TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

SCHOOL OF APPLIED TECHNOLOGY

DEPARTMENT OF INFORMATICS ENGINEERING

AN EMBEDDED PLATFORM FOR DEVELOPING DATA PROTECTION PROTOCOLS ON SMART VEHICLES

Author: Vougioukalos Giannis
Major Professor: Grammatikakis Miltos

09/2018

Abstract

In this thesis we present an automotive engineering platform that allows designing and experimenting with security solutions for automotive networks, and more specifically CAN bus. The hardware implementation of the platform is based on Arduino Uno with the addition of the DFRobot CAN bus shield v2 which integrates an MCP2515 CAN bus controller chip and MCP2551 CAN transceiver chip.

On top of this platform new protocols have been developed that support authentication, data integrity and strong defense against replay and masquerade attacks. More specifically, our open source protocol, called vatiCAN-G, extends Vetted Authenticated CAN (namely, vatiCAN protocol), designed by Stefan Nurn berger and Christian Rossow in 2016 [40], towards supporting on-the-fly secure group communication. The major components and functions used are detailed in this thesis.

The platform allows further experimentation to design large scale systems to examine scalability and energy overhead of our solution. It may also integrate other nodes, such as ECUsim 2000, to create realistic scenarios by connecting "or exposing" Engine Control Units, sensors and actuators on the CAN bus.

Περίληψη

Η εργασία αφορά την ανάπτυξη ενσωματωμένης πλατφόρμας που επιτρέπει τη μελέτη πρωτοκόλλων ασφαλούς προσπέλασης δεδομένων σε συστήματα που ελέγχουν κρίσιμες παραμέτρους λειτουργίας έξυπνων οχημάτων.

Στα πλαίσια της παρούσας εργασίας υλοποιήθηκε μια πλατφόρμα που προσφέρεται για πειραματισμό σε θέματα in-vehicle security. Επίσης παράλληλα προτάθηκε και μελετήθηκε μια λύση που εξασφαλίζει αυθεντικοποίηση, ακεραιότητα δεδομένων και προστασία απο επιθέσεις επανάληψης (replay) και πλαστοπροσωπίας (masquerade). Η προτεινόμενη λύση που ονομάζεται vatiCAN-G προσφέρει επιπροσθέτως ασφαλή επικοινωνία σε ομάδες κόμβων και βασίζεται στο vatiCAN (Vetted Authenticated, CAN bus) που προτάθηκε από τους Stefan Nurnberger και Christian Rossow το 2016. Το υλικό και οι συναρτήσεις επεξηγούνται περαιτέρω στα κεφάλαια που ακολουθούν.

Η υλοποίηση της πλατφόρμας βασίζεται σε Arduino Uno με επεκτάσεις DFRobots CAN bus shield V2 που παρέχουν ενσωματωμένα τσιπ α) MCP2515 που λειτουργεί σαν CAN controller (διεπαφή με το CAN δίκτυο σε επίπεδο data link) και β) MCP2551 που λειτουργεί σαν transceiver (διεπαφή με φυσικό επίπεδο). Η πλατφόρμα είναι επεκτάσιμη με άλλες συσκευές όπως το Ecusim 2000 για τη δημιουργία ρεαλιστικών σεναρίων, και προσφέρεται για τη μελέτη της επεκτασιμότητας και ενεργειακής κατανάλωσης πρωτοκόλλων ασφάλειας σε δίκτυα που αφορούν έξυπνα οχήματα.

Table of Contents

Index of Figures

Acknowledgments

Firstly, I would like to thank my family for supporting me all these years.

Moreover, I would like to say that, the experience and the opportunities I had in the Artificial Intelligence and System Engineering Laboratory (AISE Lab) cannot be underestimated. I would specifically like to thank very much my thesis advisor Dr. Miltiadis Grammatikakis for his guidance and patience all this time. Also I would like to thank the members of the AISE Lab: Voula Piperaki, Nikos Mouzakitis and Stratos Ntalaris for their assistance during my thesis, the help, the support and information they provided me. I would also say that Nikos Mouzakitis helped me in prototyping and validation of the CAN bus security extensions that have been developed on top of the AVR-based automotive engineering platform.Here, I would also say that Nikos Mouzakitis helped for the extended new platform and Voula Piperaki gave a great helping hand for the editing of the thesis.

1. Introduction

In recent years automotive technology has made leaps forward, as modern cars have become more and more connected among them and with other devices for the sake of comfort, efficiency and safety. This varies from the driving experience while driving through icy roads, to passenger entertainment and smoother on-board experience.

Operation of electric steering wheel, airbags, lights and other modern automotive subsystems. rely on more than 70 ECUs (Engine Control Units) with tens of sensors and actuators all of them are connected on the vehicle's network bus, usually a CAN (Controlled Area Network) which is an automotive network standard since 1990s.

Despite the features available to the driver, the smart car is still in its first steps, basically lacking the security features that are largely available for other software [48]. When the CAN bus set sail around thirty years ago, threats were invisible to designers, so the protocol remained without security extensions for many years. As the technology matured, the problems came to the surface and the lack of security was addressed as a visible problem. This made the automotive industry prone to security vulnerabilities.

In Section 1, the CAN bus is presented. Possible threats and attacks, hardware or software in nature, are defined in Section 2. Asymmetric and symmetric cryptographic algorithms are described in Section 3. Section 4 describes the embedded components used in the proposed automotive engineering platform and explains open extensions (including the software API) to support on-the-fly group security. A Section on future work and conclusions is also provided.

2. The CAN Bus

CAN (Controller Area Network) is a serial communication protocol that simplifies installation, reduces wiring, and enables very reliable, real-time data exchange among electronic control units (ECUs), providing standardization of the ECU infrastructure and network. CAN created by Bosch in 1980s is an ensemble of nodes mounted on the bus (see Figure 1). In the 1990s, CAN bus was standardized (e.g. ISO 11898-1, 11898-2, and 11898-3) and automotive industry began manufacturing it. Replacing previously heavily wired networks with a two-wire bus simplifies wiring and seriously reduces the weight of the vehicle. This also helps reduce the fuel consumption as the car becomes lighter.

Apart from sensors and actuators, Engine Control Units (ECUs) are major control components that connect to CAN bus. In upper class vehicle models, there can be more than 70 ECUs connected to multiple buses. True numbers vary between companies. On-board diagnostics (OBD) used by mechanics (and sometimes experienced owners) can search for problems and solutions on the devices connected to the CAN bus, making vehicle maintenance both fast, and reliable [1].

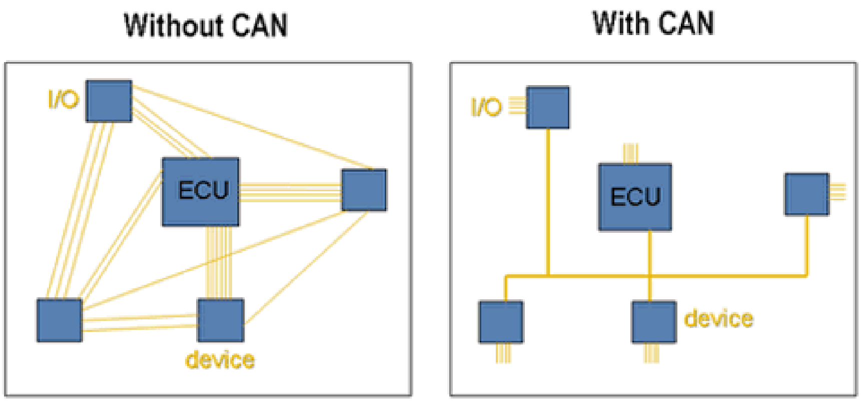


Figure 1: Example of automotive network solutions without CAN and with CAN [1]

2.1 CAN Bus Protocol

CAN protocol defines an asynchronous, event-driven prioritized communication protocol based on two OSI layers: Physical Layer specifies data rates from 125 Kbit/s to 1Mbit/s, and Data Link Layer [2].

More specifically, at the physical layer, as shown in Figure 2, the bus consists of two wires, CAN High and CAN Low. The maximum distance of the bus is 40 meters. There are two different physical layer protocols available: CAN high (specified in ISO 11898-2 protocol) with speeds up to 1Mb/s, and CAN low (specified in ISO 11898-3) which reaches a speed up to 125 Kb/s and offers higher fault tolerance.

Higher data rates are useful for safety-critical applications in powertrain and vehicle chassis areas. With CAN protocol, a CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) policy is used. Hence, although any node has the right to access the bus, at the end of the arbitration phase, only the higher priority CAN node (the one with the lowest ID) is authorized by its interface (CAN controller and transceiver) to broadcast a message to the bus. Thus, for high bus loads, CAN protocol can cause increased delay for less critical, lower-priority CAN messages. Upon message transmission, CAN nodes with lower priority messages switch to the receiving state to listen to the broadcast message 50.

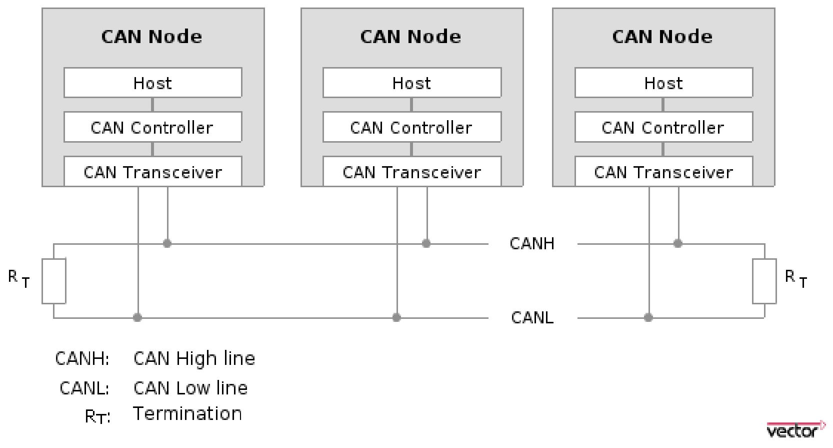


Figure 2: ECU Layers

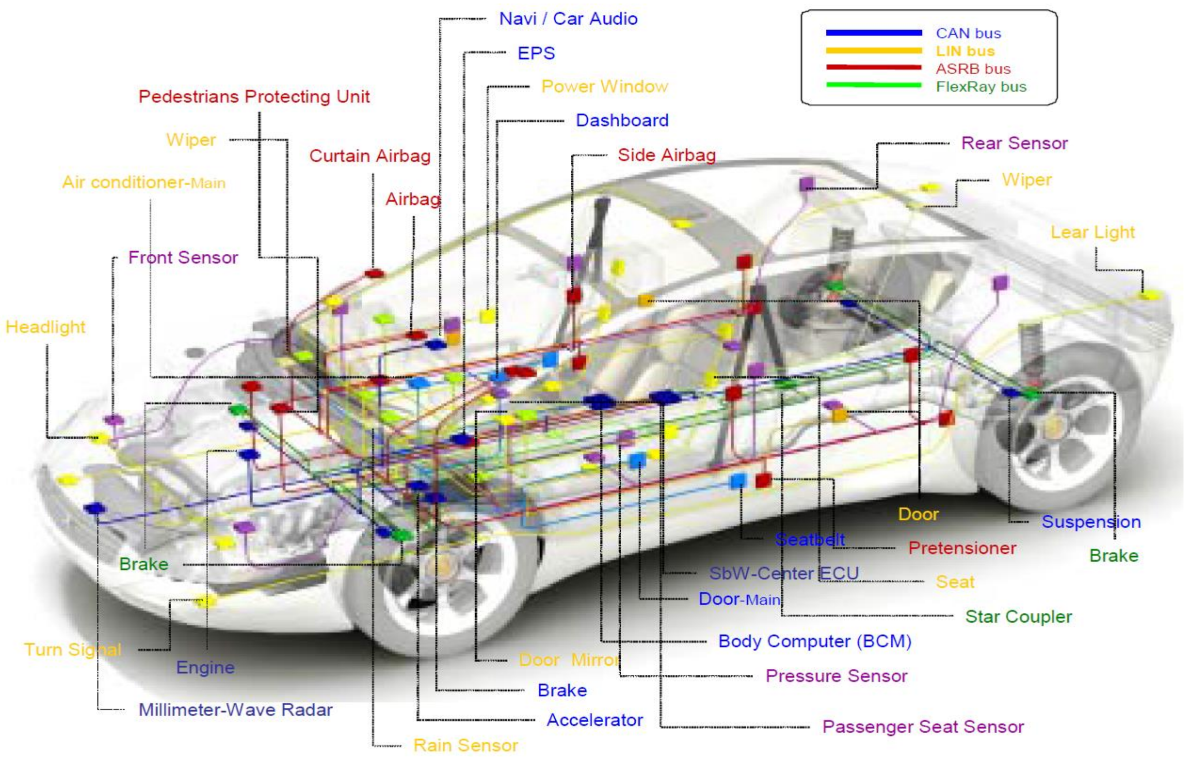


Figure 3: The car network [3]

ECUs connected to the bus (see Figure 3), are used to send information to each other in order to monitor and control the function of the car. ECUs importance varies from a secondary role, e.g. related to road plan statistics, to ensuring safety of passengers in the ECU steering or ABS. Each ECU has a transceiver that connects the node to the physical layer of the CAN, with one connection to the CAN high wire, and one to the CAN low wire. The controller implements the data link layer (ISO 11898-1) that is responsible for sending and receiving packets (so-called frames) to/from the CAN network.

2.2 Bit Arbitration

CAN nodes do not send their messages to other nodes of interest using point to point communication [1]. In contrast, they transmit their messages to all nodes connected on the bus. Only the nodes that are interested on the message react. The CAN bus protocol is based on bitwise arbitration to prevent collisions that may happen from concurrent access to the CAN bus. CAN nodes use specific IDs to communicate. When multiple nodes broadcast simultaneously only the node with the smallest CAN ID gets priority to send its message to the bus ("0" has highest priority), thus one node sends at the time. The nodes with lower priorities try to broadcast as soon as the network is available. This way all messages may eventually be transmitted, while higher priority messages are favored for real-time communication.

For example, if node 1 with ID 0010 and node 2 with ID 1010 try to broadcast at the same time on the CAN bus, the node with the smallest ID, in this case node 1, will broadcast first. Node 2 will have to wait and listen, until the bus is available for transmission again.

2.3 Frames

The messages used by the CAN bus protocol are named CAN frames [4][5] There are four different kind of frames available according to frame size.

Data Frames

The Data messages has two types: The standard or base frame format has 11 identifier bits, while the Extended frame format has 29 identifiers bits.

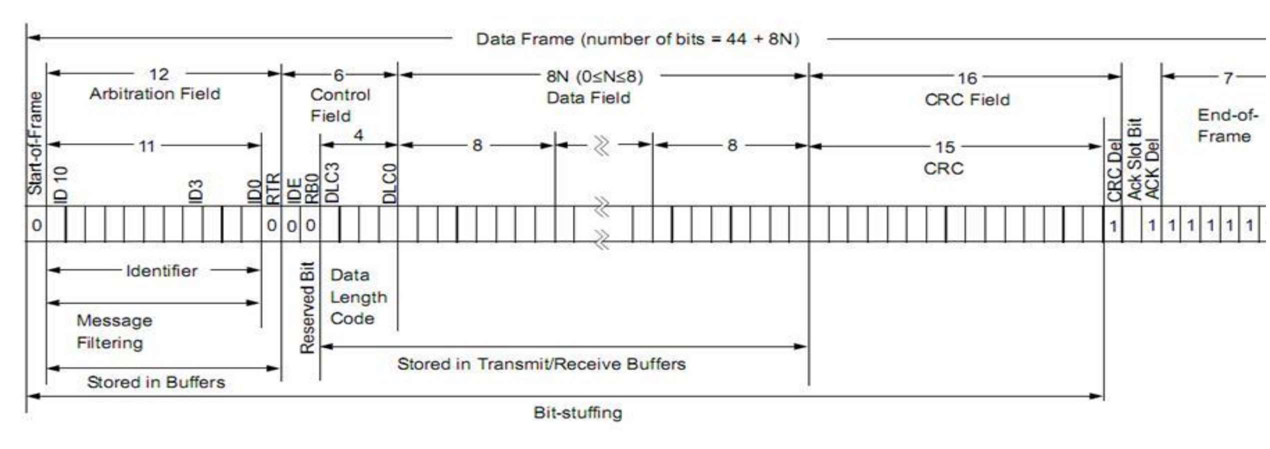


Figure 4: The Standard Data Frame

Data Frame: The most common CAN Frame is the Data Frame and this is divided to two versions with small bit changes but different size. The Standard Data (see in Figure 4) Frame (CAN 2.0A) and the Extended Data (see Figure 5) Frame (CAN 2.0B).

- The first bit of the CAN 2.0 A and B is zero to show that a broadcast began as the Interframe State of the CAN bus is logical one.
- The next 11 bits are the identifier.
- The Remote Transmission Request (RTR) is a bit whose value is zero in case of a Data Frame and one in case of a *Remote Frame* .
- The next part of this is the Identifier Extension (IDE) bit that is zero in case of a Standard Frame.
- The following bit Reserved Bit Zero (RB0) is also zero.

- After that comes a four bit representation of the size of the data field named Data Length Code (DLC).
- Data Length that can be up to 8 bytes and are the data transmitted.
- Cyclic Redundancy Check (CRC) is used to identify errors that may occurred during the broadcast and is 15 bits long [2]
- CRC Delimiter that always has the value of one.
- Acknowledgment Slot (ACK Slot) bit has the value one when a node receives a Data Frame with no mistakes. The ACK Slot takes the value zero if there is any errors.
- At the end of the frame there are 7 logical one bits.

➢ Extended vs Standard Data Frame

The major difference between the Extended and the Standard Data Frame is the size. The larger version is **20** bits more than the other and the main reason is the Identifier as it is 29 bits. The purpose for this is the need of the automotive companies to have universally unique identifiers their products even if there are not unique at their function. So, in this Frame the arbitration field is 32 bits instead of 12. After the 11-bit Identifier that exists in both versions the Extended version uses the Substitute Remote Request (SRR) bit that has the value of one. IDE comes next as logical one. Finally, we have the 18-bit part of the Extended Identifier.

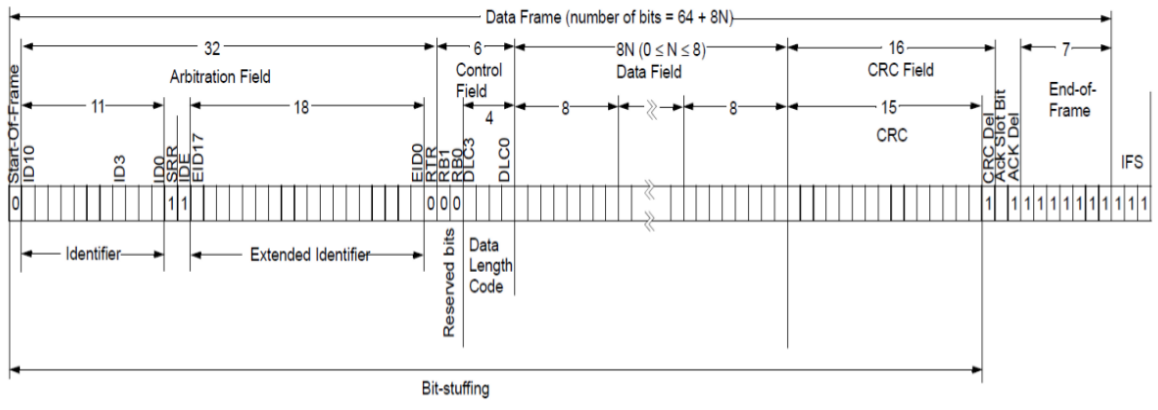


Figure 5: Extended Data Frame

2.3.1 Remote Frame

Remote Frames (see Figure 6) are used when a node needs information from another node. The data field of this frame is empty.

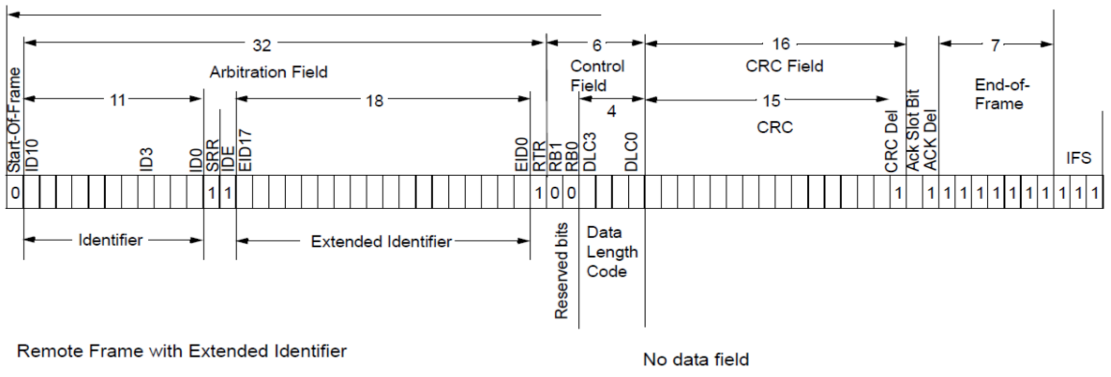


Figure 6: Remote Frame

2.3.2 Error Frame

As shown in Figure 7, nodes broadcast an Error Frame whenever an error is spotted on the bus. When more than 5 bits in the row (non-return-to-zero) have the same value. In this case the other nodes recognize the error and broadcast an Error Frame with the field of the Error Flag being filled with 8 zeros and as a result the transmission stops.

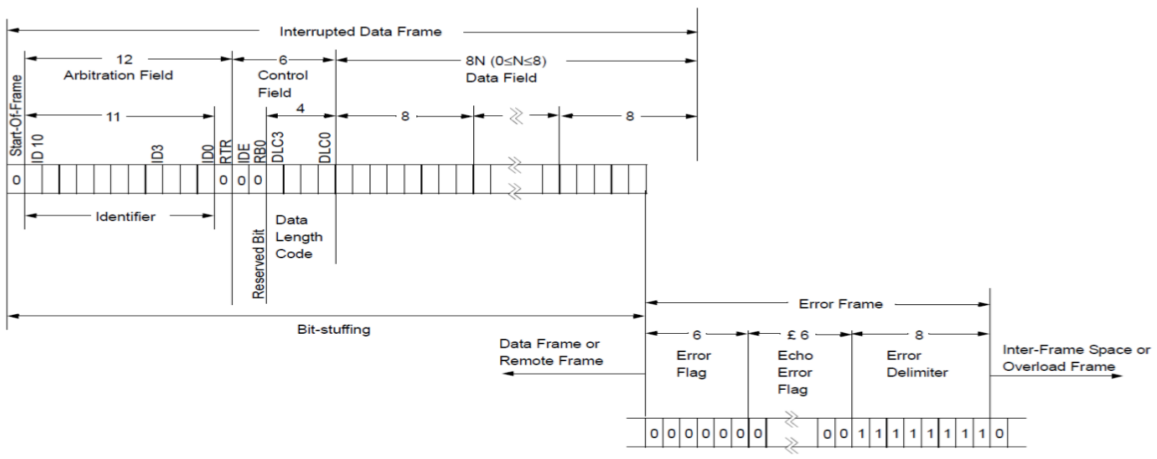


Figure 7: Error Frame

2.3.3 Overload Frame

Overload Frames, shown in Figure 8, are transmitted during the Interframe State. They are similar to Error Frames, but transmission of the frame does not stop.

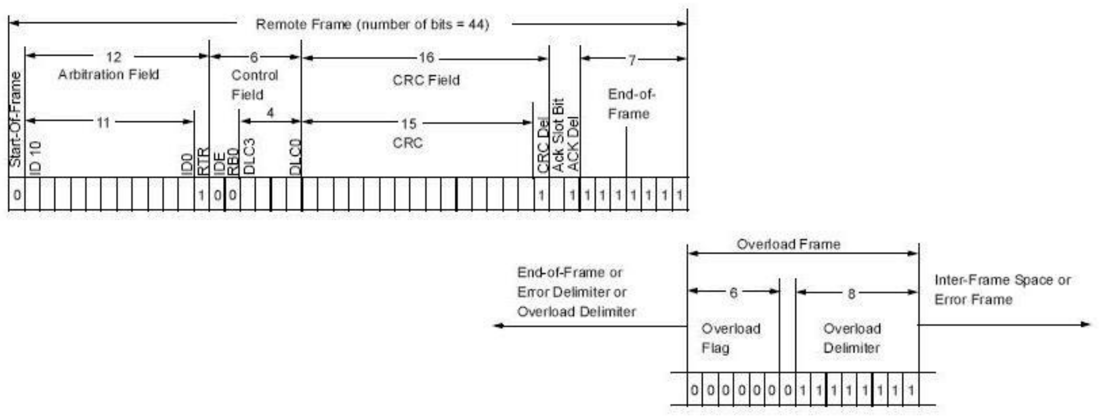


Figure 8: Overload Frame

Besides CAN bus, some other in-vehicle network technologies are available with each one dedicated to a different purpose.

- Media Oriented Systems Transport (MOST) is an optical fiber bus used for media transmission, such as audio, video, voice [6]. It spans all the OSI layers and offers bandwidth up to 150Mbp/s. Cars that use at least one MOST Network are Audi, BMW, General Motors, Honda, Hyundai, Jaguar, Lancia, Land Rover, Mercedes-Benz, Porsche, Toyota, Volkswagen, SAAB, SKODA, SEAT and Volvo.
- Local Interconnect Network (LIN) is a cost efficient, single wire bus on the vehicle that is used for connecting sensors and actuators [7]. LIN is a single master bus and each master holds up to 16 slave nodes. Speeds are limited to 19.2 Kbp/s.
- Flexray is designed as a fast bus for subnetworks with high priority [8]. It offers speeds up to 10Mbp/s, but is not widely available due to the high cost of implementation.

These networks also do not offer any security. However, since CAN remains the single most important bus used in almost all modern cars, it is the network of choice examined in this thesis.

3. Threats and Attacks

Increased level of sophistication enhances vehicle functionality, but possibly leads to increased vulnerability. Lima et al. have identified different types of automotive security weaknesses: reading or modifying *control-sensitive sensor data*, compromising *internal,* *external, or coordinated global network communications* (e.g. CAN bus, short- and long-range wireless), exploiting OS *software vulnerabilities by embedding malicious code which possibly modifies data on vehicle ECUs and road-side units*, bypassing *authentication mechanisms* from trusted authorities, e.g. during firmware update, and *physical-layer attacks* [2].

In this section we focus on some types of threats that target vehicle network communications.

3.1 Masquerade (or spoofing) attacks

Masquerade attacks happen when an attacker imitates the identity of a legitimate user or component in a network. Some specific spoofing attacks are: E-mail address spoofing, GPS spoofing, and ID address Spoofing [9]. Another spoofing attack can consider the DNS spoofing. In this case a malicious node spoofs the IP address of another user, without his knowledge and requests an address from a DNS server. The DNS server replies to the victim's server. The malevolent queries are small and chosen to need much bigger replies, all of which are being sent to the victim's server. This causes the server to be unable to handle the traffic [10].

A masquerade (or spoofing) attack in CAN Bus means that an attacker sends a fraudulent CAN message which misleads other nodes on its identity, thereby causing malicious message data to be accepted by the system.

3.2 Replay or Playback attacks

A replay attack takes place when data extracted and sent again by a harmful third party posing as a reliable member of the conversation [11].

For example, in a CAN network a perpetrator retransmits earlier data in the system, e.g., a message it has received from the CAN network, thereby misleading the system on the current value of a physical signal. While pair-wise keys can help protect from masquerade attack, they are not effective for replay attacks, i.e. a node may accept a message which it should reject (false acceptance).

Denial of Service attack

A Denial of Service (DoS) attack focuses on making a service unavailable. This type of violation takes place on a network (such as LAN), in a web service, in a device (attacks to a CPU or cache) etc. A DoS attack is created from a machine, in contrast to DDoS attacks that are caused from many machines as described in next subsection [13].

Researchers examine a DoS attack on the hardware is the HDD DoS attack. The attack was accomplished by the resonance produced by sound waves, causing the examined devices (personal computer and a CCTV circuit) to stop working and restart. The CCTV permanently lost its data [16].

3.3 Distributed Denial of Service attack

Distributed Denial of Servive (DDoS) attack is a strike to the victim's server coming from multiple sources. The attacking devices such as smartphones, IoT and computers which have been maliciously compromised and teamed up against the victim by the attack shepherd and they are called a Botnet (see Figure 9). Botnets are also increasingly available for rent by companies who offer DDoS attack services [12].

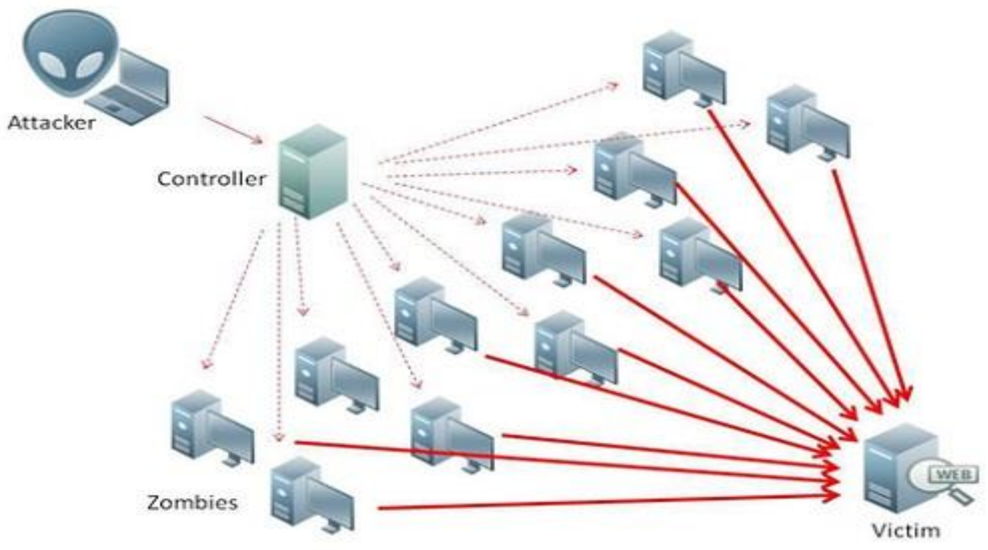


Figure 9: DDoS attacks [13]

3.4 SYN Flood

The SYN flood attacks is like a denial-of- service attacks as referred above. It works with two types of attacks the TCP SYN flood attacks that uses the TCP protocol and the SYN Flood attacks. A TCP SYN Flood is achieved by a mischievous user taking advantage of TCP's three-way-handshake. As shown in Figure 10, in the three-Layer handshake three steps take place: Firstly, a client sends a SYN message to the server, next the server rejects (if is not available at the

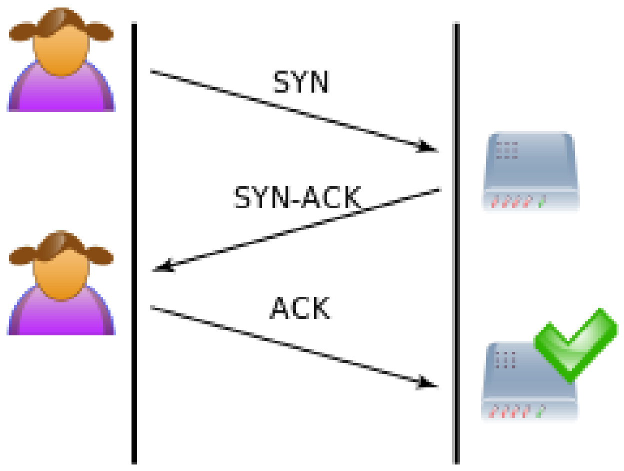


Figure 10: Three-way handshake

time) or replies with a SYN-ACK message. In the last step, the user sends an ACK message to the server and the connection can start [14].

- As shown in Figure 11, in the SYN flood scenario the client sends a request to the server. When the server replies with the SYN acknowledgement, the attacker does not send the last ACK to the server. This causes the server to spend its resources waiting the reply from the user until a connection time-out [15].

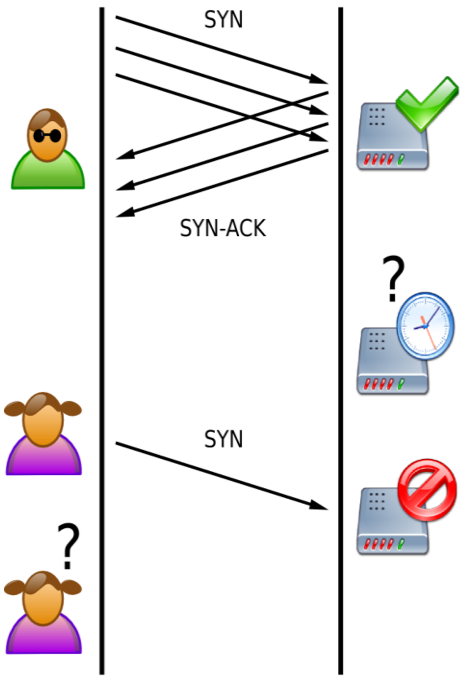


Figure 11: SYN-TCP flood attack

3.5 UDP Flood attack

UDP flood attacks are less common than TCP Flood Attacks. This technique occurs by sending multiple UDP packets to various ports of the victim's computer. The server watches and sees that no application is responding, thus sends ICMP packets to every UDP requested by the attacker. During this time legitimate users cannot communicate with the server as the ports are occupied.

3.6 HTTP Flood Attack

HTTP Flood Attack divides in two ways, HTTP GET flood and HTTP POST flood. HTTP GET flood takes place while the assailant downloads large files from the victim's server with high frequency, as a result legitimate user cannot be served. HTTP POST attack occurs when the server requests to repeatedly fetch results from the database, making the server unavailable.

4. Cryptography

Cryptography is the actions taken to protect a communication between two or more parties. There are many technologies of cryptography. These are: symmetric and asymmetric cryptography.

Basic Terminology

- **Plaintext**: The initial text (without cryptography).

- **Encryption**: Encoding initial text with cryptography algorithms.
- **Ciphertext**: The output of the initial text after encryption.
- **Decryption**: Takes the output text (encrypted text) and converting it via the same algorithm to initial text.

In general, the first step in cryptography is encryption. This is the alteration of readable plaintext to something undefined, that looks random. This output is named ciphertext. The reverse process is called decryption and is used to turn the encrypted information back to plaintext again when it is considered safe. Both procedures need the same cryptography key to take place.

4.1 Asymmetric (or public key) Cryptography

In asymmetric cryptography each member of a communication has a unique public and private key. The public key of each user is visible and potentially known to all users while the private key is only known to its owner. This solves the problem of symmetric cryptography (referred in next subsection) where the key is common and must be distributed to all involved users securely prior to the data exchange. This was not optimal as it is restrictive while secure channels are not available, or the cost can be inefficient.

Despite that the public key is known to others, it's not making the asymmetric encryption vulnerable because there is and the private key. The mathematical relation between the two keys shall be indistinguishable as the system relies on that and the secrecy of the private key. As shown in Figure 12, when a communication starts a pair of keys is created from each party. In this case (see Figure 13), Bob wants to send a secure message (Hello Alice!) to Alice. Hence, via specific algorithms and with the Alice Public Key encoding the message and send it to Alice. Next, Alice decrypts the message with her private key.

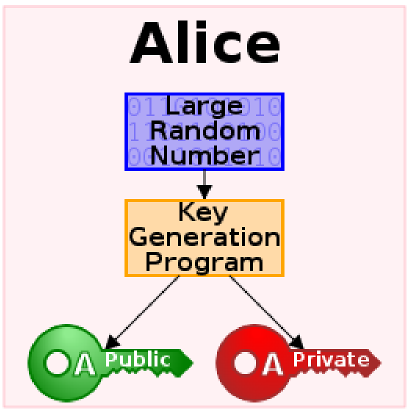


Figure 12: Alice pair of keys

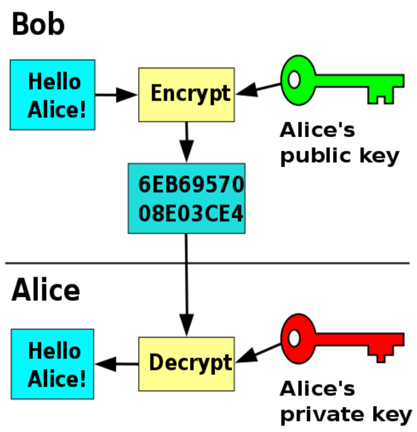


Figure 13: Asymmetric Cryptography

4.2 Symmetric Cryptography

As shown in Figure 14, symmetric cryptography uses a shared secret key (same shared key) between two or more parties for data encryption and decryption.

The communication using symmetric cryptography is as follows: A sender wants to send a message to another person. The sender encrypts the information to be transmitted with the shared key and then transmits it. Receiver uses the same shared key to decrypt the message. After this procedure the receiver can see the original information. Symmetric algorithms are divided in two categories, block and stream ciphers.

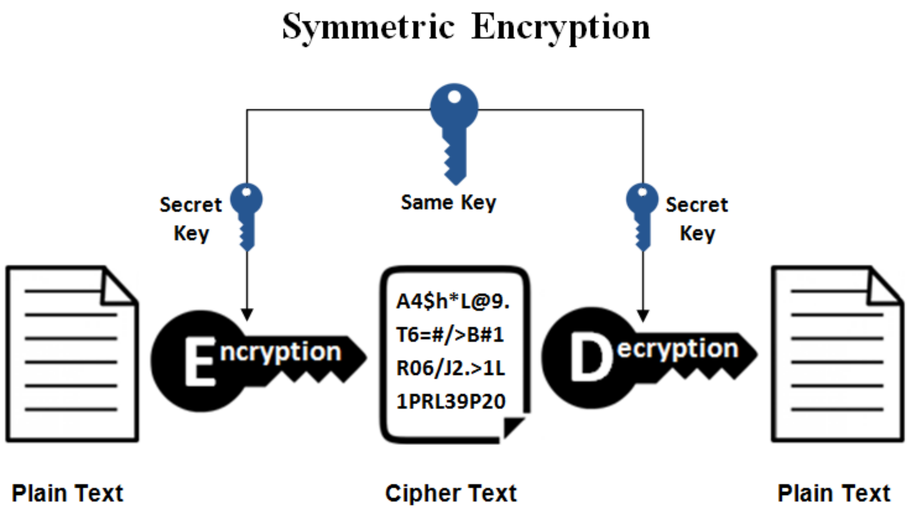


Figure 14: Symmetric Key Encryption

4.3 Block Ciphers

Block ciphers splits the data to be encrypted in predefined n-sized volumes of bytes using a n-size cryptography key. More specifically, Block ciphers take as an input a message and divide it in n-byte blocks which is multiple times smaller than the original message. Each block is manipulated in a way that along with their cryptographic key will produce a ciphertext, according to the mode of operation used. There are two main categories of Block cipher: the Feistel Cipher and the Modes of operations (ECB, CBC, CTR, GSM etc.).

4.3.1 Feistel Cipher

Feistel Cipher is the foundation for many block ciphers [17][18]. Encryption and decryption are achieved by splitting the plaintext or the ciphertext block in a left (L) and a right part (R). The following steps take place for the cipher's implementation for encryption (see Figure 15):

- An encryption function f is created based on a subkey (Ki) and R part, (f (Ki,Ri)). The L part XOR f become the input for R in the next step. The R part is kept as it is and becomes the L part of the next step.

$L_i + 1 = R_i$

$R_i + 1 = L_i$ XOR $f(K_i, Ri)$

- The subkeys used in each step are related to the key, but there are not the key itself. Another subkey is used in each step.
- In the last step of the encryption, the Ln+1 and Rn+1 parts are joined in one piece.

For the Feistel decryption, the scheme follows the same philosophy.

- The process begins with the ciphertext, split again in two L and R pieces, starting from Ln+1 and Rn+1 and each subtracts 1 in each step.
- The subkey Ki starts from Kn going down by 1 each round, until it reaches K0.

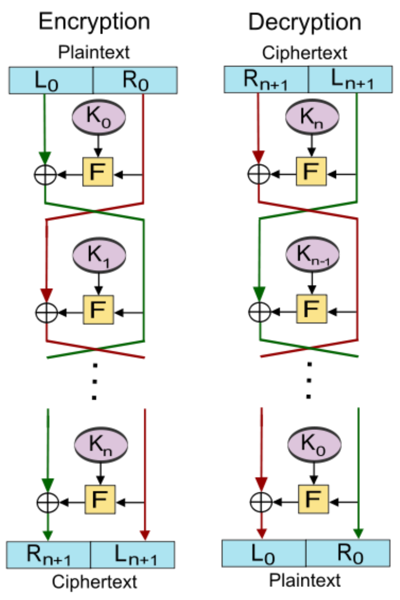


Figure 15: Feistel cipher encryption and decryption

4.3.2 Modes of Operation

As shown in Figure 16, **ECB (Electronic Code Book)** encrypts the plaintext in combination with key [19]. The outcome is the ciphertext produced. Each block is isolated and independent from all the others. The decryption (see Figure 17) follows the same approach, the ciphertext combined with the key produces the original plaintext. In ECB a plaintext that are exactly the same will produce the same ciphertexts. Repetition is not recommended therefore ECB is not ideal for type of cryptography for large files, accordingly so is being used for small size data.

ECB Encryption: $C_i = E_k(P_i)$.

ECB Decryption: $M_i = D_k(C_i)$.

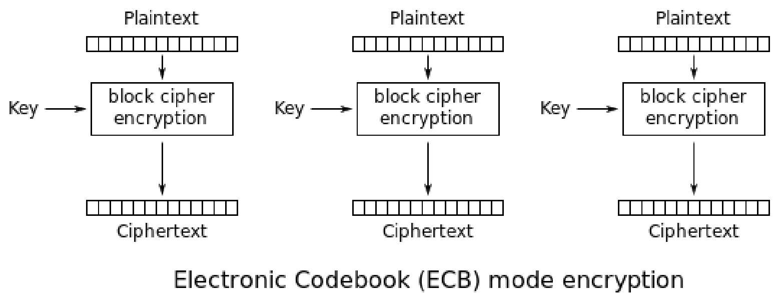


Figure 16: ECB Encryption

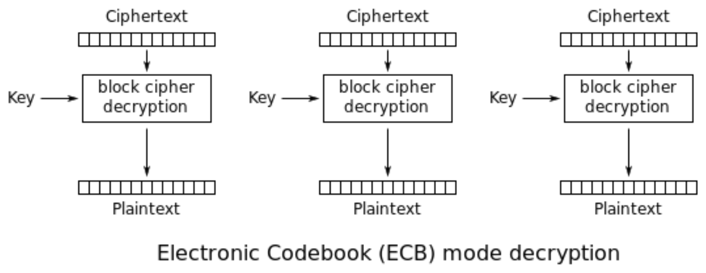


Figure 17: ECB Decryption

CBC (Cipher Block Chaining) provides confidentiality. In this procedure a ciphertext is XORed with the following plaintext. The first plaintext is XORed with an Initializing Vector (IV) due to the lack of a prior ciphertext. The outcome of the XOR is encrypted using the cryptographic key and this produces the next ciphertext.

CBC Encryption (see Figure 18):

$C_i = E_k\ (M_i \text{ XOR } C_{i-1})$

$C_0 = VI$

CBC Decryption (see Figure 19):

$M_i = D_k\ (C_i) \text{ XOR } C_{i-1}$

$C_0 = IV$

If any mistake occurs during the encryption of C_i would make the C_i block to be corrupted. Due to the function of CBC decryption thought the corruption would affect only M_i and M_{i+1}.

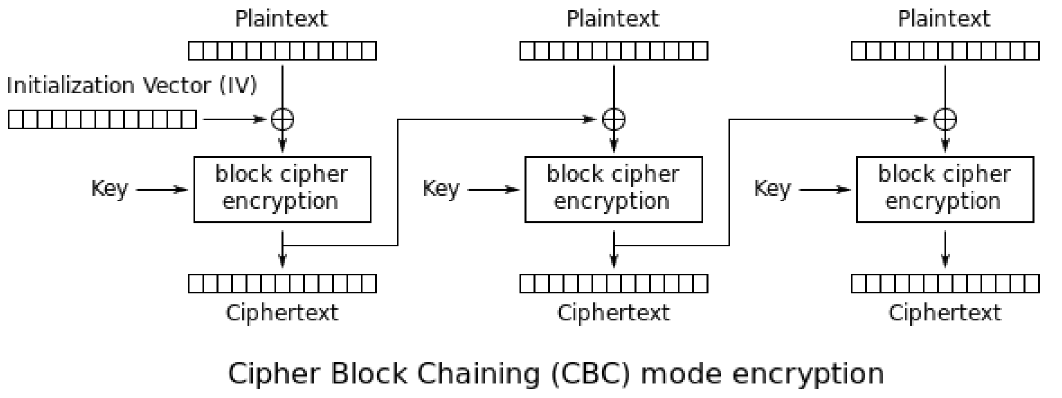


Figure 18: CBC Encryption

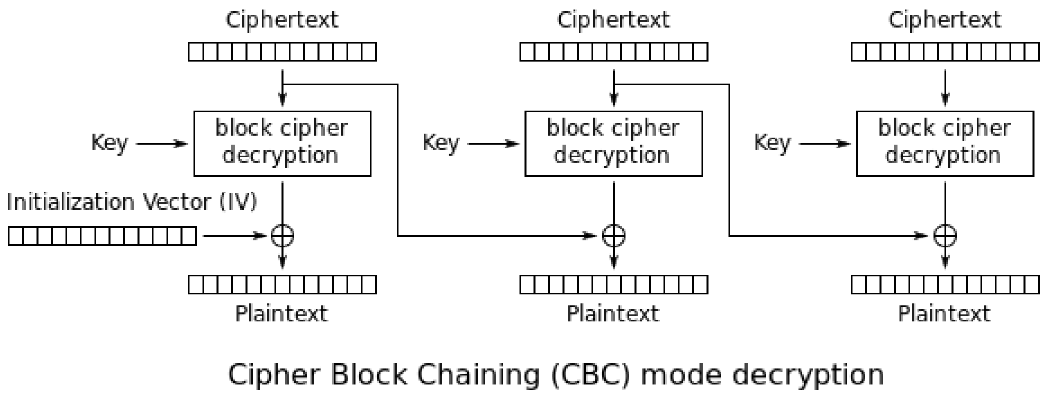


Figure 19: CBC Decryption

Counter Mode (CTR) is a confidentiality mode that a series of input blocks have stored a nonce (random value) in the first part along with a n-size incrementing counter in their second part. This is encrypted together with the key and the result is XORed with th plaintext. On this mode of operation is that encryption and decryption can occur in parallel, as in Figure 20, 21.

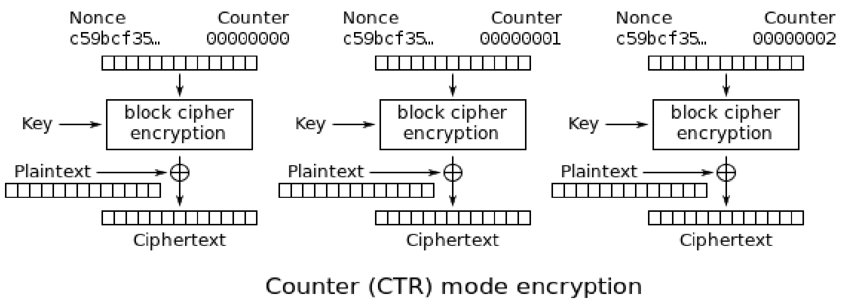


Figure 20: CTR encryption

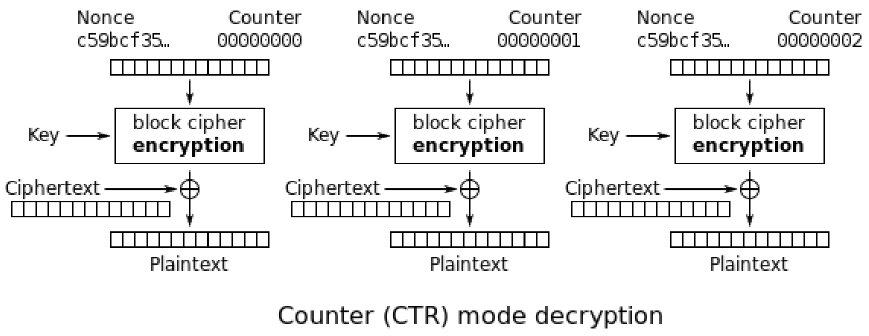


Figure 21: CTR decryption

Galois Counter Mode (GCM) is a block cipher of 128 bits, that provides authenticity and confidentiality. Similar to CTR GCM, it uses an IV and an incrementing counter. The IV is hashed along with the key and XORed with the plaintext and this has the ciphertext as a result. To ensure data integrity and authentication, the GCM makes use of the Galois Message Authentication Code (GMAC). The output of the first ciphertext along with 128 zeros that have been hashed, are used on an operation called multiplication and the result is XORed with the next block's ciphertext to produce the next multiplication block. The last multiplication is XORed with the ciphertext's length and the length of the additional data [20].

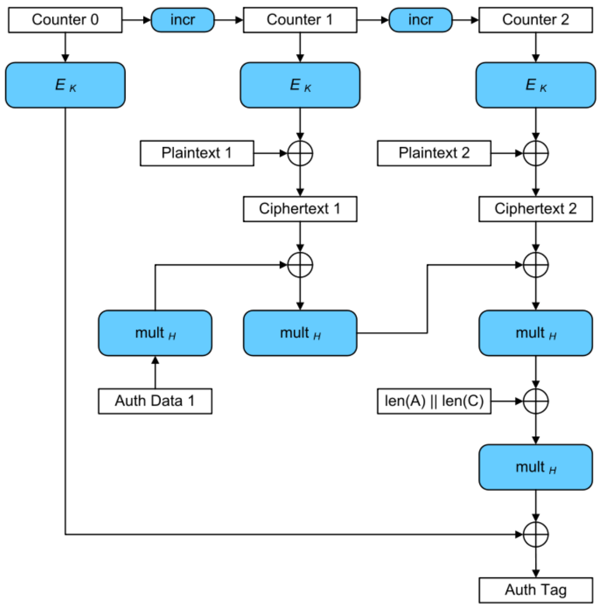


Figure 22: GCM

4.3.2.1 Advanced Encryption Standard (AES)

AES is a block cipher that is being widely used for network and storage security, that was chosen by National Institution of Standards Technology (NIST) after a contest for cryptographic algorithms for a new, secure and symmetric algorithm standard with the name AES [22][23] The algorithm is named Rijndael by the creators [24]. AES came to replace Data Encryption Standard (DES) which was the previous standard. DES at the time was not able to be as effective with the technological progress of the time. In 1998 a DES cracker was already made and was able to brute force the algorithm in a few days.

The requirements for the new algorithm were:

1) To be symmetric
2) The size of the key should be at least 128 bits, with the option to support 192- and 256-bit keys
3) To be able to provide security against the known attacks
4) To be cost and power efficient

AES key size is 128, 192 or 256 bits denoted N_k and specifies the number of columns of the key matrix, extended versions of AES have since been created with the key to reach up to 1024 bits size [21]. The possible number of columns are 4, 6 or 8 in both matrices to the 128-, 192- or 256-bit values subsequently. Accordingly, to the key and the block size the Rijndael rounds that take place are 10, 12 or 14, as in Figure 23. Both block and key sizes are multiple of 32 and can have different sizes as they are is no relation between them. The plaintext is arranged in a 4 rows * N_b column, column major order and represented as binary or decimal or hexadecimal values. The same principal applies for the key matrix with Nk columns. If there are remaining, plane cells in the matrix are padded to fill the gaps and complete the 4 * Number of columns * 8 bits of the matrix.

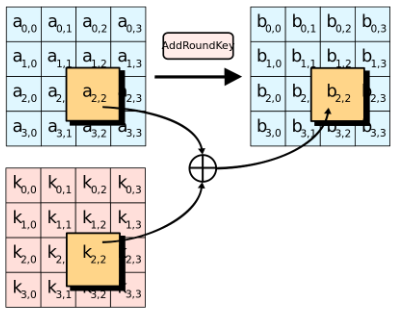
N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

Figure 23: AES Rounds

The size of the original cipher key is 4 * Number of rows. The size of the expanded key = the original cipher key * (original cipher key + 1)

Prior to the first round and in the end of each round, a subkey is used. The subkey is originates from the original cipher key, its size is equal as the state's block size and a different subkey is used in each round. The subkey is XORed byte by byte with the state block.

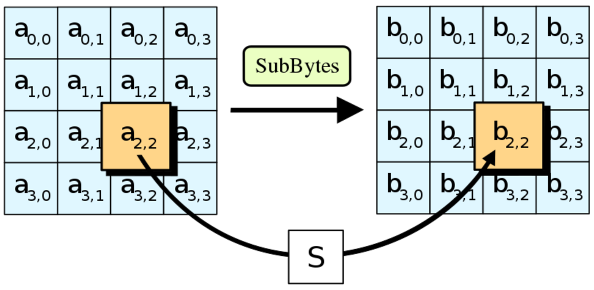
Prior to the first round and in the end of each round, a round key is used (see Figure 24). The round key originates from an extended cipher key. The extended key is equal with the original key size * (original key size + 1). In each round a different round key is used. The round key size is equal as the state's block size. The subkey is XORed byte by byte with the state block.



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
AddRoundKey
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$

Figure 24: Add Round Key

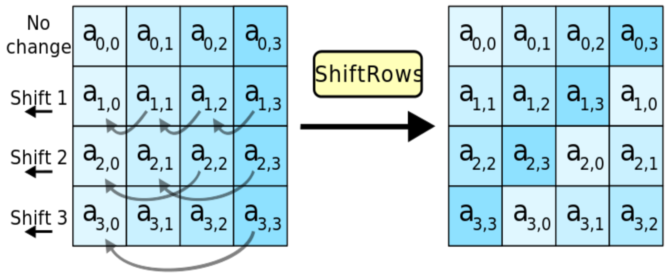
The Sub bytes step of each round in the AES cryptography corresponds to confusion. This is a byte by byte operation, as in Figure 25 that an input of the x byte will become S(x) = y, after the byte passes through the S-box.



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
S
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 25: Sub bytes operation

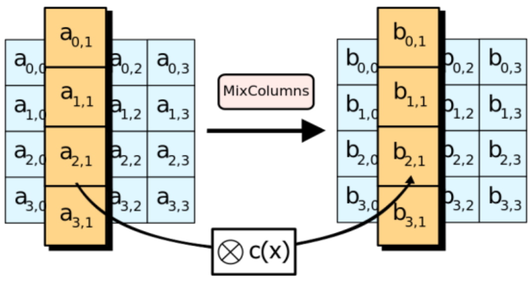
In the next step, the shift row operation the values of the matrix are being shifted left according to the number of the row they are in (see Figure 26), starting to count from row 0. Shifting rows adds diffusion to AES.



No change
Shift 1
Shift 2
Shift 3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Figure 26: Shift row

As shown in Figure 27, the Mix Column is used after the S-Box operation and Shift Row whereas both contribute to the cipher's diffusion. This operation is a transformation that takes each column and change its value by a matrix multiplication.



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes c(x)$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 27: Mix Column

4.3.3 Stream Ciphers

Stream ciphers are one of two categories of the symmetric cryptographic algorithms. In contrast to the Block ciphers who take a portion of the plaintext each time, manipulate it and produce a ciphertext. Stream ciphers, receive and encrypt the messages, character by character using pseudorandom bits XORed with the plaintext for this purpose. Block ciphers are more widely used on the Internet but due to the small computational power required by the Stream ciphers, they are popular among smaller devices e.g. embedded devices and mobile phones, also are generally faster than Block ciphers. In cases of continuous streams of bytes where errors can occur during encryption Stream ciphers are preferable due to increased fault tolerance. Although the Stream ciphers are considered less secure in comparison to Block ciphers, their advantages still make them useful and preferable for certain applications [25].

Two kind of Stream ciphers exist. The first kind is synchronous Stream ciphers, in this case the keystream is based on the key. The second case is the self-syncronizing Stream ciphers where the keystream also relies on the cipher text. To encrypt the message the keystream is XORed with the plaintext and produces the ciphertext. The same function creates the same plaintext out of the ciphertext XORed with the keystream if the same key as the encryption is used.

Encryption: $y_i = e_{si} (x_i) \equiv x_i + s_i \bmod 2$

Decryption: $x_i = d_{si} (y_i) \equiv y_i + s_i \bmod 2$

4.3.3.1 Salsa20

Salsa 20 has emerged at the eSTREAM program [27][28], which was an attempt to stimulate Stream cipher encryption, organized by the European Network of Excellence in Cryptography (ECRYPT). The eSTREAM program focuses in two profiles: Profile 1 for ciphers based on software applications with bigger requirements and profile 2 that points to cryptographic ciphers for devices with limited resources. Salsa20 belongs to profile 1 ciphers and while typically operates with 20 rounds, a 12-round edition Salsa20/12 and a 8 round edition Salsa20/8 exists. The reduced rounds versions of the algorithm exist and are some of the faster stream ciphers. Two key sizes exist, one with 128 bits and one with 256 bits [26]. As for August 2018 there are not any attack recorded against Salsa20/20 or Salsa20/12. Salsa20 produce a ciphertext by XORing the same number of bits each time from a plaintext and a keystream. The same procedure follows for the decryption.

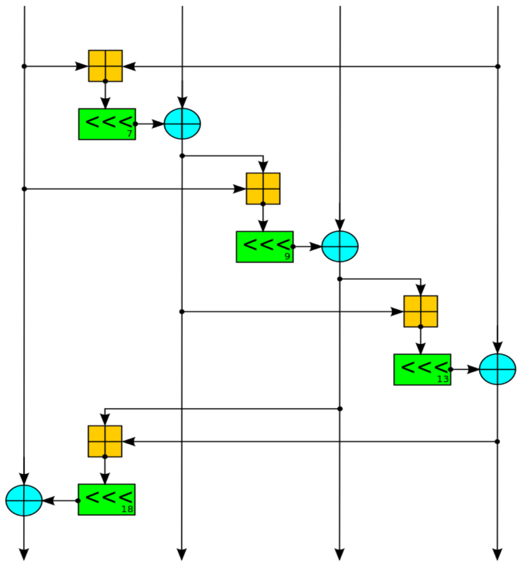


Figure 28: Salsa20

Inputs and outputs

If y is a 4-word sequence then quarterround(y) is a 4-word sequence.

Definition

If $y = (y_0, y_1, y_2, y_3)$ then quarterround(y) = (z_0, z_1, z_2, z_3) where

$z_1 = y_1 \oplus ((y_0 + y_3) <<< 7)$,

$z_2 = y_2 \oplus ((z_1 + y_0) <<< 9)$,

$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13)$,

$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18)$. [27]

4.3.4 Cryptographic Hash Function

Hash Functions are used to verify message and data integrity among users. Sites with content available for download offer their data's hash value for the users to compare the downloaded file's value with the genuine hash. Some hashing algorithms are outdated, and they no longer provide enough security because the computational power of modern computers surpassed the limits of the older algorithms (e.g. MD5). This makes the older algorithms not trustworthy with hash breakers being available online and don't require any skills for the user to crack a hash value [29]. A combination of a hash function and Message Authentication Code (MAC) creates the Hash based MAC (HMAC) that verifies the sender of a message and security against reply attacks.

A hash algorithm takes as an input data called message, of any size and turn them to a fixed smaller size hash value. The output also named digest. The ideal hash functions follows some principles.

1. The original data must be untraceable for someone who only possesses the hash value.
2. Every digest ideally must correspond to only one input. It be probabilistically impossible for two inputs to have the same hash value. (In the real-world scenario of a collision there are solutions available.)
3. Different inputs will produce different outputs, even the smallest intercept on a file would produce another hash value due to the avalanche effect. As a result, hash algorithms also provide data integrity [30].
4. The same input will always produce the same digest.
5. The functions must be fast.

The Secure Hash Algorithm family consists from SHA-1, SHA-2 and SHA-3 which is also named Keccak. The input size depends to the SHA of choice.

SHA-1 is one of the algorithms that are considered insecure and federal agencies are instructed by NIST not to use them [31]. Artificial collisions were created by Centrum Wiskunde & Informatica (CWI) and Google when they succeed to create the same digest from two different files [32]. SHA-2 and Keccak are the current algorithms in use. SHA-2 produces 224, 256, 384 for 512 bit digest size and has 80 rounds. Keccak has the same variety and value of outputs as SHA-2, is based on the sponge construction and uses 24 rounds and is the algorithm that will be used in the platform created in this thesis. The collision resistance (cr) is depended on the n-bit Keccak used each time, cr = $2^{n/2}$, the preimage resistance is 2^n and the capacity is 2n. Keccak offers the possibility to lower the capacity, which lowers the security levels and as a tradeoff boosts performance.

5. Platform

5.1 Board Project Boards

In this work, two main boards are used: Arduino Uno and DFRobot CAN Bus Shield v2.0, as discussed below. Moreover, another board Ecusim 2000 has been tested and is referenced below, but not yet integrated in our security scenarios.

5.1.1 Arduino Uno

Arduino (shown in Figure 29) is an open source board family that have been designed as open source and imitated by several other companies. It is widely used by people who

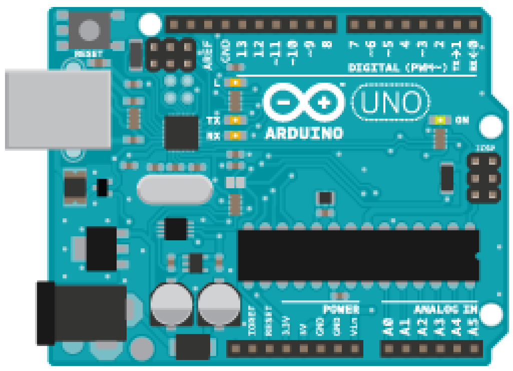


Figure 29: Arduino Uno

develop their own prototypes, e.g. in education, research, industrial automation, etc. An Arduino is relatively easy to program, in contrast to microcontroller programming in the past. As many other projects, the Arduino Uno is the backbone of this project. Arduino Uno is an inexpensive, simple and highly configurable microcontroller that facilitates different shields that mount to Arduino pins to provide additional functionalities. A microcontroller is a small 16-bit computer consuming much less computational power, and in little need of electrical power. Arduino can quickly execute small and simple programs alone, or in combination with an Arduino shield [53][54]

5.1.2 DFRobot CAN Bus Shield 2.0

The DFRobot CAN bus shield (see Figure 30, 31) is used in the proposed platform. It is compatible with Arduino Uno [35], Mega [36] and Leonardo [37] and serves as a CAN bus node able to send and receive data from/to the CAN network using different IDs. The DFRobot Can bus shield uses a common CAN controller MCP2515 and a MCP2551 transceiver chip that allows efficient data exchange. The shield offers two CAN bus channels (connectors), one through the standard DB9 (equivalently, DE9) interface and the second through a pair of CAN high/low twisted wires [38]. In this work, an Arduino library provided by DFRobots has been adapted for sending and receiving messages from Arduino Uno board.

Figure 30: CAN-Bus Shield

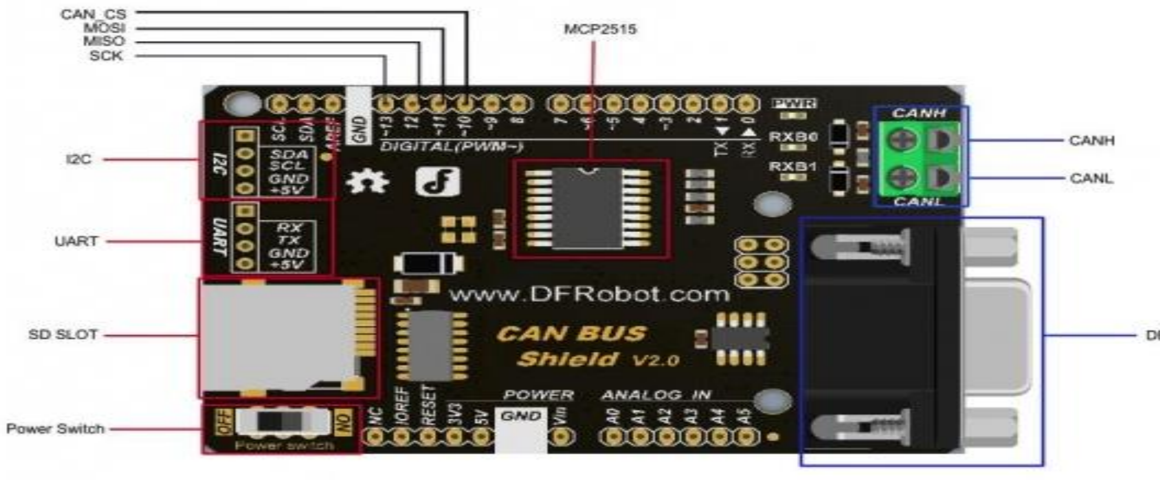


Figure 31: CAN-Bus Overview

5.1.3 Ecusim 2000

Ecusim 2000 (see Figure 32) is a simulator that generates artificial CAN events for testing and development [39]. Ecusim can be either directly connected to a CAN bus or to a PC via a USB to OBD2 cable. The board has 5 potentiometers that imitate certain PIDs and a button to create fault events called Malfunction Indicator Light. The PIDs of the potentiometers, from left to right, correspond to:

1. Coolant Temperature
2. Engine Speed (RPM)
3. Vehicle Speed
4. Oxygen Sensor Voltage
5. Mass Airflow (MAF)

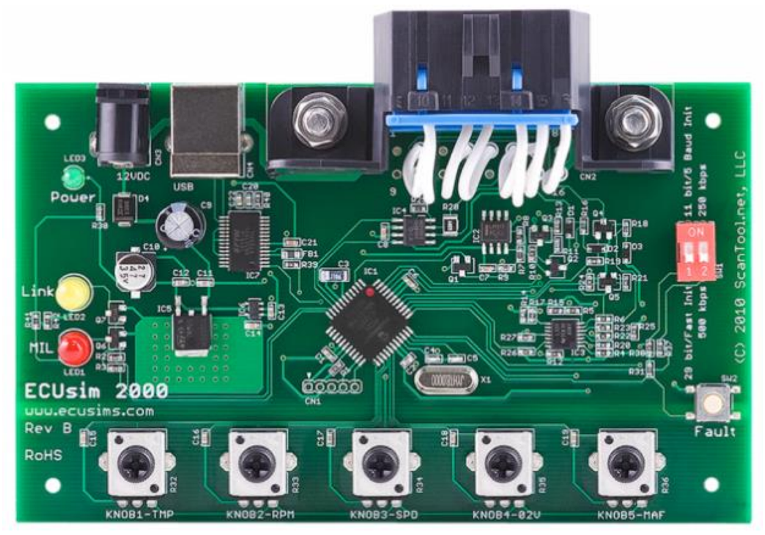


Figure 32: Ecusim 2000

5.2 Sender - Receiver Example

An example of a CAN sender and receiver is shown below. For the implementation we have used two Arduinos with two CAN bus shields which communicate via an OBD2 cable (in the example the Y cable has three ends, only two of them were used), as in Figure 33. Alternatively, they can be connected using the green CAN H/L signal interface, connecting as in Figure 2.

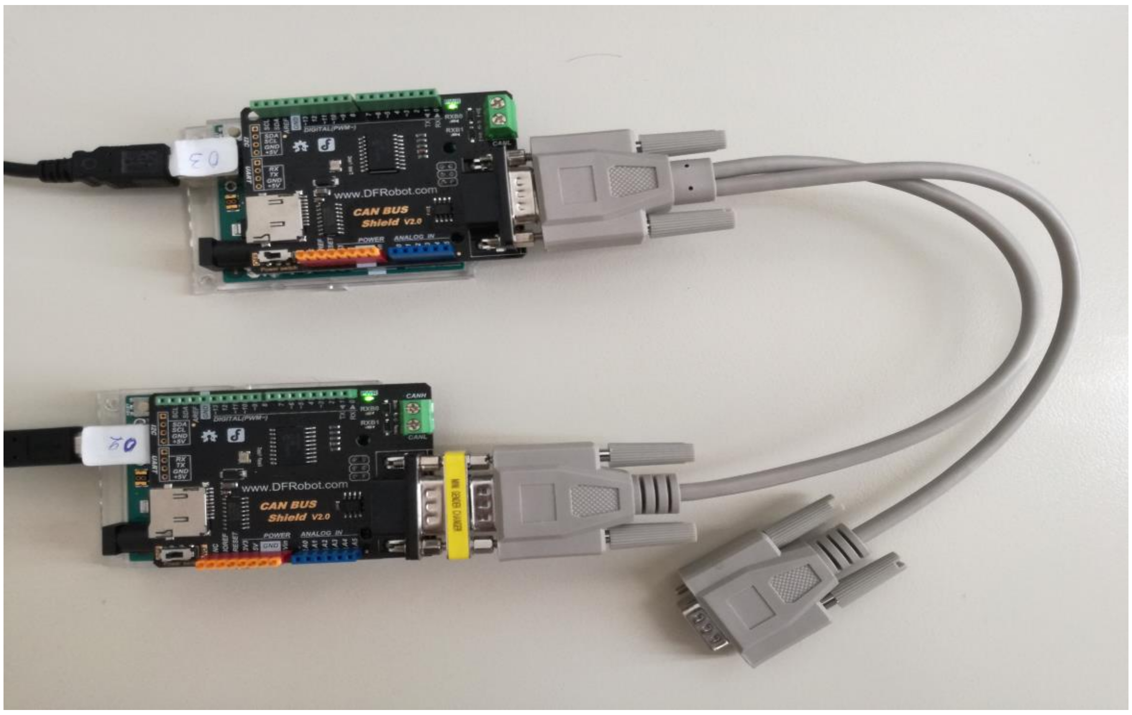


Figure 33: CAN Bus over CAN Shield - Sender, Receiver

5.2.1 Can_sender Node

Can_sender supports the following basic methods:

CAN.init`();`

CAN.begin`(speed);`

CAN.sendMsgBuf`((id, ext, len, buf);`

The method `CAN.init` is used to initialize the CAN bus interface;

The method `CAN.begin` sets the bits per second (baud transfer rate). We use 500Kb/s. A code of snippet is shown below.

```
//init can bus with //baudrate  5ps00kb

if(CAN_OK == CAN.begin(CAN_500KBPS)) {
         Serial.println("DFROBOT's CAN BUS Shield init ok!");
         break;
    } else{
         Serial.println("DFROBOT's CAN BUS Shield init fail");
         Serial.println("Please Init CAN BUS Shield again");
         delay(1000);
```

```
      if (count <= 1)
            Serial.println("Please give up trying!, trying is
useless!");
            }
```

The method `sendMsgBuf` sends data to receiver. This is accomplished by setting a specific receiver ID, the data transmission format, the data length and the data to be transferred.

For example:

```
  can_id  = 0x44;

  unsigned char data[8] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
0x07};

// 0x7e0 worked, 0x7df, 0x18db33f1
CAN.sendMsgBuf(can_id, 0, 8, data);
Serial.println("A... sending message:");
```

5.2.2 Can_receiver Node

Can receiver supports the following basic methods:

```
CAN.init();

CAN.begin(speed);

CAN.checkReceive();

CAN.readMsgBuf(&len, buf);

CAN.getCanId();
```

The methods `CAN.init` and `CAN.begin` are the same as with sender. Notice that both sender, receiver must use the same baud rate (500kbps).

The method `CAN.checkReceive` checks for new messages and saves them in a variable (here an integer variable named *code*). Returns 1 if a frame arrives, and 0 if nothing arrives. For example:

```
int code;
code = CAN.checkReceive();
```

The method `CAN.readMsgBuf` reads the data from the sender after the `CAN.checkReceive` method This is accomplished by setting a specific length and a buffer. For example:

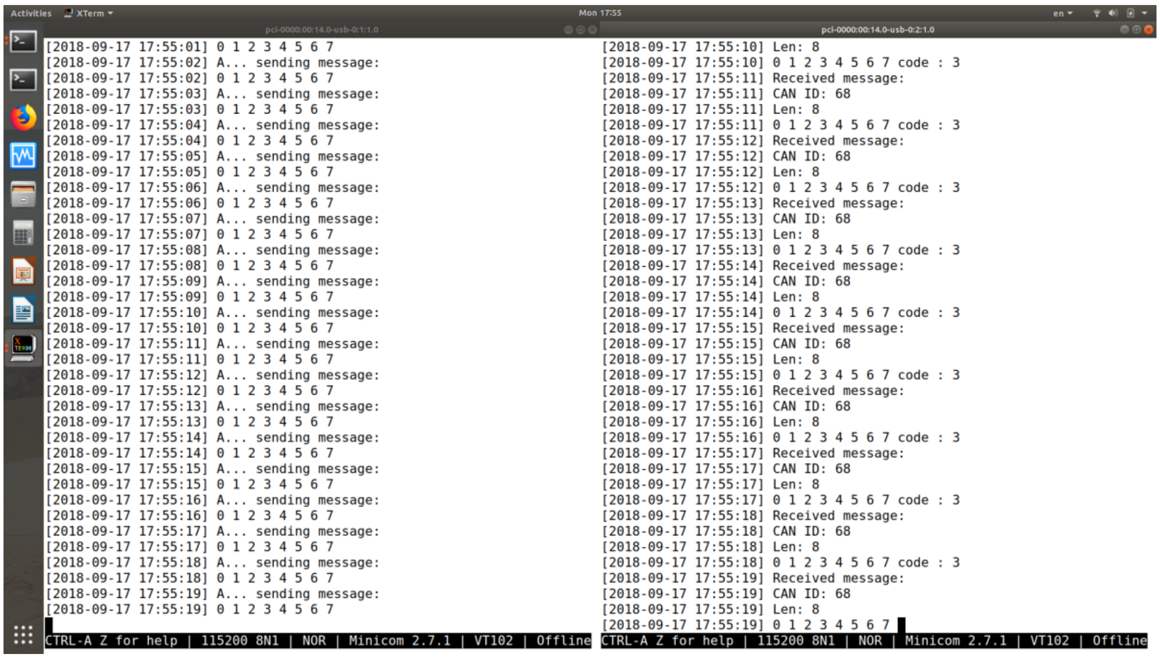
```
unsigned char len=0; //global variable

unsigned char buf[8]; // global variable
```

```
CAN.readMsgBuf(&len, buf);
```

Finally, the method `CAN.getCanId` obtains the value of the sender’s ID if data arrived.

```
can_id = CAN.getCanId();
Serial.print("CAN ID: ");
Serial.println(can_id);
```

As shown in Figure 34 (left part) the sender sends a message “0 1 2 3 4 5 6 7” to receiver. Subsequently, receiver (right part) reads the data with length 8 bytes and the sender CAN id.



```
[2018-09-17 17:55:01] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:02] A... sending message:
[2018-09-17 17:55:02] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:03] A... sending message:
[2018-09-17 17:55:03] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:04] A... sending message:
[2018-09-17 17:55:04] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:05] A... sending message:
[2018-09-17 17:55:05] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:06] A... sending message:
[2018-09-17 17:55:06] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:07] A... sending message:
[2018-09-17 17:55:07] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:08] A... sending message:
[2018-09-17 17:55:08] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:09] A... sending message:
[2018-09-17 17:55:09] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:10] A... sending message:
[2018-09-17 17:55:10] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:11] A... sending message:
[2018-09-17 17:55:11] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:12] A... sending message:
[2018-09-17 17:55:12] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:13] A... sending message:
[2018-09-17 17:55:13] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:14] A... sending message:
[2018-09-17 17:55:14] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:15] A... sending message:
[2018-09-17 17:55:15] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:16] A... sending message:
[2018-09-17 17:55:16] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:17] A... sending message:
[2018-09-17 17:55:17] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:18] A... sending message:
[2018-09-17 17:55:18] 0 1 2 3 4 5 6 7
[2018-09-17 17:55:19] A... sending message:
[2018-09-17 17:55:19] 0 1 2 3 4 5 6 7

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline
```

```
[2018-09-17 17:55:10] Len: 8
[2018-09-17 17:55:10] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:11] Received message:
[2018-09-17 17:55:11] CAN ID: 68
[2018-09-17 17:55:11] Len: 8
[2018-09-17 17:55:11] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:12] Received message:
[2018-09-17 17:55:12] CAN ID: 68
[2018-09-17 17:55:12] Len: 8
[2018-09-17 17:55:12] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:13] Received message:
[2018-09-17 17:55:13] CAN ID: 68
[2018-09-17 17:55:13] Len: 8
[2018-09-17 17:55:13] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:14] Received message:
[2018-09-17 17:55:14] CAN ID: 68
[2018-09-17 17:55:14] Len: 8
[2018-09-17 17:55:14] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:15] Received message:
[2018-09-17 17:55:15] CAN ID: 68
[2018-09-17 17:55:15] Len: 8
[2018-09-17 17:55:15] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:16] Received message:
[2018-09-17 17:55:16] CAN ID: 68
[2018-09-17 17:55:16] Len: 8
[2018-09-17 17:55:16] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:17] Received message:
[2018-09-17 17:55:17] CAN ID: 68
[2018-09-17 17:55:17] Len: 8
[2018-09-17 17:55:17] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:18] Received message:
[2018-09-17 17:55:18] CAN ID: 68
[2018-09-17 17:55:18] Len: 8
[2018-09-17 17:55:18] 0 1 2 3 4 5 6 7 code : 3
[2018-09-17 17:55:19] Received message:
[2018-09-17 17:55:19] CAN ID: 68
[2018-09-17 17:55:19] Len: 8
[2018-09-17 17:55:19] 0 1 2 3 4 5 6 7
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline
```

Figure 34: Sender (left side) and Receiver (right side)

5.3 VatiCAN and vatiCAN-G

In the original vatiCAN protocol [40], secure CAN nodes who communicate via vatiCAN messages are declared statically. Our proposed extension (called **VatiCAN-G**) [46] focuses on supporting dynamic security groups (called *Cliques*). Hence, with our protocol, any communication message, identified by its ID, can be defined on-the-fly, as a secure vatiCAN-G message, Unidentified messages are non-secure, i.e. they are treated as *legacy messages*.

In the case of secure messages, a vatiCAN-G Clique defines a group of CAN nodes who intend to participate next, in a single round of secure communications, i.e., broadcasting authenticated data from one sender to a specific set of receivers. Participants in a vatiCAN-G Clique instance are uniquely identified by an ID. These nodes share a common session key composed by hashing their corresponding secret keys. These secret keys are distributed at initialization time to define a secure clique sometime during system life. As explained in

Section 2, similar to vatiCAN, we protect from masquerade and replay attacks by authenticating payload data using the ID together with the session key associated with this ID, and a global counter (called GRC) which increments based on the number of announced vatiCAN-G Cliques.

5.4 Related work

Nilsson et al. have proposed to calculate a 64-bit MAC (CBC-MAC) over four consecutive CAN messages and transfer the MAC as four 16-bit blocks in the CRC field of the next four CAN-messages [41]. Notice that eight messages are needed for validation. Moreover, if MAC fails, the actual individual message that was wrong cannot be determined, and there is no protection against replay (data integrity).

TESLA (IETF-RFC 4082) is an efficient broadcast authentication protocol employed in wireless sensor networks and recently on CAN bus Unlike all protocols discussed here, the protocol does not provide shared keys, instead a key released in round i, is sent (possible alongside with data of subsequent message) to authenticate the previous message from the same ID [42]. This introduces additional delays unsuitable for many real-time automotive applications, except where high security is needed.

Szilagyi and Koopman have proposed an efficient authentication protocol (called One MAC Per Receiver, OMPR)[43], whereas each pair of CAN nodes shares a secret key (exchanged using one-way hash functions). This key is used to calculate a MAC for CAN messages (64-bit data) exchanged between these two nodes [44]. The MAC (message signature) is reduced to a few bits and is concatenated to the end of the message. Since it is easy to forge a message with only a few bits of the MAC available, the authors propose that authentication is provided by successfully verifying the MAC over a set of messages, making spoofing many messages in a short time period unlikely. In their scheme, a sender only computes as many MACs as its corresponding receivers. The proposed protocol can also protect from replay attacks by including a global clock, together with the pair-wise key and message data, in its MAC computation. Thus, a receiver can check the corresponding receiving counter to see if a message is fresh, However, a global clock, is only available on time-triggered CAN. While voting and TESLA are more appropriate for high assurance systems with large number of receivers and, OMPR and voting are better for low assurance systems with a small number of receivers.

Lin and Sangiovanni-Vincentelli propose an alternative mechanism to protect from masquerade and replay attacks. Their method does not need to maintain a global time, but uses symmetric secret keys and message round counters [45]. Their scheme reduces overheads by transmitting only the least significant bits of counters, and providing reset mechanisms when counters are out of synchronization due to network errors, ECU restarts, or failures. Simulation results from real testbenches prove that their security mechanism achieves sufficient security, without introducing high communication overhead (bus load and message latency).

Nürnberger and Rossow developed Vetted Authenticated CAN (VatiCAN), the first open source solution for authentication and ID/data integrity protection [40]. Their solution establishes a secure way for messaging among critical nodes, while non-protected components continue to communicate using the original CAN messages (called legacy), thus providing backward compatibility. The algorithms resemble the proposed scheme by Lin and Sangiovanni-Vincentelli in computing the MAC, i.e. as function of the packet body, pair-wise shared key and counter specific to each sender (called global nonce). However, instead of distributing the MAC over multiple messages, it is propagated in a second message that validates the previously transmitted data, so that receivers can authenticate the source of the message in a subsequent step (similar to how TESLA treats the keys).

5.5 VatiCAN-G API

Our protocol enhances security by supporting separate 32-bit MAC for group mask and 64-bit MAC for data, compared to 64-bit MAC for data in vatiCAN. Notice that an adversary listening to CAN cannot easily attempt to detect an ID and announce a spoofed group start message with the ID, since the group mask is authenticated with 32-bit MAC1 (Phase I). Similarly, an attacker cannot replay a group start message (i.e. resending a spoofed message), since group mask is protected by incorporating GRC in the authentication scheme. Similarly, confidentiality and integrity in Phases II and III as referred in next subsections, are accomplished via authentication, i.e. using the GRC in the MAC2 computation.

VATICAN-G supports secure vatiCAN-G messages. As shown in Figure 35, a group ID mask is used to define a vatiCAN-G Clique using **`InitGroupMask`** as a group of CAN nodes who intend to participate next, in a single round of secure communications. Participants in a vatiCAN-G Clique instance, uniquely identified by a group mask, share a session key, computed via **`calcSessionKey`** function by hashing on the pairwise keys of all group nodes [46]. A sender announces a Group Start Message via **`Send`** function and receives a message via **`MsgAvailable`** .

More specifically, VATICAN-G uses the following basic functions, as shown in Figure 35. Explanations of all the remaining functions will be provided in a vatiCAN-G manual, once this work matures and is delivered as open source (expected in 03/2019).

- ***void InitGroupMask(uint16_t groupID, int index);***
- ***uint64_t GetGrcCounter();***
- ***uint8_t MsgAvailable(uint16_t id, uint8_t ** buffer, uint8_t& length, uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint16_t localNodeID, uint16_t ArduID,uint8_t * server_grc, const unsigned char * getcode, uint8_t * critical);***
- ***void send(CSENDER ID, uint8_t *payload, uint8_t len,uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint32_t pmask,uint8_t *want_flag);***
- ***void MessageAuthenticate(uint16_t ID, uint64_t grc, const uint8_t *msg, int16_t msglength, uint8_t *mac, uint16_t caller, uint16_t realCaller);***
- **void calcSessionKey(uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint16_t id);**

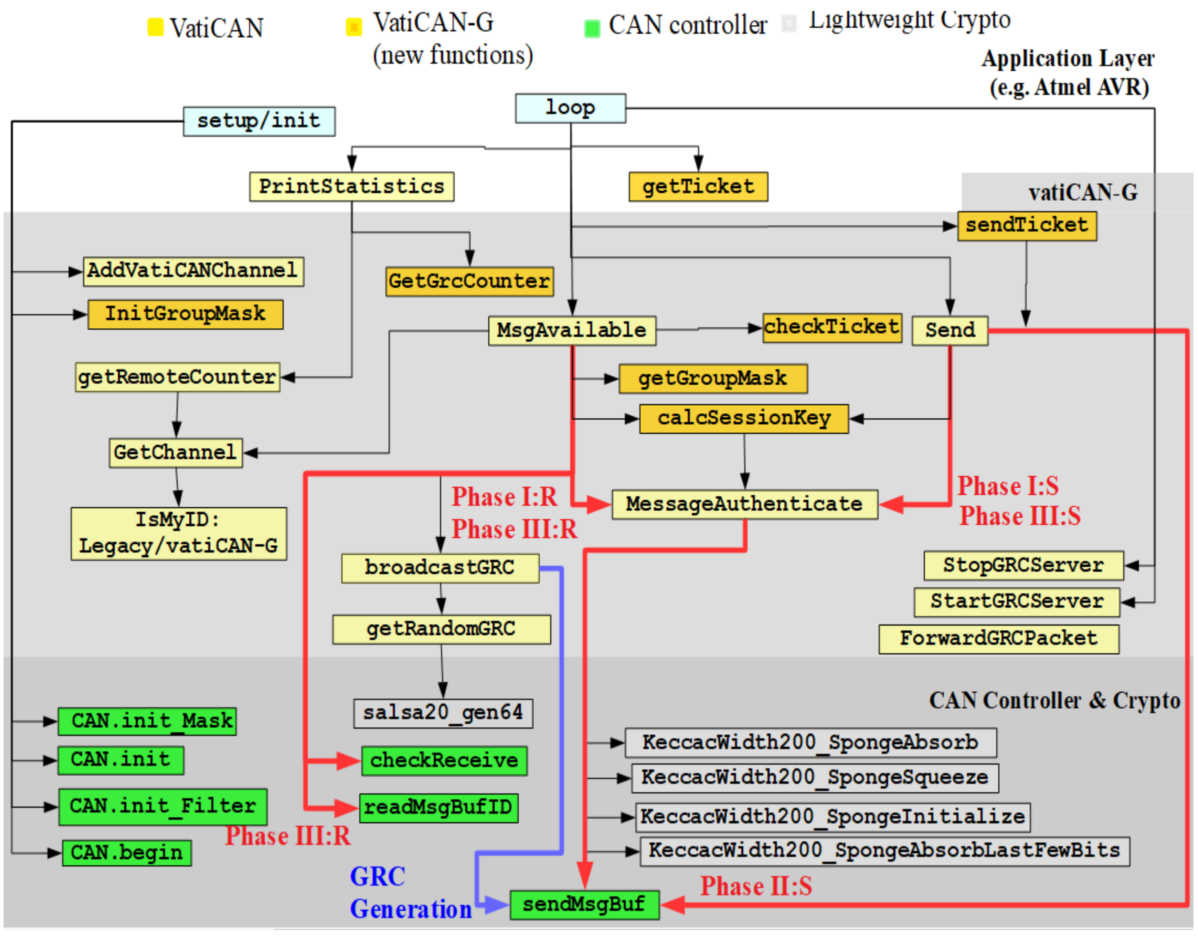


Figure 35: Vatican-G protocol with three send/receive control flow phases and GRC generator.

5.5.1 InitGroupMask

The **`InitGroupMask`** maps each group CAN id to a specific index number.

- ***`void InitGroupMask(uint16_t groupID, int index);`***

The function uses the following two arguments:

- **`groupID`** corresponds to secure group IDs. For example, 0x32, 0x34 etc.
- **`index`** defines the array number that sets the group IDs.

5.5.2 `GetGrcCounter`

The **`GetGrcCounter`** function sets the counters of the secure node to the previously broadcasted value (from GRC server). The function returns a verified state **`GRC_UPDATED`** **`(uint_8).`**

- **`uint8_t GetGrcCounter();`**

5.5.3 `Send`

The function **`Send`** broadcasts vatiCAN-G (authenticated messages) or legacy messages.

- **`void Send( uint16_t id,uint8_t *payload, uint8_t len, uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint32_t pmask,uint8_t *want_ok_flag);`**

This function extends the **`send`** function of VatiCAN [47] which defines three arguments:

- **`id`** is the sender's ID.
- **`payload`** defines the bytes to send (0 to 8).
- **`len`** defines the length of the supplied payload.

The new **send** uses the following five extra arguments:

- **`Keylist`** holds the keys of all secure group participants.
- **`Partlist`** stores the secure node IDs of secure group participants; notice that the node is a member of this group.
- **`PartListLen`** sets the length of participants **(`partList`).**
- **`pmask`** is a 32-bit mask (integer) representing the group participants via (possibly also mapped to an **`ArduID`**).
- **`want_ok_flag`** is used by the secure nodes to establish a new counter value using the GRC protocol. For example, with a 1 value, a **want** message is sent, while with a 0 value an **ok** message is sent. These messages are used to distinguish between different phases of the GRC protocol.

The **`Send`** function is used in the following three phases:

- **Phase I:** A sender announces a Group Start message that contains the **`id`**, the **`pmask`**, and a **`32-bit MAC1`** message authentication code (computed via **MessageAuthentication**, see below); these last four bytes of the payload ensure that an authorized sender has transmitted the group start message.
- **Phase II:** The sender sends 64-bit *data* (with the same *sender ID*).
- **Phase III: Send**: The final step involves transmitting the sender **`ID`** with a **`MAC2`** (as payload) for authentication. **`MAC2`** is computed using *data*, *GRC*, and *session key.*

5.5.4 `MsgAvailable`

The function **`MsgAvailable`** is called periodically to manage received messages coming from **Send** method.

- **`uint8_t MsgAvailable (uint16_t id, uint8_t** buffer, uint8_t& length, uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint16_t localNodeID, uint16_t ArduID, uint8_t * server_grc, const unsigned char *getcode, uint8_t *critical);`**

This function extends the **`Available`** function of VatiCAN [47] re-using the following three arguments:

- **id** returns a value that holds the sender's `CAN ID (legacy ID, VatiCAN-G ID).`
- **`buffer`** sets value that points to the payload buffer.
- **`length`** returns the payload's length.

The new **`MsgAvailable`** functions has eight extra arguments:

- **`Keylist`** holds the keys of all secure group participants.
- **`Partlist`** stores the secure node IDs of secure group participants; notice that the node is a member of this group.
- **`PartListLen`** sets the length of participants **`(partList).`**
- **`localNodeID`** sets the vatiCAN-G ID of the caller node.
- **`ArduID`** defines the Arduino's predefined ID written in EEPROM. By writing a unique number (e.g. 0 to 15) in the microcontroller's EEPROM, we are able to provide unique node identification and unify programming across CAN nodes, increasing reuse and software maintenance.
- **`server_grc`** is a **`uint8_t`** pointer used internally by the GRC Server in the context of the GRC protocol.
- **`getcode`** is a pointer to a buffer used as an argument in different authentication functions.
- **`critical`** is a pointer used in the GRC protocol related to the GRC server.

The **`MsgReceive`** function consists of three following phases:

Phase I: Each CAN node receiving a group start message authenticates the group mask by comparing *MAC_1* to a hash obtained by considering the *group mask*, the *secret keys* of Clique participants (indexed by mask), and a *global counter* (*GRC)*. If the *group mask* is successfully authenticated, the node shall wait next for secure data transmission (Phases II/III).

Phase II: Upon message receipt, a message authentication code is computed at each intended receiver (via **MessageAuthenticate** as explained below). This code is computed as a hash based on the *data*, *GRC*, and *session key* of all Clique participants.

Phase III: The receiver accepts the message only if the previously computed MAC_2 matches the transmitted MAC code. Otherwise, the receiver rejects the data.

5.5.5 `MessageAuthenticate`

The function **`MessageAuthenticate`** is called periodically to validate received messages in secure group communication.

- **`void EPOCH::MessageAuthentication(uint16_t ID, uint64_t grc, const uint8_t* msg, int16_t msglength, uint8_t *mac, CSENDER calltype, uint16_t realCaller);`**

This function extends the **`MessageAuthedication`** function of VatiCAN [47] which has the following five arguments:

- **`id`** defines sender/receiver CAN ID that is used for computing the MAC.
- **`grc`** sets the counter that will be used for the crypto hash functions.
- **`msg`** defines the message for which to compute the code.
- **`msglength`** is the length of that message (typically 1 to 8 bytes for CAN).
- **`mac`** is a buffer to save the MAC computed (64 bits).

`The new` **`MessageAuthenticate`** `has two extra arguments:`

- **`calltype`** is an integer variable that distinguishes internal computations in the Keccak cryptographic algorithm (i.e. vatiCAN-G secure group, or GRC/ticket protocol).
- **`realCaller`** is the secure vatiCAN-G ID of the caller used for the GRC protocol cryptographic hashing.

5.5.6 `calcSessionKey`

The **`calcSessionKey`** creates a **`SessionKey`** according to **`KeyList`** and **`partList`**. Only groups participant can calculate and use the session key.

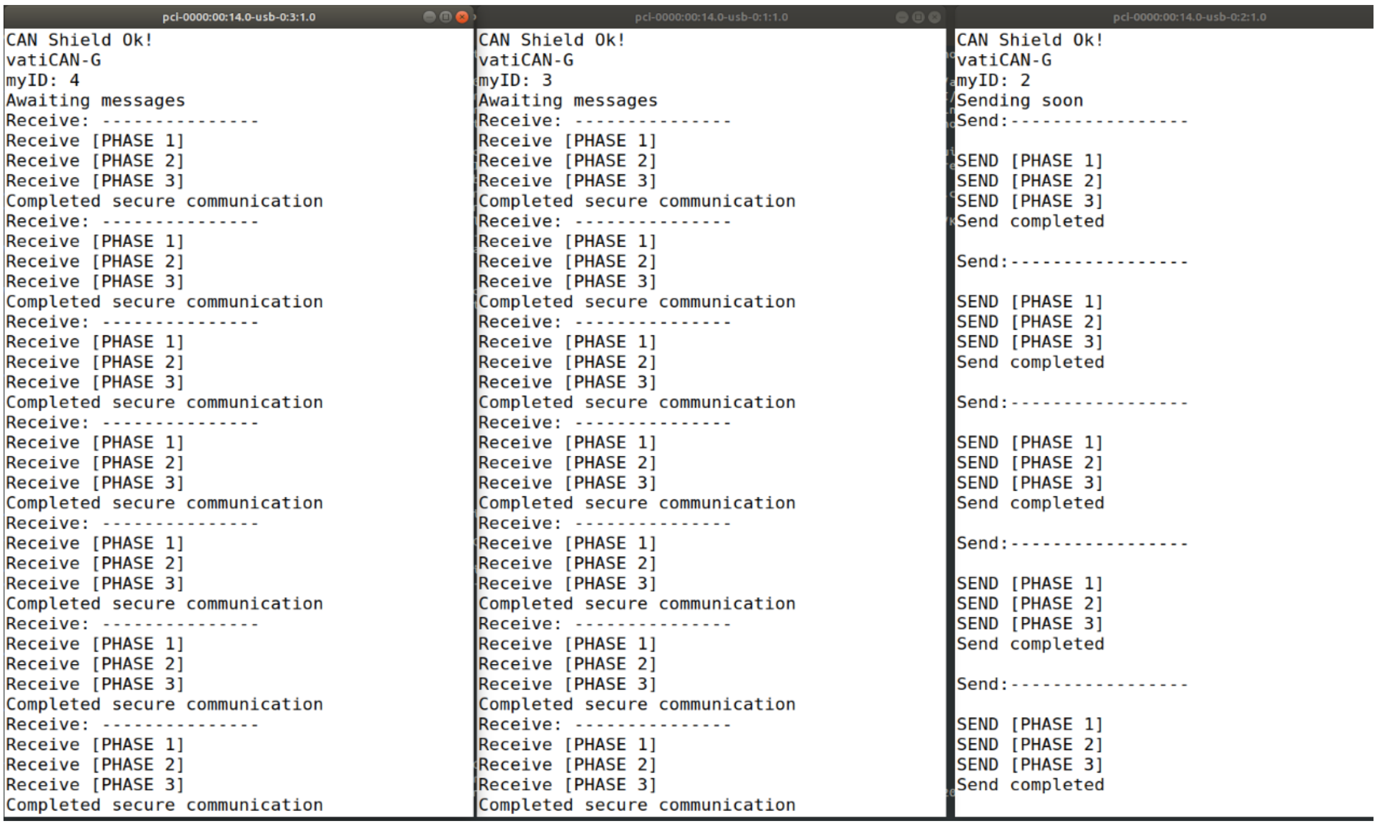
- **`void calcSessionKey(uint64_t *KeyList, uint16_t *PartList, uint8_t *PartListLen, uint16_t id);`**

This is accomplished by four arguments:

- **`Keylist`** holds the keys of all secure group participants.
- **`Partlist`** stores the secure node IDs of secure group participants; notice that the node is a member of this group.
- **`PartListLen`** sets the length of participants **`(partList).`**
- **`Id`** is the vatiCAN-G ID of the caller node.

5.5.7 vatiCAN-G example

In the following figure an example of one sender and two receivers is presented using the vatiCAN-G platform.



pci-0000:00:14.0-usb-0:3:1.0
pci-0000:00:14.0-usb-0:1:1.0
pci-0000:00:14.0-usb-0:2:1.0

```
CAN Shield Ok!
vatiCAN-G
myID: 4
Awaiting messages
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
```

```
CAN Shield Ok!
vatiCAN-G
myID: 3
Awaiting messages
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
Receive: ---------------
Receive [PHASE 1]
Receive [PHASE 2]
Receive [PHASE 3]
Completed secure communication
```

```
CAN Shield Ok!
vatiCAN-G
myID: 2
Sending soon
Send:-----------------

SEND [PHASE 1]
SEND [PHASE 2]
SEND [PHASE 3]
Send completed

Send:-----------------

SEND [PHASE 1]
SEND [PHASE 2]
SEND [PHASE 3]
Send completed

Send:-----------------

SEND [PHASE 1]
SEND [PHASE 2]
SEND [PHASE 3]
Send completed

Send:-----------------

SEND [PHASE 1]
SEND [PHASE 2]
SEND [PHASE 3]
Send completed

Send:-----------------

SEND [PHASE 1]
SEND [PHASE 2]
SEND [PHASE 3]
Send completed
```

Figure 36: vatiCAN-G send & receive phases

6. Conclusions

CAN is still the network of choice in automobiles and related automated real-time communication environments. However, CAN bus messages have no sender or receiver address, and are not protected by any Message Authentication Code (MAC) or digital signature. Thus, it is possible for an attacker to gain control of a car by directly spoofing or replaying the velocity sensors of anti-lock braking systems, thereby corrupting critical sensor data by manipulating exchanged packets in the CAN network. We design and implement open source, lightweight, group-based CAN authentication protocols for protecting from masquerade and replay attacks and evaluate performance overheads when supporting real-time task (message communication and computation) schedules.

7. Future work

In the future it would be interesting to explore the following technical aspects, and possibly add to vatiCAN-G's weaponry:

- Intrusion detection based on abnormal activity on the CAN bus; vatiCAN-G must react to such a DDoS attack.
- Event logging capabilities could possibly be used to analyze the whole system as well as cooperate with the intrusion detection system.
- Validity voting, adding overhead and increasing security.

8. References

[1] Elearning.vector.com. (2018). *CAN_E: Motivation for CAN*. (11 April 2018) [Online] Available at: https://elearning.vector.com/mod/page/view.php?id=334

[2] Lima, F. Rocha, M. Völp, et al., "Towards safe and secure autonomous and cooperative vehicle ecosystems", in Proc. 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy, 2016, pp. 59--70.

[3] Rosu, C. (2018). *EMC - EV | CAN bus (Controller Area Network)*. [Online] Available at: http://www.flexautomotive.net/EMCFLEXBLOG/post/2015/09/08/can-bus-for-controller-area-network.

[4] A. Κρίβας, Ε. Γιατράκης, "Μελέτη συστήματος διαγνωστικών αυτοκινήτων (OBD-II)," *Nemertes:Μελέτη συστήματος διαγνωστικών αυτοκινήτων (OBD-II)*, (10 Jun 2014). [Online]. Available: http://nemertes.lis.upatras.gr/jspui//handle/10889/7766

[5] "CAN bus" Wikipedia. (Sept 5, 2018) [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus#Data_frame

[6] "MOST Bus", Wikipedia,.(Sept 7, 2018) [Online]. Available:

https://en.wikipedia.org/wiki/MOST_Bus.

[7] "Local Interconnect Network" Wikipedia. (Aug 13, 2018) [Online]. Available:

https://en.wikipedia.org/wiki/Local_Interconnect_Network.

[8] "FlexRay" Wikipedia. (July 24, 2018) [Online]. Available:

https://en.wikipedia.org/wiki/FlexRay

[9] "Spoofing Attack" Wikipedia. (Sept 16, 2018)[Online]. Available: https://en.wikipedia.org/wiki/Spoofing_attack

[10] Radware "DNS Amplification Attack". (Nov 26, 2012) [Online]. Available:

https://www.youtube.com/watch?v=xTKjHWkDwP0

[11] "Reply attack" Wikipedia, (Sept 12, 2018). [Online]. Available: https://en.wikipedia.org/wiki/Replay_attack

[12] S. Linus "DDoS Attacks As Fast As Possible" (June 2, 2015) [Online]. Available: https://www.youtube.com/watch?v=0I4O4hoKzb8

[13] “The Difference between DOS and DDOS attacks” (May 1, 2017) [Online]. Available: http://www.differencebetween.net/technology/the-difference-between-dos-and-ddos-attacks/

[14] “Transmission Control Protocol” Wikipedia, (Sept 12, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol#Connection_establishment

[15] “SYN flood” Wikipedia . (Sep 14, 2018) [Online]. Available: https://en.wikipedia.org/wiki/SYN_flood

[16] C. Cimpanu, “Acoustic Attacks on HDDs Can Sabotage PCs, CCTV Systems, ATMs, More “, (27 Dec 2017) [Online]. Available: https://www.bleepingcomputer.com/news/security/acoustic-attacks-on-hdds-can-sabotage-pcs-cctv-systems-atms-more/

[17] “Feistel Cipher” Wikipedia. (une 29, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Feistel_cipher

[18] (June 29, 2018) [Online]. Available: https://www.tutorialspoint.com/cryptography/feistel_block_cipher.htm

[19] “Block cipher mode of operation” Wikipedia. (June 30, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

[20] “Galois/Counter Mode” Wikipedia. (June 30, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Galois/Counter_Mode

[21] J. Cho and et al, “Power dissipation and area comparison of 512-bit and 1024-bit key AES” Computers and Mathematics with Applications, **66(9)**, 2013, pp. 1378-1383.

[22] S. Trenhome, “Rijndael’s mix column stage”, [Online] Available: http://www.samiam.org/mix-column.html

[23] D. Selent, (2010 Nov). “Advanced Encryption Standard.” InSight: Rivier Academic Journal. **6(2). ,** [Online]. Available: https://www2.rivier.edu/journal/ROAJ-Fall-2010/J455-Selent-AES.pdf

[24] “Advanced Encryption Standard”, Wikipedia. (Sept 13, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[25] “Advantages and disadvantages of Stream versus Clock Ciphers” (Nov 14, 2010) [Online]. Available: https://security.stackexchange.com/questions/334/advantages-and-disadvantages-of-stream-versus-block-ciphers

[26] S. Babbage and et al, “The eSTREAM Portfolio ” (April 15, 2008) [Online]. Available: http://www.ecrypt.eu.org/stream/portfolio.pdf

[27] “Salsa20/12” (March 2012). [Online]. Available: http://www.ecrypt.eu.org/stream/e2-salsa20.html

[28] D. J. Bernstein. ”The Salsa20 Family of Stream Ciphers” *New Stream Ciper Designs: The eSTREAM Finalists.* 1st ed., M.Robshaw, O. Billet, Germany: Springer, 2008, pp. 84-97

[29] “Free Password Hash Cracker”, (Jan 25, 2018) [Online]. Available: https://crackstation.net/

[30] “Avalance effect”, Wikipedia, (July 15, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Avalanche_effect

[31] “NIST Policy on Hash Functions”, NIST (Aug 5, 2015) [Online]. Available: https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions

[32] “Announcing the first SHA1 collision”, (Feb 23, 2017) [Online]. Available: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

[33] Arduino Guide [Online]. Available: https://www.arduino.cc/en/Guide/Introduction

[34] M. Brain, “ How Microcontrollers work”, (April 1, 2000) [Online]. Available: https://electronics.howstuffworks.com/microcontroller6.htm

[35] “Arduino Uno”, Wikipedia, (Sept 7, 2018) [Online]. Available: https://en.wikipedia.org/wiki/Arduino_Uno

[36]“Arduino & Geduino MEGA 2560” [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardMega2560?setlang=en

[37]“Arduino Leonardo” [Online]. Available: https://www.arduino.cc/en/Main/Arduino_BoardLeonardo

[38]“CAN-BUS Shield V2 (SKU: DFR0370)” [Online]. Available: https://www.dfrobot.com/wiki/index.php/CAN-BUS_Shield_V2_(SKU:_DFR0370)

[39] “ECUsim 2000 OBD Simulator” [Online]. Available: https://www.scantool.net/ecusim-2000/

[40] S. Nürnberger , C. Rossow “– vatiCAN– Vetted, Authenticated CAN Bus” in Cryptographic Hardware and Embedded Systems, 1st ed. B. Gierlichs , A. Poschmann, Germany: Springer, 2016, pp. pp 106-124

[41] D. K. Nilsson, U. E. Larson, and E. Jonsson, “Efficient In-Vehicle Delayed Data Authentication Based on Compound Message Authentication Codes,” in Proc. of the 68th IEEE Vehicular Technology Conference (VTC 2008-Fall). IEEE, 2008, pp. 1–5.

[42] Perrig, Adrian, Ran Canetti, J. Doug Tygar and Dawn Song. “The TESLA Broadcast Authentication Protocol ∗.” CryptoBytes, 5(2), Summer/Fall 2002, pp. 2-13.

[43] C. Szilagyi and P. Koopman, “A Flexible Approach to Embedded Network Multicast Authentication,” in 2nd Workshop on Embedded Systems Security (WESS), 2008.

[44] C. Szilagyi, “Low Cost Multicast Network Authentication for Embedded Control Systems,” PhD dissertation, Dept. of Electrical and Computer Eng., Carnegie Mellon Univ., 2012;

[45] C.-W. Lin and A. Sangiovanni-Vincentelli, *Security-aware design for cyber-physical systems: a platform-based approach*, Springer Verlag, 2017. ISBN 978-3-319-51327-0.

[46] M.D. Grammatikakis,. N. Mouzakitis, and et al, , “On-the-fly Secure Group Communication on CAN Bus”, *in Proc. APPLEPIES 2018 International conference on Applications in Electronics Pervading Industry, Environment and Society*, 27 September 2018, Pisa, Italy

[47] *“VatiCAN library documentation” [Online]. Available: http://www.automotive-security.net/vatican/doc/*

[48] M. Hashem Eiza and Q. Ni, "Driving with Sharks: Rethinking Connected Vehicles with Vehicle Cybersecurity," in IEEE Vehicular Technology Magazine, vol. 12, no. 2, pp. 45-51, June 2017.