



UNIVERSITÀ DI PISA
DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE
IN COMPUTER ENGINEERING

**DESIGN AND IMPLEMENTATION OF A KEY
DERIVATION MODULE COMPLIANT WITH THE IEEE
802.1X-2010 KEY HIERARCHY**

Candidate:

Pacini Diego

Relators:

Prof. Luca Fanucci

Prof. Gianluca Dini

Ing. Berardino Carnevale

Academic Year 2013/2014

Abstract

Nowadays the automotive environment is more and more characterized by several IT applications: from infotainment systems to C2C (Car to Car) or C2X (Car to external) solutions, as LTE or Wi-Fi connections. This, together with engine and other components controls systems, reflects on many different internal and external networks, which can be found inside a car.

Carmakers are going in the direction of replacing much of these networks by Ethernet networks, to achieve more throughput in order to satisfy clients' expectations.

This work is placed in the context of the deployment of a security module for the automotive requirements compatible with the Ethernet standard, managed by the security team of Renesas Electronics Europe GmbH, one of the world's biggest microcontrollers manufacturer.

After a deep analysis of the main degrees of freedom in the project workspace, a hardware component has been implemented, which acts as accelerator for encryption keys generation, compliant with the MKA Key Hierarchy protocol in the 802.1X-2010 standard.

A testing phase has followed to validate the implemented MKA KH core by an algorithmic point of view: for this purpose, with the help of the official National Institute of Standards and Technology (NIST) test vectors, a Java software has been realized, which generates the required encryption keys compliant with the MKA KH algorithm.

Then the compliancy with respect to the IEEE 802.1AE standard and the full integration inside the MAC IP has been verified.

The realized system has been synthesized both on a FPGA Stratix V of Altera and on a 65nm standard-cell ASIC technology: the system shows an

occupation of 9938 registers and 11477 ALMs on the FPGA and 102,38kgates on the standard-cell technology. The maximum reachable throughput at 125MHz is 1Gbps.

Contents

Abstract	iii
Contents	v
Figures Index.....	viii
1. Security in automotive	1
1.1. Introduction.....	1
1.2. How secure is your car?.....	3
1.3. Ethernet backbone in cars	6
2. Automotive and cryptography	9
2.1. Symmetric Cryptography	11
2.2. Operation modes	11
2.2.1. Stream ciphers	12
2.2.2. Block ciphers	12
2.2.2.1. ECB – Electronic CodeBook.....	13
2.2.2.2. CBC – Cipher Block Chaining	14
2.2.2.3. CFB – Cipher FeedBack	16
2.2.2.4. OFB – Output FeedBack	19
2.2.2.5. CTR – Counter.....	22
2.2.3. Stream vs Block Ciphers	25
2.3. Possible attacks.....	26
2.3.1. Exhaustive key search	26
2.3.2. Data exhaustive analysis	27
2.3.3. Cryptanalysis	28
2.4. Computational security	28
2.5. Attacks in automotive systems	28
3. AES cipher	32
3.1. AES encryption algorithm.....	34
3.1.1. SubBytes() Transformation	36
3.1.2. ShiftRows() Transformation.....	38
3.1.3. MixColumns() Transformation.....	39
3.1.4. AddRoundKey() Transformation.....	41
3.2. Key Expansion algorithm.....	42

3.3. AES Inverse cipher	44
3.4. AES operation modes	44
3.5. AES – CMAC	45
3.5.1.1. Subkey generation algorithm	47
3.5.1.2. MAC generation algorithm	48
3.5.1.3. Security considerations	52
3.6. AES Key Wrap algorithm	52
4. MAC Security (MACsec)	55
4.1. Introduction.....	55
4.2. About MACsec	56
4.2.1. MACsec Benefits.....	56
4.2.2. MACsec limitations	57
4.3. 802.1X without MACsec	58
4.4. 802.1X-2010	59
4.4.1. Secure communication.....	61
4.4.2. Components and Protocols	62
4.5. MACsec Sequence	64
4.5.1. Authentication and Master Key distribution	65
4.5.2. Session key agreement (MKA).....	66
4.5.2.1. KDF (Key Derivation Function).....	67
4.5.2.2. MKA transport	69
4.5.2.3. EAPoL	70
4.5.2.4. SAK generation.....	73
4.5.2.5. CAK derivation	75
4.5.2.6. ICK derivation.....	76
4.5.2.7. KEK derivation	77
4.5.2.8. Message authentication.....	77
5. MKA Key Hierarchy SW Implementation	79
5.1. Java implementation	79
5.1.1. AES – CMAC	80
5.1.1.1. Cipher.class.....	80
5.1.1.2. Results	84
5.1.1.3. Applet Java and Web Server	84
5.1.2. Results and performance.....	86
5.2. C implementation	88

6. MKA Key Hierarchy HW Implementation	90
6.1. AES and Key Expansion modules	90
6.2. CMAC module.....	94
6.3. AES Key Wrap module	95
6.4. KDF module	98
6.5. MKA module	99
6.6. FPGA Synthesis.....	103
6.7. Synthesis on standard-cell ASIC technology	105
7. Conclusions.....	106
Bibliography	108

Figures Index

Figure 1 – Automotive ECUs Controllers by 2020	2
Figure 2 - Hackability results: A plus sign represents “more hackable,” a minus sign “less hackable.”	5
Figure 3 - In-Car Networking Scenario	8
Figure 4 - Symmetric vs Asymmetric Cryptography	10
Figure 5 - Stream cipher general schema	12
Figure 6 - ECB mode schema	13
Figure 7 - ECB equations	13
Figure 8 - CBC mode schema	14
Figure 9 - CBC equations	14
Figure 10 - CFB Encryption	16
Figure 11 - CFB Decryption	16
Figure 12 - CFB Mode	18
Figure 13 - OFB Encryption	19
Figure 14 - OFB Decryption	20
Figure 15 - OFB Mode	21
Figure 16 - CTR Encryption	23
Figure 17 - CTR Decryption	23
Figure 18 - CTR Mode	24
Figure 19 - number of pairs to avoid false positive	27
Figure 20 - An example of car attack	30
Figure 21 - AES scheme	33
Figure 22 - State array input and output	34
Figure 23 - AES cipher suite	35
Figure 24 - SubBytes() applies the S-box to each byte of the State	37
Figure 25 - S-box: substitution values for the byte xy (Hex format)	37
Figure 26 - ShiftRows() cyclically shifts the last three rows in the State	39
Figure 27 - MixColumns() operates on the State column-by-column	40
Figure 28 - AddRoundKey() XORs each column of the State with a word from the key schedule	41
Figure 29 - Key Expansion pseudo code	43
Figure 30 - Two cases of AES-CMAC	46

Figure 31 - CMAC generate_subkey() pseudo code	47
Figure 32 - AES-CMAC pseudo code	50
Figure 33 - AES Key Wrap pseudo code	53
Figure 34 - 802.1X and MACsec	56
Figure 35 - MACsec hop-by-hop basis	57
Figure 36 - 802.1X behavior prior to authentication without MACsec.....	58
Figure 37 - 802.1X behavior after authentication without MACsec.....	59
Figure 38 - MACsec enabled port.....	60
Figure 39 - Secure communication scenario	62
Figure 40 - MACsec components and protocols.....	63
Figure 41 - High Level 802.1X and MACsec sequence.....	65
Figure 42 - MKA key hierarchy	67
Figure 43 - KDF pseudo code.....	68
Figure 44 - EAPoL architecture	70
Figure 45 - EAPoL frame format.....	71
Figure 46 - EAPoL - MKA packet body with MKPDU format	73
Figure 47 - CMAC Java sample output.....	80
Figure 48 - CMACalculator homepage	84
Figure 49 - Java SW flow.....	86
Figure 50 - MKA Key hierarchy Java performance.....	87
Figure 51 - Aes C usage	88
Figure 52 - CMAC C test file output.....	89
Figure 53 - AES rolled architecture.....	92
Figure 54 - Implemented AES core	93
Figure 55 - AES core finite state machine	94
Figure 56 - (a) CMAC block diagram, (b) CMAC instantiated modules in Verilog	94
Figure 57 - <i>cmac</i> finite state machine	95
Figure 58 - <i>key_wrap</i> block diagram.....	96
Figure 59 - <i>key_wrap</i> finite state machine	97
Figure 60 - <i>kdf</i> block diagram	98
Figure 61 - <i>kdf</i> finite state machine.....	99
Figure 62 - <i>mka</i> implementing all the modules: (a) block diagram, (b) Verilog code.....	100
Figure 63 - <i>mka</i> finite state machine.....	101
Figure 64 - modules performance in clock cycles.....	102

Figure 65 - <i>mka</i> wave plot	103
Figure 66 - ALM high-level block diagram for Stratix V devices	104
Figure 67 - FPGA synthesis results	105
Figure 68 – standard-cell ASIC technology synthesis	105

1. Security in automotive

1.1. Introduction

With the following dissertation I would like to focus on IT security, in particular on security related to the automotive field. Over the last two decades vehicles have silently but dramatically changed into mobile interactive systems already carrying dozens of digital microprocessors, various external radio interfaces, and several hundred megabytes of embedded software. In fact, information and communication technology is the driving force behind most innovations in the automotive industry, with perhaps 90% of all innovations in vehicles based on digital IT systems.

Today's in-vehicle IT architectures are dominated by a large network of interactive, software driven digital microprocessors called electronic control units (ECU). However, ECUs relying on information received from open communication channels created by other ECUs or even other vehicles that are not under its control, leaves the doors wide open for manipulations or misuse.

Future cars will become even more dependent on IT security due to the following developments:

- It is predicted that an increasing number of ECUs (electronic control units) will be reprogrammable, a process that must be protected.
- Many cars will communicate with the environment in a wireless fashion, which makes strong security a necessity.
- New business models (e.g., time-limited flash images or pay-per-use infotainment content) will become possible for the car industry, but will only be successful if abuse can be prevented.

- There will be an increasing number of legislative demands which can only be solved by means of modern IT security functions, such as tamper-resistant tachographs, secure emergency call functions, secure road billing etc.
- Increasing networking of cars will allow the collection of data for each driver (e.g., driving behavior, locations visited), which will put high demands on privacy technology.
- Future cars will often be personalized, which requires a secure identification of the driver.
- Electronic anti-theft measures will go beyond current immobilizers, e.g., by protecting individual components.

...and many others...

Automotive ECUs Controllers by 2020

- Between 25 and 100 individual ECUs
- With distributed sensors and motor controllers.

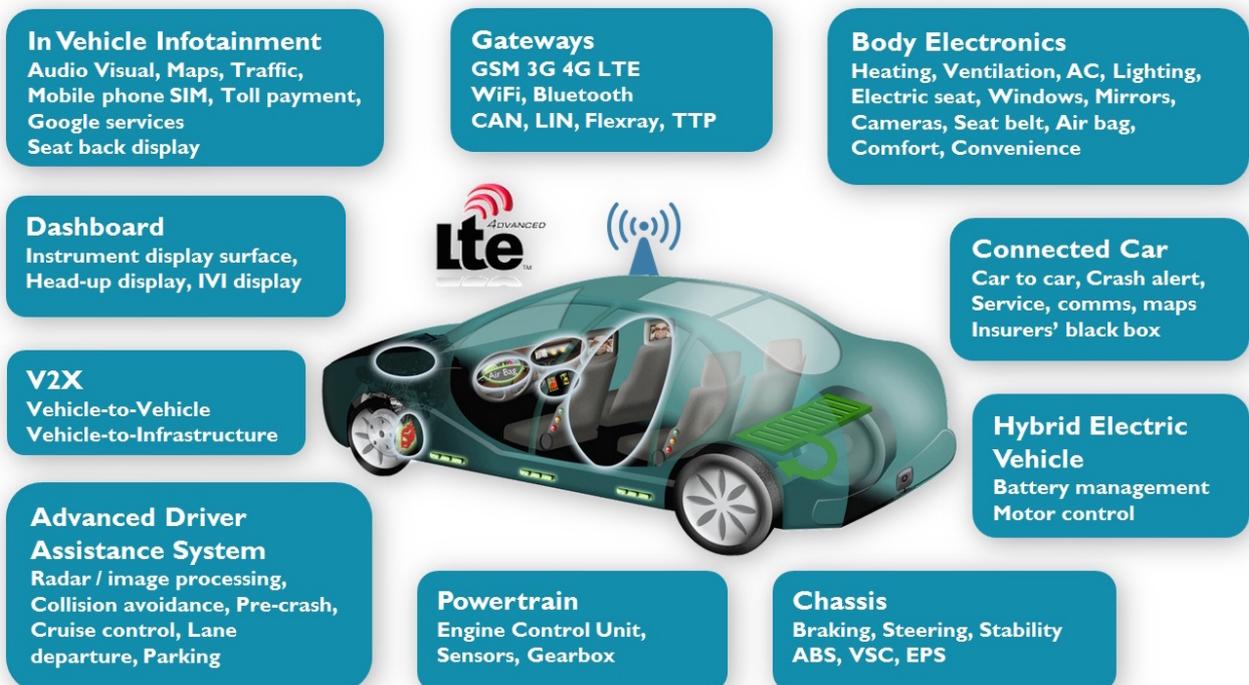


Figure 1 – Automotive ECUs Controllers by 2020

This “digital revolution” enables very sophisticated solutions considerably increasing flexibility, safety and efficiency of modern vehicles. It further helps saving fuel, weight, and costs.

Whereas in-vehicle IT safety (i.e., protection against [random] technical failures) is already a relatively well-established (if not necessarily well-understood) field, the protection of vehicular IT systems against systematic manipulations has only very recently started to emerge. In fact, automotive IT systems were never designed with security in mind. But with the increasing application of digital software and various radio interfaces to the outside world (including the Internet), modern vehicles are becoming even more vulnerable to all kinds of malicious encroachments like hackers or malware. This is especially noteworthy, since in contrast to most other IT systems, a successful malicious encroachment on a vehicle will not only endanger critical services or business models, but can also endanger human lives.

Thus strong security measures should be mandatory when developing vehicular IT systems. Today most vehicle manufacturer (hopefully) incorporates security as a design requirement. However, realizing dependable IT security solutions in a vehicular environment considerably differs from realizing IT security for typical desktop or server environments. In a typical vehicular attack scenario an attacker, for instance, has extended attack possibilities (i.e., insider attacks, offline attacks, physical attacks) and could have many different attack incentives and attack points (e.g., tachometer manipulations by the vehicle owner vs. theft of the vehicle components vs. industrial espionage).

1.2. How secure is your car?

In a talk at the Black Hat security conference in Las Vegas ^[1], Charlie Miller and Chris Valasek presented the results of a broad analysis of dozens of

different car makes and models, assessing the vehicles' schematics for the signs that hint at vulnerabilities to auto-focused hackers. The result is a kind of handbook of ratings and reviews of automobiles for the potential hackability of their networked components.

They examined how a remote attack might work on 24 different cars.

“It really depends on the architecture: If you hack the radio, can you send messages to the brakes or the steering? And if you can, what can you do with them?” said Valasek, director of vehicle security research at the security consultancy IOActive.

In the two researchers' analysis, three vehicles were ranked as “most hackable”: the 2014 models of the Infiniti Q50 and Jeep Cherokee and the 2015 model of the Cadillac Escalade. The full results, summarized in the chart below, show that the 2010 and 2014 Toyota Prius didn't fare well either.

Car	Attack Surface	Network Architecture	Cyber Physical
2014 Audi A8	++	--	+
2014 Honda Accord LX	-	+	+
2014 Infiniti Q50	++	+	+
2010 Infiniti G37	-	++	+
2014 Jeep Cherokee	++	++	++
2014 Dodge Ram 3500	++	++	--
2014 Chrysler 300	++	-	++
2014 Dodge Viper	++	-	--
2015 Cadillac Escalade	++	+	+
2006 Ford Fusion	--	--	--
2014 Ford Fusion	++	-	++
2014 BMW 3 series	++	--	+
2014 BMW X3	++	--	++
2014 BMW i12	++	--	+
2014 Range Rover Evoque	++	-	++
2010 Range Rover Sport	-	--	-
2006 Range Rover Sport	-	--	-
2014 Toyota Prius	+	+	++
2010 Toyota Prius	+	+	++
2006 Toyota Prius	-	--	--

Figure 2 - Hackability results: A plus sign represents “more hackable,” a minus sign “less hackable.”

All the cars’ ratings were based on three factors: the first was the size of their wireless “attack surface”—features like Bluetooth, Wi-Fi, cellular network connections, keyless entry systems, and even radio-readable tire pressure monitoring systems. Any of those radio connections could potentially be used by a hacker to find a security vulnerability and gain an initial foothold onto a car’s network. Second, they examined the vehicles’ network architecture, how much access those possible footholds offered to more critical systems steering and brakes. And third, Miller and Valasek assessed what they call the cars’ “cyberphysical” features: capabilities like automated braking, parking

and lane assist that could transform a few spoofed digital commands into an actual out-of-control car.

Miller and Valasek say that within the Infinity Q50's network, those radio and telematic components were directly connected to engine and braking systems. And the sedan's critical driving systems had computer-controlled features like adaptive cruise control and adaptive steering that a hacker could potentially hijack to physically manipulate the car.

The researchers pointed to Audi's A8, by contrast, as an example of a strong network layout. Its wireless features were separated from its driving functions on its internal network, with a gateway that would block commands sent to steering or brakes from any compromised radios.

1.3. Ethernet backbone in cars

In the late period we are moving in the direction where proprietary technologies in the automotive field, especially in transmission's physical layer, will be replaced with standard ones.

My thesis has been developed in cooperation with Renesas Electronics Corporation, one of the world's largest makers of semiconductor systems for mobile phones and automotive applications.

Starting from leading chip companies as Broadcom and Renesas, they think carmakers are coming around at last to the wisdom of leveraging standard technologies such as Ethernet, already well proven outside the car marketⁱⁱ.

Carmakers nowadays, and I would say people in general, are paying more attention to electronic devices and the innovation they are carrying with, instead of a car's horsepower. They need to make sure their cars can

accommodate everything, from a navigation system to displays and other gadgets that consumers use inside a car. And this is true not only for high-level cars.

Inside a car today there are many independent networks. Each automotive network technology such as low-voltage differential signaling (LVDS), media-oriented systems transport (MOST), and the controller area network (CAN), is connected to different electronics. They don't interoperate. We think Ethernet will replace these networks in some years.

Think about our smartphones, tablets and notebooks; our quality expectations are higher and higher. Would we be satisfied with a delay suffering black/white low-resolution rear camera in our car? And it's quite normal today to think about LTE networks inside cars. That's why the bandwidth needed for in-car networking grows exponentially. And scalability is another important feature carmakers are interested in; in fact they are increasingly looking to OPEN Alliance SIG, an open industry consortium designed to encourage wide-scale adoption of Ethernet-based networks as the standard in automotive networking applications, partner of Renesas as well.

	2012	2020
Engine control	Mix of high speed CAN network & Flexary	Mix of high speed CAN network & Ethernet/IP
Transmission		
Traction control		
Suspension control	CAN network & Flexary	CAN, Flexary and Ethernet/IP
Breaking control		
Active Safety		
Passive Safety		
Camera-based ADAS	LVDS >> Ethernet/IP (2013)	Ethernet/IP
Windows	CAN & LIN	CAN & LIN
HVAC & comfort		
Lighting		
Door and seats		
AV entertainment	CAN + MOST	CAN + MOST + Ethernet/IP
Device integration		
OBDX	CAN & Ethernet/IP	Ethernet/IP

Figure 3 - In-Car Networking Scenario

As we can see from Figure 3 the expectations in the next few years are about Ethernet to coexist with low-bandwidth standards like CAN.

The backbone however will be Ethernet-driven. The CAN, MOST, LIN and others will continue to exist on a small-scale basis, but Ethernet will drive the majority of the work. We need a security solution to protect Ethernet connections inside the car environment.

After a general overview of cryptography and encryption algorithms, in Sec. 2 I will deal with MACsec, a security protocol for Ethernet networks.

2. Automotive and cryptography

Even though security depends on much more than just cryptographic algorithms – a robust overall security design including secure protocols and organizational measures are needed as well – crypto schemes are in most cases the atomic building blocks of a security solution. The problem in embedded applications is that they tend to be computationally and memory constrained due to cost reasons. (Often they are also power limited, but, since automotive applications are often powered by their own battery, low-power crypto is not such an important topic in the car context).

So the main goal is to implement secure crypto algorithms on small devices at acceptable running times.

Crypto schemes are divided into two families: symmetric and asymmetric algorithms. The first group is mainly used for data encryption and message integrity checks. Symmetric algorithms tend to run relatively fast and often need little memory resources. There exists a wealth of established algorithms, with the most prominent representatives being the block ciphers DES (Data Encryption Standard) and AES (Advanced Encryption Standard). The family of stream ciphers, as we will see later, can be even more efficient than block ciphers and are, thus, sometimes preferred for embedded applications. In almost all cases it is a wise choice to use established, proven algorithms rather than unproven or self-developed ones.

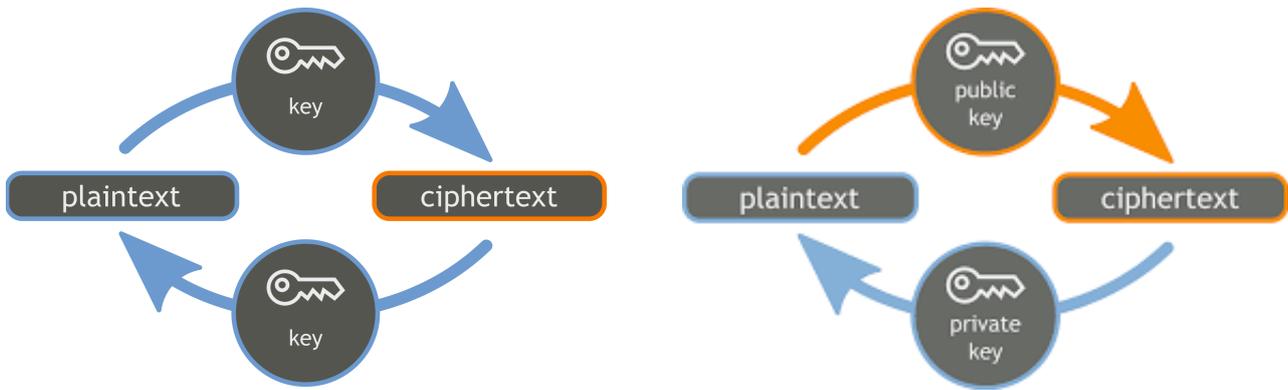


Figure 4 - Symmetric vs Asymmetric Cryptography

The second family of schemes, asymmetric or public-key algorithms, is very different. They are based on hard number theoretical problems and involve complex mathematical computations with very long numbers, commonly in the range of 160–4048 bits, depending on the algorithm and security level. Their advantage, however, is that they offer advanced functions such as digital signatures and key distribution over unsecure channels. For common automotive applications such as secure flashing, public-key algorithms are often preferred. The problem here is the computational requirement of public-key schemes. Embedded processors in the automotive domain are often only equipped with 8-bit and 16-bit processors clocked at moderate frequencies of, say, below 10 MHz. Running computationally expensive public-key algorithms on such processors can result in unacceptably long execution times, for instance several seconds for the generation of a digital signature. For this reason, it is very important that a smart parameter choice together with the latest implementation techniques are being employed.

2.1. Symmetric Cryptography

Symmetric-key algorithms are algorithms for cryptography that use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext. The keys may be identical or there may be a simple transformation to go between the two keys.

$$\forall p \in P, k \in K : D(k, E(k, p)) = p$$

Figure 5 - Symmetric Key formal definition

In Figure 5 we can see the characterized equation of the symmetric cryptography, where p is the plaintext belonging to the plaintext space P , c is the ciphertext belonging to the ciphertext space C and k is the shared secret key belonging to the key space K .

The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link. This requirement that both parties have access to the secret key is one of the main drawbacks of symmetric key encryption, in comparison to public-key encryption.

2.2. Operation modes

Symmetric cryptography can be implemented using either Stream ciphers or Block ciphers.

2.2.1. Stream ciphers

With stream ciphers plaintext digits are combined with a pseudorandom cipher digit stream (keystream). In a stream cipher each plaintext digit is encrypted one at a time with the corresponding digit of the keystream, to give a digit of the ciphertext stream.

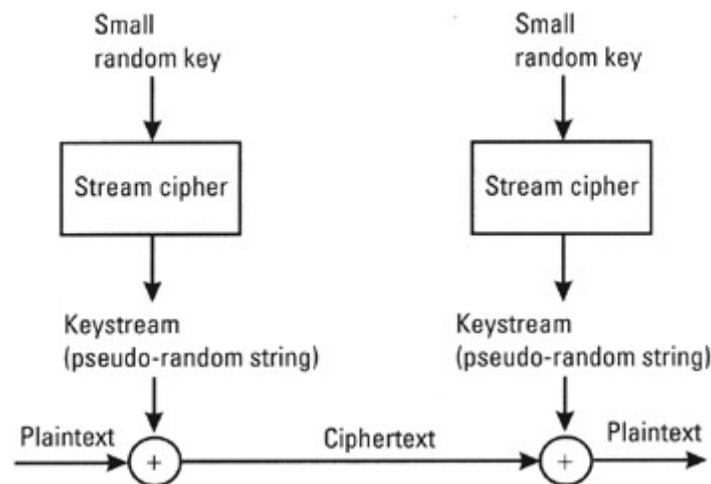


Figure 5 - Stream cipher general schema

2.2.2. Block ciphers

Block ciphers can operate into different modes:

- ECB (Electronic CodeBook)
- CBC (Cipher Block Chaining)
- CFB (Cipher FeedBack)
- OFB (Output FeedBack)
- CTR (Counter)

In the following sections all these modes are described in detail ^[iii].

2.2.2.1. ECB – Electronic CodeBook

The Electronic Codebook (ECB) mode is a confidentiality mode that features, for a given key, the assignment of a fixed ciphertext block to each plaintext block, analogous to the assignment of code words in a codebook. The Electronic Codebook (ECB) mode is defined as follows:

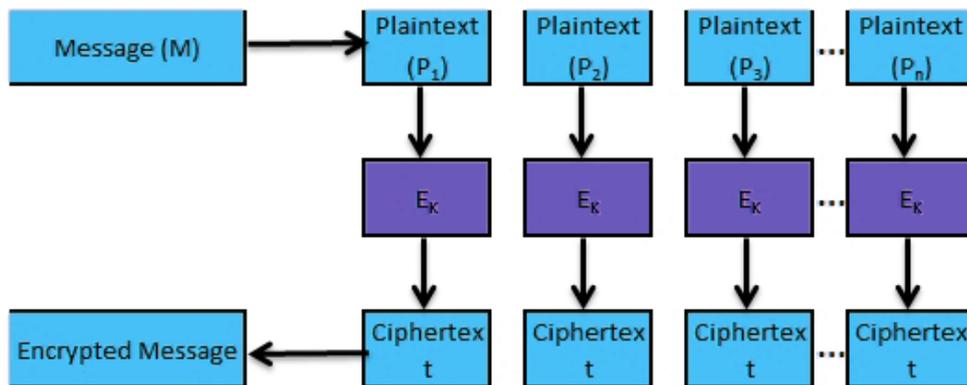


Figure 6 - ECB mode schema

$$\begin{cases} C_i = E_k(P_i) \\ P_i = D_k(C_i) \end{cases}$$

Figure 7 - ECB equations

In ECB encryption and ECB decryption, multiple forward cipher functions and inverse cipher functions can be computed in parallel.

In ECB redundancies can be present, since same plaintext blocks will have the same ciphertext. This will bring the algorithm subjected to attacks of cryptanalysis.

With ECB we don't have error propagation, i.e. if one block is received corrupted no other block will suffer for the error.

2.2.2.2. CBC – Cipher Block Chaining

The Cipher Block Chaining (CBC) mode is a confidentiality mode whose encryption process features the combining (“chaining”) of the plaintext blocks with the previous ciphertext blocks. The CBC mode requires an IV to combine with the first plaintext block (Figure 9). The IV need not be secret, but it must be unpredictable. Also, the integrity of the IV should be protected. The CBC mode is defined as follows:

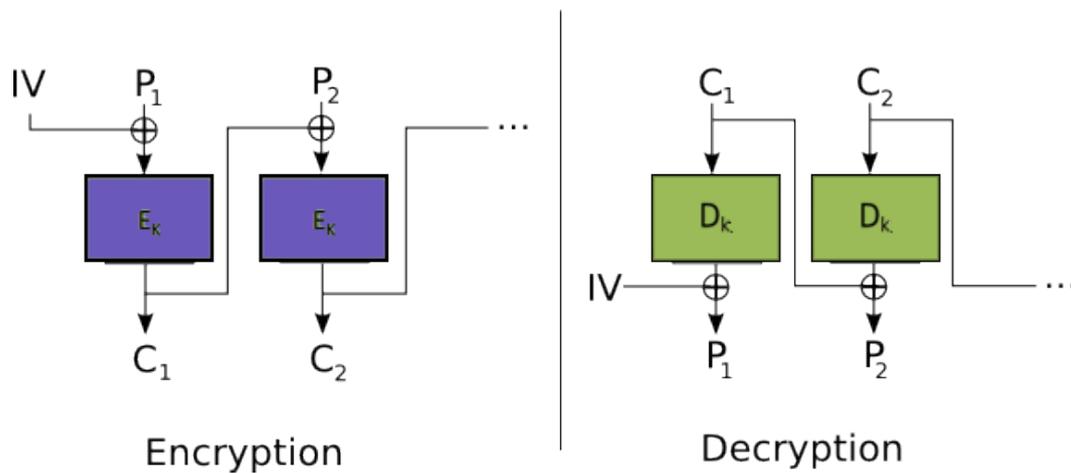


Figure 8 - CBC mode schema

$$\begin{cases} C_i = E_k(P_i \oplus C_{i-1}) \\ P_i = C_{i-1} \oplus D_k(C_i) \\ C_0 = IV \end{cases}$$

Figure 9 - CBC equations

In CBC encryption, the first input block is formed by exclusive-ORing the first block of the plaintext with the IV. The forward cipher function is applied to the first input block, and the resulting output block is the first block of the ciphertext. This output block is also exclusive-ORed with the second plaintext data block to produce the second input block, and the forward cipher function is applied to produce the second output block. This output block, which is the second ciphertext block, is exclusive-ORed with the next plaintext block to form the next input block. Each successive plaintext block is exclusive-ORed with the previous output/ciphertext block to produce the new input block. The forward cipher function is applied to each input block to produce the ciphertext block.

In CBC decryption, the inverse cipher function is applied to the first ciphertext block, and the resulting output block is exclusive-ORed with the initialization vector to recover the first plaintext block. The inverse cipher function is also applied to the second ciphertext block, and the resulting output block is exclusive-ORed with the first ciphertext block to recover the second plaintext block. In general, to recover any plaintext block (except the first), the inverse cipher function is applied to the corresponding ciphertext block, and the resulting block is exclusive-ORed with the previous ciphertext block.

In CBC encryption, the input block to each forward cipher operation (except the first) depends on the result of the previous forward cipher operation, so the forward cipher operations cannot be performed in parallel. In CBC decryption, however, the input blocks for the inverse cipher function, i.e., the ciphertext blocks, are immediately available, so that multiple inverse cipher operations can be performed in parallel.

2.2.2.3. CFB – Cipher FeedBack

The Cipher Feedback (CFB) mode is a confidentiality mode that features the feedback of successive ciphertext segments into the input blocks of the forward cipher to generate output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa. The CFB mode requires an IV as the initial input block. The IV need not be secret, but it must be unpredictable.

The CFB mode also requires an integer parameter, denoted s , such that $1 \leq s \leq b$. In the specification of the CFB mode below, each plaintext segment ($P_j^\#$) and ciphertext segment ($C_j^\#$) consists of s bits. The value of s is sometimes incorporated into the name of the mode, e.g., the 1-bit CFB mode, the 8-bit CFB mode, the 64-bit CFB mode, or the 128-bit CFB mode.

The CFB mode is defined as follows:

$$\left\{ \begin{array}{l} I_1 = IV \\ I_j = LSB_{b-s}(I_{j-1}) \mid C_{j-1}^\# \\ O_j = E_K(I_j) \\ C_j^\# = P_j^\# \oplus MSB_s(O_j) \end{array} \right. \begin{array}{l} \text{For } j=2 \dots n \\ \text{For } j=1, 2 \dots n \\ \text{For } j=1, 2 \dots n \end{array}$$

Figure 10 - CFB Encryption

$$\left\{ \begin{array}{l} I_1 = IV \\ I_j = LSB_{b-s}(I_{j-1}) \mid C_{j-1}^\# \\ O_j = E_K(I_j) \\ P_j^\# = C_j^\# \oplus MSB_s(O_j) \end{array} \right. \begin{array}{l} \text{For } j=2 \dots n \\ \text{For } j=1, 2 \dots n \\ \text{For } j=1, 2 \dots n \end{array}$$

Figure 11 - CFB Decryption

In CFB encryption, the first input block is the IV, and the forward cipher operation is applied to the IV to produce the first output block. The first ciphertext segment is produced by exclusive-ORing the first plaintext segment with the s most significant bits of the first output block. (The remaining $b-s$ bits of the first output block are discarded.) The $b-s$ least significant bits of the IV are then concatenated with the s bits of the first ciphertext segment to form the second input block. An alternative description of the formation of the second input block is that the bits of the first input block circularly shift s positions to the left, and then the ciphertext segment replaces the s least significant bits of the result.

The process is repeated with the successive input blocks until a ciphertext segment is produced from every plaintext segment. In general, each successive input block is enciphered to produce an output block. The s most significant bits of each output block are exclusive-ORed with the corresponding plaintext segment to form a ciphertext segment. Each ciphertext segment (except the last one) is “fed back” into the previous input block, as described above, to form a new input block. The feedback can be described in terms of the individual bits in the strings as follows: if $i_1i_2\dots i_b$ is the j^{th} input block, and $c_1c_2\dots c_s$ is the j^{th} ciphertext segment, then the $(j+1)^{\text{th}}$ input block is $i_{s+1}i_{s+2}\dots i_b c_1c_2\dots c_s$.

The CFB mode is illustrated in Figure 12.

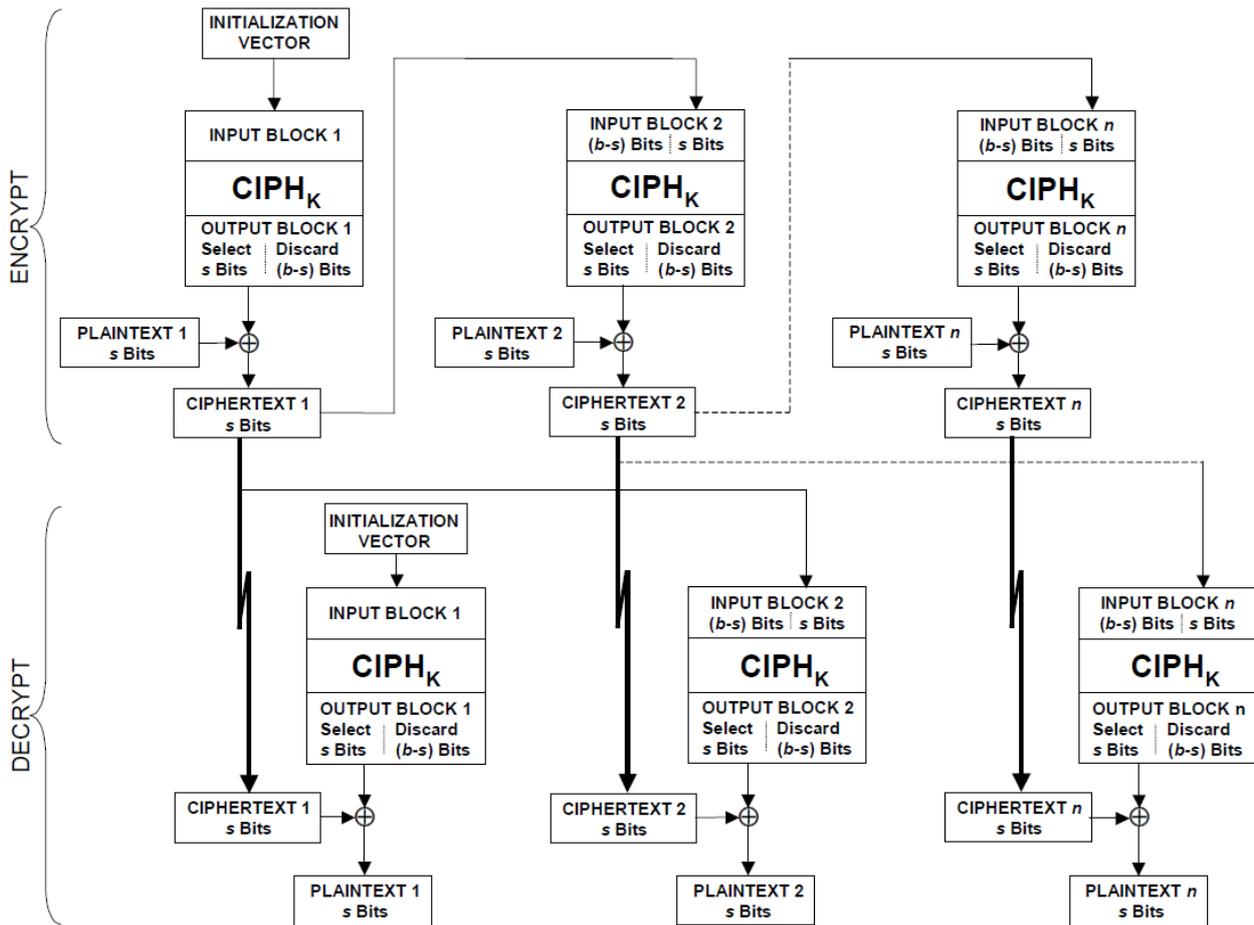


Figure 12 - CFB Mode

In CFB decryption, the IV is the first input block, and each successive input block is formed as in CFB encryption, by concatenating the $b-s$ least significant bits of the previous input block with the s most significant bits of the previous ciphertext. The forward cipher function is applied to each input block to produce the output blocks. The s most significant bits of the output blocks are exclusive-ORed with the corresponding ciphertext segments to recover the plaintext segments.

In CFB encryption, like CBC encryption, the input block to each forward cipher function (except the first) depends on the result of the previous forward cipher function; therefore, multiple forward cipher operations cannot be performed in parallel. In CFB decryption, the required forward cipher

operations can be performed in parallel if the input blocks are first constructed (in series) from the IV and the ciphertext.

2.2.2.4. OFB – Output FeedBack

The Output Feedback (OFB) mode is a confidentiality mode that features the iteration of the forward cipher on an IV to generate a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa. The OFB mode requires that the IV is a nonce, i.e., the IV must be unique for each execution of the mode under the given key; the generation of such IVs is discussed in Appendix C. The OFB mode is defined as follows:

$$\left\{ \begin{array}{l} I_1 = IV \\ I_j = O_{j-1} \\ O_j = E_K(I_j) \\ C_j = P_j \oplus O_j \\ C_n^* = P_n^* \oplus MSB_u(O_n) \end{array} \right. \begin{array}{l} \text{For } j=2 \dots n \\ \text{For } j=1, 2 \dots n \\ \text{For } j=1, 2 \dots n-1 \end{array}$$

Figure 13 - OFB Encryption

$$\left\{ \begin{array}{l} I_1 = IV \\ I_j = O_{j-1} \\ O_j = E_K(I_j) \\ P_j = C_j \oplus O_j \\ P_n^* = C_n^* \oplus MSB_u(O_n) \end{array} \right. \begin{array}{l} \text{For } j=2 \dots n \\ \text{For } j=1, 2 \dots n \\ \text{For } j=1, 2 \dots n-1 \end{array}$$

Figure 14 - OFB Decryption

In OFB encryption, the IV is transformed by the forward cipher function to produce the first output block. The first output block is exclusive-ORed with the first plaintext block to produce the first ciphertext block. The forward cipher function is then invoked on the first output block to produce the second output block. The second output block is exclusive-ORed with the second plaintext block to produce the second ciphertext block, and the forward cipher function is invoked on the second output block to produce the third output block. Thus, the successive output blocks are produced from applying the forward cipher function to the previous output blocks, and the output blocks are exclusive-ORed with the corresponding plaintext blocks to produce the ciphertext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the last output block are used for the exclusive-OR operation; the remaining $b-u$ bits of the last output block are discarded.

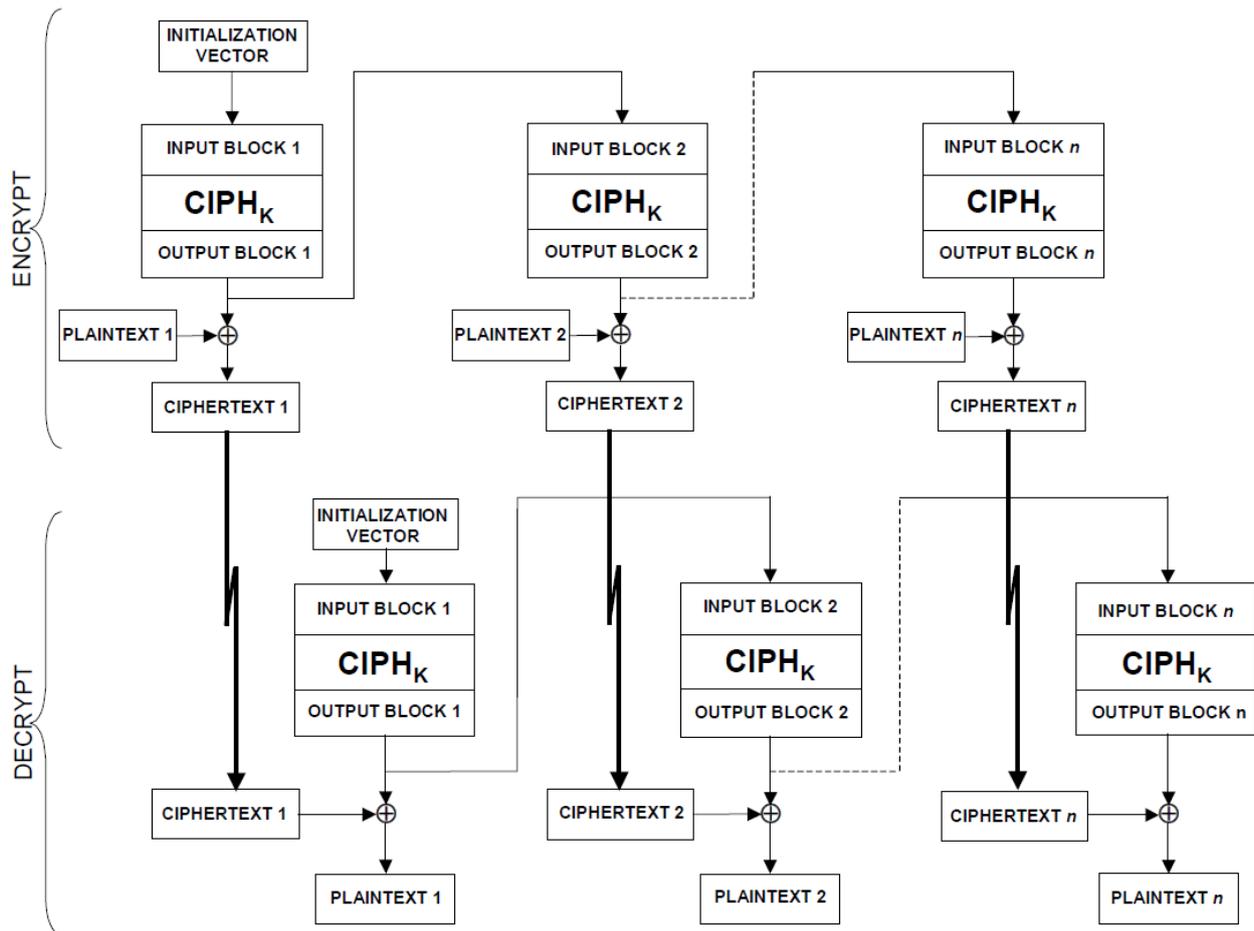


Figure 15 - OFB Mode

In OFB decryption, the IV is transformed by the forward cipher function to produce the first output block. The first output block is exclusive-ORed with the first ciphertext block to recover the first plaintext block. The first output block is then transformed by the forward cipher function to produce the second output block. The second output block is exclusive-ORed with the second ciphertext block to produce the second plaintext block, and the second output block is also transformed by the forward cipher function to produce the third output block. Thus, the successive output blocks are produced from applying the forward cipher function to the previous output blocks, and the output blocks are exclusive-ORed with the corresponding ciphertext blocks to recover the plaintext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the last output

block are used for the exclusive-OR operation; the remaining $b-u$ bits of the last output block are discarded.

In both OFB encryption and OFB decryption, each forward cipher function (except the first) depends on the results of the previous forward cipher function; therefore, multiple forward cipher functions cannot be performed in parallel. However, if the IV is known, the output blocks can be generated prior to the availability of the plaintext or ciphertext data.

The OFB mode requires a unique IV for every message that is ever encrypted under the given key. If, contrary to this requirement, the same IV is used for the encryption of more than one message, then the confidentiality of those messages may be compromised. In particular, if a plaintext block of any of these messages is known, say, the j^{th} plaintext block, then the j^{th} output of the forward cipher function can be determined easily from the j^{th} ciphertext block of the message. This information allows the j^{th} plaintext block of any other message that is encrypted using the same IV to be easily recovered from the j^{th} ciphertext block of that message.

Confidentiality may similarly be compromised if any of the input blocks to the forward cipher function for the encryption of a message is designated as the IV for the encryption of another message under the given key.

The OFB mode is illustrated in Figure 15.

2.2.2.5. CTR – Counter

The Counter (CTR) mode is a confidentiality mode that features the application of the forward cipher to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa. The sequence of counters must have the property that each block in the sequence is different from every other block. This condition is not restricted to a single message: across

all of the messages that are encrypted under the given key, all of the counters must be distinct. In this recommendation, the counters for a given message are denoted T_1, T_2, \dots, T_n . Given a sequence of counters, T_1, T_2, \dots, T_n , the CTR mode is defined as follows:

$$\begin{cases} O_j = E_K(T_j) & \text{For } j=1,2,\dots,n \\ C_j = P_j \oplus O_j & \text{For } j=1,2,\dots,n-1 \\ C_n^* = P_n^* \oplus MSB_u(O_n) \end{cases}$$

Figure 16 - CTR Encryption

$$\begin{cases} O_j = E_K(T_j) & \text{For } j=1,2,\dots,n \\ P_j = C_j \oplus O_j & \text{For } j=1,2,\dots,n-1 \\ P_n^* = C_n^* \oplus MSB_u(O_n) \end{cases}$$

Figure 17 - CTR Decryption

In CTR encryption, the forward cipher function is invoked on each counter block, and the resulting output blocks are exclusive-ORed with the corresponding plaintext blocks to produce the ciphertext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the last output block are used for the exclusive-OR operation; the remaining $b-u$ bits of the last output block are discarded.

In CTR decryption, the forward cipher function is invoked on each counter block, and the resulting output blocks are exclusive-ORed with the corresponding ciphertext blocks to recover the plaintext blocks. For the last block, which may be a partial block of u bits, the most significant u bits of the

last output block are used for the exclusive-OR operation; the remaining $b-u$ bits of the last output block are discarded.

In both CTR encryption and CTR decryption, the forward cipher functions can be performed in parallel; similarly, the plaintext block that corresponds to any particular ciphertext block can be recovered independently from the other plaintext blocks if the corresponding counter block can be determined. Moreover, the forward cipher functions can be applied to the counters prior to the availability of the plaintext or ciphertext data.

CTR mode is illustrated in Figure 18.

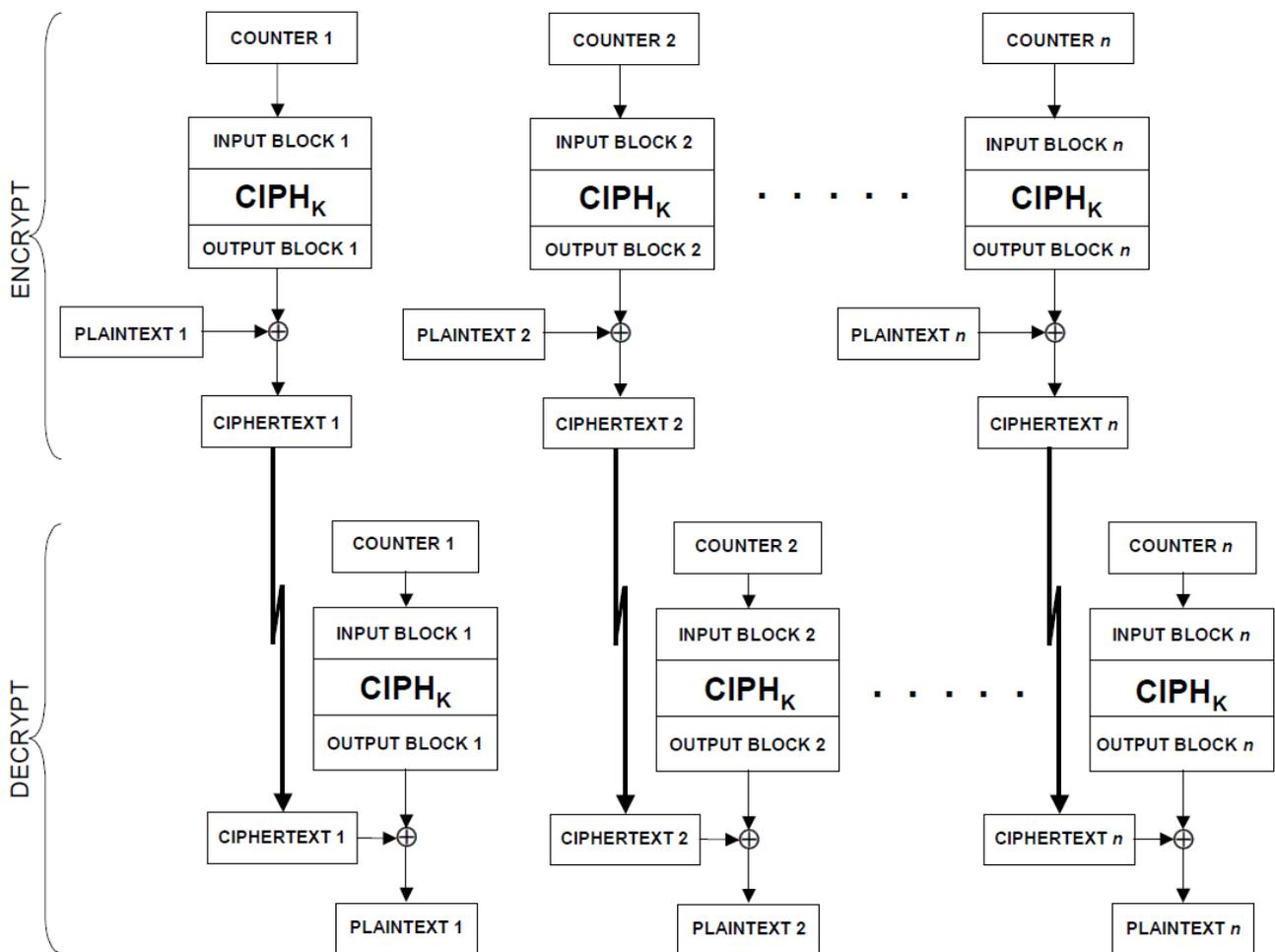


Figure 18 - CTR Mode

2.2.3. Stream vs Block Ciphers

Stream ciphers are typically faster than block, but that has its own price.

Block ciphers typically require more memory, since they work on larger chunks of data and often have "carry over" from previous blocks, whereas since stream ciphers work on only a few bits at a time they have relatively low memory requirements (and therefore cheaper to implement in limited scenarios such as embedded devices, firmware, and esp. hardware).

Stream ciphers are more difficult to implement correctly, and prone to weaknesses based on usage and the keystream has very strict requirements.

Because block ciphers encrypt a whole block at a time (and furthermore have "feedback" modes which are most recommended), they are more susceptible to noise in transmission, that is if you mess up one part of the data, all the rest is probably unrecoverable. Whereas with stream ciphers bytes are individually encrypted with no connection to other chunks of data (in most ciphers/modes), and often have support for interruptions on the line.

Also, stream ciphers do not provide integrity protection or authentication, whereas some block ciphers (depending on mode) can provide integrity protection, in addition to confidentiality.

Because of all the above, stream ciphers are usually best for cases where the amount of data is either unknown, or continuous - such as network streams. Block ciphers, on the other hand, are more useful when the amount of data is pre-known - such as a file, data fields, or request/response protocols, such as HTTP where the length of the total message is known already at the beginning.

This is the main reason why choosing the encryption algorithm we came up with a symmetric block cipher one.

2.3. Possible attacks

The idea of security comes from the need to protect data against malicious users and relative attacks. Depending on the chosen algorithm, some of these attacks can be successful or not.

In general the following types of attacks are valid under these hypothesis:

- The adversary has access to all encrypted messages.
- Kerckhoff Hypothesis: the adversary knows all the details of the encryption function but the secret key.

Here we have a list of possible attacks; they refer to what the malicious user is in posses during the attack.

Types of attacks:

- *Ciphertext-only* attack: he (the adversary) has access to the ciphertext only.
- *Known-plaintext* attack: he has the ciphertext and the message in clear, and he's able to combine them in pairs.
- *Chosen-plaintext* attack: he can obtain the ciphertexts for arbitrary plaintexts.

The previous types of attacks characterize all the so-called *force brute attacks*.

2.3.1. Exhaustive key search

Exhaustive key search is an example of known-plaintext attack, that can become a ciphertext-only attack if we have redundancies in the plain text.

The adversary has (p, c) pairs and he has to find the key that generates c from p .

Since the key is on k bit we have 2^k possible keys. The exhaustive key search tells the user to try all the possible keys to check which one can encrypt the message p on the ciphertext c . The main issue in this approach is the presence of *false positive* keys: different keys can encrypt the same message p into the same ciphertext c , so the found key can be the correct one for the given message but not for all the others.

The number of pairs (p, c) that we need to avoid false positives is:

$$J = \binom{k+4}{n}$$

Figure 19 - number of pairs to avoid false positive

where k is the key's number of bits and n is the number of bits of the message.

2.3.2. Data exhaustive analysis

Data exhaustive analysis, also called dictionary attack, is a known-plaintext attack. The adversary builds up a table with enough pair (p_i, c_i) to reuse them later to decrypt similar encrypted messages. Longer the ciphertext are harder to acquire all the needed pairs will be.

2.3.3. Cryptanalysis

Besides force brute attacks we have also cryptanalysis algorithms, which can be divided in:

- Linear Cryptanalysis (LC): used for block and stream ciphers.
- Differential Cryptanalysis (DC): used for block and stream ciphers and hash functions.

2.4. Computational security

The encryption algorithm is said to be computationally secure if the best attack is too complex for the adversary.

The attack complexity can be divided in:

- Data Complexity
- Storage Complexity
- Processing Complexity

A security schema is computationally secure to the previous described force brute attacks if:

- The key k is big enough (>64 bit), to avoid exhaustive key search.
- The messages' length is big enough (>64 bit), to avoid data exhaustive analysis.

2.5. Attacks in automotive systems

To prevent cyber attacks on vehicles, security solutions must be designed for automotive systems. There exist, however, a number of fundamental

limitations when designing such solutions. First, the ECUs inside the vehicles have limitations in computational power, memory, bandwidth, and power consumption. Second, the ECUs operate in a real-time environment where queuing of messages and delays are not tolerated. The data received from sensors on a vehicle must be processed in real-time, and decisions to affect the correct actuators must be made with no imposed delay. The design of security solutions must take the real-time constraint into consideration. Third, the traffic patterns for vehicular communication differ from traffic patterns in traditional IP networks. For example, data on the CAN bus in the in-vehicle network is broadcast. Vehicular ad hoc networks could be formed spontaneously in vehicle-to-vehicle and vehicle-to-roadside communication. In addition, automotive manufacturers could establish vehicle-to-infrastructure environments for performing wireless diagnostics and firmware updates on vehicles. The different traffic patterns and communication models require different solutions. Thus, traditional solutions developed for IP networks cannot be used.

The three most important research challenges for providing security solutions for automotive systems are described as follows. The vehicle allows interaction with the physical world, such as receiving warning signals from other vehicles or intersections and crossings. As a consequence, cyber attacks that simulate the physical world will most likely occur. Thus, a challenge is to verify the authenticity of incoming data to a vehicle. For example, a vehicle must assure that the received warning is correct and fresh (no replay) and that it was sent from the correct physical entity (e.g., vehicle or intersection).

While authenticating that incoming data is correct is one challenge, protecting the listening interface from intrusions is another. Since the wireless interface is a listening service it could possibly be subverted and allow an attacker access to the in-vehicle network. Thus, providing proper mechanisms for

preventing intrusions is an important challenge. Firewalls to prevent unauthorized accesses are necessary, and logging and detection mechanisms are needed to detect and trace attackers. However, designing these security solutions to meet the real-time requirements and the limitations in the ECUs is a challenge.

A third research challenge is to protect the security solutions in the in-vehicle network. This project defines security in this scenario. Assume various cryptographic keys are used to secure the wireless communication and access control lists are used to allow only authorized connections such that the wireless gateway is protected against intrusions. An attacker could potentially access the in-vehicle network via the OBD (on-board diagnostics) port by physically connecting a device to the vehicle. If the security solutions protect against attacks only via the wireless gateway, an attacker could choose to attack the in-vehicle network via the OBD instead. For example, the attacker could easily extract the needed cryptographic keys and update the access control lists such that he can execute future attacks via the wireless gateway. Thus, it is a challenge to protect the in-vehicle network and the security credentials against physical attacks via the OBD.

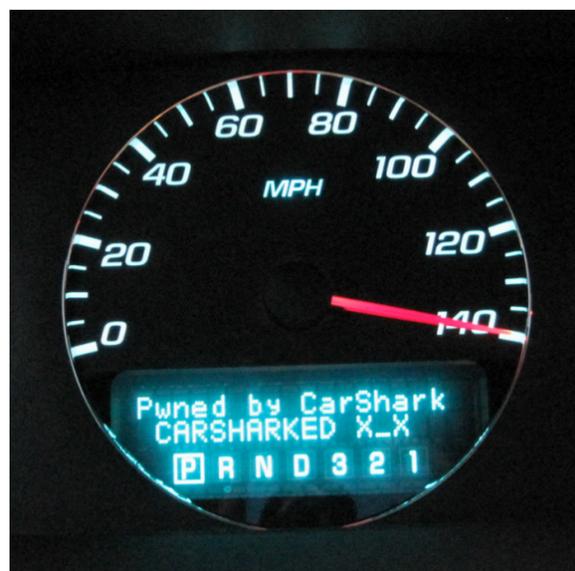


Figure 20 - An example of car attack

The usual motivation within the criminal world will be financial gain; therefore a cyber attack against automotive systems could potentially provide criminals with a repeatable, remotely exploitable mechanism for breaking into vehicles for theft of vehicle contents and/or the vehicle itself. On a more sinister level, should criminals be keen on impacting the safety of a victim's vehicle in some way then this might be achievable through cyber attack. Other criminal activity might just relate to hackers, where no financial gain is sought but merely the ability to demonstrate technical prowess through remotely attacking and controlling automotive systems.

3. AES cipher

AES will be the main encryption algorithm on which the following discussed protocols will rely on. It belongs to the symmetric-key algorithm family.

The Advanced Encryption Standard (AES), also referenced as Rijndael (its original name), is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES), which was published in 1977 ^[iv].

AES is based on a design principle known as a *substitution-permutation network*, combination of both substitution and permutation, and is fast in both software and hardware.

AES is a variant of Rijndael, which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael could work with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, called *the state*, although some versions of Rijndael have a larger block size and have additional columns in the state. Most AES calculations are done in a special finite field.

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of cycles of repetition is as follows:

- 10 cycles of repetition for 128-bit keys.
- 12 cycles of repetition for 192-bit keys.
- 14 cycles of repetition for 256-bit keys.

Each round consists of several processing steps, each containing four similar but different stages, including one that depends on the encryption key itself. A set of reverse rounds is applied to transform ciphertext back into the original plaintext using the same encryption key.

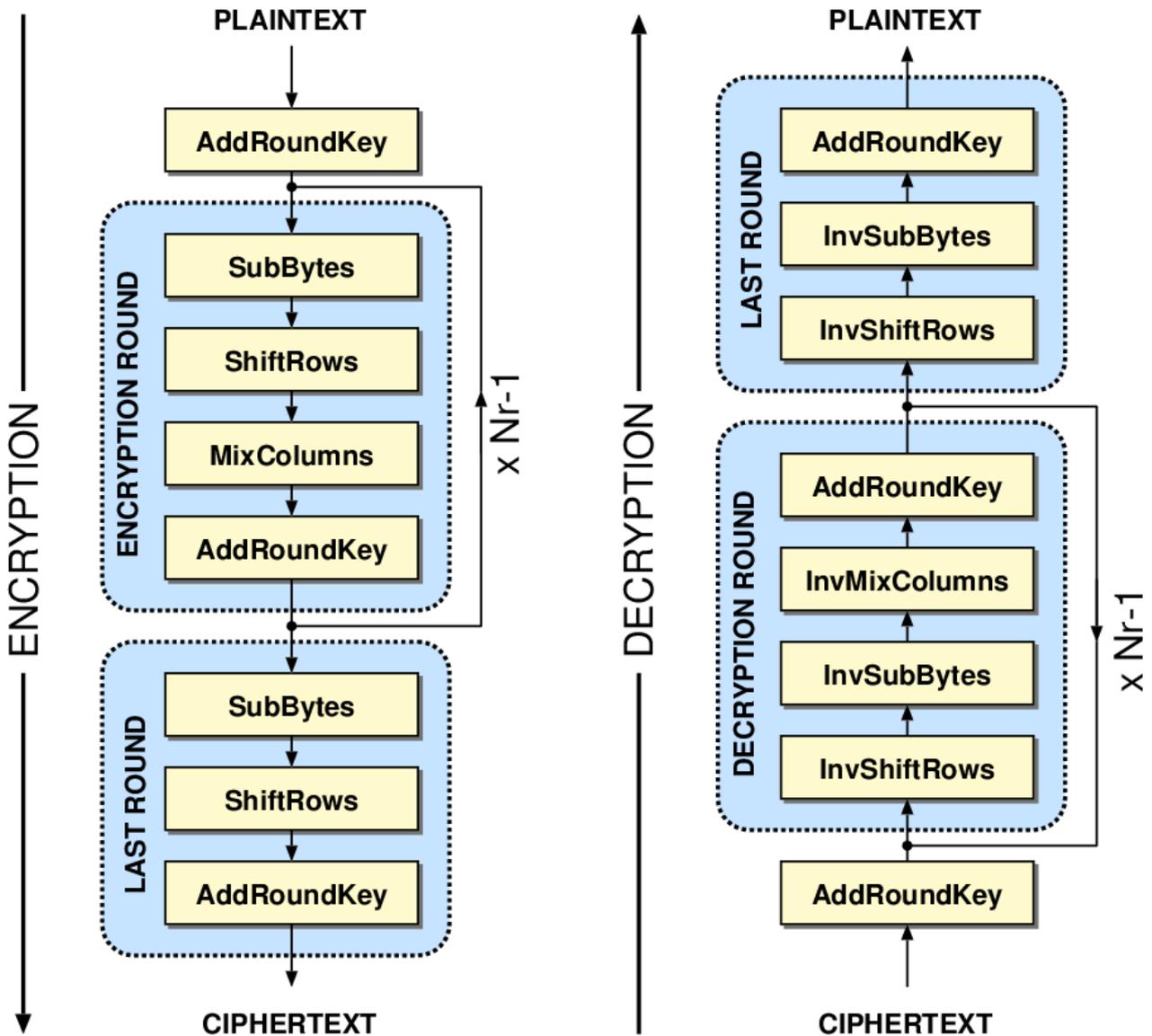


Figure 21 - AES scheme

3.1. AES encryption algorithm

At the start of the Cipher, the input is copied to the State array as described in Figure 23 ^[M].

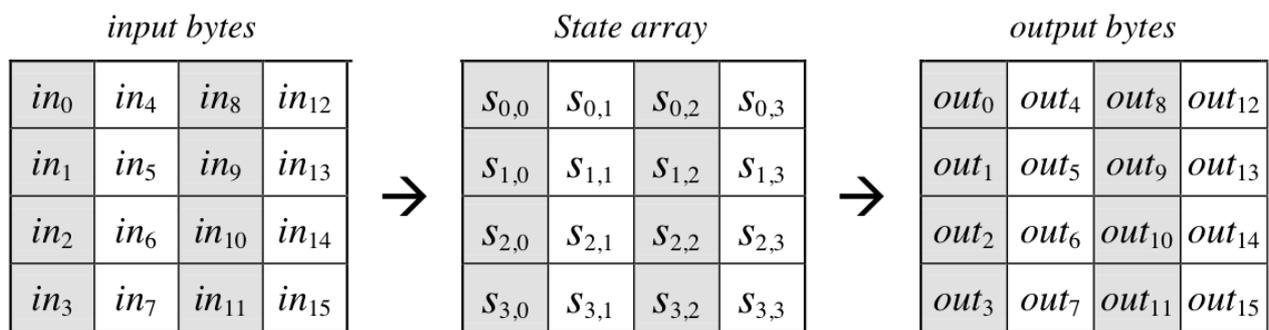


Figure 22 - State array input and output

After an initial *Round Key addition*, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first $N_r - 1$ rounds. The final State is then copied to the output as described in Figure 22.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the *Key Expansion* routine described in Sec. 3.2.

The Cipher is described in the pseudo code in Fig. 23.

```

Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin

    byte state[4,Nb]
    state = in

    AddRoundKey(state, w[0, Nb-1])

for    round = 1 step 1 to Nr-1
SubBytes(state)
ShiftRows(state)
MixColumns(state)
AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])

end for

SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state

end

```

Figure 23 - AES cipher suite

The individual transformations - *SubBytes()*, *ShiftRows()*, *MixColumns()*, and *AddRoundKey()* – process the State and are described in the following subsections. In Figure 23, the array *w[]* contains the key schedule, which is described in Sec. 3.2.

As shown in Figure 21, all *Nr* rounds are identical with the exception of the final round, which does not include the *MixColumns()* transformation.

3.1.1. SubBytes() Transformation

The *SubBytes()* transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (*S-box*).

This S-box (Figure 25), which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field $GF(2^8)$; the element $\{00\}$ is mapped to itself.
2. Apply the following affine transformation (over $GF(2)$):

$$b'_i = b_i \oplus b_{(i+4)\text{mod}8} \oplus b_{(i+5)\text{mod}8} \oplus b_{(i+6)\text{mod}8} \oplus b_{(i+7)\text{mod}8} \oplus c_i$$

for $0 \leq i < 8$, where b_i is the i^{th} bit of the byte, and c_i is the i^{th} bit of a byte c with the value $\{63\}$ or $\{01100011\}$. Here and elsewhere, a prime on a variable (e.g. b') indicates that the variable is to be updated with the value on the right.

In matrix form, the affine transformation element of the S-box can be expressed as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 24 illustrates the effect of the *SubBytes()* on the State.

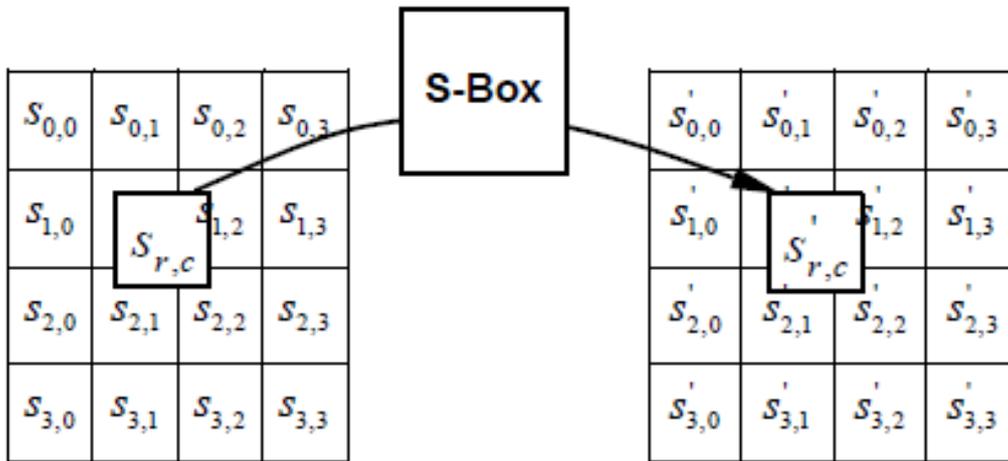


Figure 24 - SubBytes() applies the S-box to each byte of the State

The S-box used in the *SubBytes()* transformation is presented in hexadecimal form in Figure 25.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 25 - S-box: substitution values for the byte xy (Hex format)

3.1.2. ShiftRows() Transformation

In the *ShiftRows()* transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r=0$, is not shifted. Specifically, the *ShiftRows()* transformation proceeds as follows:

$$S'_{r,c} = S_{r,(c+shift(r,Nb))\bmod Nb} \quad \text{For } 0 < r < 4 \text{ and } 0 \leq c < Nb,$$

Where the shift value $shift(r,Nb)$ depends on the row number, r , as follows (recall that $Nb = 4$):

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3$$

This has the effect of moving bytes to “lower” positions in the row (i.e., lower values of c in a given row), while the “lowest” bytes wrap around into the “top” of the row (i.e., higher values of c in a given row). Figure 26 illustrates the *ShiftRows()* transformation.

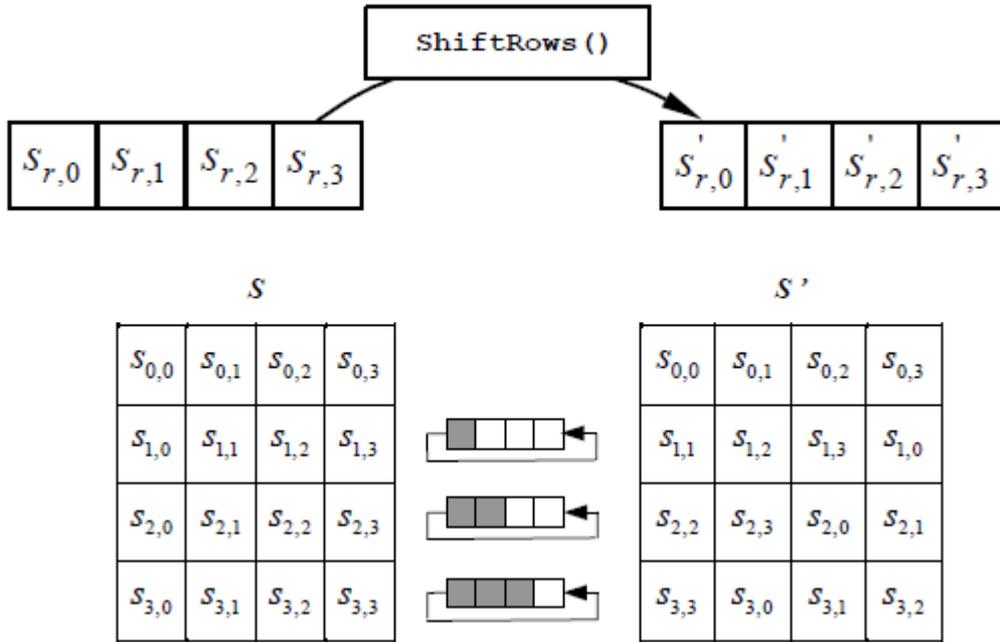


Figure 26 - ShiftRows() cyclically shifts the last three rows in the State

3.1.3. MixColumns() Transformation

The *MixColumns()* transformation operates on the State column-by-column, treating each column as a four-term polynomial. The columns are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

This can be written as a matrix multiplication. Let

$$s'(x) = a(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < Nb.$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}). \end{aligned}$$

Figure 27 illustrates the *MixColumns()* transformation.

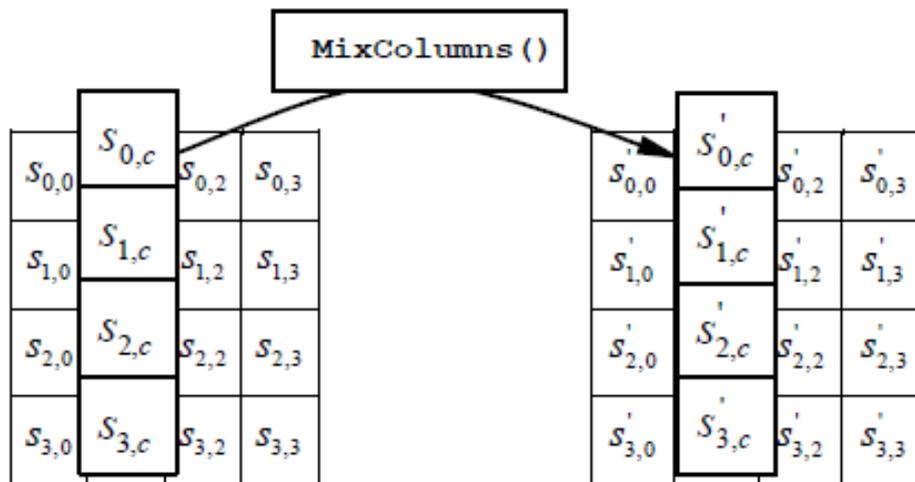


Figure 27 - *MixColumns()* operates on the State column-by-column

3.1.4. AddRoundKey() Transformation

In the *AddRoundKey()* transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of *Nb* words from the key schedule (Sec. 3.2). Those *Nb* words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{\text{round} * \text{Nb} + c}]$$

where $[w_i]$ are the key schedule words, and *round* is a value in the range $0 \leq \text{round} \leq N_r$. In the Cipher, the initial Round Key addition occurs when $\text{round} = 0$, prior to the first application of the round function (see Figure 21). The application of the *AddRoundKey()* transformation to the *Nr* rounds of the Cipher occurs when $1 \leq \text{round} \leq N_r$.

The action of this transformation is illustrated in Figure 28, where $l = \text{round} * \text{Nb}$.

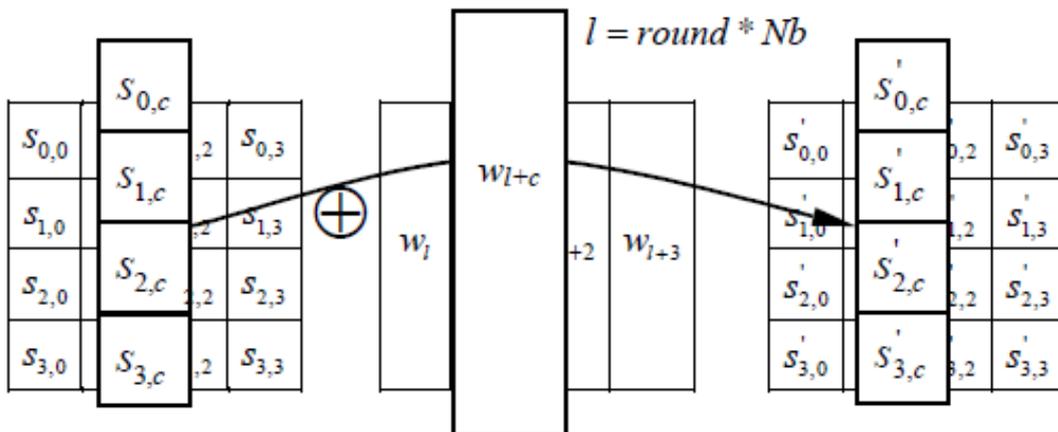


Figure 28 - AddRoundKey() XORs each column of the State with a word from the key schedule

3.2. Key Expansion algorithm

The AES algorithm takes the Cipher Key, K , and performs a *Key Expansion* routine to generate a key schedule. The Key Expansion generates a total of $Nb \cdot (Nr + 1)$ words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted $[w_i]$, with i in the range $0 \leq i < Nb(Nr + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Figure 29.

SubWord() is a function that takes a four-byte input word and applies the S-box (Figure 22) to each of the four bytes to produce an output word.

The function *RotWord()* takes a word $[a_0, a_1, a_2, a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$. The round constant word array, *Rcon[i]*, contains the values given by $[x^{i-1}, \{00\}, \{00\}, \{00\}]$, with x^{i-1} being powers of x (x is denoted as $\{02\}$) in the field $GF(2^8)$ (note that i starts at 1, not 0).

```

KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i = 0
    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1))
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else
            if (Nk > 6 and i mod Nk = 4)
                temp = SubWord(temp)
            end if
            w[i] = w[i-Nk] xor temp
            i = i + 1
        end while
    end
end

```

Figure 29 - Key Expansion pseudo code

From Figure 29, it can be seen that the first Nk words of the expanded key are filled with the Cipher Key. Every following word, $w[i]$, is equal to the XOR of the previous word, $w[i-1]$, and the word Nk positions earlier, $w[i-Nk]$. For words in positions that are a multiple of Nk , a transformation is applied to $w[i-$

1] prior to the XOR, followed by an XOR with a round constant, $Rcon[i]$. This transformation consists of a cyclic shift of the bytes in a word ($RotWord()$), followed by the application of a table lookup to all four bytes of the word ($SubWord()$).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys ($Nk=8$) is slightly different than for 128 and 192-bit Cipher Keys. If $Nk = 8$ and $i-4$ is a multiple of Nk , then $SubWord()$ is applied to $w[i-1]$ prior to the XOR.

3.3. AES Inverse cipher

The Cipher transformations in Sec. 3.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - $InvShiftRows()$, $InvSubBytes()$, $InvMixColumns()$, and $AddRoundKey()$ - process the State; they are not described in the following sections as they haven't been implemented.

3.4. AES operation modes

Every symmetric key block cipher algorithm can work in different modes of operation, as described in Sec. 2.2.2.

AES, being a symmetric block cipher, can operate in one of those modes.

The block cipher modes ECB, CBC, OFB, CFB, CTR provide confidentiality, but they do not protect against accidental modification or malicious tampering. Modification or tampering can be detected with a separate message authentication code such as CBC-MAC, or a digital signature. The cryptographic community recognized the need for dedicated integrity

assurances and NIST responded with AES-CMAC, in a way another AES mode of operation ^[vi].

3.5. AES – CMAC

AES-CMAC provides stronger assurance of data integrity than a checksum or an error-detecting code. The verification of a checksum or an error-detecting code detects only accidental modifications of the data, while CMAC is designed to detect intentional, unauthorized modifications of the data, as well as accidental modifications.

AES-CMAC achieves a security goal similar to that of HMAC.

Since AES-CMAC is based on a symmetric key block cipher, AES, and HMAC is based on a hash function, such as SHA-1, AES-CMAC is appropriate for information systems in which AES is more readily available than a hash function.

AES-CMAC uses the Advanced Encryption Standard (AES) as a building block. To generate a MAC, AES-CMAC takes a secret key, a message of variable length, and the length of the message in octets as inputs and returns a fixed-bit string called a MAC.

The core of AES-CMAC is the basic CBC-MAC. For a message, M, to be authenticated, the CBC-MAC is applied to M. There are two cases of operation in CMAC. Figure 31 illustrates the operation of CBC-MAC in both cases. If the size of the input message block is equal to a positive multiple of the block size (namely, 128 bits), the last block shall be exclusive-OR'ed with K1 before processing. Otherwise, the last block shall be padded with 10^i and exclusive-OR'ed with K2.

The result of the previous process will be the input of the last encryption. The output of AES-CMAC provides data integrity of the whole input message.

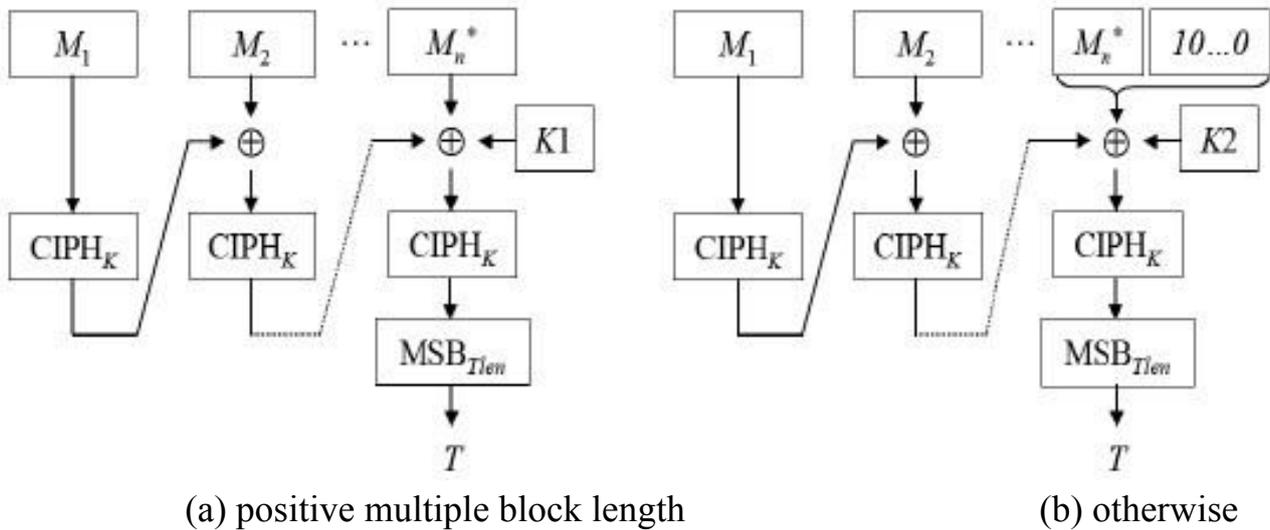


Figure 30 - Two cases of AES-CMAC

- $CIPH_K$ is AES-128 with key K .
- The message M is divided into blocks M_1, \dots, M_n , where M_i is the i -th message block.
- The length of M_i is 128 bits for $i = 1, \dots, n-1$, and *the* length of the last block, M_n , is less than or equal to 128 bits.
- $K1$ is the subkey for the case (a), and $K2$ is the subkey for the case (b).
- $K1$ and $K2$ are generated by the subkey generation algorithm described in section 3.4.1.1.

3.5.1.1. Subkey generation algorithm

The subkey generation algorithm, *Generate_Subkey()*, takes a secret key, K, which is just the key for AES-128. The outputs of the subkey generation algorithm are two subkeys, K1 and K2.

$$(K1, K2) := \text{Generate_Subkey}(K).$$

Subkeys K1 and K2 are used in both MAC generation and MAC verification algorithms. K1 is used for the case where the length of the last block is equal to the block length. K2 is used for the case where the length of the last block is less than the block length.

```
Input   : K (128-bit key)
Output  : K1 (128-bit first subkey)
         K2 (128-bit second subkey)

Constants: const_Zero is 0x00000000000000000000000000000000
          const_Rb  is 0x00000000000000000000000000000087
Variables: L          for output of AES-128 applied to 0^128

Step 1. L := AES-128(K, const_Zero);
Step 2. if MSB(L) is equal to 0
        then K1 := L << 1;
        else K1 := (L << 1) XOR const_Rb;
Step 3. if MSB(K1) is equal to 0
        then K2 := K1 << 1;
        else K2 := (K1 << 1) XOR const_Rb;
Step 4. return K1, K2;
```

Figure 31 - CMAC *generate_subkey()* pseudo code

- In step 1, AES-128 with key K is applied to an all-zero input block.
- In step 2, K1 is derived through the following operation:
 - If the most significant bit of L is equal to 0, K1 is the left-shift of L by 1 bit.
 - Otherwise, K1 is the exclusive-OR of *const_Rb* and the left-shift of L by 1 bit.
- In step 3, K2 is derived through the following operation:
 - If the most significant bit of K1 is equal to 0, K2 is the left-shift of K1 by 1 bit.
 - Otherwise, K2 is the exclusive-OR of *const_Rb* and the left-shift of K1 by 1 bit.
- In step 4, (K1,K2) := *Generate_Subkey*(K) is returned.

3.5.1.2. MAC generation algorithm

The MAC generation algorithm, AES-CMAC(), takes three inputs, a secret key, a message, and the length of the message in octets. The secret key, denoted by K, is just the key for AES-128. The message and its length in octets are denoted by M and len, respectively. The message M is denoted by the sequence of M_i, where M_i is the i-th message block. That is, if M consists of n blocks, then M is written as:

$$- M = M_1 || M_2 || \dots || M_{n-1} || M_n$$

The length of M_i is 128 bits for i = 1,...,n-1, and the length of the last block M_n is less than or equal to 128 bits.

The output of the MAC generation algorithm is a 128-bit string called a MAC, which is used to validate the input message. The MAC is denoted by:

$$T := \text{AES-CMAC}(K, M, \text{len})$$

Validating the MAC provides assurance of the integrity and authenticity of the message from the source.

It is possible to truncate the MAC. According to CMAC, at least a 64-bit MAC should be used as protection against guessing attacks. The result of truncation should be taken in most significant bits first order.

The block length of AES-128 is 128 bits (16 octets). There is a special treatment if the length of the message is not a positive multiple of the block length. The special treatment is to pad M with the bit-string 10^i to adjust the length of the last block up to the block length.

For an input string x of r-octets, where $0 \leq r < 16$, the padding function, *padding(x)*, is defined as follows:

$$\text{padding}(x) = x \parallel 10^i \quad \text{where } i \text{ is } 128 - 8 * r - 1$$

That is, *padding(x)* is the concatenation of x and a single '1', followed by the minimum number of '0's, so that the total length is equal to 128 bits.

Figure 32 describes the MAC generation algorithm.

```
Input  : K ( 128-bit key )
        : M ( message to be authenticated )
        : len ( length of the message in octets )
Output : T ( message authentication code )
Constants: const_Zero is 0x00000000000000000000000000000000
          const_Bsize is 16
Variables: K1, K2 for 128-bit subkeys
          M_i is the i-th block (i=1..ceil(len/const_Bsize))
          M_last is the last block xor-ed with K1 or K2
          n for number of blocks to be processed
          r for number of octets of last block
          flag for denoting if last block is complete or not

Step 1. (K1,K2) := Generate_Subkey(K);
Step 2. n := ceil(len/const_Bsize);
Step 3. if n = 0
      then
        n := 1;
        flag := false;
      else
        if len mod const_Bsize is 0
          then flag := true;
          else flag := false;
Step 4. if flag is true
      then M_last := M_n XOR K1;
      else M_last := padding(M_n) XOR K2;
Step 5. X := const_Zero;
Step 6. for i := 1 to n-1 do
      begin
        Y := X XOR M_i;
        X := AES-128(K,Y);
      end
      Y := M_last XOR X;
      T := AES-128(K,Y);
Step 7. return T;
```

Figure 32 - AES-CMAC pseudo code

- In step 1, subkeys K1 and K2 are derived from K through the subkey generation algorithm.
- In step 2, the number of blocks, n, is calculated. The number of blocks is the smallest integer value greater than or equal to the quotient determined by dividing the length parameter by the block length, 16 octets.
- In step 3, the length of the input message is checked. If the input length is 0 (null), the number of blocks to be processed shall be 1, and the flag shall be marked as not-complete-block (false).
Otherwise, if the last block length is 128 bits, the flag is marked as complete-block (true); else mark the flag as not-complete-block (false).
- In step 4, M_last is calculated by exclusive-OR'ing M_n and one of the previously calculated subkeys. If the last block is a complete block (true), then M_last is the exclusive-OR of M_n and K1.
Otherwise, M_last is the exclusive-OR of padding(M_n) and K2.
- In step 5, the variable X is initialized.
- In step 6, the basic CBC-MAC is applied to M_1,...,M_{n-1},M_last.
- In step 7, the 128-bit MAC, T := AES-CMAC(K,M,len), is returned.
- If necessary, the MAC is truncated before it is returned.

3.5.1.3. Security considerations

The security provided by AES-CMAC is built on the strong cryptographic algorithm AES. However, as is true with any cryptographic algorithm, part of its strength lies in the secret key, K, and the correctness of the implementation in all of the participating systems. If the secret key is compromised or inappropriately shared, it guarantees neither authentication nor integrity of message at all.

If and only if AES-CMAC is used properly it provides the authentication and integrity that meet the best current practice of message authentication.

3.6. AES Key Wrap algorithm

United States of America has chosen AES Key Wrap algorithm for AES keys encryption. This algorithm is described in this section because it will be useful, together with AES-CMAC, during the developing of MACsec Key Agreement protocol in Sec. 4.5.2.

The AES key wrap algorithm is designed to wrap or encrypt key data ^[vii].

The key wrap operates on blocks of 64 bits. Before being wrapped, the key data is parsed into n blocks of 64 bits.

The only restriction the key wrap algorithm places on n is that n be at least two.

The inputs to the key wrapping process are the KEK and the plaintext to be wrapped. The plaintext consists of n 64-bit blocks, containing the key data being wrapped. The key wrapping process is described below (Figure 33).

```

1) Initialize variables.
   Set A[0] to an initial value (see 2.2.3)
   For i = 1 to n
     R[0][i] = P[i]

2) Calculate intermediate values.
   For t = 1 to s, where s = 6n
     A[t] = MSB(64, AES(K, A[t-1] | R[t-1][1])) ^ t
     For i = 1 to n-1
       R[t][i] = R[t-1][i+1]
     R[t][n] = LSB(64, AES(K, A[t-1] | R[t-1][1]))

3) Output the results.
   Set C[0] = A[0]
   For i = 1 to n
     C[i] = R[t][i]

```

Figure 33 - AES Key Wrap pseudo code

The initial value (IV) refers to the value assigned to A[0] in the first step of the wrapping process. This value is used to obtain an integrity check on the key data. In the final step of the unwrapping process, the recovered value of A[0] is compared to the expected value of A[0]. If there is a match, the key is accepted as valid, and the unwrapping algorithm returns it. If there is not a match, then the key is rejected, and the unwrapping algorithm returns an error.

The default initial value (IV) is defined to be the hexadecimal constant:

$$A[0] = IV = A6A6A6A6A6A6A6A6$$

The use of a constant as the IV supports a strong integrity check on the key data during the period that it is wrapped. If unwrapping produces $A[0] = A6A6A6A6A6A6A6A6$, then the chance that the key data is corrupt is 2^{-64} . If unwrapping produces $A[0]$ any other value, then the unwrap must return an error and not return any key data.

4. MAC Security (MACsec)

4.1. Introduction

After the discussion of symmetric cryptographic algorithms (Sec. 3), with special emphasis on AES and its modes of operations AES-CMAC and AES Key Wrap, in this section a new protocol is introduced, MACsec, which takes advantage of the previously treated algorithms.

MACsec (Media Access Control security) ^[viii] has been chosen as security standard by Renesas, in the developing of a project whose aim is to establish secure communications between devices inside automotive environment on Ethernet links.

The main problem to deal with is the access by unauthorized people or devices to controlled or confidential information.

The best and most secure solution to vulnerability at the access edge is to use the intelligence of the network. The standard IEEE 802.1X provides port-based access control using authentication, but authentication alone does not guarantee the confidentiality and integrity of data on the LAN. While physical security and end-user awareness can mitigate threats to data on an IEEE 802.1X–authenticated LAN, in the case of automotive field there may be situations in which the LAN needs additional protection. When additional protection is needed we can enable data confidentiality and integrity on the LAN by using MAC Security (MACsec). Defined by the IEEE 802.1AE standard, MACsec secures communication for authorized endpoints on Ethernet links.

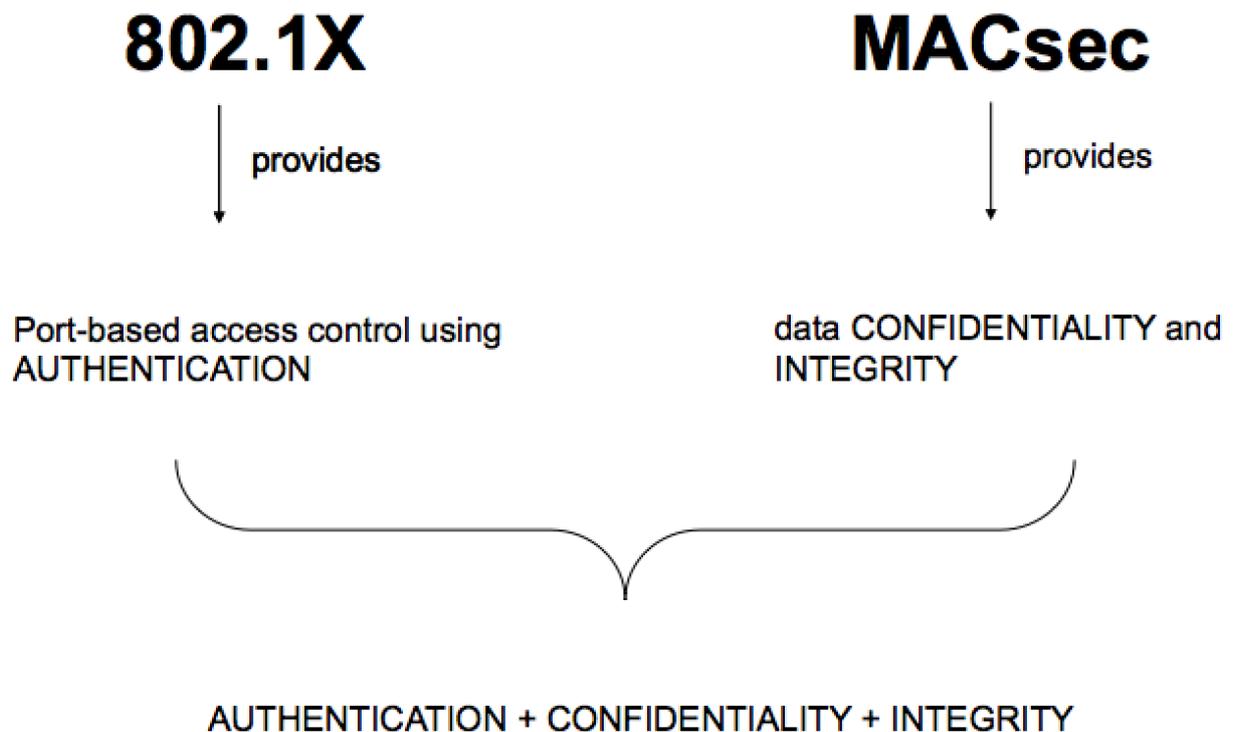


Figure 34 - 802.1X and MACsec

4.2. About MACsec

4.2.1. MACsec Benefits

The reasons why Renesas, and maybe other companies, are going in the direction of MACsec implementations are to be found in the following benefits of the protocol on wired networks:

- *Confidentiality*: MACsec helps ensure data confidentiality by providing strong encryption at Layer 2.
- *Integrity*: MACsec provides integrity checking to help ensure that data cannot be modified in transit.
- *Flexibility*: You can selectively enable MACsec using a centralized policy, thereby helping ensure that MACsec is enforced where required

while allowing non-MACsec-capable components to access the network.

- *Network intelligence*: Unlike end-to-end, Layer 3 encryption techniques that hide the contents of packets from the network devices they cross, MACsec encrypts packets on a hop-by-hop basis at Layer 2, allowing the network to inspect, monitor, mark, and forward traffic according to your existing policies.

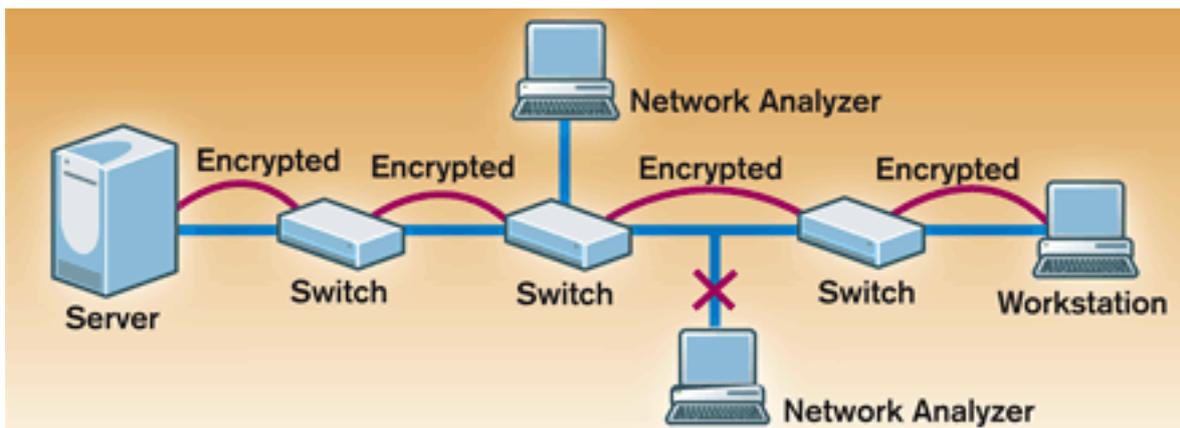


Figure 35 - MACsec hop-by-hop basis

4.2.2. MACsec limitations

Although MACsec offers outstanding data security, it has some limitations:

- *Endpoint support*: Not all endpoints support MACsec.
- *Hardware support*: Line-rate encryption typically requires updated hardware on the access switch.
- *Technology integration*: Enabling MACsec may affect the functions of other technologies that also connect at the access edge, such as IP telephony. Understanding and accommodating these technologies is essential to a successful deployment.

4.3. 802.1X without MACsec

MACsec was primarily designed to be used in conjunction with IEEE 802.1X-2010. IEEE 802.1X provides port-based access control using authentication. An IEEE 802.1X-enabled port can be dynamically enabled or disabled based on the identity of the user or device that connects to it. Figure 36 illustrates the default behavior of an IEEE 802.1X-enabled port prior to authentication: we can see that if the endpoint's identity is unknown all traffic is blocked.

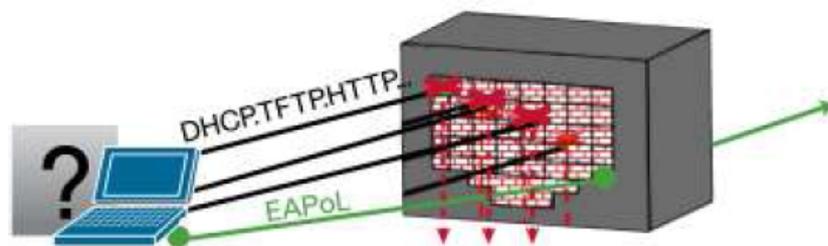


Figure 36 - 802.1X behavior prior to authentication without MACsec

After authentication instead (Figure 37), the endpoint's identity is known and all traffic from that endpoint is allowed. The switch performs source MAC address filtering and port state monitoring to help ensure that only the authenticated endpoint is allowed to send traffic.

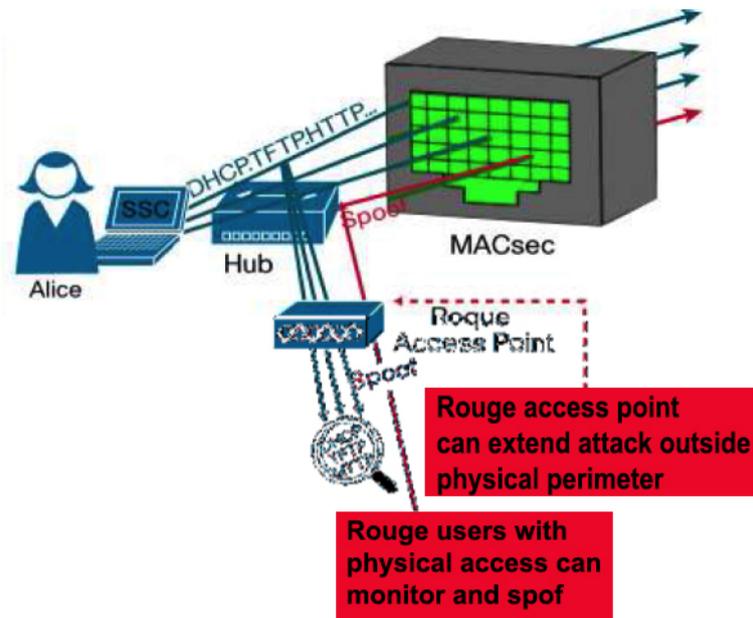


Figure 37 - 802.1X behavior after authentication without MACsec

Before the 2010 revision of IEEE 802.1X, there was no mechanism to help ensure the confidentiality or integrity of the traffic sent after authentication. Because traffic was sent in the clear with no integrity checks, rogue users with physical access to the authenticated port could monitor, modify, and send traffic.

4.4. 802.1X-2010

IEEE 802.1X-2010 defines the way that MACsec can be used in conjunction with authentication to provide secure port-based access control ^[ix] [x].

IEEE 802.1X authenticates the endpoint and transmits the necessary cryptographic keying material to both sides.

Using the master keys derived from the IEEE 802.1X authentication, MACsec can establish an encrypted link on the LAN, thereby helping ensure the security of the authenticated session.

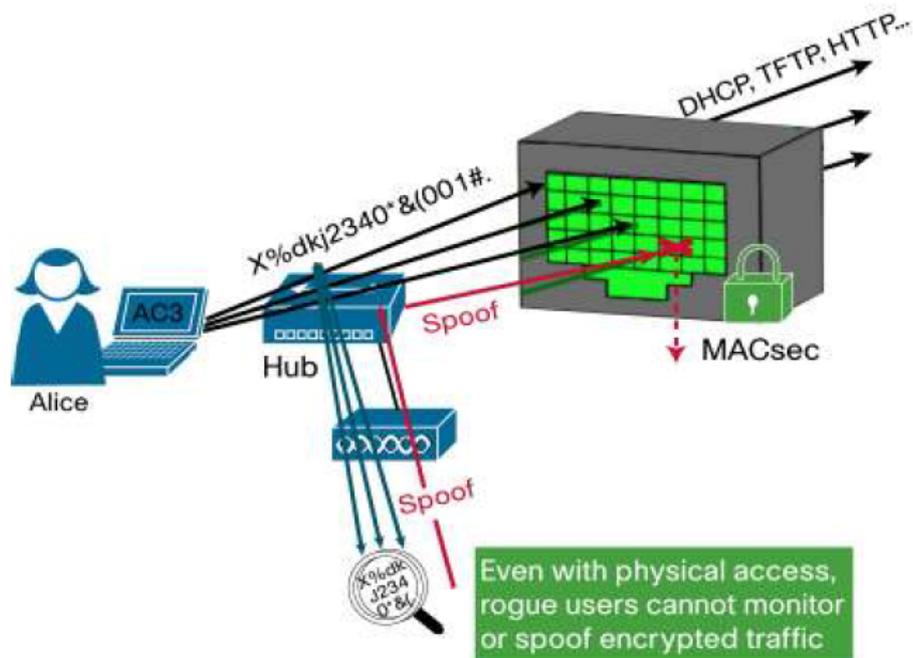


Figure 38 - MACsec enabled port

As we can see from Figure 38, rogue users, even with physical access, can't monitor or spoof encrypted traffic on the wire.

When MACsec is applied on both the uplink and the downlink, the MACsec sessions are completely independent. Moreover, while all traffic is encrypted on the wire, the traffic is in the clear inside each switch. This feature allows the switch to apply all the network policies (quality of service [QoS], deep packet inspection, NetFlow, etc.) to each packet without compromising the security of the packet on the wire. With hop-by-hop encryption, MACsec secures communication while maintaining network intelligence.

4.4.1. Secure communication

Each port that is capable of participating in an instance of the secure MAC Service comprises:

- MAC Security Key Agreement (MKA) Entity (KaY)
- MAC Security Entity (SecY)

A secure Connectivity Association (CA) is created to meet the requirements of the MAC Service and MACsec for connectivity between the stations attached to an individual LAN.

Each CA is supported by unidirectional Secure Channels (SCs), each SC supporting secure transmission of frames through the use of symmetric key cryptography, from one of the systems to all the others in the CA.

Each SC is supported by an overlapped sequence of Security Associations (SAs).

Each SA uses a fresh Secure Association Key (SAK) derived by the MKA to provide the MACsec service guarantees and security services for a sequence of transmitted frames.

In the following Figure we can see a typical scenario where the CA is created by the MACsec Key Agreement following mutual authentication and authorization of A, B and C, and the two SCs that support the CA.

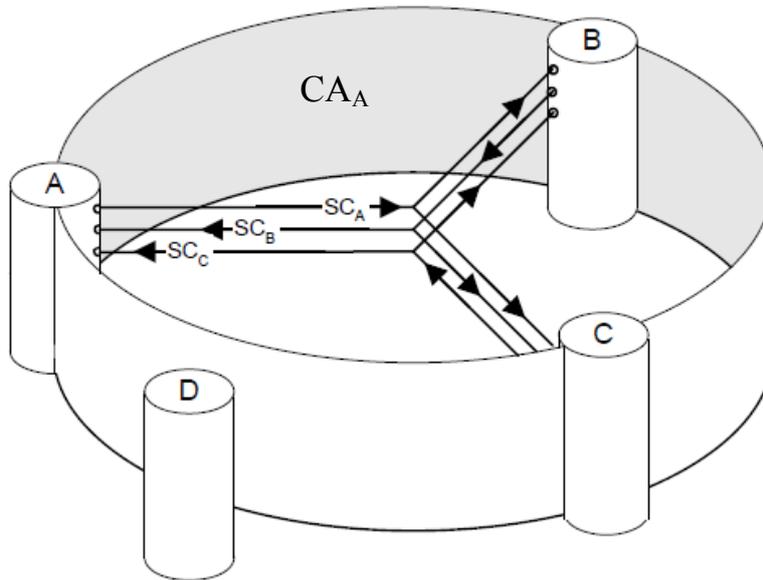


Figure 39 - Secure communication scenario

4.4.2. Components and Protocols

MACsec uses three components (as shown in Figure 40):

1. *Supplicant*: The supplicant is a client that runs on the endpoint and submits credentials for authentication. To support MACsec, the supplicant must also be able to manage MACsec key negotiation and encrypt packets.
2. *Authenticator*: The authenticator (switch) is the network access device that facilitates the authentication process by relaying the supplicant's credentials to the authentication server. The authenticator enforces the network access policy, including MACsec. Like the supplicant, the authenticator must be capable of MACsec key negotiation and packet encryption. The authenticator typically needs special hardware to support MACsec at line rate.

3. *Authentication server*: The authentication server validates the supplicant's credentials and determines what network access the supplicant should receive. In MACsec, the authentication server plays an important role in the distribution of master keying material to the supplicant and authenticator. In addition, the authentication server can define the MACsec policy to be applied to a particular endpoint.

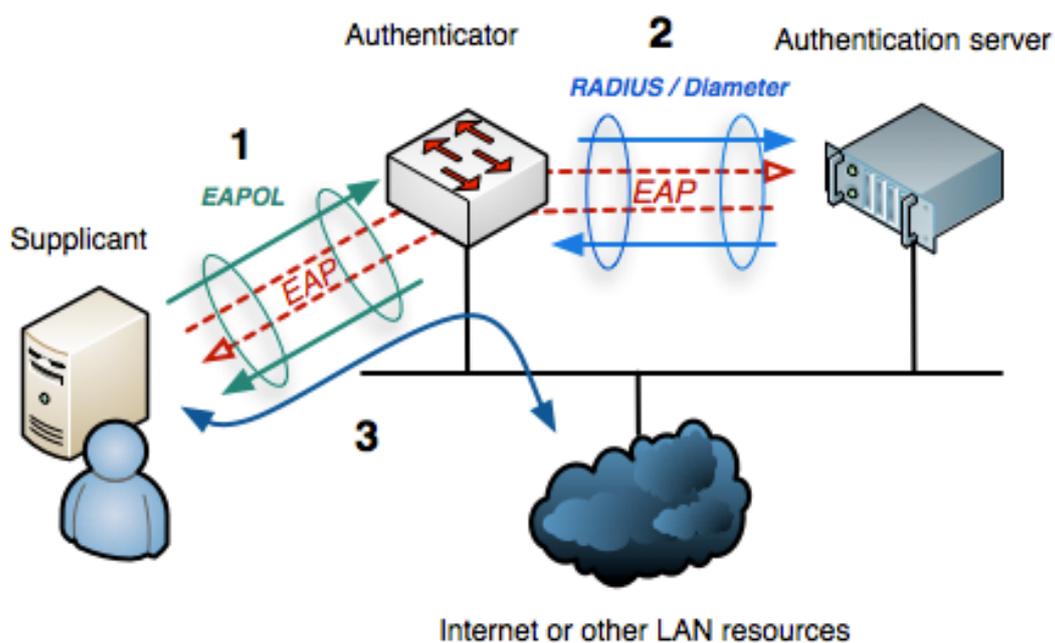


Figure 40 - MACsec components and protocols

MACsec uses several protocols:

- *Extensible Authentication Protocol (EAP)*: The message format and framework defined by RFC 4187 that provides a way for the supplicant and the authenticator to negotiate the EAP authentication method and MACsec association.

- *EAP method*: Protocol that defines the authentication method—that is, the credential type and how it will be submitted from the supplicant to the authentication server using the EAP framework; for MACsec, the EAP method must be capable of generating keying material to export a master session key (MSK) to the supplicant and authentication server.
- *MACsec Key Agreement (MKA)*: Protocol that discovers MACsec peers and negotiates the keys used by MACsec; MKA is defined in IEEE 802.1X-2010.
- *Security Association Protocol (SAP)*: A pre-standard key agreement protocol similar to MKA.
- *EAP over LAN (EAPoL)*: An encapsulation defined by IEEE 802.1X for the transport of EAP from the supplicant to the switch over IEEE 802 wired networks; EAPoL is a Layer 2 protocol (Sec. 4.5.2.2).
- *RADIUS*: Essentially the standard for communication between the switch and the authentication server - the switch extracts the EAP payload from the Layer 2 EAPoL frame and encapsulates the payload inside a Layer 4 RADIUS packet; RADIUS is also used to deliver keying material to the authenticator.

4.5. MACsec Sequence

With the following schema (Figure 41) we can see how the components and protocols of MACsec work together. Messages exchange is divided into three stages: *master key distribution*, *session key agreement*, and *session secure*.

A fourth stage, session termination, is not shown. Each stage is described in the sections that follow.

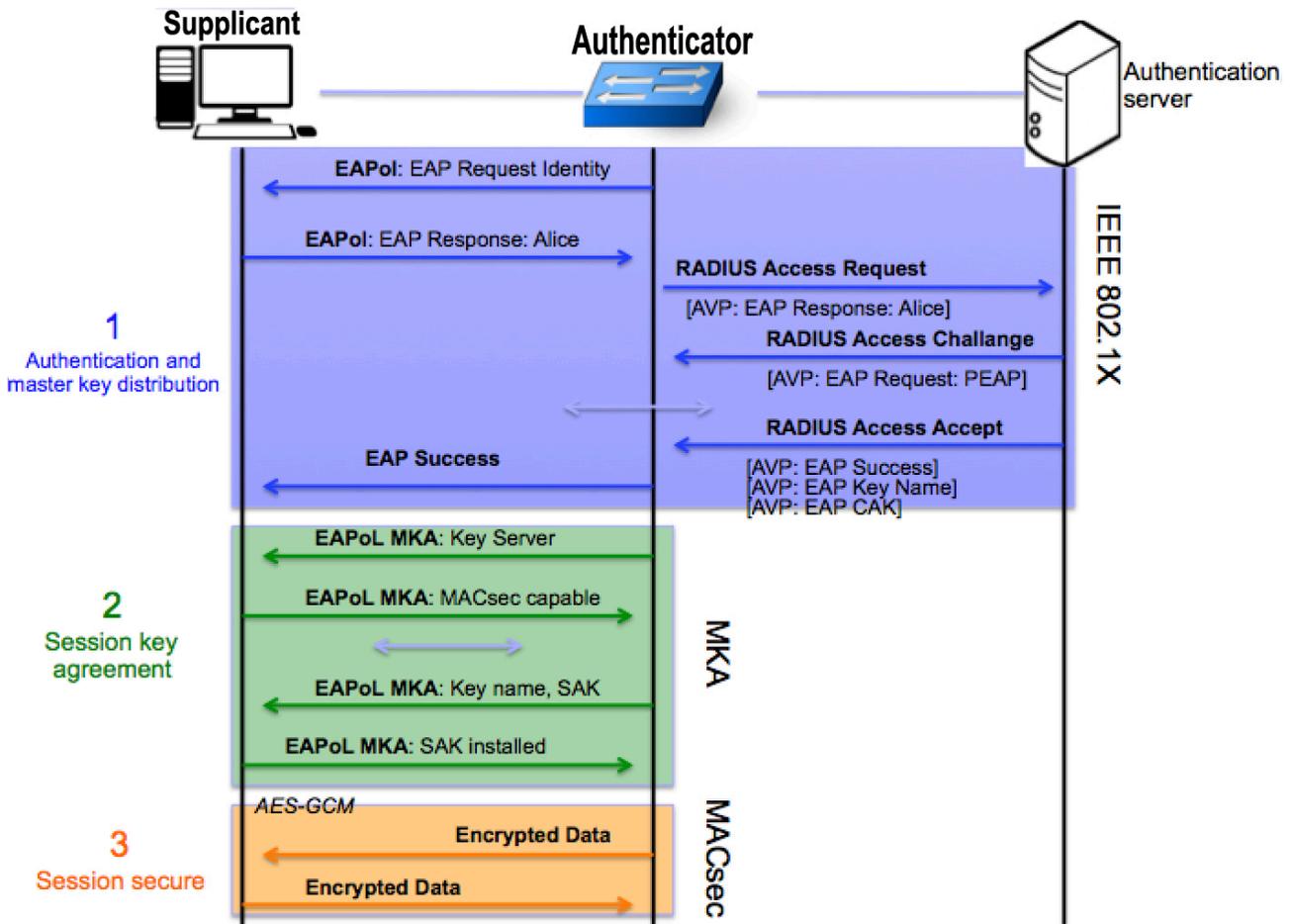


Figure 41 - High Level 802.1X and MACsec sequence

4.5.1. Authentication and Master Key distribution

Through these messages (IEEE 802.1X) master key material will be provided to the supplicant and switch that will subsequently be used by MACsec.

By using an EAP method that supports the generation of encryption keys, the supplicant and the authentication server independently derive the same MSK (Master Session Key). The MSK passes through a key derivation function to

generate a Connectivity Association Key (CAK) on the supplicant and the authentication server. The CAK is a long-lived master key that is used to generate all other keys needed for MACsec in the MKA.

The switch has no visibility into the details of the EAP session between the supplicant and the authentication server, so it cannot derive the MSK or the CAK directly. Instead, the switch receives the CAK from the authentication server in the Access-Accept message at the end of the IEEE 802.1X authentication. The CAK is delivered in the RADIUS vendor-specific attributes (VSAs) MS-MPPE-Send-Key and MS-MPPE-Recv-Key. Along with the CAK, the authentication server sends an EAP key identifier that is derived from the EAP exchange and is delivered to the authenticator in the EAP Key-Name attribute of the Access-Accept message.

4.5.2. Session key agreement (MKA)

In this stage of the protocol both the Supplicant and the Authenticator have the same CAK key: the Supplicant derived it from the MSK using a Key Derivation Function (Sec. 4.5.2.1), while the Authenticator received it from the Authentication Server in the first stage.

This stage takes the name of MACsec Key Agreement (MKA) protocol. The Authenticator's goal is to deliver the SAK (Secure Association Key) to the Supplicant as an encryption key for the future MACsec messages. To do this, the Authenticator derives other two keys (ICK and KEK) starting from the CAK, which is not used directly. ICK (ICV Key) is used for integrity of the SAK, while the KEK (Key Encryption Key) is used in the AES Key Wrap algorithm to wrap the SAK to be distributed.

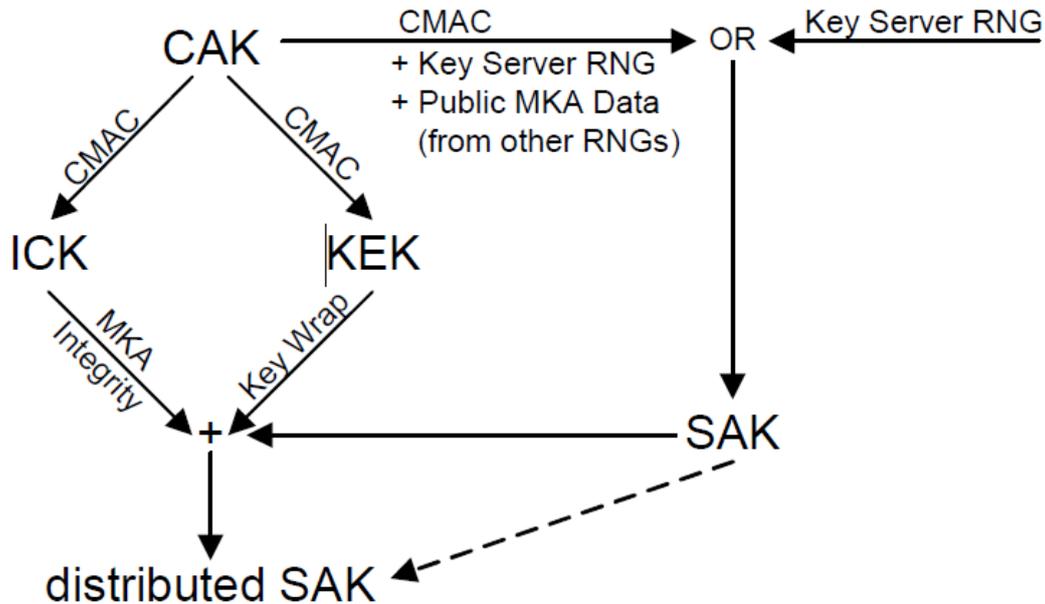


Figure 42 - MKA key hierarchy

In the above Figure the MKA key hierarchy is shown.

The root of key hierarchy for any given instance of MKA is the secure Connectivity Association Key (CAK), a secret key. Possession of a CAK for the CA is a prerequisite for membership in each CA supported by MACsec, and all potential members possess the same CAK and are attached to the same LAN.

The main function, which is used in the creation of the two keys ICK and KEK, is the KDF (Key Derivation Function), defined in the standard (Sec. 4.5.2.1). KDF uses a PRF (Pseudo Random Function), which in this case is AES-CMAC.

4.5.2.1. KDF (Key Derivation Function)

The key derivation function (KDF) defined in the 802.1X-2010 standard is the main function used to derive the keys in the MKA key hierarchy (see Figure 42).

The KDF uses a pseudorandom function (PRF), which shall be AES-CMAC-128 when the derivation key is 128 bits.

The KDF is described as follows:

```
Output ← KDF (Key, Label, Context, Length)
```

where

Input:

Key, a key derivation key of 128 or 256 bits

Label, a string identifying the purpose of the keys derived using this KDF
Context, a bit string that provides context to identify the derived key

Length, the length of the output in bits encoded in two octets with the most significant octet first

Output: a Length-bit derived value

Fixed values:

h, the length of the output of the PRF in bits

r, denoting the length of the binary representation of the counter *i*

iterations ← (Length + (h-1))/h

if iterations > 2^{*r*}-1, **then** indicate an error and stop.

result ← ""

do *i* = 1 to iterations

result ← result | PRF(Key, *i* | Label | 0x00 | Context | Length)

od

return first Length bits of result, and securely delete all unused bits

Figure 43 - KDF pseudo code

4.5.2.2. MKA transport

MKA provides a secure multipoint-to-multipoint transport between the members of the same CA, suitable for conveying information that is constant, or refreshed or acknowledged by the MKA applications that make use of that transport. The CAK is used to authenticate each protocol data unit (MKPDU) transmitted, providing proof of its transmission by a CA member, and each station includes its own randomly chosen identifier and a message number in the MKPDU. By transmitting MKPDUs that contain the identifiers and recent message numbers of the other participants, each member proves that it is in current possession of the CAK and is actively participating in the protocol, thus demonstrating the ‘liveness’ of the MKPDU and distinguishing it from MKPDUs that could have been captured by an attacker and played or replayed later—with the aim of disrupting the protocol or of influencing its outcome. MKPDUs are transmitted at regular intervals of MKA Hello Time or MKA Bounded Hello Time.

The message numbers also serve to enforce in-order delivery, and each of the MKA applications is designed so that the information conveyed in each MKPDU is idempotent, i.e., can be repeated without further changing the state of a recipient, and complete, i.e., fully expresses the desire of the transmitter for state change at the recipient. This design philosophy simplifies protocol analysis and allows a receiver to discard MKPDUs with prior message numbers.

The MKA transport is fully distributed and, as a consequence, robust in the face of the failure of any participant or of the LAN connectivity to that participant.

4.5.2.3. EAPoL

MKA Protocol Data Units (MKPDUs) are transmitted as body of EAPoL MKA messages.

EAP (Extensible Authentication Protocol) is an authentication framework which supports multiple authentication methods.

The encapsulation of EAP over IEEE 802 is defined in IEEE 802.1X and known as "EAP over LANs" or EAPoL.

The same three main components are defined in EAP and EAPoL to accomplish the authentication conversation:

1. Supplicant (Port Authentication Entity (PAE) seeking access to network resources)
2. Authenticator (PAE that controls network access)
3. Authentication Server (a RADIUS/AAA server)

The following figure shows how these LAN components are connected in a wired environment (as discussed for MACsec in Sec. 4.5).

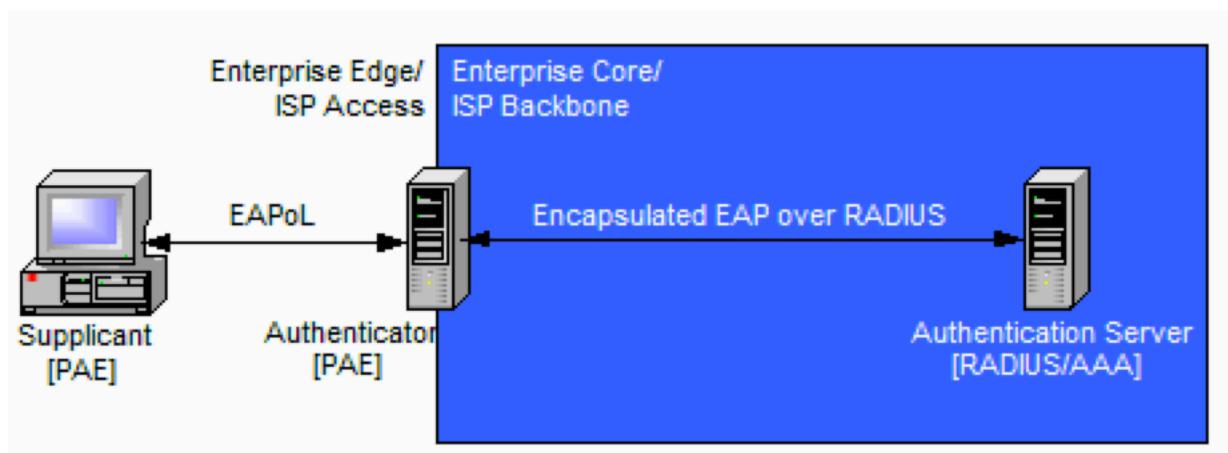


Figure 44 - EAPoL architecture

The EAPoL frame has the following format:

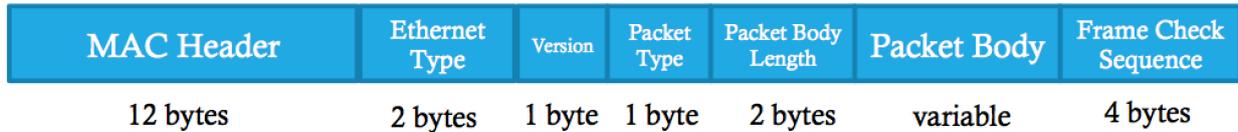


Figure 45 - EAPoL frame format

The fields in the frame are:

MAC Header

The first 6 bytes of the MAC header are the Destination Address and the last 6 bytes are the Source Address.

Ethernet Type

The Ethernet Type contains a 88-8e, this is the two byte type code assigned to EAPoL.

Version

In 2004 Version 2 was standardized, nothing has been standardized since.

Packet Type

The Packet Type field is a byte long and represents the type of package the frame is.

Packet Type	Name	Description
00000000	EAP-Packet	Contains an encapsulated EAP frame (this is what majority of EAPoL frames are)
00000001	EAPOL-Start	A supplicant can issue an EAPOL-Start frame instead of waiting for a challenge from the authenticator
00000010	EAPOL-Logoff	Used to return the state of the port to unauthorized when the supplicant is finished using the network
00000011	EAPOL-Key	Used to exchange Cryptographic Keying information
00000100	EAPOL-Encapsulated-ASF-Alert	Provided as a method of allowing Alerting Standards Forum (ASF) alerts (ex. specific SNMP traps) to be forwarded through a port that is in the Unauthorized state
00000101	EAPOL-MKA	Used to exchange

During the MKA protocol the EAPoL method used is the EAPOL-MKA (type 5).

Packet Body Length

The Packet Body Length field is a 2 byte value representing packet body length (It is set to 0 when there is no packet body)

Packet Body

The Length field is two bytes long and contains the number of bytes in the entire packet. EAP assumes anything in excess of the Length is padding that can be ignored.

Frame Check Sequence

The Frame Check Sequence (FCS) is checksum value added to the frame for error detection and correction.

Each MKPDU (Figure 46) comprises a number of parameter sets. The first of these, the Basic Parameter Set, is always present, and is followed by zero or more further parameter sets, followed by the ICV. The ICV comprises the last 16 octets of the MKPDU, as indicated by the EAPOL Packet Body Length.

Protocol Version		
Packet Type = EAPOL-MKA		
Packet Body Length		Size
Packet Body (MKPDU)	Basic Parameter Set	Multiple of 4 octets
	Parameter Set	Multiple of 4 octets
	Parameter Set	Multiple of 4 octets
	ICV	16 octets

Figure 46 - EAPOL - MKA packet body with MKPDU format

4.5.2.4. SAK generation

The Key Server is responsible for generating and distributing MACsec SAKs, using AES Key Wrap, to each of the other members of the CA, using the MKA transport.

Each SAK is identified by a 128-bit Key Identifier (KI), comprising the Key Server's MI (providing the more significant bits) and a 32-bit Key Number (KN) assigned by that Key Server (sequentially, beginning with 1). Each KI is used to identify the corresponding SAK for the purposes of SA assignment, and appears in the clear in MKPDUs, so network management equipment and personnel can observe and diagnose MKA operation (if necessary) without having access to any secret key.

Each SAK should be generated using the KDF specified in Sec. 4.5.2.1. using the following transform:

$$\text{SAK} = \text{KDF}(\text{Key}, \text{Label}, \text{KS-nonce} \mid \text{MI-value list} \mid \text{KN}, \text{SAKlength})$$

Where:

- Key = CAK Label = "IEEE8021 SAK"
- KS-nonce = a nonce of the same size as the required SAK, obtained from an RNG each time an SAK is generated.
- MI-value list = a concatenation of MI values (Member Identifier, randomly chosen by each participant at the beginning of the protocol) from all live participants.
- KN = four octets, the Key Number assigned by the Key Server as part of the KI
- SAKlength = two octets representing an integer value (128 for a 128 bit SAK, 256 for a 256 bit SAK) with the most significant octet first.

4.5.2.5. CAK derivation

A pairwise CAK is derived directly from the EAP MSK using the following transform:

$$\text{CAK} = \text{KDF}(\text{Key}, \text{Label}, \text{mac1} \mid \text{mac2}, \text{CAKlength})$$

Where:

- Key = MSK[0-15] for a 128 bit CAK, MSK[0-31] for a 256 bit CAK.
- Label = "IEEE8021 EAP CAK"
- mac1 = the lesser of the two source MAC addresses used in the EAPOL-EAP exchange.
- mac2 = the greater of the two source MAC addresses used in the EAPOL-EAP exchange.
- CAKlength = two octets representing an integer value (128 for a 128 bit CAK, 256 for a 256 bit CAK) with the most significant octet first.

A 16 octet CKN is derived from the EAP session ID using the following transform:

$$\text{CKN} = \text{KDF}(\text{Key}, \text{Label}, \text{ID} \mid \text{mac1} \mid \text{mac2}, \text{CKNlength})$$

Where:

- Key = MSK[0-15] for a CKN naming a 128 bit CAK, MSK[0-31] for naming a 256 bit CAK.
- ID = EAP-Session-ID Label = "IEEE8021 EAP CKN"

- mac1 = the lesser of the two source MAC addresses used in the EAPOL-EAP exchange.
- mac2 = the greater of the two source MAC addresses used in the EAPOL-EAP exchange.
- CKNlength = two octets representing an integer value (128) with the most significant octet first.

4.5.2.6. ICK derivation

ICK (Integrity Check value Key) is a 128bit key derived using the KDF with the following parameters:

- Key : CAK (16 octets)
- Label : "IEEE8021 ICK" (12 octets)
- Context: first 16 octets of the CKN for the CAK
- Length: ICK length ("0080", 2 octets)

This is a test vector for the ICK derivation.

- Key : *135bd758 b0ee5c11 c55ff6ab 19fdb199*
- Label : *49454545 38303231 2049434b*
- Context: *96437a93 ccf10d9d fe347846 cce52c7d*
- Length: *0080*
- Output: *8f1c5cb1 c8ed2e5f 047906e0 473aad4d*

The ICK key is used to produce an ICV for integrity protection as described in Sec. 4.5.2.8.

4.5.2.7. KEK derivation

KEK (Key Encryption Key) is a 128bit key (Figure 42) that will be used as key in the AES Key Wrap algorithm (Sec. 3.6) to protect the session key SAK for MACsec communication.

The KEK is derived from the CAK using the following transform:

$$\text{KEK} = \text{KDF}(\text{Key}, \text{Label}, \text{Keyid}, \text{KEKLength})$$

Where:

- Key = CAK Label = "IEEE8021 KEK"
- Keyid = the first 16 octets of the CKN, with null octets appended to pad to 16 octets if necessary
- KEKLength = two octets representing an integer value (128 for a 128 bit KEK, 256 for a 256 bit KEK) with the most significant octet first

4.5.2.8. Message authentication

Each protocol data unit (MKPDU) transmitted is integrity protected by an 128 bit ICV, generated by AES- CMAC using the ICK:

$$\text{ICV} = \text{AES-CMAC}(\text{ICK}, \text{M}, 128)$$
$$\text{M} = \text{DA} + \text{SA} + (\text{MSDU} - \text{ICV})$$

In other words, M comprises the concatenation of the destination and source MAC addresses, each represented by a sequence of 6 octets in canonical format order, with the MSDU (MAC Service Data Unit) of the MKPDU

including the allocated Ethertype, and up to but not including, the generated ICV.

5. MKA Key Hierarchy SW Implementation

After the conceptual and algorithmic description of AES and MACsec, with particular attention to the MKA Key Hierarchy protocol, it's possible, after the commitment of Renesas, to show a software implementation of the key hierarchy described in Sec. 4.5.2.

The implemented software will be used to test the hardware solution (Sec. 6), to show its output correctness.

5.1. Java implementation

One of the main reasons Java has been chosen at first is its *platform independence*, which means that Java programs can be run on many different types of computers. A Java program runs on any computer with a *Java Runtime Environment*, also known as a *JRE*, installed. A JRE is available for almost every type of computer — PCs running Windows, Macintosh computers, Unix or Linux computers, huge mainframe computers, and even cell phones.

Regarding automotive environment Java is not the best solution, but as will be shown later (Sec. 5.2) a C implementation will be preferred.

However Java software is very useful for future hardware results test and it's faster and easier to implement respect to C.

5.1.1. AES – CMAC

The AES-CMAC Java module has been developed to obtain a CMAC value starting from an input message and key of variable length. The realization followed the specifications of the [rfc4493] standard.

INPUTS:

- *Message* to be encrypted
- *Key* for encryption

OUTPUT:

- *CMAC* 128bit value

```
Start time (ms):1430211064318
Input message : a8de55170c6dc0d80de32f508bf49b70
End time (ms):1430211064545
Final value: cfef9b7839841fdbccbb6c2cf238f7a3
```

Figure 47 - CMAC Java sample output

5.1.1.1. Cipher.class

As described in Sec. 3.5 the AES-CMAC relies on AES encryption algorithm. The final software takes advantage of the Java *Cipher.class* class: this class provides the functionality of a cryptographic cipher for encryption and decryption. It forms the core of the Java Cryptographic Extension (JCE) framework ^[xi].

In order to create a Cipher object, the application calls the Cipher's *getInstance* method, and passes the name of the requested *transformation* to it. Optionally, the name of a provider may be specified.

A *transformation* is a string that describes the operation (or set of operations) to be performed on the given input, to produce some output.

A transformation always includes the name of a cryptographic algorithm (e.g., *AES*), and may be followed by a feedback mode and padding scheme.

A transformation is of the form:

- "*algorithm/mode/padding*" or
- "*algorithm*"

(in the latter case, provider-specific default values for the mode and padding scheme are used).

The transformation used in this case has been of the type:

```
Cipher aesCipher = Cipher.getInstance("AES/CBC/NOPADDING");
```

As we can see the algorithm is of course AES, while the mode is CBC with no padding; there is no CMAC mode in the modes list of the transformations which can be requested by the *getInstance* method.

After having the cipher instance, the *init* method is called to initialize the cipher with a key and a set of algorithm parameters.

```
public final void init(int opmode, Key key, AlgorithmParameterSpec params)
    throws InvalidKeyException, InvalidAlgorithmParameterException
```

opmode - the operation mode of this cipher (this is one of the following: ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE or UNWRAP_MODE)

key - the encryption key

params - the algorithm parameters

The cipher is initialized for one of the following four operations: encryption, decryption, key wrapping or key unwrapping, depending on the value of *opmode*.

In the CMAC case, the *init* method is called as follows:

```
aesCipher.init(Cipher.ENCRYPT_MODE, key, ZERO_IV);
```

where:

- *opmode* is ENCRYPT_MODE since we have to encrypt the message created with the CMAC algorithm
- *key* is a 128bit key
- ZERO_IV is a 128bit initialization vector (as requested in the CMAC algorithm) of 16 octets equals to 0x00; it belongs to the IvParameterSpec class which specifies an initialization vector (IV).

To start the encryption the method *update* has to be called:

```
public final int update(byte[] input, int inputOffset, int inputLen, byte[] output,  
int outputOffset)  
    throws ShortBufferException
```

The update method used in the case of CMAC continues a multiple-part encryption processing another data part.

The first *inputLen* bytes in the input buffer, starting at *inputOffset* inclusive, are processed, and the result is stored in the output buffer, starting at *outputOffset* inclusive.

Parameters:

- *input* - the input buffer
- *inputOffset* - the offset in input where the input starts
- *inputLen* - the input length
- *output* - the buffer for the result
- *outputOffset* - the offset in output where the result is stored

At the end of a multi-part encryption done with update method, the doFinal method has to be called:

```
public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output,  
    int outputOffset)  
    throws ShortBufferException, IllegalBlockSizeException,  
    BadPaddingException
```

The method encrypts data in a single-part operation, or, in this case, finishes a multiple-part operation.

The first *inputLen* bytes in the *input* buffer, starting at *inputOffset* inclusive, and any input bytes that have been buffered during a previous *update* operation, are processed, with padding (if requested) being applied.

Parameters:

- *input* - the input buffer
- *inputOffset* - the offset in input where the input starts
- *inputLen* - the input length
- *output* - the buffer for the result
- *outputOffset* - the offset in output where the result is stored

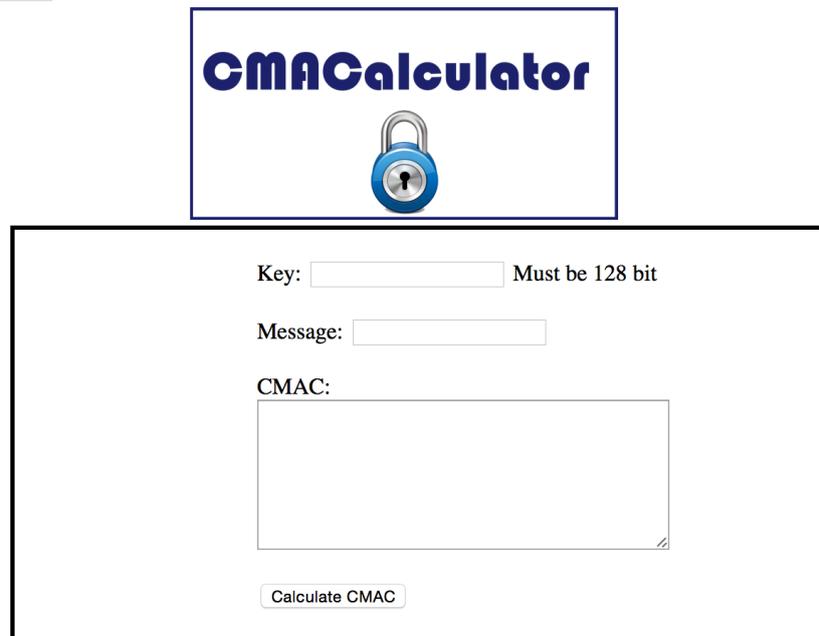
5.1.1.2. Results

In order to test the software solution of the AES-CMAC the results are being compared to the test vectors of AES-CMAC 128 given by NIST ^[xii].

5.1.1.3. Applet Java and Web Server

After the AES_CMACH Java implementation, a Web Applet has been created to give the possibility to use the CMAC calculator on a webpage available on the net.

The homepage of the so called *CMACCalculator* (Figure 48) consists in two simple fields asking for the *key* and the *message* for the encryption algorithm.



CMACCalculator



Key: Must be 128 bit

Message:

CMAC:

Figure 48 - CMACCalculator homepage

To make our Java `AesCmac` class run by a web browser, a Java applet has to be realized.

A Java applet is a small application which is written in Java and delivered to users in the form of bytecode. The user launches the Java applet from a web page, and the applet is then executed within a Java Virtual Machine (JVM) in a process separate from the web browser itself.

A Java applet extends the class `java.applet.Applet`. The class which must override methods from the applet class to set up a user interface inside itself (*Applet*) is a descendant of *Panel* which is a descendant of *Container*. As applet inherits from container, it has largely the same user interface possibilities as an ordinary Java application, including regions with user specific visualization.

So to turn the `AesCmac` class in a Java applet it must extend `Applet`:

```
public class AesCmac extends Applet { ... }
```

Now the html page has to include the `<applet>` tag in the `<head>` of the page:

```
<APPLET id="cmac" CODE="AesCmac.class">
```

Of course to make the applet run in the web browser, Java must be activated.

5.1.2. Results and performance

The main module in the MKA Key Hierarchy protocol from a software point of view is the AES-CMAC described in Sec. 5.1.1.

Another implemented module is AES Key Wrap algorithm, which is compliant with the *rfc3394*.

The whole MKA Key Hierarchy Java implementation flow is shown below:

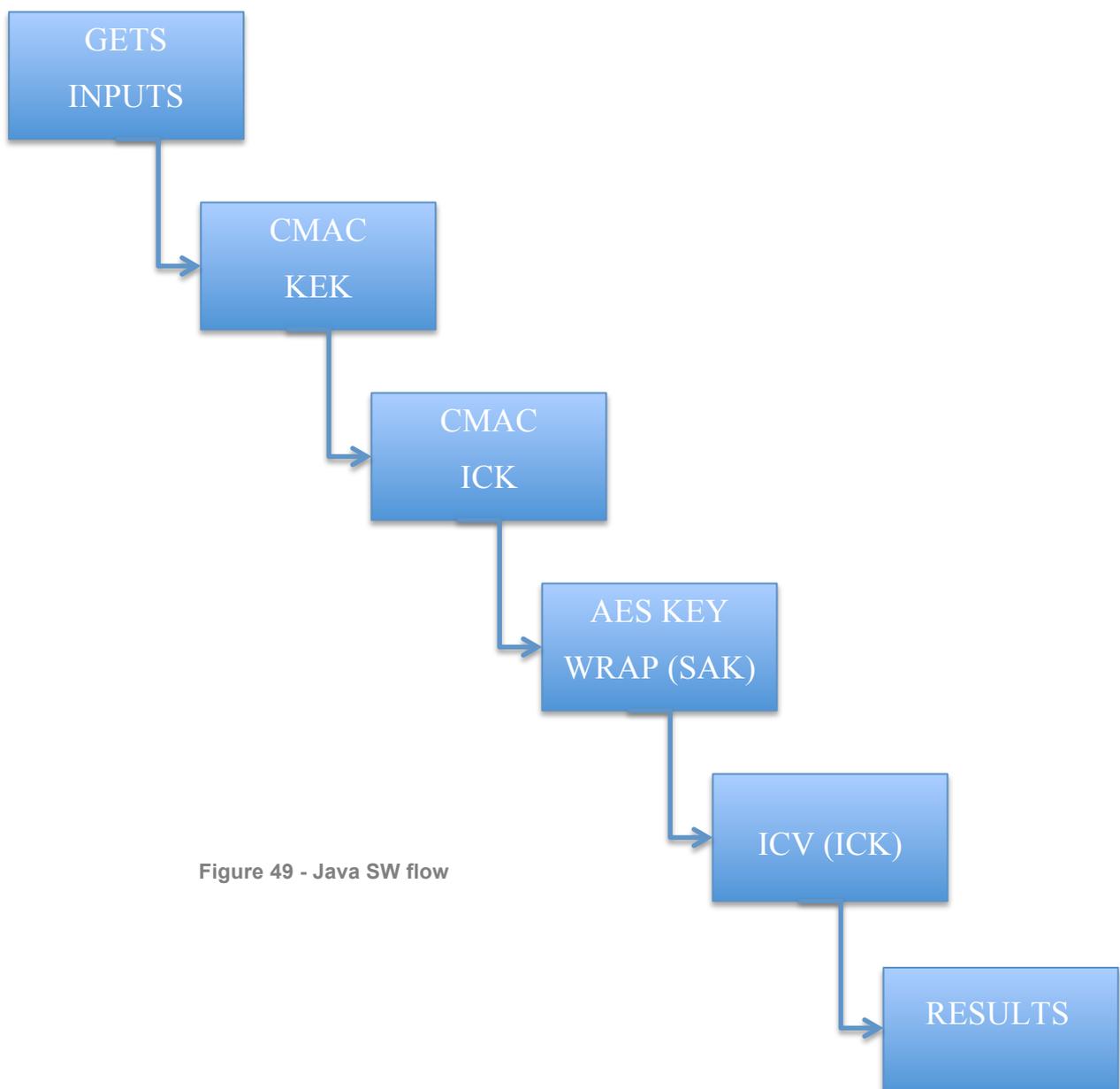


Figure 49 - Java SW flow

To test the performance of the Java implementation we have to face the time precision issues related to:

- Clock resolution (less accuracy)
- Java Virtual Machine (JVM) implementation in each operating system
- Computer architecture

Java.lang.System.currentTimeMillis() is used to get timing information; it runs faster than others (5/6 CPU clocks).

Protocol	Speed in ms
AES-CMAC	727
AES KEY WRAP	534
ICV	230
TOTAL	1491

Figure 50 - MKA Key hierarchy Java performance

These results are obtained with the following computer architecture:

- CPU: Intel i7 2,3GHz
- RAM: 8GB DDR3
- OS: Win 7 64bit
- JVM version 8, build 1.8.0_45-b14

As visible from the above table, the most time expensive software module is the AES-CMAC, since it has to encrypt multiple times using the AES cipher.

5.2. C implementation

The C implementation of the AES-CMAC algorithm came straightforward after the need to authenticate the devices to the main board, which are not MACsec already capable. The other modules of MKA Key Hierarchy are not implemented yet.

To deal with AES encoding in C there is the need of the OpenSSL library, which has to be included in the way:

```
#include <openssl/aes.h>
```

The version of OpenSSL used is the 1.0.2.

With OpenSSL, including AES.h, we can encrypt using AES in this way:

```
Char *key; //String containing the key
unsigned char IN[16] = “.....”;
unsigned char OUT[16] = “.....”;
AES_KEY aes; //structure to hold the key
AES_set_encrypt_key (key, 128, &aes);
AES_encrypt (IN , OUT , &aes); // final encryption
```

Figure 51 - Aes C usage

The CMAC software structure is as follows:

SOURCE FILE

Implementation

cmac.cpp

```
#include "cmac.h"
```

```
#include <openssl/aes.h>
```

HEADER FILE

Interface

cmac.h

The test file main.cpp (#include "cmac.h") produces the following screen outputs:

```
Insert 32 Hex KEY (128bit)
c595ee7655c8eeecd3e8fbbbc439dbe2
Specify message length: from 0 to 128 digits (not odd numbers)
64
Insert 64 Hex MESSAGE
39fb12288a67f15fa4191d597c834dc0a049a4fc6ca686b1810ca988730a6f33
Insert output MAC length (hex digits)
30
Final CMAC:
8398e6153ba580cf3c4254bcdda2f
```

Figure 52 - CMAC C test file output

6. MKA Key Hierarchy HW Implementation

After the conceptual and algorithmic description of the AES and the AES-CMAC ciphers, of AES KEY WRAP and the software implementation of the MKA Key Hierarchy, it's possible to illustrate the different hardware architectures to implement them.

6.1. AES and Key Expansion modules

Concerning the AES encryption algorithm, the attention must be focused to the *SubBytes()* transformation which is the most expensive step of the whole AES cipher in terms of resources and presents the longest critical path (see Sec. 3.1.1). The byte substitution is performed by the S-box which is byte oriented, so to substitute the whole State (128-bit), 16 instances of the S-box are needed: 16 S-boxes can be used in parallel in one time or less S-boxes in more than one time (e.g. 4 S-boxes in 4 times). The S-box hardware implementation is critical because of the computation of the multiplicative inverse of a byte on the Galois field $GF(2^8)$: as the same Rijndael cipher's authors and the NIST indicate, the Extended Euclidean algorithm should be used. This procedure is very expensive in terms of hardware delay because it requires the integer division, which is a serial operation. As the literature suggests ^[xiii], there are two usual implementations of the S-box: one based on the lookup tables, or memory supports like the dedicated RAM or ROM blocks on an FPGA, or one based on the Galois composite fields. The chosen solution is the first one.

The LUT (Look-Up Table)-based implementation of the S-box simply consists in storing and arranging appropriately the output values of the S-box in relation to all possible values of the input data, that is the byte to be

substituted. As already hinted in Sec. 3.1.1, the result is a 256 bytes table arranged as a 16×16 matrix in which the most significant nibble of the input byte selects the matrix row, while the least significant nibble of the input bytes selects the matrix column. This solution is widely diffused because its implementation requires a very few effort and it brings to a significant gain in terms of maximum achievable frequency. Anyway it can be almost expensive in terms of area consumption. The LUT-based S-box is usually employed when the AES cipher is implemented on an FPGA device, while it's usually discharged for the ASIC realizations.

Concerning the other AES round transformations, *ShiftRows()*, *MixColumns()* and *AddRoundKey()*, they do not leave space to any optimization or significant architectural variation. Focusing on the overall architecture of an AES cipher, there are many possibilities. The primary aspect concerns the number of rounds physically implemented. The structure created works with only one round used iteratively: in this case some multiplexer are needed to bypass the preliminary *AddRoundKey()* transformation and the *MixColumns()* one in the last round execution and the system is characterized by a low area consumption, even if with a higher latency.

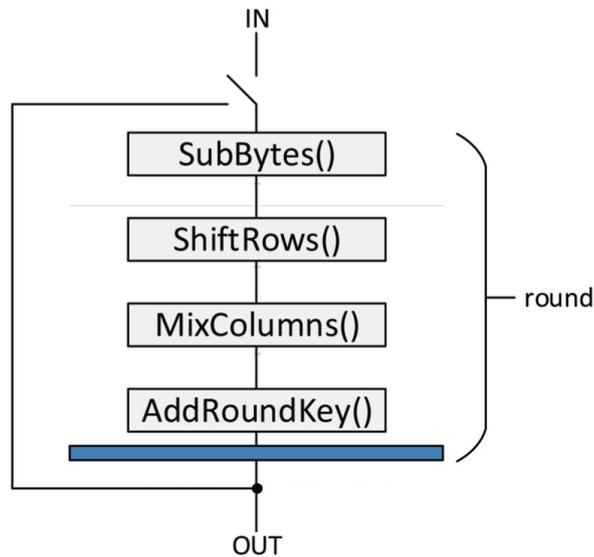


Figure 53 - AES rolled architecture

Lastly some few words can be spent about the key expander (see Sec. 3.2). The first point faced when projecting this module is the decision of when perform the key expansion: we chose to perform the expansion before the encryption, storing all the round keys in appropriate memory supports, instead of computing the round keys at runtime (or "on the fly"). In Figure 54 we can see the AES core architecture with a buffer on the output to store the value.

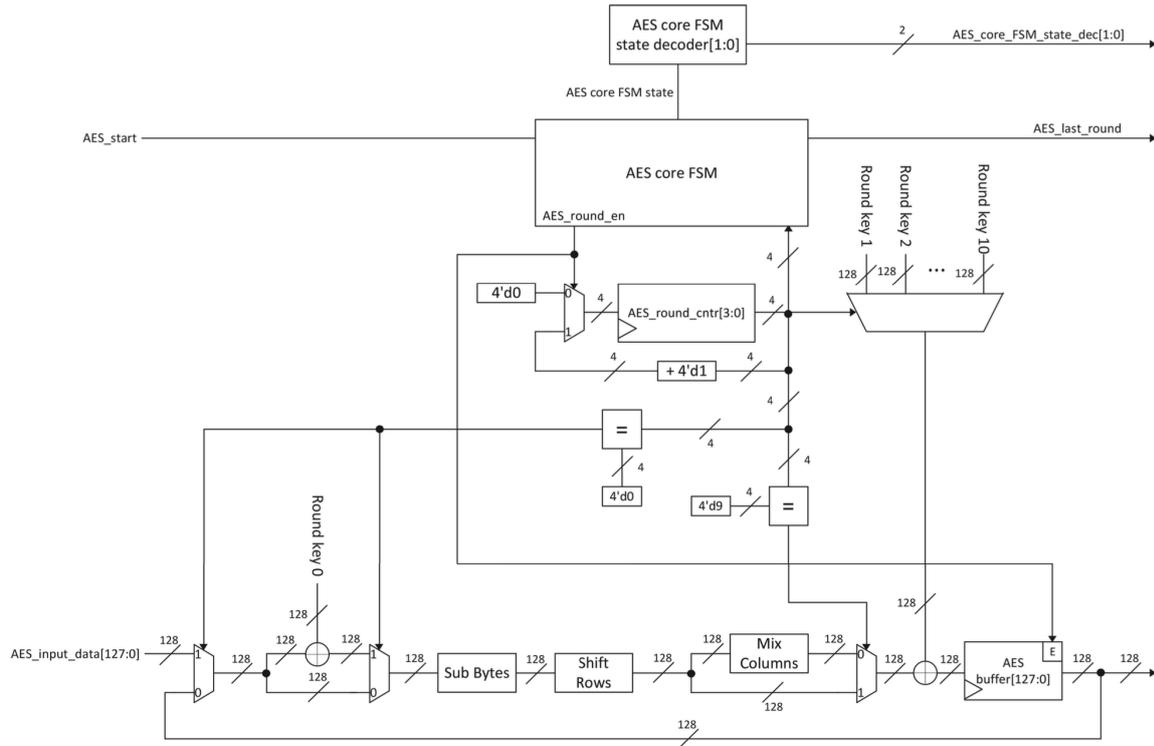


Figure 54 - Implemented AES core

Each time that a start signal occurs, the FSM enables the AES core, this one processes the data block with the help of the counter that drives the multiplexers and when the encrypted block is ready this is signaled through a last step signal, while the FSM return to the idle state. The Figure 55 shows the states map of the AES core.

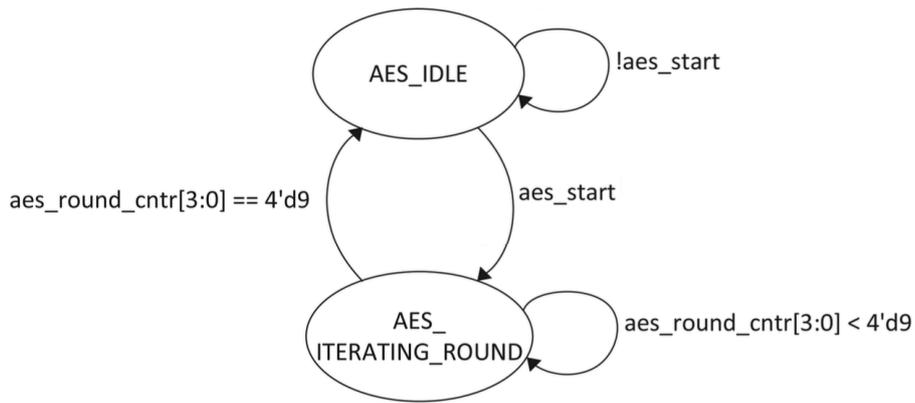
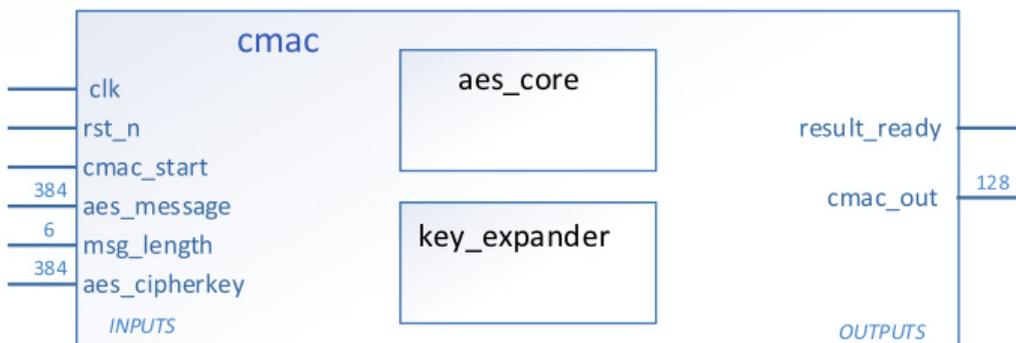


Figure 55 - AES core finite state machine

6.2. CMAC module

The other implemented module is the *cmac* module. It instantiates the two previously described modules: *aes_core* and *key_expander*.



(a)

```

key_expander U1 (clk, rst_n, cmac_start, aes_cipherkey, aes_out_key, key_1, key_2,
  key_3, key_4, key_5, key_6, key_7, key_8,
  key_9, key_10, key_expansion_done);
  
```

```

aes_core U2 (clk, rst_n, key_expansion_done | aes_start, aes_input, aes_cipherkey,
  key_1, key_2, key_3, key_4, key_5, key_6, key_7, key_8,
  key_9, key_10, aes_last_round, aes_buffer );
  
```

(b)

Figure 56 - (a) CMAC block diagram, (b) CMAC instantiated modules in Verilog

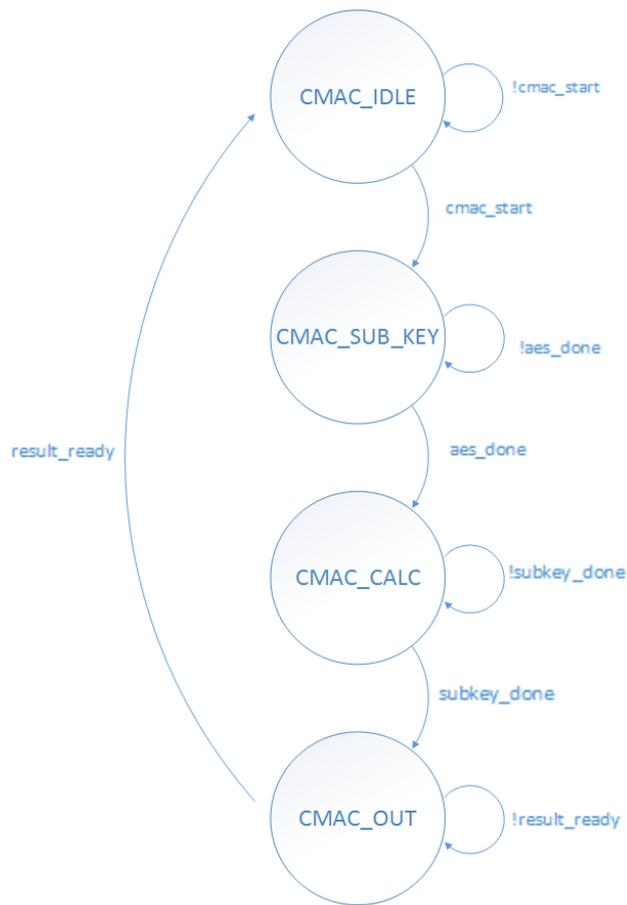


Figure 57 - *cmac* finite state machine

From the above *cmac* finite state machine figure, we can understand how the module is following the AES-CMAC algorithm steps described in *rfc4493*: the first step is the sub-key generation algorithm, which creates the two keys K1 and K2 (*cmac_sub_key* state).

After that, the *cmac* core algorithm is implemented; the output is given on a 128bit bus together with the *result_ready* pin (see Figure 56a).

6.3. AES Key Wrap module

Another module that instantiates the same *aes_core* and *key_expander* modules as the *cmac* module, is the *key_wrap* module which implements the AES Key Wrap algorithm described in Sec. 3.6.

In the following figure we have the *key_wrap* diagram box.

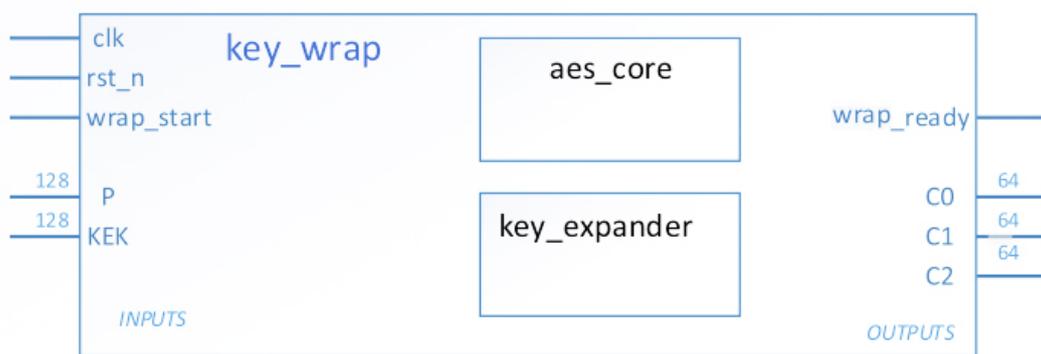


Figure 58 - *key_wrap* block diagram

The output of the wrapping algorithm is given on three buses of 64 bits, as this is the way it is defined in the *rfc3394*. As it will be described in the following *mka* module (Sec. 6.5), these buses will be merged in a bigger register for the needed calculations.

The next figure shows the finite state machine of the *key_wrap* module.

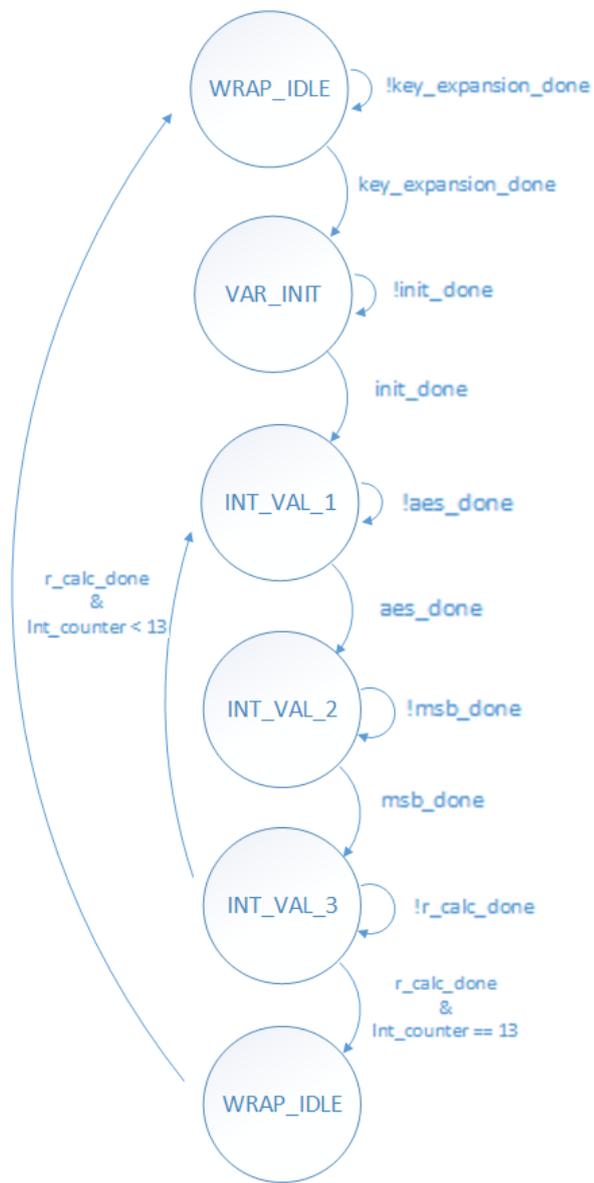


Figure 59 - *key_wrap* finite state machine

6.4. KDF module

The *kdf* module implements the key derivation function described in the 802.1X-2010 standard (see Sec. 4.5.2.1.). It implements the *cmac* module described in Sec. 6.2 as we can see from the block diagram in the following figure.



Figure 60 - *kdf* block diagram

The output is on a 128bit bus, and a *kdf_ready* pin is present. As described in the MKA Key Hierarchy algorithm, the *kdf* module will be started twice to have the keys ICK and KEK as output.

Here is the *kdf* finite state machine.

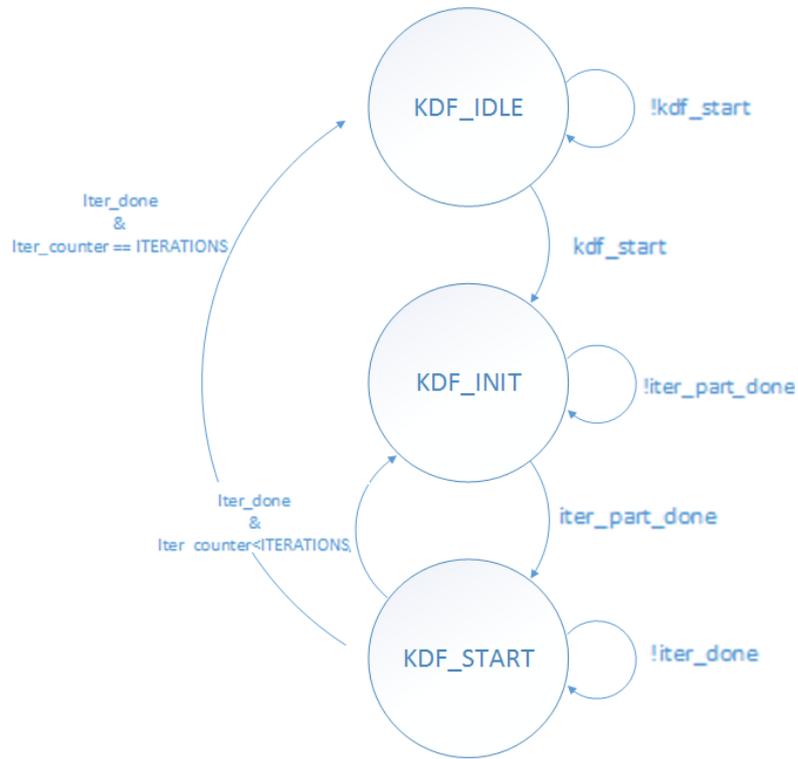
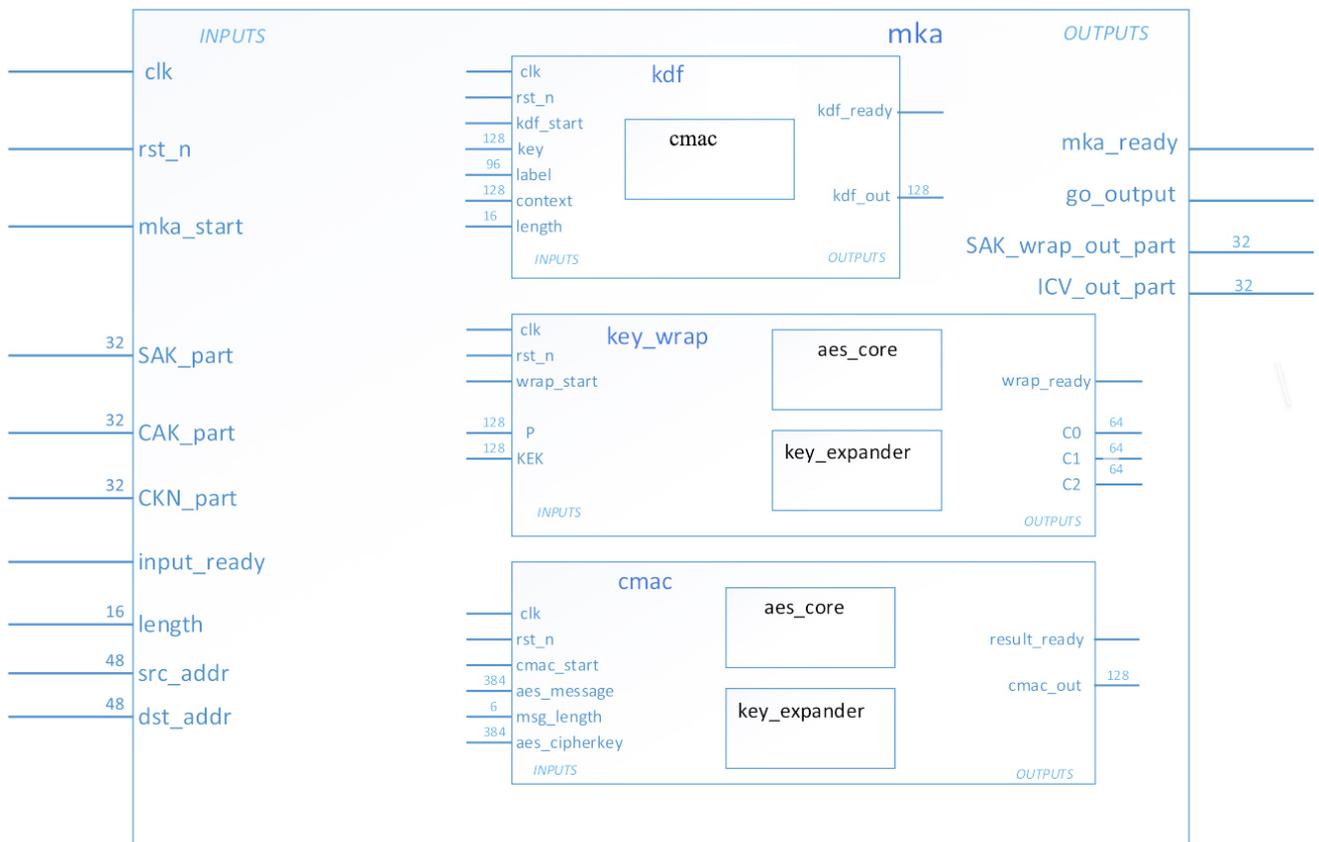


Figure 61 - *kdf* finite state machine

6.5. MKA module

The *mka* module is the top-level module of the MKA Key Hierarchy implementation.

It implements all the previous mentioned modules, as showed in the following figure, taken from the Verilog code.



(a)

```

kdf U1(clk, rst_n, kdf_start, CAK, kdf_label, CKN, length , kdf_ready, kdf_out);
key_wrap U2(clk, rst_n, wrap_start, SAK, buffer_kek, wrap_ready, C0, C1, C2 );
cmac U3(clk,rst_n,cmac_start, cmac_message, cmac_msg_length, buffer_ick, cmac_ready,cmac_out);

```

(b)

Figure 62 - *mka* implementing all the modules: (a) block diagram, (b) Verilog code

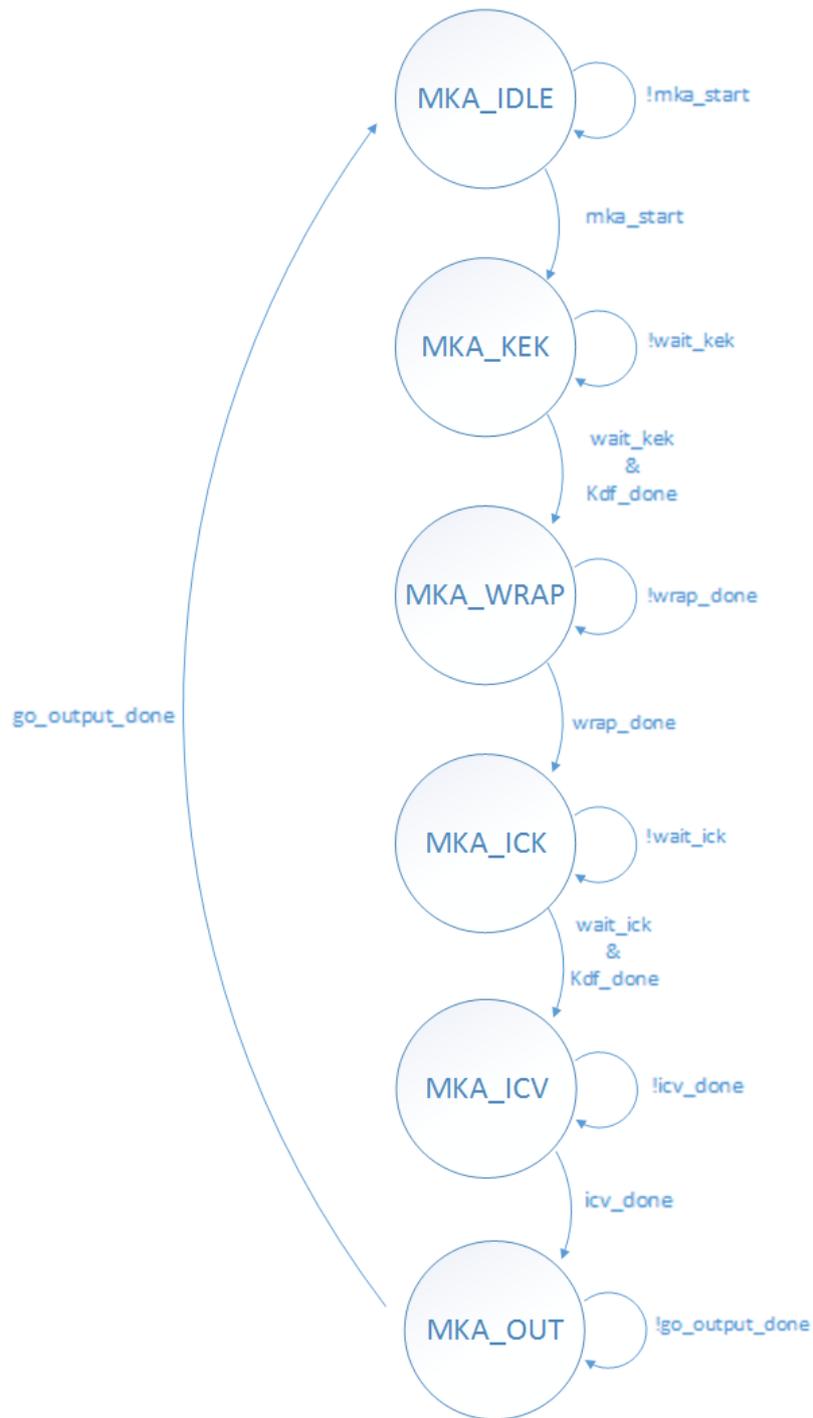


Figure 63 - *mka* finite state machine

The above *mka* finite state machine reflects the algorithmic steps of the MKA Key hierarchy protocol.

The *mka* module receives the SAK serially on a 32bit bus: every received 32bit chunks is stored in a 128bit register. This holds also for the CKN and CAK inputs.

When all the three internal registers, linked to the three input keys, are full (that is each of the four chunks of every key has been stored), the *input_ready* pin is set and the internal state change to MKA_KEK.

This state is responsible if the creation of the KEK: the *kdf* module is involved giving it the correct kek label as input.

With the obtained KEK the Key Wrap can be applied to the SAK, which is stored in the internal SAK register.

After, the ICK is derived and then the related ICV. All the results are stored in internal registers; at the end, when all the wanted values are ready, the output is ready and is sending in series on a 32bit bus.

In the following table are showed the relative CPU clock cycles for each module during the *mka* module test bench.

Module	Clock cycles
cmac	100
key_wrap	246
kdf	95
mka	525

Figure 64 - modules performance in clock cycles

As visible from the above table the most expensive in time is the *key_wrap* module: to notice that the *mka* module instantiates all the other modules, in particular twice the *kdf* module.

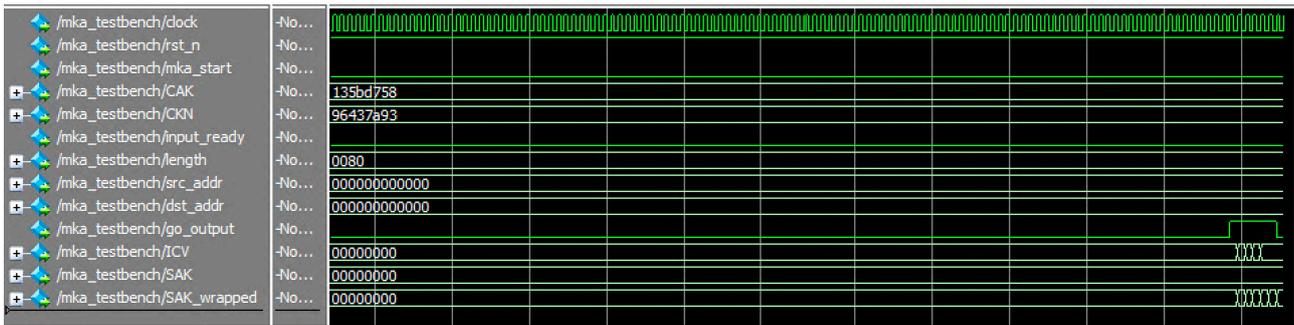


Figure 65 - mka wave plot

6.6. FPGA Synthesis

The implemented system, described in Sec. 5, has been synthesized both on a FPGA Stratix V of Altera and on a 65 nm standard-cell ASIC technology, to verify the respect of the design constraints (i.e. support of the maximum frequency of 125MHz) and document the statistics relevant to the area consumption. Even if the final target is the realization of an ASIC device, the synthesis on FPGA have been necessary to confirm that the implemented system was able to be used on the FPGA demo board.

The TX MACsec and RX MACsec modules have been synthesized on the FPGA Stratix V 5SGXMABK3H40C4 of Altera, using the Altera software Quartus II (version 14.1). The selected device is an high performance and high size FPGA realized through the 28-nm TSMC process technology and has logic cores supplied with 0.9 V or 0.85 V. The programmable logic cores are called ALM (adaptive logic module) and they implement LUT-based logic functions. Each ALM contains a variety of LUT-based resources that can be divided between two combinational adaptive LUTs (ALUTs) and four registers, as it is depicted in Figure 66.

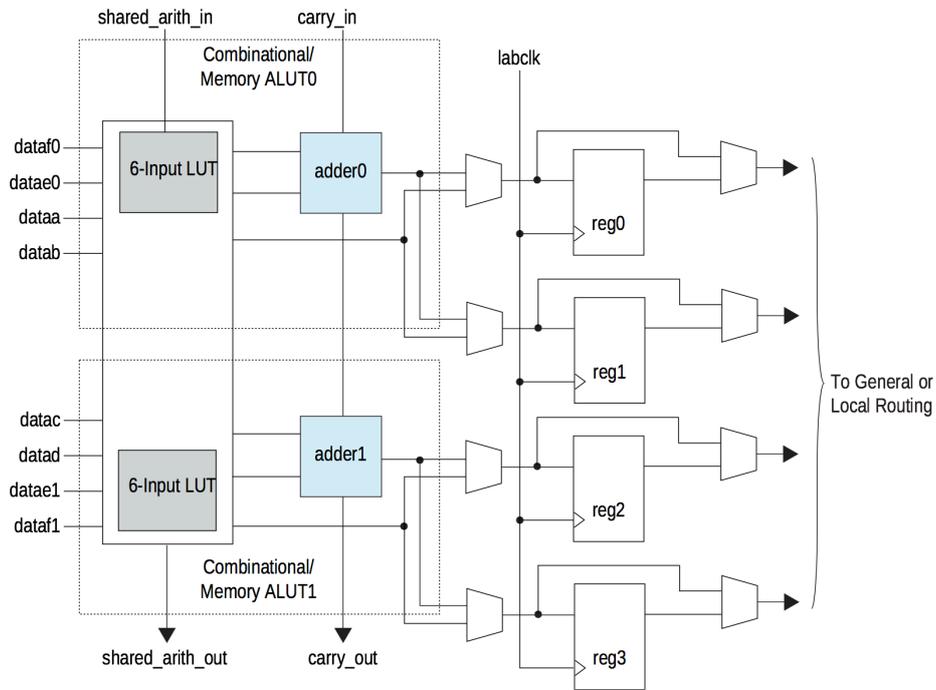


Figure 66 - ALM high-level block diagram for Stratix V devices

Furthermore a Stratix V device is provided with embedded block dedicated to specific functionalities, as DSP blocks or M20K memory blocks.

For the synthesis they have been specified the following constraints:

- clock period = 8 ns (corresponding to the frequency of 125 MHz);

The synthesis results are reported in the following table.

Compiler Optimization mode	Logic utilization (ALMs)	Total Registers	Total pins	Max Frequency @ 85C
BALANCED	11477/359200 (3%)	9938	278/864	135,26 MHz

Figure 67 - FPGA synthesis results

6.7. Synthesis on standard-cell ASIC technology

The *mka* module has been synthesized also on a 65 nm standard-cell technology, using again a clock period of 8 ns as constraint: the following table reports the statistics related to the area occupation (in kgate) and to the frequency that the module can support.

Module	Frequency	Area occupancy (kgate)
mka	125 MHz	102,38

Figure 68 – standard-cell ASIC technology synthesis

7. Conclusions

This work entered Renesas project flow, which goal is to secure the next generation Ethernet networks that will replace the heterogeneous network automotive environment present nowadays.

The realized system has been integrated as the above layer of the MACsec message exchange protocol developed by Renesas' security team.

Compliant with IEEE 802.1X-2010 standard, it is able to supply the essential security keying material for the whole MACsec protocol, which will add security services to the automotive area Ethernet networks.

The synthesized modules fit the company's requirements in order of area occupancy and latency.

All these aspects makes the realized system a good entry point for the security requirements of the automotive field and to create a MACsec module which is fully compliant with the IEEE 802.1AE standard.

Bibliography

- ⁱ Kerstin Lemke · Christof Paar · Marko Wolf (Eds.) - Embedded Security in Cars
- ⁱⁱ Ethernet Backbone in Car: Hype or Reality?
http://www.eetimes.com/document.asp?doc_id=1319157
- ⁱⁱⁱ NIST - Recommendation for Block Cipher Modes of Operation
<http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>
- ^{iv} Advanced Encryption Standard
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- ^v NIST - Advanced Encryption Standard (AES)
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- ^{vi} JH. Song, R. Poovendran, J. Lee, T. Iwata - The AES-CMAC Algorithm
<https://tools.ietf.org/html/rfc4493>
- ^{vii} J. Schaad, R. Housley - Advanced Encryption Standard (AES) Key Wrap Algorithm
<https://www.ietf.org/rfc/rfc3394.txt>
- ^{viii} IEEE Computer Society – Media Access Control (MAC) Security
- ^{ix} Cisco - Identity-Based Networking Services: MAC Security
- ^x IEEE Computer Society - Port-based Network Access Control
- ^{xi} Class Cipher
<http://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html>
- ^{xii} NIST - CMAC test vectors
<http://csrc.nist.gov/groups/STM/cavp/documents/mac/cmactestvectors.zip>

^{xiii} A. Satoh, S. Morioka, K. Takano and S. Munetoh - A Compact Rijndael Hardware Architecture with S-Box Optimization. In: ed. by ASIACRYPT. 2001.

