide attacks. In L. Knudsen, editor, *Fast Software Encryption*. Springer Berlin Heidelberg, Berlin, Heidelberg, pages 245–259, 1999.

Boesgaard, M., Vesterager, M., and Zenner, E. *The Rabbit Stream Cipher*, page 69–83. Springer-Verlag, Berlin, Heidelberg, 2008.

Crutchfield, C. *Implementing and Optimizing Encryption Algorithms for the ARMv8-A Architecture*. Master's thesis, California State University - Sacramento, 6000 J St, Sacramento, CA 95819, USA, 2014.

Dai, W. Crypto++. 2020. URL https://www.cryptopp.com/. Accessed: 2020-12-16.

Dang, Q. H. The keyed-hash message authentication code (hmac) - fips 198-1. Technical report, Gaithersburg, MD, USA, 2008.

Dang, Q. H. Secure hash standard - fips 180-4. Technical report, Gaithersburg, MD, USA, 2015.

de Sá, A. O., d. C. Carmo, L. F. R., and Machado, R. C. S. Covert attacks in cyber-physical control systems. *IEEE Transactions on Industrial Informatics*, 2017. 13(4):1641–1651.

Duong, T. and Rizzo, J. Here come the ⊕ ninjas, 2011. Unpublished.

Dworkin, M. J. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, Gaithersburg, MD, USA, 2001.

Gupta, R. A. and Chow, M. Performance assessment and compensation for secure networked control systems. In *2008 34th Annual Conference of IEEE Industrial Electronics*. pages 2929–2934, 2008.

Hespanha, J. P., Naghshtabrizi, P., and Xu, Y. A survey of recent results in networked control systems. *Proceedings of the IEEE*, 2007. 95(1):138–162.

Jithish, J. and Sankaran, S. Securing networked control systems: Modeling attacks and defenses. In *2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. pages 7–11, 2017.

Lera, F. J. R., Balsa, J., Casado, F., Fernández, C., Rico, F. M., and Matellán, V. Cybersecurity in autonomous systems: Evaluating the performance of hardening ros. *Málaga, Spain*, 2016. 47.

Maritime Robotics. The portable usv system. 2020. URL https://www.maritimerobotics.com/otter. Accessed: 2020-12-18.

NIST. Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

OpenSSL Software Foundation. Openssl. 2020. URL https://www.openssl.org/. Accessed: 2020-12-20.

Osvik, D. A. Speeding up serpent. In *AES Candidate Conference*. 2000.

Pang, Z., Zheng, G., Liu, G., and Luo, C. Secure transmission mechanism for networked control systems under deception attacks. In *2011 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems*. pages 27–32, 2011.

Rodríguez-Lera, F. J., Matellán-Olivera, V., Balsa-Comerón, J., Guerrero-Higueras, n. M., and Fernández-Llamas, C. Message encryption in robot operating system: Collateral effects of hardening mobile robots. *Frontiers in ICT*, 2018. 5:11. doi:10.3389/fict.2018.00002.

Solnør, P. CryptoToolbox. https://github.com/pettsol/CryptoToolbox, 2020.

Teixeira, A., Sou, K. C., Sandberg, H., and Johansson, K. H. *Quantifying Cyber-Security for Networked Control Systems*, pages 123–142. Springer International Publishing, Heidelberg, 2013. doi:10.1007/978-3-319-01159-2_7.

Volden, Ø. and Solnør, P. Crypto ROS: Real-time authenticated encryption of vision-based sensor signals in ROS. https://github.com/oysteinvolden/Real-time-sensor-encryption, 2020.

wolfSSL Inc. wolfcrypt. 2020. URL https://www.wolfssl.com/products/wolfcrypt-2/. Accessed: 2020-12-20.

Wu, H. The stream cipher hc-128. In *The eSTREAM Finalists*. 2008.

Wu, H. and Preneel, B. Aegis: A fast authenticated encryption algorithm. In T. Lange, K. Lauter, and P. Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*. Springer Berlin Heidelberg, Berlin, Heidelberg, pages 185–201, 2014.

A. CryptoToolbox Algorithm Details

This appendix gives the reader a more detailed introducton to the algorithms and the implementations found in the CryptoToolbox.

A.1. Algorithm implementations

This section presents an overview of the algorithms that are accessible in the CryptoToolbox. The interfaces of the algorithms are explained, along with compilation options that exist for specific algorithms.

A.1.1. The Advanced Encryption Standard

The Advanced Encryption Standard (NIST, 2001) (AES) was the result of an international effort to develop a new block cipher around the year 2000. The winner, the Rijndael cipher, was designed by Vincent Rijmen and Joan Daemen and is a substitution-permutation network. Figure 5 illustrates the structure of the AES cipher. Note that like all block ciphers, AES is stateless. The AES cipher operates on blocks of 128 bits, thus resulting in a fixed $\{0,1\}^{128} \times \{0,1\}^{K} \mapsto \{0,1\}^{128}$ substitution parametrized by the K-bit key if operated directly. The official AES standard accepts three key sizes; 128, 192, and 256 bits, respectively. The CryptoToolbox implementations accept 128-bit keys. The direct operation of a block cipher is known as the Electronic Codebook (ECB) mode and leaks structural information from the plaintext to the ciphertext. This leak is an unfortunate characteristic, and block ciphers are therefore primarily operated in other modes of operation such as the Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes (Dworkin, 2001). The CryptoToolbox contains implementations of AES operating in the CTR mode of operation and a modified CFB mode of operation.

As seen in Figure 5, the AES round function consists of four operations. A byte substitution element, commonly referred to as an S-box, provides the nonlinearity. A shift row and a mix column operation provide the diffusion. Finally, a round key is added to prevent slide attacks (Biryukov and Wagner, 1999). The round keys are derived from the secret key. Because

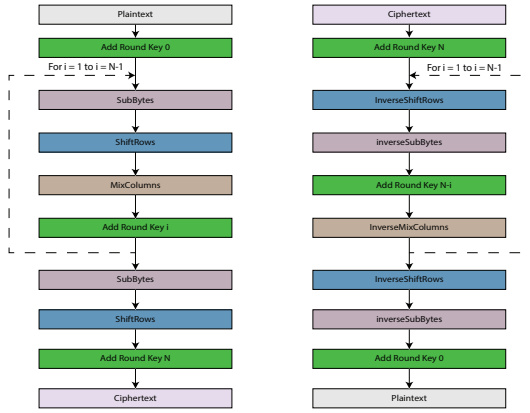

Figure 5: The overall structure of AES, with the encryption mode on the left and the decryption mode on the right. For CFB and CTR mode the encryption mode is used both during encryption and decryption.

the byte substitution operates on bytes and consists of computationally expensive operations such as exponentiations, and because the mix column operation consists of matrix multiplications, the round function is very inefficient if implemented directly. At the very least, the byte substitution should be pre-computed and implemented as a lookup table. However, because most systems today have 32 or 64-bit word sizes and because we still have to deal with the matrix multiplication step, such an implementation is not very efficient. Therefore, the CryptoToolbox implementation of the AES round function uses a time-memory trade-off in which the byte substitution, shift row, and mix column operations are pre-computed and stored in four 1 KB lookup tables. As such, an iteration of the AES round function requires only 16 table lookups and 16 bitwise XOR ($\oplus$) operations. The AES documentation provides a detailed description of how to compute these lookup tables.

Counter mode

The Counter (CTR) mode transforms the block cipher into a synchronous stream cipher by introducing a state determined by a nonce and a counter value. The nonce combined with the counter value is often referred to as the initialization vector (IV) and serves as input to the block cipher. The output of the block cipher is called the keystream, and after each iteration, the cipher increments the counter value. The keystream is then mixed with the plaintext or ciphertext through

the $\oplus$-operator to form the ciphertext or plaintext, respectively. If packets arrive out of order, or if a message is lost or injected, the transmitter and the receiver of a transmission encrypted with a synchronous stream cipher lose synchronization. The IV acts as a synchronization mechanism to provide robustness against such events. Because the IV is a public parameter it may be transmitted along with the ciphertext in the plaintext. Note that only the nonce needs to be transmitted, as the counter value can be agreed upon beforehand (e.g. by always initializing the counter value to zero for each message). The size of the nonce and the counter value depends on the application; if small packets are transmitted at a high frequency, the nonce value is chosen to be large (e.g. 96 bits for AES). If large packets are transmitted less frequently, more bits can be reserved to the counter value. A common configuration for AES consists of 96 bits reserved to the nonce value and 32 bits reserved to the counter value. This is the configuration used by the CryptoToolbox implementation, and the counter value is always initialized to 1 for a new message.

The AES CTR cipher is accessed through the interface seen in Listing 1.

Listing 1: AES CTR Interface

```
void aes_load_key(aes_state *cs, uint8_t key[16]);
void aes_load_iv(aes_state *cs, uint8_t nonce[12])
    ;
void aes_ctr_process_packet(aes_state *cs, uint8_t
     *out, uint8_t *in, int size);
```

Note that the `aes_load_key()` function is only called once per key to derive the round keys, while the `aes_load_iv()` function is called to resynchronize the transmitter and the receiver using the public nonce, usually on a per-message basis. Both encryption and decryption is achieved through the `aes_ctr_process_packet()` function.

Cipher Feedback mode

For some applications, it may be desirable to minimize the amount of data that is to be transmitted. Because stateful ciphers often require IVs to guarantee synchronous behavior between the transmitter and the receiver, each message must carry a (unique) IV in addition to the ciphertext.

The Cipher Feedback (CFB) mode converts the block cipher to a self-synchronizing stream cipher by making the state uniquely determined by a finite number of ciphertext bits. By modifying the CFB mode slightly, the need for IVs can be removed by using the final ciphertext block of the previous message as the IV for the next message, thus reducing the amount of data that must be transmitted. This is referred to this as a carry-over IV design. It may be tempting to apply a similar modification to the Cipher Block Chaining mode, but due to the nature of the CBC decryption mode, such an implementation is vulnerable to attacks as shown by the famous BEAST attack by Duong and Rizzo (2011). For this reason, the National Institute of Standards and Technology (NIST) recommends that the IVs for both CFB and CBC mode should be *unpredictable* in addition to being unique. Thus, even though no attacks are known against this modified CFB mode, we warn that this implementation defies best-practice as defined by NIST.

Because the state is uniquely determined by a finite number of ciphertext bits, a transmission error propagates and results in burst errors. This can happen e.g. if packets are received out-of-order, if packets are injected or if packets are lost in transmission.

An illustration of the carry-over IV CFB mode can be seen in Figure 6. Unlike a block cipher operating in CTR mode, a block cipher operating in CFB mode must be aware of whether is it used to encrypt or decrypt data. This is done by passing either of the predefined macros `ENCRYPT` and `DECRYPT` in the final function argument.

The AES CFB cipher is accessed through the interface seen in Listing 2.

Listing 2: AES CFB Interface

```
void aes_cfb_initialize(aes_state *cs, uint8_t key
    [16], uint8_t iv[16]);
void aes_cfb_process_packet(aes_state *cs, uint8_t
     *out, uint8_t *in, int size, int mode);
```

The cipher is only initialized once per fixed key using `aes_cfb_initialize()`, after which the `aes_cfb_process_packet()` function is used to encrypt and decrypt.

AES on `x86` and `ARMv8`

Because of the wide adoption of AES, microprocessor manufacturers have included enhanced instruction sets that provide hardware-acceleration of the AES operations. In 2010 Intel included the Intel Advanced Encryption Standard New Instructions (AES-NI) on their `x86`-processors. Advanced Micro Devices followed shortly after, and included AES-NI on their `x86`-processors. Later, ARM provided an optional cryptographic extension to their `ARMv8`-processors, the `ARMv8` Crypto Extension. These instructions may easily be accessed through intrinsic functions.

On systems with a modern `x86` processor with the AES-NI instruction set available, the user may compile AES CTR and AES CFB using the `g++` commands seen in Listing 3 to take advantage of the AES-NI instructions.

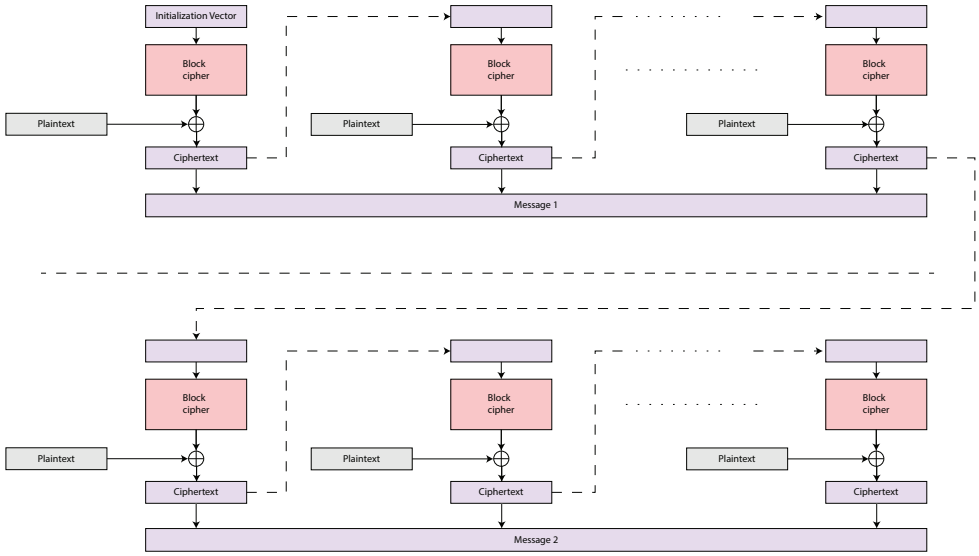

Figure 6: A block cipher operated in CFB mode, with a carry-over IV.

Listing 3: AES x86 AES-NI Compilation.

```
g++ test_vectors.cpp aes_ctr.cpp ../../../../
    Encoders/Hex/encoder.cpp -o test_vectors -D
    x86_INTRINSICS -march=native
g++ main.cpp aes_cfb.cpp -o main -D x86_INTRINSICS
     -march=native
```

On systems running an `ARMv8` processor with the `ARMv8` Crypto Extension instruction set available, the user may compile AES CTR and AES CFB using the `g++` commands seen in Listing 4 to take advantage of the `ARMv8` Crypto Extension instructions.

Listing 4: AES ARMv8 Crypto Extension Compilation.

```
g++ test_vectors.cpp aes_ctr.cpp ../../../../
    Encoders/Hex/encoder.cpp -o test_vectors -D
    ARM_INTRINSICS -march=armv8-a+crypto
g++ main.cpp aes_cfb.cpp -o main -march=armv8-a+
    crypto -D ARM_INTRINSICS
```

Note that these hardware-accelerated variants are, in addition to being faster, less prone to *side-channel attacks*, i.e. attacks that target the algorithm implementations rather than the algorithms themselves. An example of such a side-channel attack is the timing attack in which an adversary attempts to extract information based on the time certain operations take. For example, there may be variations in the time required to compute multiplication operations depending on the inputs, and the time needed to access lookup tables depends on where the lookup tables are stored, such as the level-1 cache or level-2 cache.

A.1.2. Sosemanuk

The Sosemanuk stream cipher was the result of a cooperative effort between multiple French cryptologists and was submitted by Berbain et al. (2008) to the eSTREAM competition. The Sosemanuk stream cipher consists of a linear feedback shift register composed with a nonlinear output function. The nonlinear output function is constructed using components from the Serpent block cipher designed by Anderson et al. (2000), which was the runner-up submission to the AES-process. An overview of the Sosemanuk cipher can be seen in Figure 7.

The Sosemanuk cipher is accessed through the interface seen in Listing 5.

Listing 5: The Sosemanuk Interface.

```
void sosemanuk_load_key(sosemanuk_state *cs,
    uint8_t *key, int keysize);
void sosemanuk_load_iv(sosemanuk_state *cs,
    uint8_t iv[16]);
void sosemanuk_process_packet(sosemanuk_state *cs,
     uint8_t *out, uint8_t *in, uint64_t size);
```

The `sosemanuk_load_key()` function is called once per key, while the `sosemanuk_load_iv()` function is called to resynchronize the transmitter and the receiver by deducing an initial state of the cipher from the pre-

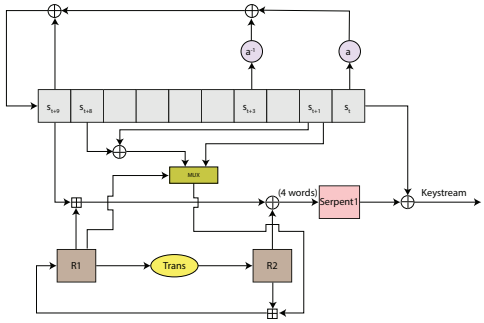

Figure 7: An overview of the Sosemanuk stream cipher.

loaded key and an IV. This is usually done on a per-message basis. Encryption and decryption is achieved through the `sosemanuk_process_packet()` function.

Serpent

The Serpent block cipher is a substitution-permutation network like AES. As in AES, the nonlinear component of the cipher consists of S-boxes. However, because the Serpent S-boxes are $\{0,1\}^4 \mapsto \{0,1\}^4$ mappings, they do not lend themselves well to lookup table implementations. Instead, a bit-slicing technique may be applied. In the CryptoToolbox implementation, the bit-slicing techniques proposed by Osvik (2000) is used to implement the Serpent S-boxes. The Serpent block cipher is accessed indirectly through the Sosemanuk function calls, and it is noted that only the parts used in the Sosemanuk cipher are implemented. The Serpent block cipher is therefore not available as a standalone cipher. An implementation of a bit-sliced Osvik S-box used in the Serpent cipher can be seen in Listing 6.

Listing 6: A Bitsliced Osvik S-Box for the Serpent Block Cipher.

```
inline void S4(uint32_t *r0, uint32_t *r1,
    uint32_t *r2, uint32_t *r3, uint32_t *r4)
{
        *r1 ^= *r3; *r3 = ~(*r3);
        *r2 ^= *r3; *r3 ^= *r0;
        *r4 = *r1; *r1 &= *r3;
        *r1 ^= *r2; *r4 ^= *r3;
        *r0 ^= *r4; *r2 &= *r4;
        *r2 ^= *r0; *r0 &= *r1;
        *r3 ^= *r0; *r4 |= *r1;
        *r4 ^= *r0; *r0 |= *r3;
        *r0 ^= *r2; *r2 &= *r3;
        *r0 = ~(*r0); *r4 ^= *r2;
}
```

A.1.3. Rabbit

The Rabbit stream cipher is a cipher designed by Boesgaard et al. (2008) that was a successful entrant to the eSTREAM competition. The theoretical foundation of the Rabbit cipher comes from the theory of chaotic systems. The cipher deduces a secret *master state* from the key, and each IV is mixed with the master state to produce an initial state of the cipher.

The Rabbit stream cipher is accessed through the interface seen in Listing 7.

Listing 7: The Rabbit Interface.

```
void rabbit_load_key(rabbit_state *cs, uint8_t key
    [16]);
void rabbit_load_iv(rabbit_state *cs, uint8_t iv
    [8]);
void rabbit_process_packet(rabbit_state *cs,
    uint8_t *output, uint8_t *input, uint64_t
    size);
```

The `rabbit_load_key()` function deduces the master state, and is called once per key. The `rabbit_load_iv()` function derives an initial state from the master state and a public IV. This is usually done on a per-message basis. The `rabbit_process_packet()` function is used to encrypt and decrypt.

A.1.4. ChaCha

The ChaCha stream cipher is a variant of the Salsa-family of stream ciphers and was designed by Bernstein (2008). The ChaCha stream cipher follows an Add-Rotate-XOR-design, and is generally used in three configurations; the full cipher consisting of twenty rounds (ChaCha20), a round reduced variant consisting of twelve rounds (ChaCha20/12), and a further round reduced variant consisting of eight rounds (ChaCha20/8). The round reduced variants offer increased performance at the cost of reduced security. The CryptoToolbox provides the full ChaCha20 cipher as default, however, the round reduced variants may be accessed by passing the `-D TWELVE_ROUNDS` and `-D EIGHT_ROUNDS` preprocessor flags for the twelve and eight round variants, respectively, as seen in Listing 8:

Listing 8: The ChaCha Compilation Options for Round-Reduced Variants.

```
g++ main.cpp chacha.cpp ../../Encoders/Hex/encoder
    .cpp -o main -D TWELVE_ROUNDS
g++ main.cpp chacha.cpp ../../Encoders/Hex/encoder
    .cpp -o main -D EIGHT_ROUNDS
```

If compiled using `CMAKE`, the preprocessor flags can be set using `add_definitions()`. All variants of the ChaCha stream cipher are accessed through the interface seen in Listing 9.

Listing 9: The ChaCha Interface.

```
void chacha_initialize(chacha_state *cs, uint8_t
    key[32], uint8_t nonce[12]);
void chacha_process_packet(chacha_state *cs,
    uint8_t *output, uint8_t *input, uint64_t
    size);
```

The `chacha_initialize()` function is used to derive an initial state from the secret key and the public IV, normally on a per-message basis, after which `chacha_process_packet()` is used to encrypt and decrypt.

The core of the ChaCha stream cipher revolves around the quarter-round function shown in Listing 10. Notice that only modular additions, 32-bit rotations and bitwise $\oplus$-operations are used.

Listing 10: The ChaCha Quarter-Round Function.

```
inline void q_round(chacha_state *cs, int a, int b
    , int c, int d){

  cs->state[a] += cs->state[b];
  cs->state[d] ^= cs->state[a];
  cs->state[d] = ROTL_32((cs->state[d]), 16);

  cs->state[c] += cs->state[d];
  cs->state[b] ^= cs->state[c];
  cs->state[b] = ROTL_32((cs->state[b]), 12);

  cs->state[a] += cs->state[b];
  cs->state[d] ^= cs->state[a];
  cs->state[d] = ROTL_32((cs->state[d]), 8);

  cs->state[c] += cs->state[d];
  cs->state[b] ^= cs->state[c];
  cs->state[b] = ROTL_32((cs->state[b]), 7);
}
```

A.1.5. HC-128

The HC-128 stream cipher was designed by Wu (2008) and rely on large permutation tables. The HC-128 cipher offers excellent performance on bulk-encryption, at the cost of a large initialization overhead. The cipher, therefore, suffers from poor performance if small packets are encrypted frequently.

The HC-128 stream cipher is accessed through the interface seen in Listing 11:

Listing 11: The HC-128 Interface.

```
void hc128_initialize(hc128_state *cs, uint8_t key
    [16], uint8_t iv[16]);
void hc128_process_packet(hc128_state *cs, uint8_t
     *output, uint8_t *input, uint64_t size);
```

The `hc128_initialize()` function derives an initial state from the secret key and IV by mapping the key and the IV to the tables containing the state, and iterating the cipher 1024 times. Once initialized, the `hc128_process_packet()` function is used to encrypt and decrypt.

The remarkably efficient keystream generator function of the HC-128 stream cipher can be seen in Listing 12. Note that $g_{1,2}$ and $h_{1,2}$ are functions consisting only of 32-bit rotations, modular additions, and bitwise $\oplus$-operations, while P and Q denote the tables that make up the state of the cipher.

Listing 12: The HC-128 Keystream Generator Function.

```
void hc128_generate_keystream(hc128_state *cs,
    uint32_t *keystream, uint64_t size)
{
  // Generate keystream
  for (int i = 0; i <= (size-1)/4; i++)
  {
    int j = (i&0x1FF);
    if ( (i&0x3FF) < 512 )
    {
      // Operate on P
      cs->P[j] = cs->P[j] + g1(cs->P[(j-3)&0x1FF],
          cs->P[(j-10)&0x1FF],
          cs->P[(j-511)&0x1FF]);
      *keystream = h1(cs, cs->P[(j-12)&0x1FF]) ^ (
          cs->P[j]);
      keystream++;
    } else {
      // Operate on Q
      cs->Q[j] = cs->Q[j] + g2(cs->Q[(j-3)&0x1FF],
          cs->Q[(j-10)&0x1FF],
          cs->Q[(j-511)&0x1FF]);
      *keystream = h2(cs, cs->Q[(j-12)&0x1FF]) ^ (
          cs->Q[j]);
      keystream++;
    }
  }
}
```

A.1.6. AEGIS

The AEGIS stream cipher was designed by Wu and Preneel (2014) and submitted to the Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR). The AEGIS stream cipher is a cipher that is heavily based on the AES round function and provides authenticated encryption directly. Note that AEGIS also can be used to provide message authenticity without encryption or to authenticate additional data that is not encrypted. This is commonly used to authenticate the IV in plaintext, in addition to the ciphertext. The AEGIS stream cipher is accessed through the interface displayed in Listing 13.

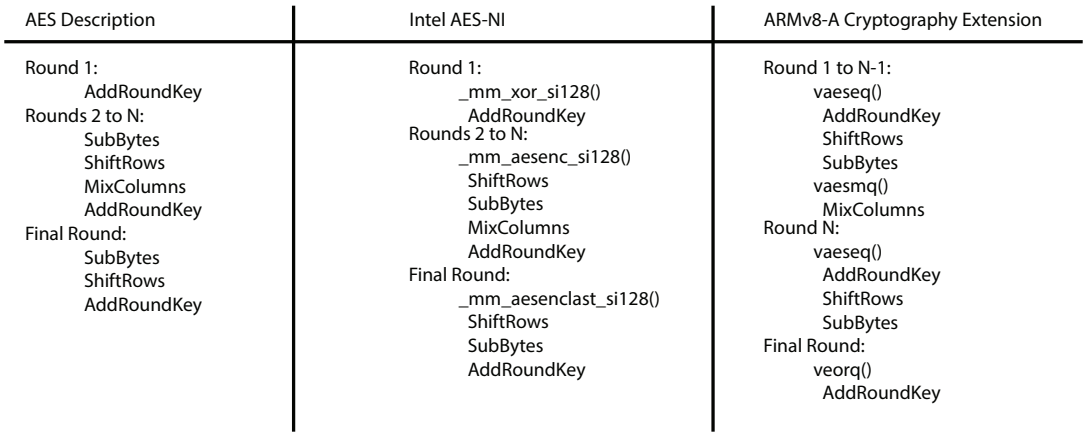

Figure 8: An illustration of the AES description, the AES-NI operations and the `ARMv8` cryptography extension operations. The difference between the AES-NI and `ARMv8` cryptography extension round function means that extra operations are required when using ARM hardware-acceleration. This figure is based on a figure from Crutchfield (2014).

Listing 13: The AEGIS Interface.

```
void aegis_load_key(aegis_state *cs, uint8_t key
    [16]);
void aegis_encrypt_packet(aegis_state *cs, uint8_t
     *ct, uint8_t tag[16], uint8_t *pt, uint8_t *
    ad, uint8_t iv[16], uint64_t adlen, uint64_t
    msglen);
int aegis_decrypt_packet(aegis_state *cs, uint8_t
    *pt, uint8_t *ct, uint8_t *ad, uint8_t iv
    [16], uint8_t tag[16], uint64_t adlen,
    uint64_t msglen);
```

The `aegis_load_key()` function is called once per key, while the `aegis_encrypt_packet()` and `aegis_decrypt_packet()` functions are used to encrypt and decrypt, respectively. Note that the `aegis_decrypt_packet()`-function returns 1 if the (message,tag)-pair is valid. If the (message,tag)-pair is invalid, the pt-buffer is zeroized. This is done to prevent chosen ciphertext attacks.

AEGIS on `x86` and `ARMv8`

Because the AEGIS stream cipher utilizes AES operations, the cipher can take advantage of the enhanced instruction sets provided by some modern microprocessors.

On systems running an `x86` processor with the AES-NI instruction set available, AEGIS is compiled using the `g++` command seen in Listing 14:

Listing 14: AEGIS x86 AES-NI Compilation.

```
g++ test_vectors.cpp aegis_128.cpp ../../../
    Encoders/Hex/encoder.cpp -o test_vectors -D
    x86_INTRINSICS -march=native
```

On systems running an `ARMv8` processor with the `ARMv8` Crypto Extension instruction set available, AEGIS is compiled using the `g++` command seen in Listing 15:

Listing 15: AEGIS ARMv8 Crypto Extension Compilation.

```
g++ test_vectors.cpp aegis_128.cpp ../../../
    Encoders/Hex/encoder.cpp -o test_vectors -
    march=armv8-a+crypto -D ARM_INTRINSICS
```

Notice in Figure 8, however, that the ARMv8 Cryptography Extension intrinsic functions are not perfectly aligned with the 'true' AES round function. Since AEGIS only utilizes the 'true' AES round function and not the first and last rounds, the round keys must be pre- and post-added. An excerpt from the ARMv8 AEGIS implementation in the CryptoToolbox illustrates this in Listing 16.

Listing 16: Reconstruction of AES Round using ARMv8 Intrinsics.

```
#ifdef ARM_INTRINSICS
  // ARM INTRINSICS
  B_S3 = veorq_u8(B_S3, B_KEY);
  B_S3 = vaesmcq_u8(vaeseq_u8(B_S3, B_KEY));
  B_S3 = veorq_u8(B_S3, B_KEY);
  B_TMP = B_KEY;
  vst1q_u8((uint8_t*)cs->s3, B_S3);
```

```
#else
```

A.1.7. Keyed-Hash Message Authentication Code

The Keyed-Hash Message Authentication Code (Dang, 2008) (HMAC) is a construction that converts an *unkeyed* cryptographic hash function to a *keyed* MAC. In the CryptoToolbox, the HMAC algorithm is implemented with the Secure Hash Algorithm 2 (Dang, 2015) (SHA-2), SHA-256 to be more specific. Note that the tag size is a parameter in the range $[0, 32]$ bytes determined by the user. A larger tag size provides greater security against forgery attacks. The size of the key is also a parameter determined by the user. A key size of 32 bytes is recommended, provided that it is generated from a sufficiently random source. The interface to the HMAC-SHA-256 algorithm is displayed in Listing 17.

Listing 17: The HMAC Interface

```
void hmac_load_key(hmac_state *cs, uint8_t *key,
    int keysize);
void hmac_tag_generation(hmac_state *cs, uint8_t*
    tag, uint8_t *message, uint64_t dataLength,
    int tagSize);
int hmac_tag_validation(hmac_state *cs, uint8_t *
    tag, uint8_t *message, uint64_t dataLength,
    int tagSize);
```

The `hmac_load_key()` function is only called once per key, while the `hmac_tag_generation()` and `hmac_tag_validation` is used to generate a valid tag, and authenticate the validity of a (message,tag)-pair, respectively. The `hmac_tag_validation` function returns 1 if the (message,tag)-pair is valid and 0 if invalid.

SHA-256

The unkeyed SHA-256 algorithm is also accessible through the interface displayed in Listing 18. Note that an unkeyed cryptographic hash algorithm *should not* be used to provide message authenticity and integrity directly.

Listing 18: SHA-256 Interface

```
void sha256_process_message(uint8_t *digest,
    uint8_t *message, uint64_t size);
```

A.1.8. Hexadecimal encoding

In addition to the cryptographic algorithms, the CryptoToolbox contains a hexadecimal encoder. The hexadecimal encoder is useful for printing the output of the cryptographic algorithms in a printable format. Because the algorithms operate on buffers of type `uint8_t`, each byte represents a number in the interval $[0, 255]$. However, only numbers in the interval $[32, 255]$ represent printable characters, some of which are unintelligible. The hexadecimal encoder abates this problem by interpreting each byte as a hexadecimal number. The CryptoToolbox also provides a decoder that interprets a buffer of hexadecimal numbers as `uint8_t`. The decoder is generally used in scenarios in which correctly formatted input is needed to confirm the correct operation of an algorithm with official test vectors.

The interfaces for the hexadecimal encoder and decoder are displayed in Listing 19.

Listing 19: The Hexadecimal Encoder and Decoder Interfaces.

```
void hex_encode(char* output, const uint8_t* input
    , int size);
void hex_decode(uint8_t* output, const char* input
    , int size);
```

B. Algorithm Applications

This appendix contains examples of how cryptographic algorithms from the CryptoToolbox may be applied to obtain data confidentiality, data authenticity, or both.

B.1. Encryption and decryption using Rabbit

Listing 20: E-block implemented with Rabbit for data confidentiality

```
#include "rabbit.h"
#include <cstring> // for memcpy

int main()
{

  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
  uint8_t iv[8] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */


  /*One buffer for plaintext and one
    buffer for the ciphertext and IV*/
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE];

  while(1)
  {
    /* Get new data */
    plaintext <- LoadData();
```

```
    /* RABBIT ENCRYPT */
    std::memcpy(message, iv, 8);
    rabbit_load_iv(&cs, iv);
    rabbit_process_packet(&cs, &message[8],
        plaintext, DATA_SIZE);
    (*(uint64_t*)iv)++;
    /* ENCRYPT FINISHED */

    /* Transmit (IV || Ciphertext) */
    Transmit(message);
  }
}
```

Listing 21: D-block implemented with Rabbit for data confidentiality

```
#include "rabbit.h"

int main()
{

  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */

  /*One buffer for plaintext and one
    buffer for the ciphertext and IV*/
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE];

  while(1)
  {
    /* Receive message (IV || Ciphertext) */
    message <- Receiver();

    /* RABBIT DECRYPT */
    rabbit_load_iv(&cs, message);
    rabbit_process_packet(&cs, plaintext, &message
        [8], DATA_SIZE);
    /* DECRYPT FINISHED */

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

B.2. Authentication and verification using HMAC-SHA-256

Listing 22: E-block implemented with HMAC-SHA-256 for data origin authenticity

```
#include "hmac.h"
#include <cstring> // for memcpy

int main()
{

  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
  /* SETUP FINISHED */

  /* One buffer holds the plaintext,
     and the other holds the plaintext
     and the 32-byte tag. */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[DATA_SIZE+32];

  while(1)
  {
    /* Get new data */
    plaintext <- LoadData();

    /* COMPUTE TAG */
    std::memcpy(message, plaintext, DATA_SIZE);
    hmac_tag_generation(&as, &message[DATA_SIZE],
        plaintext, DATA_SIZE, 32);
    /* TAG GENERATION FINISHED */

    /* Transmit (Plaintext || Tag) */
    Transmit(message);
  }
}
```

Listing 23: D-block implemented with HMAC-SHA-256 for data origin authenticity

```
#include "hmac.h"

int main()
{
  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
  /* SETUP FINISHED */

  /* One buffer holds the plaintext,
     and the other holds the plaintext
     and the 32-byte tag. */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[DATA_SIZE+32];

  while(1)
  {
    /* Receive message (Plaintext || Tag) */
    message <- Receiver();

    /* HMAC-SHA-256 VALIDATE MESSAGE */
    if (!hmac_tag_validation(&as, &message[
        DATA_SIZE], message, DATA_SIZE, 32)){
      /* TAG IS INVALID */
      continue;
```

```
    } else {
      std::memcpy(plaintext, message, DATA_SIZE);
    }
    /* MESSAGE VALIDATION FINISHED */

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

B.3. Authenticated encryption using Rabbit and HMAC-SHA-256

Listing 24: E-block implemented with Rabbit and HMAC-SHA-256 for data confidentiality and data origin authenticity

```
#include "rabbit.h"
#include "hmac.h"
#include <cstring> // for memcpy

int main()
{
  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
  uint8_t iv[8] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */

  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
  /* SETUP FINISHED */

  /* Buffer for plaintext, and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE+32];

  while(1)
  {
    /* Get new data */
    plaintext <- LoadData();

    /* RABBIT ENCRYPT */
    std::memcpy(message, iv, 8);
    rabbit_load_iv(&cs, iv);
    rabbit_process_packet(&cs, &message[8],
        plaintext, DATA_SIZE);
    (*(uint64_t*)iv)++;
    /* ENCRYPT FINISHED */

    /* COMPUTE TAG OVER IV AND CIPHERTEXT,
       PLACE BEHIND IV AND CIPHERTEXT */
    hmac_tag_generation(&as, &message[DATA_SIZE
        +8], message, DATA_SIZE+8, 32);
    /* TAG GENERATION FINISHED */

    /* Transmit (IV || Ciphertext || Tag) */
    Transmit(message);
  }
}
```

Listing 25: D-block implemented with Rabbit and HMAC-SHA-256 for data confidentiality and data origin authenticity

```
#include "rabbit.h"
#include "hmac.h"

int main()
{
  /* RABBIT SETUP */
  rabbit_state cs;
  uint8_t key[16] = {0};
  rabbit_load_key(&cs, key);
  /* SETUP FINISHED */

  /* HMAC-SHA-256 SETUP */
  hmac_state as;
  uint8_t a_key[32] = {0};
  hmac_load_key(&as, a_key, 32);
  /* SETUP FINISHED */

  /* Buffer for plaintext, and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[8+DATA_SIZE+32];

  while(1)
  {
    /* Receive message (IV || Ciphertext || Tag)
       */
    message <- Receiver();

    /* HMAC-SHA-256 VALIDATE MESSAGE */
    if (!hmac_tag_validation(&as, &message[8+
        DATA_SIZE], message, DATA_SIZE+8, 32)){
      /* TAG IS INVALID */
      continue;
    }
    /* MESSAGE VALIDATION FINISHED */

    /* RABBIT DECRYPT */
    rabbit_load_iv(&cs, message);
    rabbit_process_packet(&cs, plaintext, &message
        [8], DATA_SIZE);
    /* DECRYPT FINISHED */

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

B.4. Authenticated encryption using AEGIS

Listing 26: E-block implemented with AEGIS for data confidentiality and data origin authenticity

```
#include "aegis_128.h"
#include <cstring> // for memcpy

int main()
{
  /* AEGIS SETUP */
  aegis_state cs;
  uint8_t key[16] = {0};
  uint8_t iv[16] = {0};
  aegis_load_key(&cs, key);
  /* SETUP FINISHED */

  /* Buffer for plaintext and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[16+DATA_SIZE+16];

  while(1)
  {
    /* Get new data*/
    plaintext <- LoadData();

    /* AEGIS ENCRYPT & AUTHENTICATE */
    std::memcpy(message, iv, 16);
    aegis_encrypt_packet(&cs, &message[16], &
        message[16+DATA_SIZE], plaintext, iv, iv,
        16, DATA_SIZE);
    (*(uint64_t*)iv)++;

    /*Transmit (IV || Ciphertext || Tag) */
    Transmit(message);
  }
}
```

Listing 27: D-block implemented with AEGIS for data confidentiality and data origin authenticity

```
#include "aegis_128.h"

int main()
{
  /* AEGIS SETUP */
  aegis_state cs;
  uint8_t key[16] = {0};
  aegis_load_key(&cs, key);
  /* SETUP FINISHED */

  /* Buffer for plaintext and
     (IV || Ciphertext || Tag) */
  uint8_t plaintext[DATA_SIZE];
  uint8_t message[16+DATA_SIZE+16];

  while(1)
  {
    /* Receive message */
    message <- Receiver();

    /* AEGIS VALIDATE & DECRYPT */
    if (!aegis_decrypt_packet(&cs, plaintext, &
        message[16], message, message, &message
        [16+DATA_SIZE], 16, DATA_SIZE))
    {
      // Invalid msg
      continue;
    }
    /*COMPLETED*/

    /* Pass on the data */
    Accept(plaintext);
  }
}
```

Petter Solnør

Authenticated Encryption Methods for Feedback Control Systems

NTNU
Norwegian University of
Science and Technology