Modeling, Identification and Control, Vol. 41, No. 4, 2020, pp. 313–332, 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 S 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.2020.4.3

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 Rodrguez-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

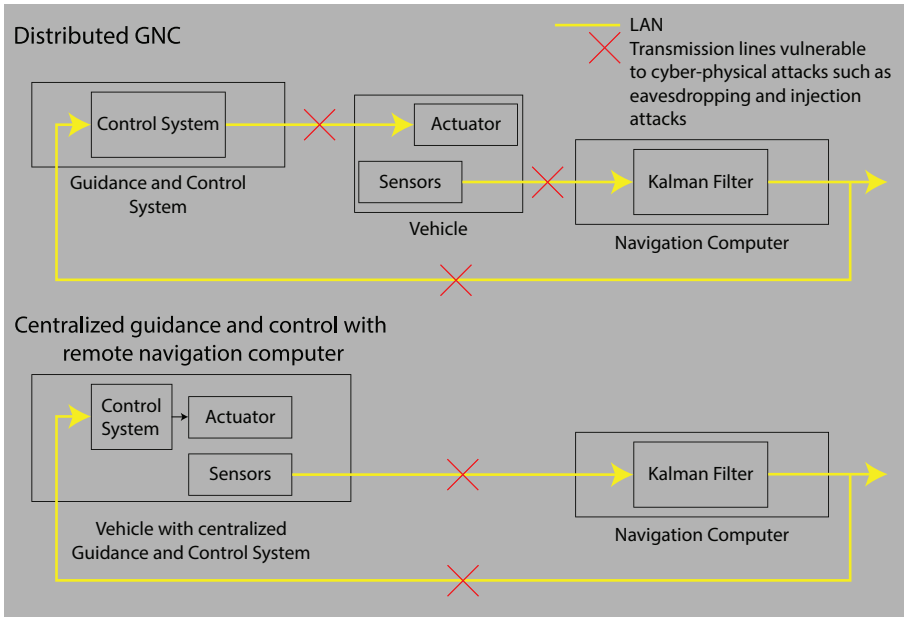

Figure 2: A generic schematic of the signal flow in vehicles with a distributed GNC architecture and a centralized guidance and control computer with a remote navigation computer. Surfaces that are vulnerable to attacks are marked, and the goal is to make the system resistant against attacks on these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

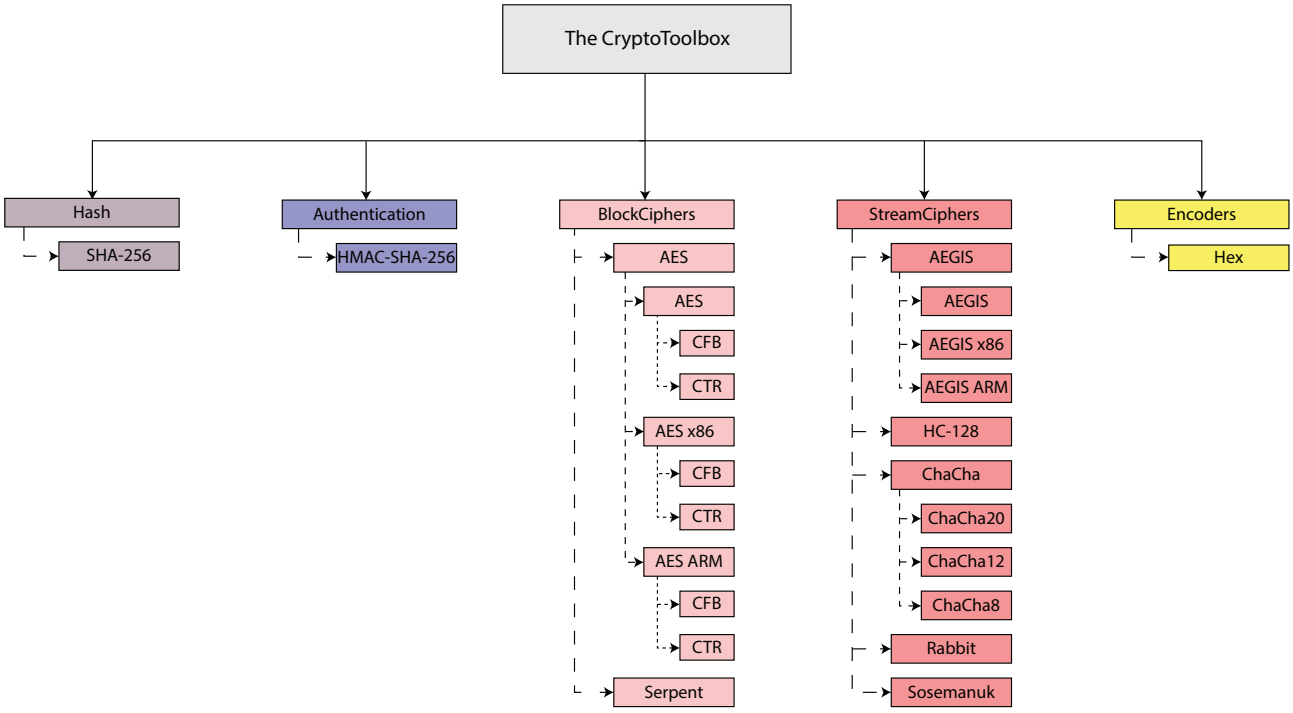

Figure 3: An overview of the algorithms available through the CryptoToolbox.

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.

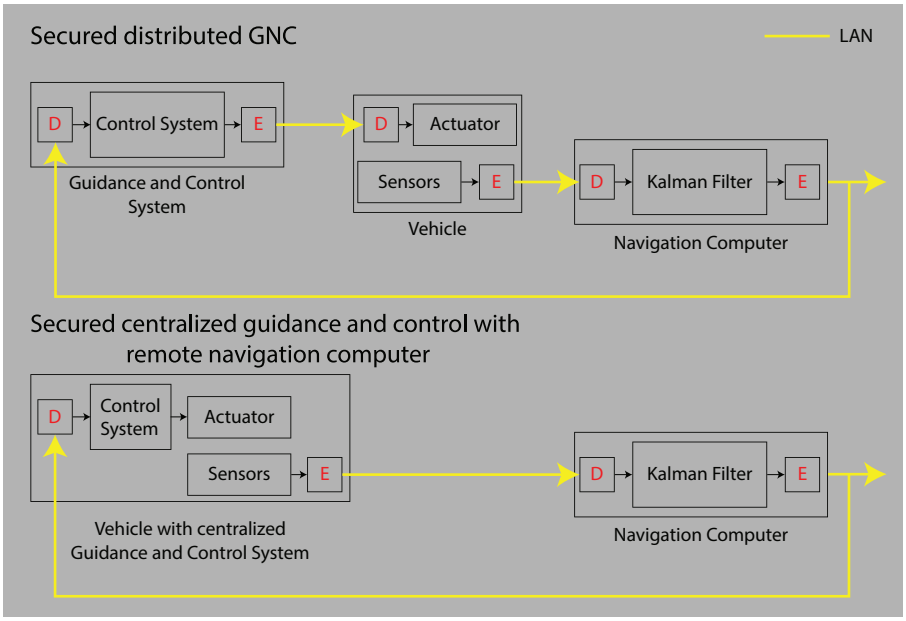

Figure 4: An overview of how a vehicle may be enhanced with secure signal transmission. The cryptographic algorithms are applied immediately before transmission and upon reception. Whether encryption, authentication, or authenticated encryption is applied would depend on which cryptographic properties 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' ← $\text{D}_{K,IV'}$(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):469491. 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 98118. 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 6983. 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 S, 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. doi:10.1109/TII.2017.2676005.

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. doi:10.1109/IECON.2008.4758425.

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. doi:10.1109/JPROC.2006.887288.

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. doi:10.1109/ICCE-ASIA.2017.8309317.

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. doi:10.1109/CYBER.2011.6011758.

Rodrguez-Lera, F. J., Matelln-Olivera, V., Balsa-Comern, J., Guerrero-Higueras, n. M., and Fernndez-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.

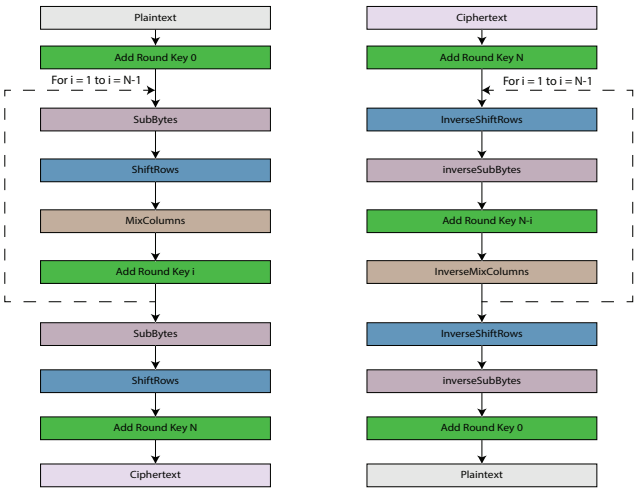

Figure 5: The overall structure of AES, with the encryption mode on the left and the decryption mode on the right. For CFB and CTR mode the encryption mode is used both during encryption and decryption.

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 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`-

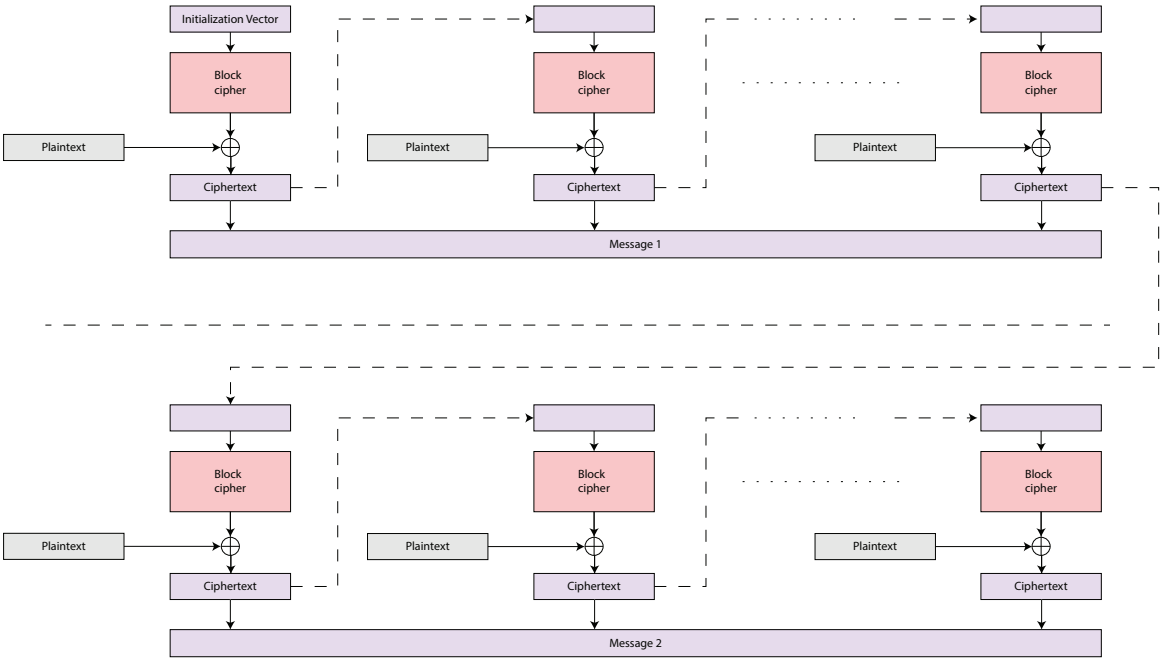

Figure 6: A block cipher operated in CFB mode, with a carry-over IV.

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.

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.

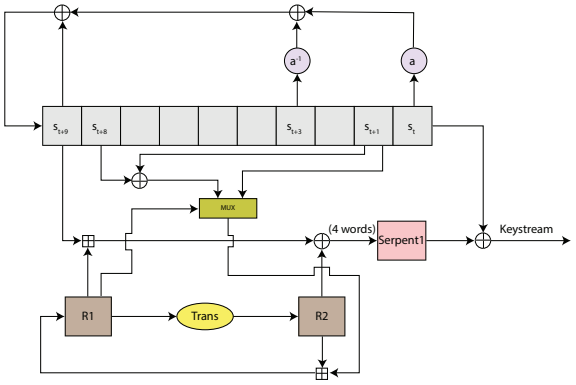

Figure 7: An overview of the Sosemanuk stream cipher.

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 preloaded 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.

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,

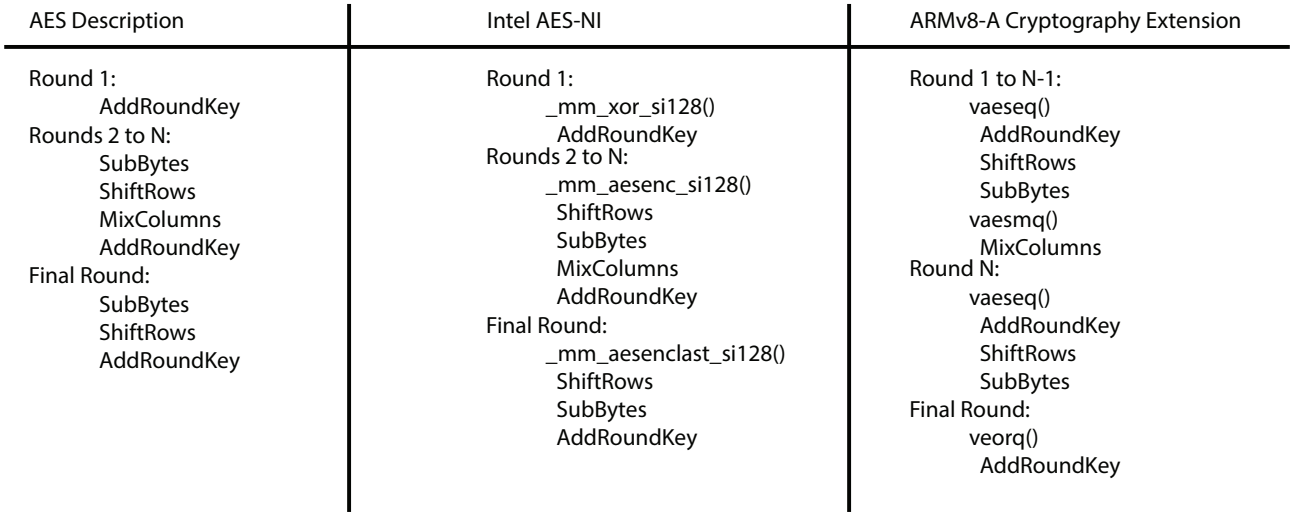

Figure 8: An illustration of the AES description, the AES-NI operations and the ARMv8 cryptography extension operations. The difference between the AES-NI and ARMv8 cryptography extension round function means that extra operations are required when using ARM hardware-acceleration. This figure is based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

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);
  }
}
```