Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Matematica

ANALYSIS AND PARTIAL SOLUTIONS OF SECURITY PROBLEMS IN AUTOMOTIVE ENVIRONMENTS

Tesi di Laurea Magistrale in Crittografia

Relatore:
Chiar.mo Prof.
Davide Aliffi

Presentata da:
Elisa Bragaglia

Correlatore:
Chiar.mo Dott.
Cosimo Senni

III Sessione

Anno Accademico 2013/2014

Introduzione

I moderni veicoli non sono piú ‘semplici’ macchine meccaniche, ma contengono una miriade di diverse componenti elettroniche collegate tra loro. Sono i computer, oggi, che coordinano e monitorano questi componenti, i sensori, il guidatore e i passegeri per ottimizzare le prestazioni del veicolo e la sicurezza degli utilizzatori. Ma questa transformazione ha introdotto anche una gamma di nuovi potenziali rischi: questioni sulla robustezza e l’affidabilità della comunicazione tra questi dispositivi e la prevenzione della loro manomissione devono essere sollevate.

Questo scritto mira a fare una panoramica di questi problemi in campo automobilistico e delle soluzioni oggigiorno disponibili.
Partendo con una descrizione generale del circuito interno dell’auto, analizzeremo i suoi punti di accesso, e discuteremo i danni prodotti dalla sua manomissione illecita.
In seguito vedremo se é possibile prevenire tali attacchi, dando un’occhiata alle soluzioni disponibili e soffermandoci in particolare sui moduli crittografici e le loro applicazioni.
Infine, presenteremo l’implementazione pratica di un protocollo di autenticazione tra ECUs e una dimostrazione matematica della sua sicurezza.

Niente di tutto ciò sarebbe stato possibile, senza l’azienda **Magneti Marelli** di Bologna, dove ho passato gli ultimi cinque mesi come tirocinante del **CTO** (Chief Technical Office) e dove ho studiato e appreso ciò che ora racconterò a voi.

Introduction

Modern vehicles are no longer ‘simple’ mechanical devices, but contain a myriad of different electronic components networked together. Computers coordinate and monitor these components, the sensors, the driver and the passengers to optimize the performance of the vehicle and increase the safety of the users. But this transformation has also introduced a range of new potential risks: questions about the robustness and reliability of the communication between those devices and the prevention of their tampering shall be raised.

This report aims to give an overlook of these problems in automotive environments and the solutions today’s available.
Starting with a general description of cars internal network, we will analyze its entry points and we will discuss the damages originated by its illicit alteration.
Then we will see if it’s possible to avoid these attacks, having a look at the available solutions and considering in particular the cryptographic modules and their applications.
Finally we will report the practical implementation of an ECUs authentication protocol and a mathematical proof of its security.

None of this would not have been possible without the **Magneti Marelli Company** of Bologna, where I passed the last five months as a trainee of the **CTO** (Chief Technical Office) and where I studied and learned what I’m gonna write now.

Contents

Part I

Analysis of Threats

Chapter 1

How car's brain works

1.1 ECUs and CAN packets

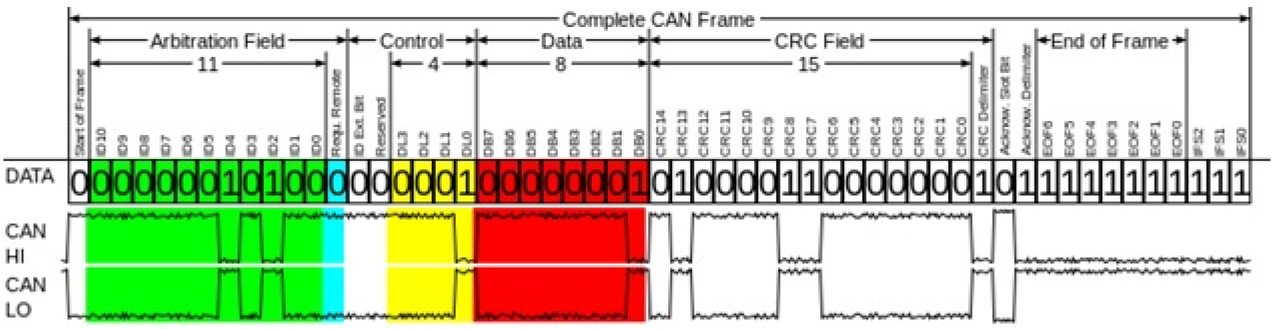
Today, electronics is more and more used in the automotive industry.
Mechanic devices make ways for millions lines of code and automated components that test, analyze, and take decisions in order to allow the motion, optimizing at the same time the total performance of the vehicle.
Security, consumption, emission, drivetrain, brakes, lighting and entertainment are only some examples of systems that are submitted to the control of electronics.
A modern vehicle contains between 50 and 80 independent computers: the **Electronic Control Units** (**ECUs**).
ECUs exchange data with sensors and communicate each other over one or more shared internal network buses; fundamental for this study is the **Controller Area Network** (**CAN**) standard: the **CAN-bus**.
Available transmission rates are low-speed (around 33Kbps), mid-speed (around 128Kbps) and high-speed CAN-buses (around 500Kbps).
This interconnection between ECUs increases safety cause it allows to share safety relevant information. For example pre-tensioning of seat-belts, detect skids, perform anti-lock braking, etc. and allows some convenient features like automatically varying radio volume as a function of speed.
At the same time, this architecture permits that, on a given bus, each component has

implicit access to every other component, and, since some ECUs are connected to more than one bus, the final result is that if an adversary were able to compromise and control a single ECU then this capability would be sufficient to control critical components across the entire car.

The table below presents the main ECUs in a modern car.

Component	***Functionality***
ECM	*Engine Control Module* Controls the engine using information from sensors to determine the amount of fuel, ignition timing, and other engine parameters.
EBCM	*Electronic Brake Control Module* Controls the Antilock Brake System (ABS) pump motor and valves, preventing brakes from locking up and skidding by regulating hydraulic pressure.
TCM	*Transmission Control Module* Controls electronic transmission using data from sensors and from the ECM to determine when and how to change gears.
BCM	*Body Control Module* Controls various vehicle functions, provides information to occupants, and acts as a firewall between CAN-buses of different speed.
Telematics	*Telematics Module* Enables remote data communication with the vehicle via cellular link.
RCDLR	*Remote Control Door Lock Receiver* Receives the signal from the car's key fob to lock/unlock the doors and the trunk. It also receives data wirelessly from the Tire Pressure Monitoring System sensors.
HVAC	*Heating Ventilation, Air Conditioning* Controls cabin environment.
SDM	*Inflatable Restraint Sensing and Diagnostic Module* Controls airbags and seat belt pretensioners.
IPC/DIC	*Instrument Panel Cluster/Driver Information Center* Displays information to the driver about speed, fuel level, and various alerts about the car's status.
Radio	*Radio* In addition to regular radio functions, funnels and generates most of the incabin sounds (beeps, buzzes, chimes).
TDM	*Theft Deterrent Module* Prevents vehicle from starting without a legitimate key.

On the CAN-bus the ECUs communicate with each other by sending **CAN packets**, broadcast to all components on the bus; each component decides whether it is intended for them or not.
There're two forms of CAN packets: **standard** or **extended**.



(from: `http://en.wikipedia.org/wiki/File:CAN-Bus-frame_in_base_format_without_stuffbits.svg`*)*

As we can see in the figure, in a standard CAN packet we find:

- 11/29 bits for the Identifier (standard or extended)
- 1 bit for the IDE (Identifier Extension): 0 for standard CAN, 1 for extended.
- 4 bits for the size of the data (from 0 to 8 bytes)
- 0 to 64 bits (8 bytes) maximum of data (in the above example: 1 byte)

We are not interested now in the other bits, so we will ignore them.

We can classify CAN packets in two main types: **normal** and **diagnostic**.
Normal packets are sent from the ECUs and can be seen on the network at any given time; they could be information for other ECUs, commands for other ECUs to act on,...
Diagnostic packets are sent by diagnostic tools used by mechanic to communicate with and interrogate an ECU and usually they are not present during the normal operation of the vehicle.

For **normal CAN packets** the identifier identify the content of the message and, if more packets are sent at the same time, it's essential to decide the priority of the transmission: the lower the number of the identifier, the higher the priority.
ECUs and devices don't have a fixed ID, everyone could send multiple arbitrary IDs, so it's impossible, a priori, to know which ECU is sending or receiving a particular packet. This fact makes reverse engineering traffic complicated, but, on the other hand, during an attack, it guarantees the anonymity of the sender.

On the contrary, in the case of **CAN diagnostic packets**, each ECU has a particular ID assigned to it, to recognize it and to decide the priority of transmission, and usually their structure follow a precise international standard: one of the most common is the **ISO-TP**(*Transport Protocol).*
This standard defines a way to send arbitrary length data over the bus, and the two specifications **ISO 14229** and **ISO 4230** describe the format of the actual data sent.
We won't see the standard in detail but it's useful to know that one byte of the data reports the service ID, that is the number of the service to implement.
Some of the most important services are:

- 0x10 - Initiate diagnostics
- 0x11 - ECU Reset
- 0x14 - Clear Diagnostic Codes
- 0x22 - Read Data by ID
- 0x23 - Read Memory by Address
- 0x27 - Security Access
- 0x2e - Write Data by ID
- 0x34 - Request Download
- 0x35 - Request Upload
- 0x36 - Transfer Data
- 0x37 - Request Transfer Exit
- 0x3d - Write Memory By Address
- 0x3e TesterPresent

1.2 Challenge-Response Authentication and Parallel Attack

Let's focus for a while on the **SecurityAccess (0x27)**.
This one is used to authenticate someone (e.g. a device) to the ECU in order to access some protected information or important commands (e.g. flashing ROMs).
The ECU and the device have a shared cryptographic function and a key:

1. the ECU sends to the device a seed
2. the device computes the seed, using the function and the key, and sends it back to the ECU
3. the ECU makes the same computation and controls if the one it recieved is correct:
    - if yes, the device is authenticated and his request is accepted
    - if no, the request is ignored

This authentication protocol is an example of a more general process, called ***'Challenge-Response'***, used in Cryptography to **authenticate an entity to another**.
In simple words, in the 'Challenge-Response' method, an entity Y that wants to prove X's identity, sends him a random number as a challenge. X, in order to be authenticated, has to send back to Y the right response. The computation of the response depends on the technique chosen (e.g. symmetric key, public key, digital signature,..), but in any case it requires the use of a secret information that only X and sometimes Y know.
In the example above a symmetric key technique is used for a one way authentication, with a random number as challenge.

Here is the scheme of this process: B has to prove A's identity.

- r_B is a random number
- A and B share:
    - k, the secret key
    - $E_{k,ID}(x)$, $E_{k,ID}$ is the cryptographic function/algorithm that computes r_B with k and user's ID

Algorithm:

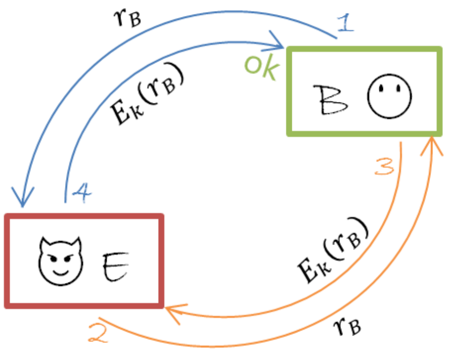
1. **B** takes a random number r_B and sends it to **A** *(challenge)*
2. **A** calculates $E_{k,ID}(r_B) = E_{k,ID}(r_B, k, ID_A)$ and sends it to **B** *(response)*
3. **B** calculates $E_{k,ID}(r_B) = E_{k,ID}(r_B, k, ID_A)$ and controls A's result: if it's correct the authentication is done.

Why is important to encrypt A's ID with r_B and the key?

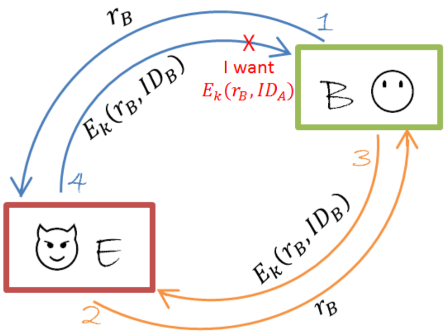
One reason could be to avoid a **'Parallel Session attack'** that, under certain conditions, permits an attacker to be correctly authenticated, without knowing any secret key.
Let's see the steps of this kind of attack when we don't use any ID during the coding process:
E is an attacker and B the entity that E wants to cheat

1. **E** asks B to be authenticated
2. **B** answers with the challenge r_B
3. **E** sends to B a request of identification, and then the same challenge r_B
4. **B** in order to answer E's request, calculates $E_k(r_B)$ and sends it to E
5. **E** sends back $E_k(r_B)$ to B
6. **B** verify E's computation of its challenge and validates E's identity

Result: an attacker is correctly authenticated with B.



Since in normal CAN communication a unique ID to identify a component is not contemplated, we should pay attention not to fall in this kind of trap.
If each component has a unique ID, coding it with r_B is effectively a solution; in fact, while in step n·4 B answers with $E_k(r_B, ID_B)$, E in step n·5 should answer with $E_k(r_B, ID_A)$, where A is the legitimate entity that E wants to impersonate, that means E needs ID_A and k to succeed.



Chapter 2

Tampering a car

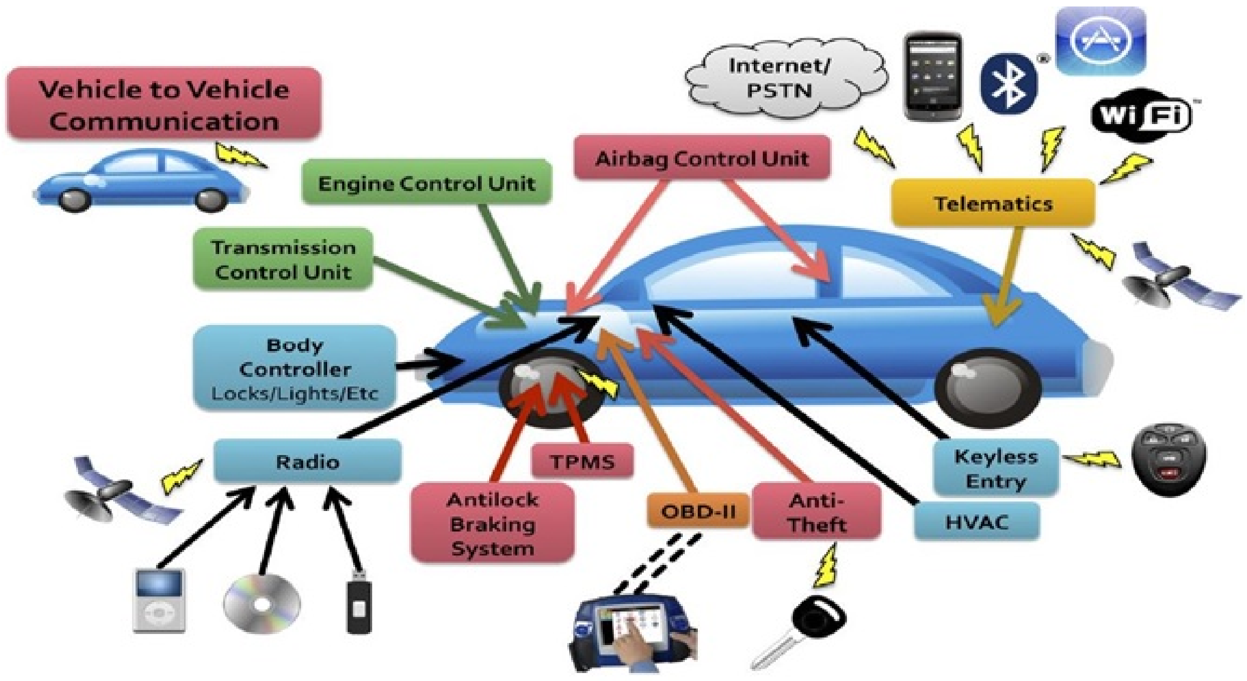
2.1 How to enter into car's network

Now that we know how the brain of our car works, let's see how a malicious attacker could have access to its network.
Let's put ourselves in hacker's shoes: here are some attacks that we can implement growing in ability:

- **Fuzzing**: inject random or partially random CAN packets in the networks, in order to create indiscriminate disruption and overload the BUS.
- Extract the firmware, **reverse engineer** its I/O code, and inject precise CAN traffic.
- **Create a malware** with our malicious code, reflash an ECU in order to memorize it: control/damage remotely and repeatedly the car.

In any case, primarily, we need to find a way to enter in car's hidden system in order to spy/inject CAN traffic: we need an **'entry point'**.

This figure summarizes how vehicles communicate with the outside world:



*(*from [2])

It's necessary to make a distinction between gates who require a **prior physical** access (we need to insert something phisically in the car in order to inject data: e.g. connect a data's transmitter device to the OBD-II) and those who are susceptible to **remote compromise** (we don't need physical acces to the car: wireless communication). Theoretically every gate in the figure represents a possible threat, but an attack to the CAN-bus via prior physical access is considered less realistic, because someone that has direct access to the vehicle could compromise it manually (e.g. cutting the brake lines), saving time and study.
So, without forgetting any entry points, the second group is considered as the real future threat and the interesting part.

Prior physical access

In the first group we find:
a. the OBD-II Port

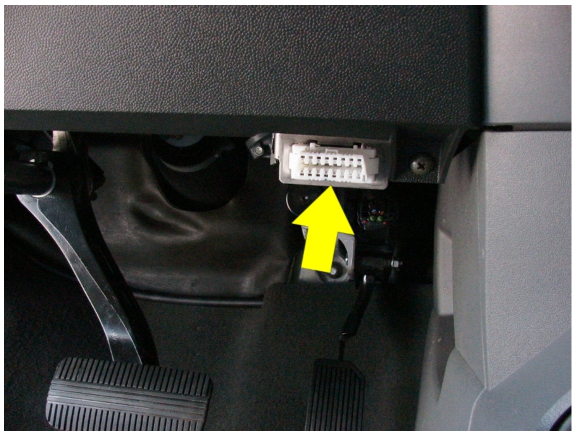
- PassThru device
- CAN Hacking Tool

b. the Infotainment System

c. CD player, USB, iPod/iPhone docking port

a. The OBD-II Port

On Board Diagnostic is a term referring to the vehicle's capability of self-diagnosing errors and breakdowns.
The OBD-II port is the gate that permits to enter directly in the CAN-bus to read all these information (very useful for our mechanic that doesn't have to check manually every component of the automobile!).



The OBD-II port

It's inside the cabin, usually on the steering column, and it can be connected to a common laptop to observe/steal/inject CAN packets and **reprogram car's ECUs**.
We can find specific cables and adaptors, available on the market, that permit a direct connection between the port and the computer; then we need a software on the laptop able to create an interface to make readable the CAN flow.

Otherwise, it exists a device, called **'PassThru device'**, that connects a laptop to the CAN-bus and provides an interface to communicate with it to reprogram some

vehicle's control modules.
Since the way to reprogram ECUs changes depending on the manufacturer, this device can't connect to all cars, but only with vehicles that accept reprogrammation via a specific standard: the J-2534 (frequently used in North America).
There's a cable joining the device to the OBD-II port, and a USB cable or a Wi-Fi option to connect it to the laptop.
As the communication between the client application and the PassThru device is unauthenticated, the second solution presents a big threat: everyone connected to the same Wi-Fi line and with the same interface has access to the data, that means that, if it's not well protected, we can penetrate in the Wi-Fi line and try to control the car.
A limitation is that the device can communicate with one application at a time, so we need to wait for the device to be connected but not in use.
Furthermore, we can compromise the PassThru device itself, implant malicious code and then infect a great number of vehicles.
It's also possible to install a malware in an infected car that corrupts every PassThru device we will connect to.
Moreover it's proven[1] that a worm can pass from one PassThru device to another, infecting any device in range.

While this device is quite expensive, another similar gadget has been built with less than $20 (luckily is not on the market!):
Two Spanish security researchers, Javier Vazquez-Vidal and Alberto Garcia Illera, presented on march 2014, at the Black Hat Asia security conference in Singapore, a device they created, called **CHT (CAN Hacking Tool)**. With less than $20 they built this object, smaller than a smartphone, that can be physically connected to a car's internal network (OBD-II port), draws power from the car's electrical system, and controls remotely components of the car (which components it depends on car's model) sending wireless commands from an attacker's computer.
For a practical demonstration video, see: `http://goo.gl/sZxqUl`

[1]source: [2], page 8

b. The Infotainment System

The Infotainment System is the touchscreen interface that permits the driver to control the options of his car (i.e. radio, CD player, Bluetooth, cell connection, GPS, ...).

Usually on this device runs an operation system (Windows CE, Linux,..); it's possible to enter the system and compromise it uploading executable files, apps and plugins.
It's not faraway the day when the customer will be able to manage independently, e.g. via Wi-Fi, the uploading of the software or the installation of apps.
As for the cell phone or the computer, everything we download from the network, if it's not well controlled and certified is a threat for the device's security, so this situation represents for the user higher danger to be duped and to install malicious programs.
If this device is not connected to the CAN-bus a hacker could potentially distract a driver by blasting the stereo, or disable the navigation system, but these types of attacks are more annoying than life-threatening (even if a spoofed GPS [2] could put you into a hazardous situation: e.g. send you the wrong way down a one-way street).
Unluckily from this point of view, this connection already exists so the threat is real. Moreover, the infotainment system is always connected to the telematics units that, we will see, has a direct and dangerous access to the CAN-bus.

c. CD player, USB, iPod/iPhone docking port

Since the car's media system ECU is frequently connected to the CAN-bus, all devices it supports are indirectly connected to the CAN-bus too. So, in theory, if these devices

[2]source: [5] This article presents a study about how to spoof a GPS.

permit the transmission of data to the ECU, understanding how it works, it is possible to inject some malicious data in the CAN network via, for example, the CD player or the USB port (via a USB flash drive or connecting an iPod/iPhone).
Speaking about the CD player, researchers at the University of California and the University of Washington discovered two vulnerabilities, analyzing the unit: [3]

- the media player capability of automatically installing firmware upgrades permits the unit to:
  - automatically recognize an ISO 9660(a file system standard)-formatted CD with a particular name file
  - present the user a cryptic message
  - after the message, if a particular button is not pressed, reflash the unit with the data contained in the file.

  This capability allowed the researchers to reflash the media player ECU:
  they created a CD taping on it that particular file and they let it play in the CD player. The program inside the file reflashed the ECU in order to send some selected message in the CAN-bus each time the unit recieved a specific message over the FM RDS channel.

- a weakness in the firmware, in the input code for WMA files (see the article for details) that permits to construct a WMA audio file and encode it in a CD: it plays perfectly on a PC, but it sends chosen CAN packets in the bus, if played in the car.

Remote compromise

In the remote compromise group we place all the devices that receive a wireless signal:
Over short range:
a. Bluetooth
b. Radio Frequency Identification (RFID):Immobilizer, Remote Keyless Entry and

[3]source: [2] pages 7, 12

Passive Keyless Entry and Start
c. Tire Pressure Monitoring System (TPMS)
d. Vehicle to Vehicle Communications (WiFi)
And over long range:
e. Global Position System (GPS)
f. Digital Radio
g. Traffic Message Channel (TMC)
h. Cellular Connection

a. Bluetooth

Bluetooth capabilities, built in the Telematics ECU, allow the user's cell phone to connect to the car.
A study [4] made by the University of Washington and the University of California shows that every paired Bluetooth device could compromise the car's telematics unit and consequentially all the ECUs in the car. But, since the telematics interface and system is custom-built, we don't really know if the weakness found are generalizable to any vehicles or only a few.
The researchers observed that during a Bluetooth configuration process some instructions were copied using the `strcpy` function directly in the unit's *stack*. The stack is a part of the unit's memory where data are temporarily stored, waiting to be analized and executed by the program.
Since this copy is unchecked (i.e. none controls if the data copied are the real configuration instructions), a previously paired Bluetooth device could penetrate in the system and replace the good data with arbitrary ones (e.g. code that makes the unit send some messages in the CAN-bus).
The researchers did so with a Trojan Horse application uploaded on a smartphone running Android 2.1; the app was innocuous until they paired the smartphone to the car's telematics unit via Bluetooth, in this case it sent the attack payload, corrupting the ECU.
Then we could examine the situation in the event that **we don't have a paired**

[4]source: [2]: pages 8-9

Bluetooth device. It is proven [5] that it is possible to pair a device with a car surreptitiously, if the car's Bluetooth MAC address is known. This is the procedure to do so:

- first we **steal the car's Bluetooth MAC address** sniffing the Bluetooth traffic of a device, previously paired to the car, that has its Bluetooth unit enabled
- then to pair our device, we should **enter in the phone a secret PIN code**, that the car automatically generates and shows on the dashboard.
  Since the car replays to the pairing request and generates the code automatically, without driver's approval, we have time to brute force it without letting the drive suspect anything: the average is approximately 10 hours.

Someone could think that this attack is unrealistic because it could be successful only if the PIN code doesn't have a fixed period of time (e.g. 3 minutes) and only if the car runs for all the time we need to force the code, in fact the PIN will surely change if we restart the car.
Actually, in some particular situations, this attack could be a real threat: let's see an example:
Imagine that, during a day, you could sniff the MAC address of all cars parked in a parking garage. Know that in 1 minute you can try 8 different PINs [6], probabilities* say that: if a thousand of such cars leave this place in a day, you have good chances to brute force the PIN for at least one of them within three minutes.
Of course there's a limit: you can't choose exactly your target.

**Let's try to give an explanation:*

[5] source [2]: page 9
[6] source [2]: page 9, calculated on car's response time

Given 1000 cars and a 4 digits PIN, for the first car we consider the events:

- $A_{1,1}$ *: my* 1st *PIN is equal to the* 1st *car's PIN*
  $A_{1,2}$ *: my* 2nd *PIN is equal to the* 1st *car's PIN*
  $\vdots$
  $A_{1,24}$ *: my* 24th *PIN is equal to the* 1st *car's PIN*

Now, we do the same with all the other 999 cars:
*in general the $A_{k,i}$ event, where **k** is referred to the car and **i** to the PIN we try, is:*

$A_{k,i}$ *: my i^{th} PIN is equal to the k^{th} car's PIN, for* $k = 1, \cdots, 1000, \quad i = 1, \cdots, 24$

Now,

$$P(A_{k,i}) = \frac{1}{10^4} \quad for \quad k = 1, \cdots, 1000, \quad i = 1, \cdots, 24$$

So, the probability to guess the PIN of the k^{th} car with 24 attemps is:

$$P\left(\bigcup_{i=1}^{24} A_{k,i}\right) \underset{\downarrow}{=} \sum_{i=1}^{24} P(A_{k,i}) = \frac{24}{10^4}$$

the $A_{k,i} \quad i = 1, \cdots 24$ events are <u>*mutually exclusive*</u> *cause the PIN of a given car is unique*

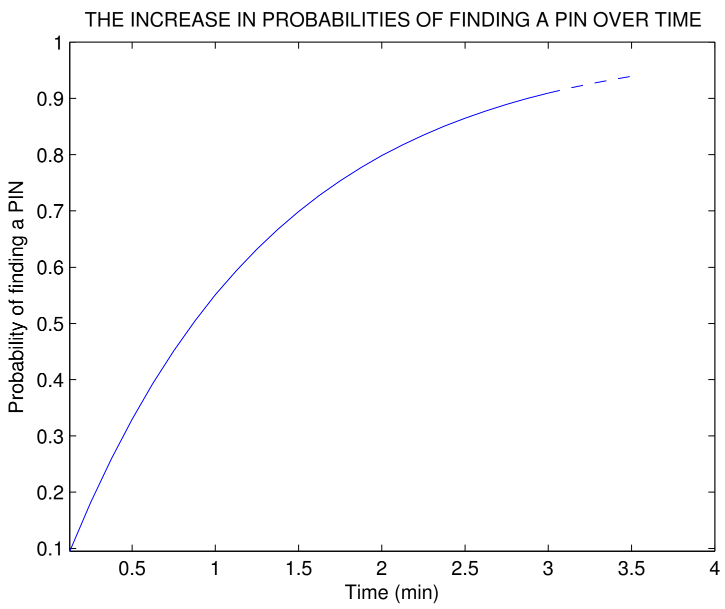
<u>*Note*</u>*: we call* $\bigcup_{i=1}^{24} A_{k,i} = A_k^* \quad for \quad k = 1, \cdots, 1000$

So*: given 1000 cars the probability to guess one of their PIN in 3 minutes (= 24 attemps) is:*

$$P\left(\bigcup_{k=1}^{1000} A_k^*\right) = 1 - P\left(\bigcap_{k=1}^{1000} \bar{A}_k^*\right) \underset{\downarrow}{=} 1 - \prod_{k=1}^{1000} P(\bar{A}_k^*) = 1 - \left(1 - \frac{24}{10^4}\right)^{1000} \sim 0.91$$

the $\bar{A}_k^* \quad k = 1, \cdots 1000$ events are <u>*independent*</u>

♣



b. Radio Frequency Identification (RFID): Immobilizer, Remote Keyless Entry and Passive Keyless Entry and Start

RFID appeared for the first time in U.S. in 1990's; it's a security system implemented between the car and his key.

An RFID Immobilizer is a transponder embedded in the top part of an ignition key. This transponder sends out an encrypted string of radio-frequency signals, basically a particular number of impulses broadcast on various radio frequencies to create a specific code, when the driver inserts it into the ignition-key slot.

Without this code, the car either won't start or won't activate the fuel pump. So even if someone hotwires the car or copies an ignition key, the ignition isn't going to work because it hasn't received the proper radio-frequency code.[7]

Recently this system improved into the Remote Keyless Entry (RKE).

The RKE, inserted in a KeyFob, permits, thanks to the RFID, to open/close car's doors, activate alarms, flash lights and disable immobilizers, by pressing a button in the fob.

Modern systems, as an additional protection, typically change the RFID code every time, using a rolling system.

[7][Web: 14]

(a) *KeyFob*

(b) *Start Ignition Button*

This obstacle could be bypassed jamming the keyfob signal by passing garbage data within the passband of the receiver: this will prevent the receiver from changing the rolling code and allow the attacker to view the correct key sequence.[8]
Some other problems could be given exploiting the facts that:

- sometimes Immobilizers have the key still in memory minutes after the key has been removed
- it is possible to dump the memory of the transponder and get the secret key
- an attacker could simulate the 'lock' button press, preventing the car from locking and allowing a person to enter in the vehicle.

The last step of this evolution is the Passive Keyless Entry and Start (PKES).
This system is very similar to the RKE except that we don't neeed to 'take out of pockets' the keyfob: the only presence of the device in a short range (few meters) permits to lock/unlock doors and to start ignition by pressing a button placed inside the car.
The PKES system is particularly sensitive to a relay attack; in this attack two paired devices are placed respectively next to the car and next to the victim: the device relays the signals from the victim to the vehicle and back, enabling the attaker to start the car.
In 2005 students at Johns Hopkins University in Maryland demonstrated how to break the Immobilizer security controller without the physical key[9] and in 2010 researchers at the ETH in Zurich showed how to intercept and extend the signal of Passive Keyless Entry and Start system up to 10 meter.

[8]source: [4]
[9]source: [6]

c. Tire Pressure Monitoring System (TPMS)

In tires are embedded sensors that communicate to the car information about tires pressure, temperature and rotation, via a radio frequency (RF) transmitter.
The receiver ECU analyzes data and can send results or commands to other ECUs over CAN-bus, for example to trigger a warning message on the vehicle dashboard if the tires are deflated.
Usually sensors broadcast information automatically every 60-90 seconds but they could be forced to do it by sending a 125kHz LF signal.

Each pressure sensor contains a unique 32-bits ID, so merely eavesdropping enables an attacker to identify and **track vehicles** remotely.[12]
The unique ID permits also to **trigger additional events** when the vehicle is near: for example a system that opens authomatically garage's door when the owner arrives: in this case a radio receiver near the garage recognizes the unique signal of the car and triggers a device to open the door. 😎
The **dangerous** part is that in the same way someone could trigger evil events, like detonating an explosive, and make with excellent precision an threat to the life of people in the vehicle.☠
In closing, an attacker, broadcasting his own messages, could also alter and forge the data sent by the sensors, in order to cause **warning lights on the dashboard** to turn on without a real need. [10]

d. Vehicle to Vehicle Communication[11]

This is an emerging study, that we won't consider in the followings sections; it concerns the communication between two or more vehicles and it includes the idea of using the 802.11 type protocol to create a mesh network between vehicles.

Speaking about the long range wireless communication, probably the most important part is the cellular connection, because it provides continuous connectivity via cellular voice and data networks.

[10]source: [4], [2]
[11]source: [4]

We will overlook the others.

e. Cellular Connection

Even if the cellular channels provide a broad range of important features (e.g. hands free calling, crash reporting, remote track and disable a stolen vehicle,...), they offer many advantages for attackers: they can be accessed over arbitrary distance in a totally anonymous way, they support interactive control and data exfiltration, they are individually addressable.
For example, a hacker could disable a car's ignition the same way an anti-theft system would.
The cell phone interface supports voice, SMS and 3G data; in particular the voice channel is used for critical telematics functions (e.g. crash notification) because it's the medium with the widest service area.
In this channel, there's a software that aims to synthetize a digital channel in this environment.
Researchers of the University of California and the University of Washington[12] used reverse engineering to enter in one of those: the Airbiquity's aqLink software, very common in North America. The study shows how we can bypass the authentication step and then discover some vulnerabilities in the implementation of the code, that permits to send modified packets without being controlled by anyone.

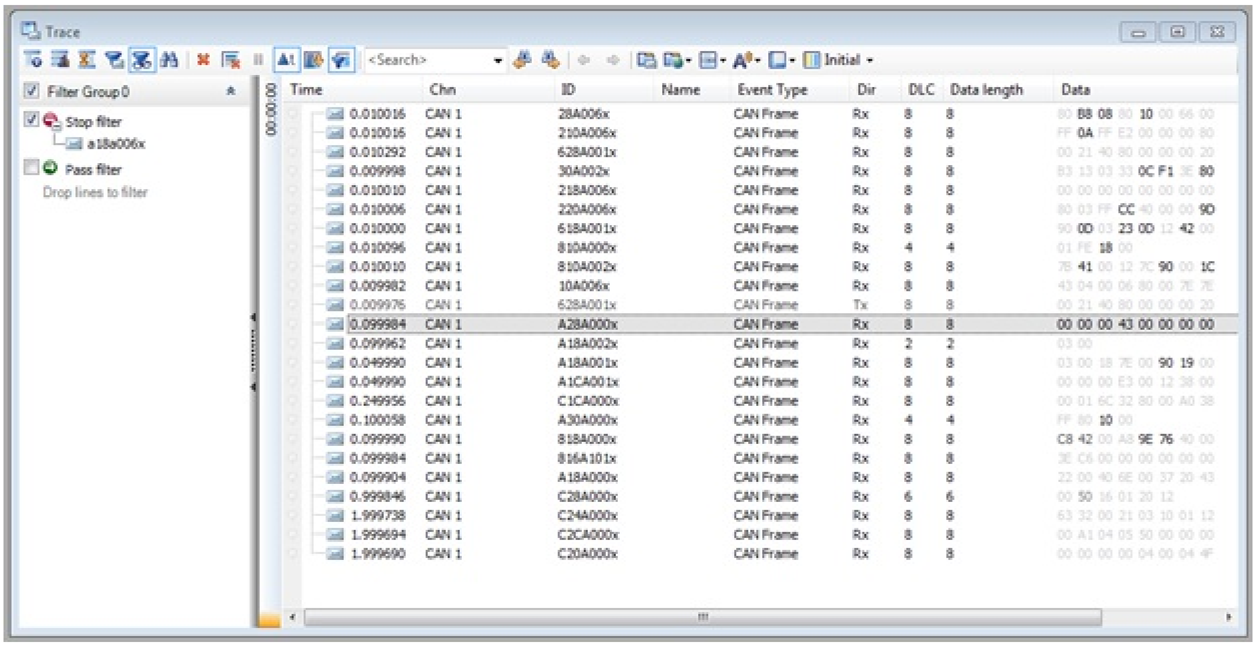
[12]source: [2]

2.2 Inject CAN data

Now the main question: if we have access to the car's network, actually what could we do?
Unluckily the answer is: with time, money, patience, and not too much knowledge, serious damages.

Software and devices exist on the market that permit to transform the binary traffic of cars in a readable form where you can see ID, length and data, usually written in the practical hexadecimal form.

The layout will be something like this: a list of packets that moves and updates continuously:



Time	Chn	ID	Name	Event Type	Dir	DLC	Data length	Data
0.010016	CAN 1	28A006x		CAN Frame	Rx	8	8	80 B8 08 80 10 00 66 00
0.010016	CAN 1	210A006x		CAN Frame	Rx	8	8	FF 0A FF E2 00 00 00 80
0.010292	CAN 1	628A001x		CAN Frame	Rx	8	8	00 21 40 80 00 00 00 20
0.009998	CAN 1	30A002x		CAN Frame	Rx	8	8	83 13 03 33 0C F1 3E 80
0.010010	CAN 1	218A006x		CAN Frame	Rx	8	8	00 00 00 00 00 00 00 00
0.010006	CAN 1	220A006x		CAN Frame	Rx	8	8	80 03 FF CC 40 00 00 9D
0.010000	CAN 1	618A001x		CAN Frame	Rx	8	8	90 0D 03 23 0D 12 42 00
0.010096	CAN 1	810A000x		CAN Frame	Rx	4	4	01 FE 18 00
0.010010	CAN 1	810A002x		CAN Frame	Rx	8	8	7B 41 00 12 7C 90 00 1C
0.009982	CAN 1	10A006x		CAN Frame	Rx	8	8	43 04 00 06 80 00 7E 7E
0.009976	CAN 1	628A001x		CAN Frame	Tx	8	8	00 21 40 80 00 00 00 20
0.099984	CAN 1	A28A000x		CAN Frame	Rx	8	8	00 00 00 43 00 00 00 00
0.099962	CAN 1	A18A002x		CAN Frame	Rx	2	2	03 00
0.049990	CAN 1	A18A001x		CAN Frame	Rx	8	8	03 00 18 7E 00 90 19 00
0.049990	CAN 1	A1CA001x		CAN Frame	Rx	8	8	00 00 00 E3 00 12 38 00
0.249956	CAN 1	C1CA000x		CAN Frame	Rx	8	8	00 01 6C 32 80 00 A0 38
0.100058	CAN 1	A30A000x		CAN Frame	Rx	4	4	FF 80 10 00
0.099990	CAN 1	818A000x		CAN Frame	Rx	8	8	C8 42 00 A8 9E 76 40 00
0.099984	CAN 1	816A101x		CAN Frame	Rx	8	8	3E C6 00 00 00 00 00 00
0.099904	CAN 1	A18A000x		CAN Frame	Rx	8	8	22 00 40 6E 00 37 20 43
0.999846	CAN 1	C28A000x		CAN Frame	Rx	6	6	00 50 16 01 20 12
1.999738	CAN 1	C24A000x		CAN Frame	Rx	8	8	63 32 00 21 03 10 01 12
1.999694	CAN 1	C2CA000x		CAN Frame	Rx	8	8	00 A1 04 05 50 00 00 00
1.999690	CAN 1	C20A000x		CAN Frame	Rx	8	8	00 00 00 00 04 00 04 4F

CAN traffic

If we are not experts of car hacking, the easiest way to try to control (or damage) the car is to ***fuzzing*** the CAN, that means sending random data to a casual ID in the list and looking for something to act strange.

Maybe, this is not really useful in this case, because the range of valid CAN packets is rather small or sometimes is a collection of packets that causes a change, in addition some of them are visible only with a moving vehicle (that makes the observation more difficult and dangerous).
Instead, fuzz testing could be really useful when we have already discovered a connection between the ID and the message or the device that will receive the packet; in fact, in this case, sending arbitrary data and observing a specific reaction or analyzing an error will let us know which one is the right input and how to control it.

But... how to make the first step?
We did a little practical experiment to figure it out.

2.2.1 The little practical experiment

First, we needed a ***car***: Magneti Marelli provided one of hers, employed for tests.
Then, a planning with objectives, procedure and materials for the test:
Objectives:

- *Recognize the CAN packets that control these actions, modifiy them, and inject them again in the network to control the result:*
  - *Control the Radio*
  - *Tachometer reading (RPM)*
  - *Lock/Unlock the doors*
  - *Door ajar*
- *Inject a packet with ID=0x and observe possible consequences.*

Looking also at articles of people that did it before me, we chose these actions because:

1. We could observe the effects without making the vehicle move
2. They are simple actions, that means a higher probability that they are controlled by a single packet, not related with others.

Then, as the packet's ID defines its priority, in theory, it could be possible to overload the channel by sending repeatedly a packet with the lowest ID (0x), preventing, in this

way, the transmission of all other messages.
Procedure:
Start recording CAN traffic, do the action, stop the recording.
Replay the record:

- *If the action repeats, figure out which packet controls it by dividing in two parts the record and play them to see where the instruction is. Repeat it until you find it. Then verify it modifying and injecting it in the CAN bus.*
- *If the action doesn't repeat, start again the record at least 2 times.*
  If it doesn't work maybe:
  - *you can't control this actions only with CAN packets*
  - *the command is on a different BUS*
  - *?*

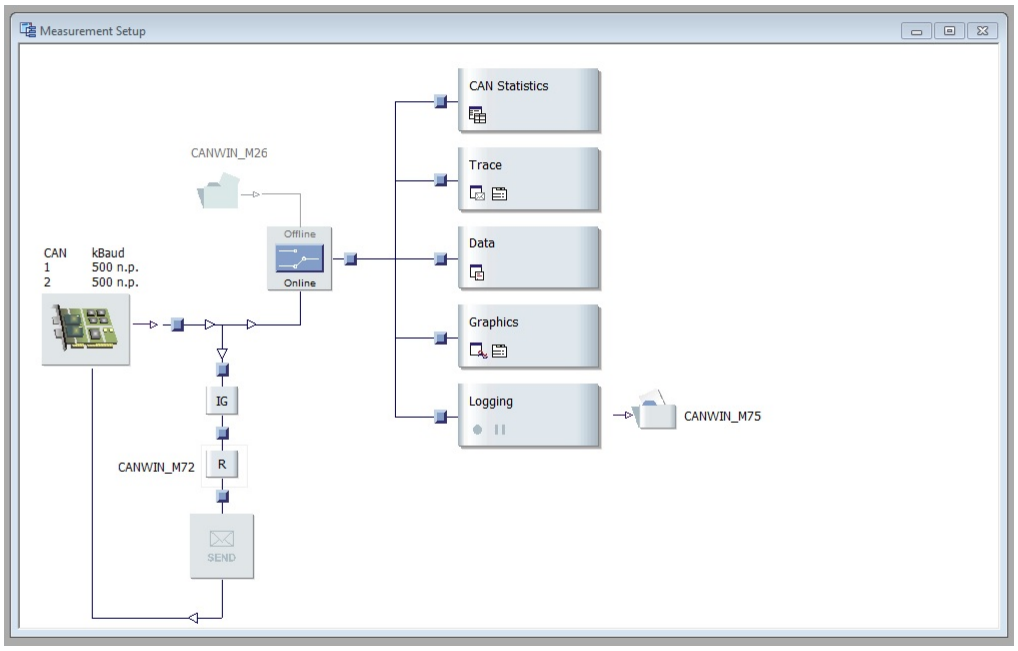
We found this modality in a study[13], and we liked it because it permits not to waste time, searching for something that maybe couldn't be found in that situation. On the other side injecting so many packets at the same time, could seriously damage the car's circuit and, if we're moving, endanger the user's safety.
Anyway, the reality is that with my car it didn't work at all, later we'll try to explain why. So we proceeded only with direct observations.

Materials:
Vector CANcase, Laptop (with the software Vector CANalyzer), cable for the OBD-II port.

Experiment:
Connected the CANcase to the car and to the laptop, we opened the software.

Then, we started the capture of CAN traffic and we observed it in the 'trace' section. The Radio was not present on the car, so we concentrate on the ***Tachometer reading****. We followed the procedure we planned and we recorded the traffic pressing the accelerator, then we injected it continuously in the BUS. Nothing happened.*

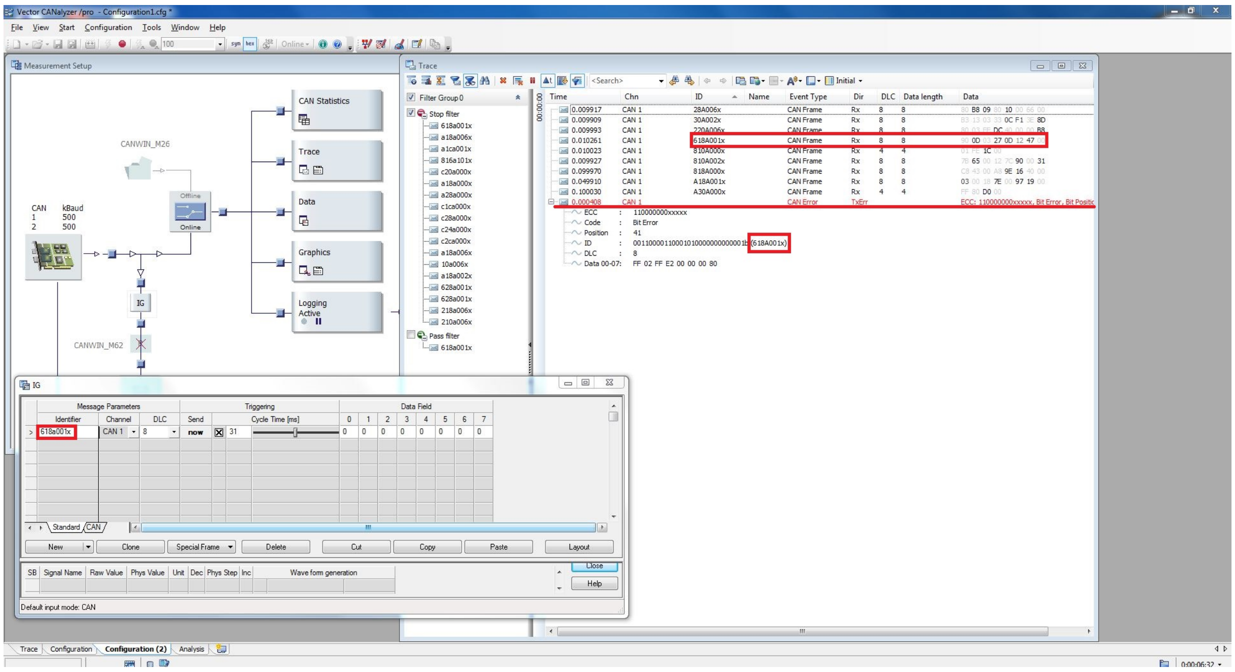
[13]source: [4]



Configuration of the software

When you inject new packets in the bus you don't cancel the original ones that are sent continuously from the real car's components, so we thought that maybe the frequency of my input wasn't so high to cover the real one and produce some effects. We tried to modify it but we couldn't reach any result, so we passed to the direct observation.
We observed for each ID the effect of pressing the accelerator, and, if we reached the conclusion that the ID was not the one we were looking for, we filtered it.
Finally the choice was between less than 5 messages, all possible candidates: we randomly selected one.
It was an 8 bytes length packet, but not all of them changed pressing the pedal so in reality we modified three bytes, putting a higher value, and then we sent it with a frequency of 1 every 20 millisecond (ms). We noticed a variation of the RPM indicator, so we increased the frequency until 1 every 10 ms, to have the confirmation that was the right one.
The result was the trembling of the indicator around our value.
We had to put the maximum frequency supported by my device (1 every ms): to obtain the indicator's complete stability on our value.
This first little victory took me more or less 40 minutes (it was the first time after all!).

Later, with another packet, we tried to modify also the fixed bytes, but the consequence was the appearance of an error, and none of my data were sent.


```
Time      Chn    ID        Name  Event Type  Dir    DLC  Data length  Data
0.009917  CAN 1  28A006x         CAN Frame   Rx     8    8            80 B8 09 80 10 00 66 00
0.009909  CAN 1  30A002x         CAN Frame   Rx     8    8            B3 13 03 33 0C F1 3E 8D
0.009993  CAN 1  220A006x        CAN Frame   Rx     8    8            80 03 FE DC 40 00 00 B8
0.010261  CAN 1  618A001x        CAN Frame   Rx     8    8            90 0D 03 27 0D 12 47 00
0.010023  CAN 1  810A000x        CAN Frame   Rx     4    4            01 FE 1C 00
0.009927  CAN 1  810A002x        CAN Frame   Rx     8    8            7B 65 00 12 7C 90 00 31
0.099970  CAN 1  818A000x        CAN Frame   Rx     8    8            C8 43 00 A8 9E 16 40 00
0.049910  CAN 1  A18A001x        CAN Frame   Rx     8    8            03 00 18 7E 00 97 19 00
0.100030  CAN 1  A30A000x        CAN Frame   Rx     4    4            FF 80 D0 00
0.000408  CAN 1                  CAN Error   TxErr                    ECC: 110000000xxxxx, Bit Error, Bit Positic
    ECC        :  110000000xxxxx
    Code       :  Bit Error
    Position   :  41
    ID         :  001100001100010100000000000001b (618A001x)
    DLC        :  8
    Data 00-07:   FF 02 FF E2 00 00 00 80
```

```
Identifier  Channel  DLC  Send  Cycle Time [ms]  0  1  2  3  4  5  6  7
618a001x    CAN 1    8    now   31               0  0  0  0  0  0  0  0
```

We proceeded in the same way to recognize the other commands: unluckily we didn't succeed.
We spent another hour in the car, but we couldn't find any clear connection between CAN packets and the light indicating 'Door ajar', or the command to lock/unlock the doors.
Instead, we found connections between packets and turn on/off lights, put on/off the safety belt, insert/disable the city and the handbrake. Unluckily none of them if replayed obtained any consequence. My hypothesis is that the packets we found were part of the information statement or an incomplete command that didn't trigger any reaction (i.e. turn on/off the light of the handbrake on the instrument panel).

Finally, we injected the 0x ID packet but, also in this case, nothing happened. We tried to modify the bytes of data, in order to find the right message to overload the channel, but the other packets continued to be sent, without any consequence.
The manufacturer probably prevented this kind of attacks, for example making the car

ignoring the 0x ID at all or assigning it some specific data.

The table below summarizes the discoveries:

ID	***Data***	***Command***	***Description***
A18A001x	03 00 18 80 **03 62 4F** 00	**Tachometer reading**	**03 62 4F** = RPM
C1CA000x A18A000x	00 01 6C 33 40 00 A0 **A8/38** 22 **20/00** 40 6E 00 37 20 43	**Light on/off**	**A8**, **20** = lights on **38**, **00** = lights off
C1CA000x	00 01 6C 33 40 00 **A0/B0** 38	**Safety belt**	**A0** = belt inserted **B0** = belt desabled
A18A000x	22 **02/00** 40 6E 00 37 20 43	**City**City Option	**02** = City on **00** = City off
A18A000x	**22/02** 00 40 6E 00 37 **20/00** 43	**Handbrake**	**22, 20** = Handbrake on **02, 00** = Handbrake off

Observations and Conclusions:
We can notice that some IDs are repeated for different devices, but the bytes involved are different: so we can hypothesize that their message is not the one that triggers the event, but it reports only the state of the objects.

In conclusion, we have seen how with zero experience, and little knowledge, we could start to understand something simple about CAN traffic.
Next step is of course the hardest one, and it requires more time and patience, but the researches in next pages will prove that is not impossible.
We wanted to continue our experiment, obtaining, for example, the control of the speed reading, but we couldn't do it, because of safety reasons.
In fact, finding that kind of messages demands a moving car, and tamper a moving car should happen only in a controlled and safety context (e.g. with the car immobilized on jack stands), in order to protect reasearchers' safety.
Since this was not possible, in next section we will restrict our seach to explore what other people had been able to do in months of work.

2.2.2 A case study

Once understood to whom each ID is related and what each packet or sequence of packets can control, it's time to act and take the control of the car.
Of course, not everything can be controlled via CAN bus, for example [14] in the Ford Escape the packet with ID 0200 has a byte that indicates how much the accelerator is depressed; contrary to what we could think, replying this packet with different values, doesn't change the car's acceleration or speed. This is because the packet is sent out from the Power-train Control Module (PCM) to the Antilock Braking System, probably to figure out if there's a traction control event in progress, and, if changed, it doesn't produce any consequence to the throttle.
Furthermore, each car is different, so it's not obvious that if something works for one car it has to work in all vehicles, not even in a car built by the same manufacturer. In particular the devices and the optionals embedded in the car play an important role: the cellular connection, the Antilock Braking System, the Cruise Control, the Pre-Collision System, etc. Usually, because of the large number of parameters that these systems have to receive, their presence or their absence entails a different network, and for us, a network with a dense presence of interconnections between components means, during an attack, an higher probability to succeed.

In 2013, Charlie Miller and Chris Valasek, researchers at DARPA (Defense Advanced Research Projects Agency), wrote in a report, '*Adventures in Automotive Networks and Control Units*', the experiments and the observations they made hacking a Ford Escape and a Toyota Prius.
They injected CAN packets directly in the CAN bus via physical connection between laptop and OBD-II port.

Inject Normal CAN packets
These are the results they reached for the Ford Escape, injecting **normal CAN packets**:[15]

[14]source: [3]
[15]see [3] for all details.

Command	*What they discovered*	*What they did/Consequences*
Door Ajar, Lights	The messages that indicate lights' and doors' state on the dashboard	Make appear the messages even if door are closed and lights off and vice versa.
Speedometer	The message that controls the speed's indicator.	Move the speed indicator indicate any value they want.
Odometer	The message that increases the odometer's value.	Write a short program and make grow the odometer's value as they want.
On board Navigation	The messages that report where you're going.	Write a program that sends fake information about car's position
Limited Steering	The overload of the CAN network prevents the delivery of CAN messages ⇒ in particular: Power Steering Control Module shuts down.	Most of the dashboard's lights turn on. The car can make only gradual turns: the wheel can't move more than ∼ 45%.
Prevent Ignition	The ID 0000 has the highest priority.	If played continuously before the car is started the vehicle doesn't switch on; after, it overloads the network.
Stearing	The message that the PSCM uses to control the steering, when the Parking Assistant Module (PAM) is auto-parking.	Write a program that replays the curve of the steering wheel and use it to steer the wheel to any position.

And these are the results for the Toyota Prius:

Command	*What they discovered.*	*What they did/Consequences.*
Speedometer	The message that controls the speed's indicator.	Move the speed indicator indicate any value they want.
Braking	The message that control the braking of the **Pre-Collision System (PCS)**	Send the messages at any time to make the car slow down, prevent the acceleration and even stop.
Acceleration	The accelerator pedal is not directly connect to the Engine Control Module/Throttle Body Controls, but the Power Management Control receives the physical signals and transmit them to the ECM using CAN messages.	Since this message is not viewable all over the network (e.g. not from the OBD-II port), they should connect directly to the Power Management ECU to catch it and reproduce it to tamper the acceleration.

Steering: *at speed < 4mph* *at any speed.*	The Intelligence Park Assist System uses a combination of two messages to control the steering, but only if the speed is < 4mph. Tampering the steering at any speed is more complicated: it requires the use of 2 cables at the same time, one that sends bogus speed messages and the other the steering control messages.	Write a program that uses those messages and control the steering when the speed is < 5mph. Anyway they don't obtain the complete control, the results are some sporadic jerks of the wheel, which cause vehicle instability.
Steering: *with the option: Lane Keep Assistant (LKA)*	The message used by the LKA is designed to be used at any speed, but it can't turn the wheel more than about 5 degrees.	Use this message to turn a little bit the wheel: it could be dangerous when we're driving fast on a small road or in traffic conditions.

Inject Diagnostic CAN packet

The researchers showed also what we can do controlling **Diagnostic CAN packets**: but, before we can perform most diagnostic operations on a ECU, we need to authenticate against it, using the **SecurityAccess process**, seen at page 7. The difficulty of this step depends on the car and on the single ECU.
For example, in the ***Ford Escape***, authentication against the Parking Assistant Module (PAM) is quite easy, in fact, after the SecurityAccess request, the PAM always sends the same seed, that means that the response also is always the same. So, if we sniff the CAN traffic of a tool performing a SecurityAccess against the PAM, we can just replay it and be authenticated too.
On the contrary, the other Ford ECUs change seed every time, to prevent the replay attack, so, apparently, the only way to be authenticated is to get the secret key. This could be accomplished by extracting the firmware and reversing the key out of it, or, more simply, by reverse engineering the actual Ford Integrated Diagnostic Software (IDS) tool.
This device and the license to use it could be bought (the total is quite expensive) by anyone and used to auto-check the state of his own car. It can't be used to authenticate directly against all ECUs in the car (only a few), but it has the capability to do so: in fact it contains a file with all keys of car's ECUs (in plain). They are 407: once we have found the file, we could just try them all to get the one we need.
Furthermore, in the IDS tool there are also some proprietary services, used for cars

check-up; reverse engineering the IDS tool permits to analyze and take the control of these functions too:[16]

Command	***What they discovered***	***Remarks***
Brakes Engaged	The IDS function: *DiagnosticCommand_B1*, used to check the brakes, contains a message that engages the brakes.	It could be used to engage the brakes, only if the car is already stopped, and it will prevent the motion, even if we push hard on the accelerator.
No brakes	Similar to the last one, another *DiagnosticCommand* bleeds the brakes, preventing the brake pedal's physical depressing.	The message is considered valid only if the car's speed is $<$ 5mph; even if at these low speeds it doesn't seam too dangerous, it could be enough to cause incidents.
Lights Out	The message that shuts down Smart Junction Box (SJB) $\Rightarrow$ any device that depends on the SJB stops working: radio, Heating, Ventilating and Air Conditioning (HVAC), **HEADLIGHTS**, **BRAKE LIGHTS**,etc.	The attack could be carried only when the vehicle is stopped, but **it continues to work after that, even if the car is at speed**.
Lights Flashing	Erasing data on the SJB (i.e. while reprogramming it) causes all car's lights break, except the interior ones, that are permanently turned on.	The situation continues **permanently even when we stop sending packets and even if the car is restarted**, until a new reprogramming of the SJB is made.
Kill Engine	The message that **kills the engine** and prevent to start up the car until we stop sending it.	We could shut down the engine **at all times** and **at all speeds** ! **N.B.: We don't need to establish a diagnostic session before using it.**

Speaking about the **Toyota Prius**, for all vehicles that could be sold in North America, is available a software: '*Toyota Techstream*'[17], supporting a J2534 PassThru device, that permits to control some diagnostic actions, simulate active tests and reprogram some ECUs.

As for the Ford Escape, Charlie Miller and Chris Valasek used reverse engineering to discover some of the commands used in Toyota's diagnostic session. They observed

[16]Remember: we should authenticate before performing any of these actions.

[17][Web: 15]

that only few diagnostic functions (specifically the ones that re-flash ECUs) required a previous SecurityAcces.
Breaking the Toyota's SecurityAccess by brute force is not useful cause the seed changes after a specific number of wrong attempts (usually 10), so, as in the previous case, it's necessary to reverse out the secrets from the firmware or the Toyota Techstream software. Since it is less complex, they chose the second way, let's see what they discovered:

Command	*What they discovered*	*Remarks*
Braking	The messages that teste the solenoids within the ABS and the Electronically-Controlled Braking System (EBS).	?
Kill Engine	Two messages that **kill the engine**.	One works only if the car is in park state, the other could be used **at any time**.
Lights On/Off	The messages that turn on/off the headlights	It works only if the car is in 'auto' state
Horn On/Off.	The messages that turn on/off the horn	The horn could be turned on forever as long as the packet is sent: it will continue even if the car is turned off.
Seat Belt	The message that tests the ability of the PCS to pre-tighten the seatbelts in the event on an impending accident.	Pre-tighten the seatbelts at any time.
Doors Lock/Unlock	The messages that lock and unlock car's doors.	They could be used at any time. The locking one doesn't prevent the doors from being physically opened from the inside.
Fuel Gauge	The message that controls the fuel gauge	Modifying the message the fuel gauge could indicate an arbitrary value.

Re-flash ECUs

Furthermore, using diagnostic messages, it is possible **to reprogram some ECUs**.
For example, in the Ford Escape, with the 0x34 Diagnostic Service, the **RequestDownload**[18], the researchers were able to upload their own code to the PAM and to the SJB, make it memorize by the ECUs and then make it execute, calling the Diagnostic Service: RoutineControl.
Their code make the PAM read and write arbitrary CAN packets, which, as we've seen,

[18]The Diagnostic Services have to be read from ECU's point of view, so the 'Download' service actually means upload something to the ECU

can be used to control the vehicle in different ways.
Luckily reprogramming an ECU in the Ford Escape, even if it follows more or less the Diagnostic Standard, is not very easy.
Going over all the steps that took us until here, we should:

1. Extract the ECU's firmware or obtain the Ford Integrated Diagnostic tool
2. Reverse engineer the software or the firmware in order to find the ECUs-IDs connections and some diagnostic commands that we want to implement
3. Find the key or, if it exists, another way to bypass the SecurityAcces for the ECU we want to attack, in order to open a diagnostic session
4. Understand how the RequestDownoload service of the ECU works: in particular which parameters it requires, in which form the data should be sent, which address we should write in order to make data check by the RoutineControl,...
5. Find if the data are controlled by the ECU, e.g. with a checksum, and eventually modify them in order to cheat it
6. Call the RoutineControl in order to check data and make them execute.

For the Toyota Prius the situation is more difficult: the ECUs re-flashing process integrates some manufacturer's own functions to the Diagnostic Standard. A consequence is, for example, that the RequestDownload way, used before, doesn't work for this car.
The reverse engineering complexity grows, and the security too.
However Miller and Valasek succeeded also in this conditions, observing the process directly.
Some software allow the ECUs reprogramming, in particular they used the Toyota Calibration Update Wizard[19], that supports the *.cuw* files and works with the Teachstream, seen before.
They downloaded a new legitimate calibration update for the ECU they wanted to reprogram (the ECM). They installed it and they observed the exchange of CAN messages, in order to understand the method and reproduce it.
The CAN traffic shows an initial SecurityAccess exchange, but then alternates some custom-built messages to some diagnostic functions (e.g. 0x76 GetMemoryInfo, 0x26

[19][Web: 13]

EraseBlock, 0x45 WriteBlock, ...).
To understand the costum-built ones, they needed to reverse engineer the software, extract the calibration *.cuw* file and read it to get information such as number of calibrations, the new calibration ID after the update is applied (used by the ECM to control that the version to upload is newer than the previous one; if it's not the case it will kill the process), the IDs they should use during the process, etc.
Furthermore they discovered that after the SecurityAccess the ECU requires another value, like a password, in order to continue. This data is written in the calibration file and it changes, in an unknown way, every time.
So, in order to re-flash the ECM we should reverse every time the calibration file of a new legitimate update, to obtain the new ID and 'password'; the ID and value of a previous update won't be considered valid by the ECU.
Speaking about other ECUs the process is similar and requires the same efforts, summarized as a whole below:

1. Extract the ECU's firmware or obtain the Toyota Teachstream Software
2. Reverse engineer the software or the firmware in order to find the ECUs-IDs connections and some diagnostic commands that we want to implement
3. Find the Key to pass the SecurityAccess
4. Obtain a reprogrammation software, like Toyota Calibration Update Wizard, and reverse engineer it to understand the custom-built dialogue and to satisfy all the ECU's requests
5. Use correctly 6 different diagnostic functions of the standard to write and verify data.

Not impossible ... but what an effort!

2.2.3 Other Damages

Other few, less detailed, practical studies[20] have been made about this argument. The attacks that succeeded, with their dangerous consequences, are now reported. The entry point is always the OBD-II port and, when is known, it's specified if normal or diagnostic packets are used.

1. *Radio*

   The complete control -and disable user control of- the radio and to display arbitrary messages.

   ⚠ An attacker could increase the volume and prevent the user from resetting it. He could control other car's sounds that are overseen by the radio: turn signal clicks, seat belt warning alert,...

2. *Instrument Panel Cluster*

   The complete control of the Instrument Panel Cluster.

   ⚠ An attacker could falsify the fuel level and the speedometer reading, modify the illumination of instruments and display arbitrary messages

3. Body Controller

   The control of the Body Control Module's functions.

   ⚠ An attacker could lock/unlock doors, jam the doors locked, open the trunk, control interior and exterior lights, honk the horn, control window and wipers, continuously shoot windshield fluid,...

   ℹ In order to control the BCM he needs to reverse engineer its packets on the low-speed bus, and fuzzing packets on the high speed bus.

4. *Engine*

   Implement some Engine Control Module functions.

   ⚠ An attacker could disturb engine timing by resetting the learned crankshaft angle sensor error, kill the engine and prevent restart, ...

   ℹ These commands were found fuzzing DeviceControl requests to the Engine Control Module.

[20]source: [1]

5. *Brakes*

   Discover the Electronic Brake Control Module functions and messages.
   ⚠ An attacker could engage one or more brakes and lock them preventing the manual override, even through a battery removal.
   ℹ These commands were found fuzzing the EBCM.
   In some cars the EBCM doesn't require any safety control in order to implement these attacks (e.g. Toyota Prius), in others it requires a SecurityAccess when the speed passes the 5mph.

6. *Heating Ventilation, Air Conditioning (HVAC)*

   Control of the cabin environment.
   ⚠ An attacker could turn on/off the fans, the A/C and the heat.
   ℹ In some cases, the manual override was not permitted.

7. *Generic Denial of Service*

   Disable, at arbitrary time, communication from/to an ECU, overflowing the bus.
   ⚠ From/to the ECM: it reduces the reported speed at zero.
   From/to the BCM: it freezes the IPC in its current state; the car could be turned off, but not restarted again.
   ⚠ An attacker could prevent the car to be turned off
   ℹ He could do this by activating the ignition output of the BCM and overriding the key lock solenoid, preventing the key to be removed or allowing the removal while the car is in drive.

2.3 Other attacks

Injecting CAN messages is certainly extremely dangerous but it's not the only threat we should be protected from: for example, installing counterfeit components could be very risky too.
An example?
The U.S. Department of Transportation's National Highway Traffic Safety Administration (NHTSA) had issued some safety advisories for drivers and repair professionals to dissuade them to buy not certified Airbags for their cars.
In fact, in case of accident, against regulations Airbags, instead of inflating, could explode, becoming more dangerous then the accident itself.
Here the warning video: `https://www.youtube.com/watch?v=uEYExJhYbg8`
The NHTSA in 2012 estimated that 250 thousand of counterfeit airbags had been sold in U.S. up to that moment.[21] ⚠
Moreover, the 31 August 2013, the Times of India related that up to 20 percent of all road accidents in India were due to counterfeit auto parts.[22]
From an economic point of view, the General Motors Corporation reports that the counterfeit auto parts is a worldwide business of around 12US$ billion, that means an equal loss for the automotive industry, and it causes a job loss of 250,000 places. [23]

[21] [Web: 11]
[22] [Web: 16]
[23] [Web: 12]

Conclusions:

How serious are the threats?

In this first part, we have seen how a spiteful person could enter in our car's network, using a physical connection, direct (a cable) or indirect (a WMA file, an App,...), or a wireless channel (Bluetooth, cellular channel,...); then we discussed which damages the injection of CAN packets could trigger and how, and finally other security and economic damages.
Now, to complete the puzzle, we should ask ourselves:

How serious are the threats?

Speaking about messages' injection, the answer isn't clear: we can't really know how much time, efforts and tests have been done to reach those results.
What we could say is that some attacks are easier than others: the ability to track a vehicle, for example, doesn't require a real reverse engineering work, that means that probably not so much time (on the order of weeks) is enough to succeed.
On the other hand, all those attacks that demand a reverse engineering effort, belong to a higher level of difficulty, and their complexity is directly proportional to the quantity of data that needs to be reversed: the time is at least several months for diagnostic messages and something less for normal ones.
Furthermore, usually, diagnostic messages can be seen only during a check-up situation, so the direct observation is quite rare and mechanics' devices and software are required.
Moreover, some security and authentication controls need to be bypassed before sending diagnostic commands.
Then, the price of the equipment is high; and the risk of seriously damaging the test car, with consequent high-priced costs to the mechanic, too.
Otherwise, the web community of car's hackers is extended and really active. Finding

the software descriptions and specifications, and starting the reverse engineering is not a job for an inner circle anymore.

The system is not impenetrable, on the contrary it presents several exploitable bugs, especially in the implementation of the diagnostic standard, and underestimating the problem can be lethal.
The *American Corporation Caterpillar* knows something about this, since the Department of Homeland Security for demostration purposes has been able to use a cyber-attack causing the self-destruction of a large diesel generator.
This is the video released by the CNN that reports the experiment:
`http://www.cnn.com/2007/US/09/26/power.at.risk/index.html#cnnSTCVideo`
This was happening in 2007 ... after 7 years of development, we can only imagine what people could be able to do now!
The field of cyber security is in rapid expansion, and the Department of Homeland Security is not the only one that works on it: a lot of enterprises all over the world are developing hacking researches, and consequent solutions, in order to contain the problem.
Most of the security is now committed to the morality of those people.
The situation could rapidly collapse if criminal and terroristic organizations start to work on the subject, too. In fact, even if the effort needed to succeed (or the information's price) is really high, the recompenses are priceless.
A new entire **'theft business'**, that tracks, opens and steals vehicles, could be born; and the dealers are already gaining no little advantages **decreasing car's mileage** before selling it.
Furthermore: what would happen if suddenly all cars in Manhattan shut down in the same moment? I can't really imagine, not only the inconveniences, but the **economic loss** of this terroristic act.
Again, any citizen driving a vehicle won't be safe anymore from this kind of life, kidnapping and extortion attacks.
Finally: given the high number of daily car crashes, it will be complicated to know for certain if a tragedy is a real accident or the result of a car's tampering; and, above all, the fact that **the injection of CAN messages, in general, doesn't leave any trace**

and it's totally anonymous will make investigation work impossible ⚠.
Afterward it won't be impossible to transfer the experience acquired to focus on targets different from cars, like plains, military vehicles and tanks, compromising the security of entire nations.
Lastly, but not for importance, we remember the real and current economic (12US$billion), safety, and job loss (250,000 places) due to the **black market of counterfeit auto parts**.[24]

Surely the list is not complete, but it's enough to put in evidence the need to protect car's component from any criminal pursues.
Next section is committed to the current solutions on today's market, and in particular to their use of Cryptography.

[24][Web: 12]

Part II

Current Solutions to the Automotive Security Problems

Short Introduction

The previous study highlighted the need to develop a security system to protect next cars generation.
Now, we will list some reasonable points that represent our expectations about future cars (of course they are all related to each other):

- **Authentication between car's components** (sensors, ECUs, devices,...):
  - to prevent an unauthorized entity to act as an authentic one in order to reduce the counterfeiting risk
  - to prove sender's identity, for each message, in order to ignore fake CAN packets.
- **Secure communication on the CAN bus**
- **Detect abnormalities in the CAN traffic**:
  - to alert the driver in case of tampering
- **Secure wireless communication**:
  - with Immobilizer and TPMS
  - to prevent the injection of malicious CAN messages by a remote user
  - to allow additional advantages, like secure firmware updates via WiFi or cellular channel
  - between vehicles (V2V)
  - between a vehicle and another entity (V2X)
- **Protect data sets**

- mileage
- engine maps [25]
- fee-based services and features.

Security is a complex problem, and pretending to find an easy solution is not realistic, but automotive companies all over the world are studying the subject and proposing solutions more and more progressive.
So, let's take a look at some proposals of today's market.

[25]e.g.: little modifications of engine maps values could increase the speed and the pollution, bigger ones could damage seriously the engine and endanger user's safety.

Chapter 3

SHE specification

In 2008 the Hersteller Initiative Software (HIS, a group that involves the main German manufacturers) presented a new specification called **SHE** (Secure Hardware Extension). SHE describes a small hardware extension for adding essential security functionality (e.g. protection of cryptographic keys, hardware crypto module, or secure boot) to standard automotive microcontrollers.
It is considered the first specification that tries to answer to the previous security problems and sets a standard in automotive security environment.

The objectives of SHE were:

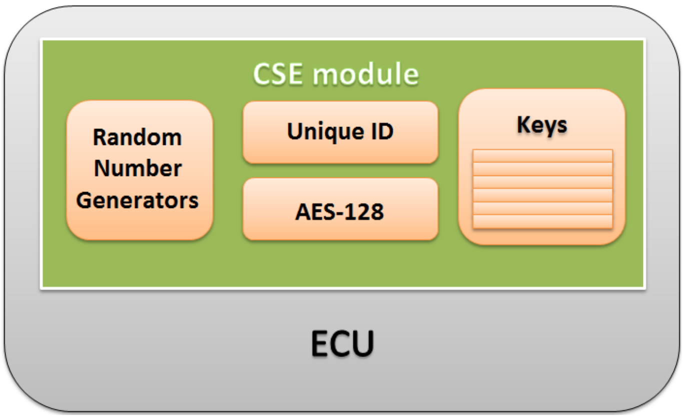
- Protect cryptographic keys from software attacks
- Provide an authentic software environment
- Make security depend only on the strength of the cryptographic algorithm and the confidentiality of the keys
- Allow for distributed key ownership
- Keep the flexibility high and the costs low

And the specifications:

- Hardware implementation of a crypto algorithm acceleration (AES-128)
- Secure Boot mechanism to verify custom firmware after reset
- 19 security specific functions
- Up to 10 general and 5 special purpose crypto keys

In 2009/2010 the American multinational corporation Freescale Semiconductor[1] developed the **Cryptographic Services Engine (CSE)** module, that implements the HIS SHE-Specification.
The CSE is just an example of implementation of the SHE specification, and it's not the only one; the following CSE's and protocols' description, even if referred in particular to this tool, could be related to any module that implements the SHE standard.

The CSE is a module embedded in the ECU, that could count on: his own core, a unique 120-bit ID, the AES-128 algorithm (with Electronic Codebook (ECB) and Cipher-block Chaining (CBC) modes), some secret 128-bit keys, shared with the components it has to communicate with, and random number generators.
In a very very simple way, we could imagine it like this:



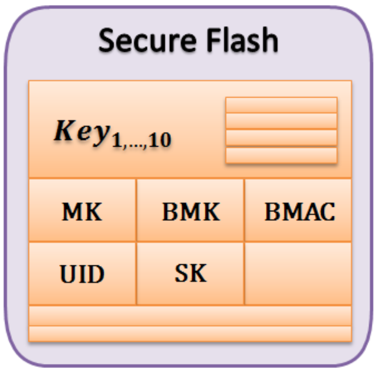
CSE module

Of course in the CSE there're a lot of other components (memory tools, interfaces to communicate with the ECU,...), but, since we are not electrical engineers, we will overlook them.
Furthermore we will consider that the communication inside the CSE is secure and that an attacker can't penetrate in the module and have access to secret information (e.g. the keys).

[1][Web: 12]

3.1 Secret Keys and Secure Flash

Speaking about the secrets keys, in the figure above there's an imprecision: in reality they are all stored in the secure part of the flash, and the Unique ID too.
In particular the '**secure flash**' contains:



1. 10 user keys
2. the Master Key (MK)
3. the Boot MAC Key (BMK)
4. the Boot MAC (BMAC)
5. the Unique ID (UID)
6. a Secret Key (SK)

1. **User Keys** are programmed by the user, and they are needed for message encryption/ decryption or authentication. An option permits to write them in an indelible way, to prevent their modification: the process is not reversible.
2. **Master ECU Key** could be programmed and written in an indelible way by the user; this key permits to update other keys without knowing them, if they are not written in an irreversible way.
3. **Boot MAC Key** is used to calculate the: MAC[2] value of the boot code; the key and the value could be written in a indelible way too.
4. **Boot MAC value** is calculated using the previous key.
   A computation of the boot MAC value is made every time we start the car, followed by a check against the stored value in order to prevent any modification. If it's not written in an irreversible way, we could modify the boot code and the corresponding boot MAC value using the correct keys
5. the **Unique ID** is written in an irreversible way by CSE's manufacturer
6. the **Secret Key** is a random number written in an indelible way by CSE's manufacturer,

[2]MAC (Message Authentication Code): given a message m, its MAC is $f(m)$ where f is a MAC function, that is a random association between all the possible input and output of n bit. [10]
In our case $MAC(m)$ is the last block of the encryption of m with AES-128.

and used by the Random Number Generators.
Each key has an ID to identify it in an unique way, and a counter that should be increased during an uploading, to prevent a malicious update with replay attack.
In case of compromised keys, knowing all of them, it's possible to erase all those that are not written in an irreversible way and set the secure flash back to factory state. We will see later [3] how to write data in an irreversible way.

3.2 Authentication SHE Compliant Protocol

Now, let's see an example of how we can use a SHE compliant system to implement a 'challenge- response' protocol, in order to verify the identity of an ECU.

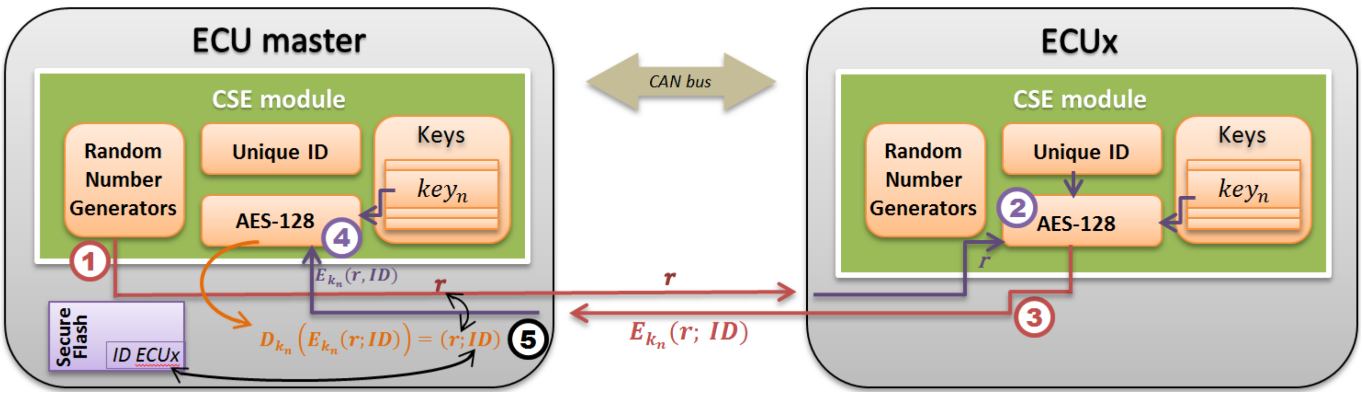
We suppose that the *ECU master* wants to prove *ECUx*'s identity:
key_n is a 128bits secret key, shared by the two ECUs

1. **ECU master** calculates a random challenge r thanks to the random number generators, and sends it to **ECUx**
2. **ECUx** uses AES-128 and the key key_n, to encrypt together r and its ID
3. **ECUx** sends $E_{key_n}(r; ID)$ back to ECU master
4. **ECU master** decrypts the message received, calculating $D_{key_n}(E_{key_n}(r; ID))$, and it obtains (r;ID).
5. **ECU master** verifies if r is the same random number it sent before and if ID is ECUx's ID.

3.3 Comments and Security Observations

The previous protocol does not specify in which mode is implemented the encryption with AES-128. We said before that the CSE can support the **ECB** and **CBC** modes for block cipher: we recall that the first one divides the message in 128-bit blocks and

[3]Appendix B 69: OTP



Challenge-Response Authentication

ciphers each block independently from the others; the second one divides the message in 128-bit blocks and it encrypts each block summing it with the previous coded one. If we want to use this last modality we have to choose a starting vector IV (Initialization vector), that we will sum with the first block: it could be fixed, or different each time and it can be public.[4]

The main difference between the two modalities is:

- ECB:
  - using the same key, encodes equal plaintexts into equal ciphertexts
- CBC:
  - using the same key, if the IV is different, equal plaintexts are enciphered into different ciphertexts (this is not the case, if IV and the key are fixed)

Both of them can encode only blocks of **fixed length of 128-bit**.

If the key is fixed, Using ECB or CBC with fixed IV reveals information about the frequency of the original plaintext, so if the coding purpose is to give less information as possible to a malicious observer, the CBC mode with random IV is allowed.

In our case, we could think to use the random number r as IV, but the result will be always the same: to equals plaintexts (i.e. equal $r \Rightarrow$ equal IV) we will have equal ciphertexts. In this case, to prevent a replay attack, the sequence of random numbers should never be repeated: as this is guaranteed for cryptographic random number generators, the solution is valid.

[4]source: [10], [11]

Anyway, if we are not worried by a frequency analysis, we can pass over the problem.

Speaking about AES-128, the main proof of its security is that none, in 13 years, has never discovered an attack to find the secret key considerably more efficient than the brute force one (i.e. check every possible key combination).
This last attack requires to find it between 2^{128} ($\sim 3.4 \cdot 10^{38}$) possibilities, whereas the best recovery key attack to AES-128 ever implemented decreases the computational complexity to $2^{126.1}$ ($\sim 9.12 \cdot 10^{37}$).[5]
The fastest computer in the world, up to November 2014, is the China's Tianhe-2, built at the China's National University of Defense Technology: it can do $33.86 \cdot 10^{15}$ Flops (Floating point operations per second)[6]; knowing that AES-128 takes about 1000 operations to check a key, we could try $\frac{33.86 \cdot 10^{15}}{1000} = 33.86 \cdot 10^{12}$ keys per second, using it.
This corresponds to: $\frac{2^{126.1}}{33.86 \cdot 10^{12} \cdot 31536000} = 8.54 \cdot 10^{16}$ years.

[5][18]
[6][Web: 28]

Chapter 4

Immobilizers Authentication

Since 1994 BMW added in his cars an anti-theft technology called **"EWS"** (Elektronische Wegfahrsperr). EWS means literally Electronic Drive Away Protection, but today it's better known as **Electronic Immobilizer**. Being an immobilizer, its scope is to prevent to run a car without the correct key[1].
Starting from the first version in the 90's the EWS evolved until today to EWS III.
To give an idea of how the system works we report the current BMW description of his tool:

> *'The electronic immobilizer secures your car using a chip with an electronic code integrated in the car key. This code consists of a permanent* ***personal code*** *(1) and a* ***second code*** *(2) changed by the immobilizer each time you start the engine. Whenever the ignition is switched on, the immobilizer first reads the personal code and then asks for the changing code. If both answers are correct, the immobilizer will send another coded signal to the Digital Motor Electronics* ***(DME)*** *to unlock the engine. Without which the engine cannot be started - not even by short-circuiting it.*
> *This data is transmitted wirelessly via an aerial in the steering wheel lock and the key with integrated remote control.*
> *The key is equipped with a maintenance-free battery which recharges automatically while driving. And should you ever lose your key, you can have it deactivated by*

[1]see also page 20

your authorized BMW dealer and receive a replacement key without delay' [2].

We found the detailed description[3], of the immobilizer protocols of a BMW E_38; this car is not made any more since 2001 and we don't know exactly how the process is changed in these years for BMW and cars in general; however, laying together different information caught on the web[4], we can figure out how modern protocols work.
Two different processes are now described:

- one **to authenticate the key to the EWS**
- the other **to authenticate the EWS to the DME**

4.1 Key to EWS Authentication

4.1.1 1998 Original Protocol

This protocol prevents the usage of an unauthorized key to run a car.
The original process needs:

1. an **EWS Control Module ECU**, that communicates wirelessly with the key (e.g. via radio waves) and wired with the DME. The EWS has a unique identification number: the **VIN** (Vehicle Identification Number)
2. a **key identification code (key ID)** that identifies a key in a unique way.
   This number is used by the EWS to verify if a key is correct and enabled: the EWS stores indelibly in his memory 10 different enable key IDs, 4 of them correspond to the 4 keys delivered with the vehicle; the others to the 6 additional keys that may be ordered as replacement. If we lose a key, his ID in the EWS could be deactivated, making the lost key harmless.
3. a **password** shared between the key and the EWS
4. a **changing code** sent by the EWS to the key

The original operation of key authentication follows these steps:

[2][Web: 18]
[3][Web: 19]
[4][Web: 20], [Web: 21], [Web: 22]

1. When the key is inserted in the car, the EWS module sends a 125kHz AM signal to the antenna in the key
2. The signal powers up the key's transponder, that sends the **key identification code** to the EWS
3. The EWS verifies the key ID and checks if the key is enabled to prevent the use of a lost key
4. If the key is enabled and correct the EWS sends a **password** to the transponder via a 125kHz AM signal
5. If the password is correct the transponder answers with a **changing code** which it received from EWS in a previous authentication process
6. EWS compares the changing code received with the one stored in its memory:
   - If they match the car can crank but not start and the EWS sends to the transponder a new changing code which it will use in a following authentication request in step #5
   - If they don't nothing happens and a message of EWS error appears on the dashboard.

The problem of this protocol is that the changing code transmission from the EWS to the key **happens in an insecure way**. In fact, if an attacker eavesdrops the key ID and the final changing code, when the car is restarted, he will be able to be correctly authenticated to the EWS not only one time, but all the followings too, and this will permit him, for example, to steal and run the car.
The idea of changing password each time is valid, but not the implementation. We could have a better result using a *'One-Time Password'* protocol.

4.1.2 One-Time Password Identification Protocols

This kind of protocols have been created with the purpose of generating, from a starting key, a sequence of passwords which should be used only one time for an authentication process.
The benefit is the impossibility, for an attacker, to guess the next password knowing the current one.

To do so we could use:

1. one-time password sequences based on a one-way function, or
2. sequentially updated one-time passwords.

We recall that f is a **one way function** if:
$\forall x \in X$, $f(x)$ is easy to compute but for essentially all elements $y \in Im(f)$ it is not computationally possible to find any $x \in X : f(x) = y$.
Hash functions are an example.

1. In the fist case, to obtain a one-time password algorithm we could follow the '*Lamport's Protocol*'.

   Lamport's Protocol:

   - *H = shared one way function*

   A identifies itself to B using one-time passwords from a sequence. [5]

   (a) *One-time Setup.*
      i. *A chooses a secret number w and fixes a constant t (e.g. t=100 or t=1000). t represents the length of the sequence.*
      ii. *A computes $w_0 = H^t(w)$ and sends it to B*
      iii. *B initializes its counter for A: $i_A = 1$.*

   (b) ***Protocol*** of the i^{th} identification, $1 \leq i \leq t$
      i. A computes $w_i = H^{t-i}(w)$ and sends to B (i, w_i)
      ii. B, known the previous password $w_{i-1} = H^{t-i+1}(w)$, checks that $H(w_i) = w_{i-1}$ and that $i = i_A$.
      iii. If everything is correct the authentication is valid, and B sets $i_A = i_A + 1$ and saves w_i for the next session.

   The Lamport's scheme is a one-way authentication protocol: A doesn't acquire any information about B's identity or legality and it remains vulnerable to an attacker

[5] [17]

that aims to impersonate B.
However an attacker that eavesdrops the communication can't impersonate A, without knowing the secret w, because the hacker, to compute the p^{th} password, knowing the last $p-1$, should calculate $w_p = H^{-1}(w_{p-1})$ that is computationally impossible since H is a one-way function.
The counter i_A is useful to prevent replay attacks and should never be decreased until the end of the sequence. If the synchronization goes out, and $i_A < i$, something (it depends of the context of application) should be done (e.g. B increments i_A until i).
The process presents two main limits:

- the finite length of the sequence: in fact we can't choose t too big or the probability to find a collision between two values of H will become too high.
- the synchronization: A and B shall maintaining an updated internal state, but this could succumb to Denial of Service (DoS) attacks. Moreover, when a sequence ends, the passage from that one to another can be problematic.

2. The other group of one-time password protocols is represented by those algorithms that, given a shared starting seed, generate the same sequence in a not predictable way. The main examples are the *'Rolling Codes'*.

Rolling Codes

A Rolling Code[6] is a code programmed to change each time a precise action occurs (e.g. the car is restarted), or after a precise time interval. Usually it follows a precise sequence depending on the initial value and a calculation.
To preserve the security, an observer should see a deterministic Rolling Code as a sequence of random numbers. To do so we can use a **Pseudo Random Number Generator (PRNG)**.
A PRNG is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The sequence has no repeated values until it's gone through every number it can generate, and then it

[6] [Web: 23]

starts over again with the same order. The PRNG-generated sequence is not truly random, because it is completely determined by a relatively small set of initial values, called the PRNG's seed: **a PRNG initialized with the same seed will always give the same sequence**. Therefore to create the same Rolling Code, it's enough to share the same algorithm to calculate pseudo random numbers and the same PRNG's seed.
In this case the real problem is to obtain a real random number for the seed, different for each car and to keep it secret. This last condition entails the seeds' memorization in some secure sever, that should never be violated in order to preserve cars' security.
Moreover, different seeds should be used for different cars. In fact if the same seed is used in two different cars, it's possible to take the EWS of one car and the DME of the other and reset them, in order to make the sequence start again and synchronize their rolling codes even without knowing the seed.[7]
Speaking about the seed, if the PRNG period is long enough, it's very hard to find out the feedback number, because the number of values that needs to be memorized, in order to try to find out the algorithm behind, is very high.
Anyway **using only a PRNG is not cryptographically secure**.
We recall that a random number generator is cryptographically secure if:[8]

- nobody, knowing the last k numbers generated (i.e. also the seed), can reliably determine the $k + 1$ value with a polynomial-time algorithm.

In our case a **Cryptographically Secure Pseudo Random Number Generator (CSPRNG)** is recommended. A way could be to encrypt the pseudo random value, making each electronic keyfob have billions of possible codes.
Blocks ciphers and hash functions are often employed in this environment, in particular the 64 bit block cipher "KEELOQ"[9] was commonly used for cars immobilizers in automotive context, until 2008.
In March 2008, researchers from the Ruhr University in Bochum (Germany) presented

[7][Web: 20]
[8][17]
[9][15]

a complete break of remote keyless entry systems based on the KeeLoq RFID technology, thanks to a side channel attack based on the DPA (Differential Power Analysis). Their attack works on all known cars and building access control systems that rely on the KeeLoq cipher[10].
New systems, using different ciphers (e.g. AES-128), are required.
Furthermore, using automatic rolling codes, the loss of a key's transmission by the receiver, and the consequent desynchronization, is quite frequent, so after a failure authentication, EWS's code usually rolls forward for a maximum of n-times (n chosen by the manufacturer, usually around 200), searching for key's value. If it finds it, it stops so that the devices will become synchronize with each other again and the subsequent authentication proof will succeed.
We expect that a valid key can take at most two/three attempts to be approved.

As we saw the implementation of one-time password protocols, especially in case of desynchronization, is always troublesome, so if it's possible we suggest to give up this solution and choose this one: **use a symmetric cryptosystem** with a secret key to implement a double *challenge-response protocol* (e.g. using EWS's VIN) for the authentication of both sides.
However, since in modern immobilizer authentication protocols the use of rolling codes is frequent, we will continue to take them as an example in the following section.

4.1.3 Modern Key to EWS Authentication

We repeat the two previous solutions to increase the security in our authentication protocol that are commonly used in automotive context:

- insert a valid rolling code, shared between the key and the EWS
- share a secret key, and perform a challenge-response authentication protocol.

Protocol

A modern protocol for immobilizer and key's control could work like this:

[10] [12] [14]

1. When the key is inserted in the car, the EWS module sends a signal on a frequency between 300 and 400 MHz to the antenna in the key
2. The signal powers up key's transponder, that sends the **key identification code** to the EWS
3. The EWS verifies the key ID and checks if the key is enabled to prevent the use of a lost key
4. If the key is enabled and correct the EWS sends a not secret **password/ challenge** to the transponder
5. If the password is correct the transponder answers with **his rolling code/ the encryption of the challenge** and, in case of rolling code, the transponder rolls to next value
6. The EWS calculates his own code and verify key's ones:
   - if they match, it allows the car to start and, if the code is a rolling one, when the car is restarted it rolls to next code
   - if they don't the authentication fails and an EWS error message will appear on the dashboard.

Another simpler way for the identification could be the use of a **secret shared pre-programmed algorithm**: e.g. key and EWS take the same car's parameters and compute them with the same calculation, then EWS verifies key's result: if it's correct the key is valid.
In reality, every Cryptography book advises against an algorithm, in which security is based only on the secrecy of its operations: cipher history teaches that you can't keep a procedure secret for longtime.[11]
In this case, since the algorithm would be the same for lots of cars, finding the computation once will allow you to enter in them all. For this reason we won't consider this procedure as an option.

[11] [9]

4.1.4 Implementation Observations

In both cases (rolling code/symmetric cipher) we need to do some observations about the implementation: **a cryptographic secure algorithm could lose his efficiency if it's not well implemented in practice.**
For example, it's important to think where to store sensitive information in the ECU to keep them safe from simple attacks: e.g. we can easily dump a memory placed outside the microprocessor and read its data, so this data storage should never contain private values or secret procedures.

As an additional security control, to prevent the substitution of the pair EWS-key, the DME, before allowing car's starting, can verify EWS's authenticity:

4.2 ECUs marriage: EWS to DME Authentication

The process should prevent the substitution of the pair EWS-ignition key with a counterfeit one by **'marrying'** the EWS and the DME together.
It needs:

- a **rolling code ISN** (Individual Serial Number) shared between the EWS and the DME
- the same VIN for the EWS and the DME

And it operates like this:

1. The DME verifies EWS's VIN (how, could be various)
2. The EWS sends its rolling code to the DME
3. The DME calculates his own rolling code and verifies EWS ones:
    - if they match, the ignition can now start
    - if they don't, the DME 'rolls forward' to the next n-codes, as in the previous protocol, and tries the authentication again
4. When the engine is switched off, the EWS and the DME will automatically roll forward to the next code. This will be used in the next EWS to DME authentication process.

To each VIN is associated a specific seed for the rolling code. If we try to substitute an ECU, for example the EWS, with a not original one, the VIN and the rolling code won't match anymore: this will prevent the EWS to communicate with the DME and start the ignition.

What about to substitute one of them in a legal way?

- A new EWS for our car could be requested to an authorized dealer: it will have the right VIN and the right rolling code on it. After the installation on the car a reset of the DME's rolling code to the 'Rolling Code #1' is required.
- To substitute a DME we need to install on the car a new virgin DME, and inform the EWS that a new ECU has been embedded. The next time we switch on the car the EWS will send the entire Rolling Code (usually is just the seed) to the DME and reset it to the 'Rolling Code #1'.
  The DME will automatically burn the Rolling Code into its memory in an **indelible way**. Finally probably we have to synchronize the EWS and the DME again to the same initial value.

For this reason, once a DME is **'married'** to the vehicle it won't work in any other vehicle.

Of course some sagacity is needed during the implementation, like using different seeds for different cars, to reduce the risk of being attacked.
Some high level threats are still possible: for example, with the right tools and knowledge, it's possible to take a DME, with a rolling code burned permanently in it, reprogram arbitrarily its VIN and substitute the part with the rolling code with a new one, making the unit virgin and ready to be married with the EWS desired[12].

[12][Web: 20]

Appendices

Appendix A

Detect attacks: Bus Guardian

Speaking about cyber attacks we reflected upon cryptography and its innovative responses to increase security, but we haven't mentioned yet the more immediate and 'simple' solution.
Like in the computer case, we could think to create a guardian, that, as an antivirus, detects harmful messages and filters them to protect the system.
Actually, system like this, are already existing and they are called: '***Bus Guardian***'. [1]

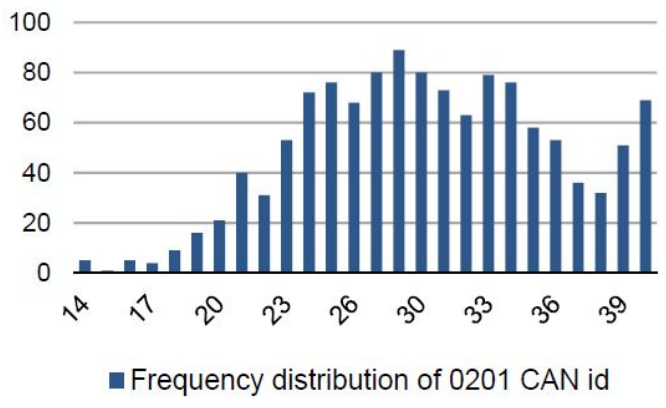
In really few words, their validity is based on the facts that:

- the frequency of each CAN ID is predictable
- it's difficult for an attacker to injects some messages and at the same time delete the original ones.

So we could study the frequency of a Normal CAN packet:
the figure below [2] shows the frequency distribution of the Ford Escape CAN packet with ID 0210.

The maximum is in (28,90) that means that for 90 times the numbers of messages with this ID sent in a second was 28.

[1] [Web: 29], [Web: 30], [Web: 31]
[2] [3]



Ford Escape

Instead, during an injection attack, the message was replayed at 10 to 20 times these frequencies, that means that the maximum was probably in (56,90). [3]

On the other hand, speaking about Diagnostic CAN packets, it's even simpler, because, with few exceptions, they shouldn't be present during the normal operation of the car.

Those the reasons why, in many cases, it is possible to detect frequency abnormalities and prevent certain attacks.

[3][3]

Appendix B

One Time Programmable (OTP) memory

In the paper we said repeatedly that we can write data in a 'indelible/ not reversible/ permanent' way, but we didn't give any information about how this is effectively possible. Well, let's see now how the **One Time Programmable (OTP) memory** (or **Programmable Read-Only Memory (PROM)**) works:
As mathematicians, we will say, in few and simple words, that this memory is a form of digital memory where the setting of each bit is locked by a fuse.[1]

A fuse is a safety device with a conductor thread that lies together an electrical source and its associated load(s). When the current that passes in it exceeds its designed threshold, the fuse is calibrated to permanently open the series circuit, thereby disconnecting the load(s) from the power source.
Once a fuse disconnects, to restore initial settings, it's necessary to discard it and replace it with a new one .
Fuses protects a circuit from overheating due to excessive current flow.[2]
In our case, the setting of bits is locked with fuses: if fuses are working, we can write and reprogram data inside, when we want to fix the information memorized in a permanent

[1][Web: 24]
[2][Web: 25]

way we can send a voltage higher than the supported one, burning out the fuses and making the memory a read only one.
This process is known as **burning the PROM**.[3]
Like said before, the only way to have rewrite access of this memory is to substitute the fuses with new ones.
If the PROM is inside the microprocessor the replacement operation is not possible without specific and expensive equipment and very high knowledge, so we can consider **secure** to memorize sensible data, like cryptographic keys, in this way.

[3][Web: 26]

Appendix C

Hardware Secure Module (HSM)

EVITA Project

"The objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering and sensitive data are protected against compromise when transferred inside a vehicle.[1]*"*

The EVITA (E-safety Vehicle Intrusion proTected Applications) was a European Union project born in 2008 and ended in 2011, which has established state of the art specifications of automotive Cyber Security.
Between the main participants there was: BMW, Bosch, Continental, Infineon, Fujitsu and Escypt.
As reported before, the objective of the project was to create a new architecture for cars network in order to increase their security.
The work made by the team merged in the description of a **Hardware Secure Module (HSM)**[2] for automotive purpose: an integrated chips specifically developed and designed for security automotive use-cases. In technical words the HSMs are tamper-resistant cryptographic coprocessors with a programmable secure core, integrated on the same chips as the ECUs.

[1][Web: 27]
[2][16]

To enable cost-efficiency and flexibility, different classes of HSMs have been specified with different security requirements[3]:

- *Full HSM* for protecting the in-vehicle domain against vulnerabilities due to V2X communication: This includes an asymmetric cryptographic engine for creating and verifying electronic signatures. The full HSM provides the maximum level of functionality, security, and performance of all the different HSM variants.
- *Medium HSM* for securing the on-board communication: The medium HSM resembles the full HSM, but contains a little less performing microprocessor and no asymmetric cryptographic engine in hardware. However, it is able to perform some non-time-critical asymmetric cryptographic operations in software, e.g. for the establishment of shared secrets.
- *Light HSM* for securing the interaction between ECUs and sensors and actuators: It only contains a symmetric cryptographic engine and an I/O component in order to fulfill the strict cost and efficiency requirements that are typical for sensors and actuators.

In particular a Full HSM should be able to:

- prevent unauthorized manipulations of vehicular on-board electronics,
- prevent unauthorized modifications of vehicle applications especially regarding safety and m-commerce applications,
- protect privacy of vehicle drivers,
- protect intellectual property of vehicle manufacturers and suppliers,
- maintain the operational performance of applications and security services.

The table shows the components of the different HSMs:

[3] [16]

	HSM		
	Full	*Medium*	*Light*
RAM	√	√	optional
NVM (non-volatile memory)	√	√	optional
Symmetric cryptographic engine	√	√	√
Asymmetric cryptographic engine	√		
Hash engine	√		
Counters	√	√	optional
Random-number generator	√	√	optional
Secure CPU	√	√	
I/O component	√	√	√

Components for HSM

HSM is compliant to the SHE specification. CSE security level is placed between a Light and a Medium HSM.[4]

Nowadays some Medium HSM are available on the market: Freescale Semiconductor Corporation and Infineon Technologies AG are the referential companies for their production. Implementation of Full HSM is still on working.
The main differences between a medium HSM and a SHE specification module are that in this case, the **firmware** is not pre-installed, but it is **programmable** by car's manufacturers, in order to be more compliant to their specific needs.
Furthermore with HSM, it's possible to implement a valid control of Debug access.

To have an idea of what we are working with, we report the technical specifications of the Freescale's Calypso MPC5747G microprocessor and its Medium HSM:

[4]See chap. 3 pag. 49 for CSE description

Calypso	Number of cores	3
	Maximum Operating Frequencies	160Mhz 160Mhz 80Mhz
	Flash	4 MB
	RAM	768 KB
	HSM	Available (Level Medium)
	LIN	17
	CAN	8
	MOST	Available
	FlexRay	Available
	USB 2.0	Available

Calypso MPC5747G

HSM	Number of cores	1
	Maximum Operating Frequencies	80Mhz
	Dedicated Secure Flash	144 (application) + 32 (data) KB
	Dedicated Secure RAM	32 KB
	Debug Interface	Secure
	Cryptographic HW engine	AES-128 (symmetric)
	Firmware	Programmable

Details of HSM

Thanks to the presence of a secure storage, the Random Number Generator and a symmetric cipher, HSMs, or a cryptographic modules in general, can be used to establish secure communications and identity proofs between multiple ECUs.
In the third part of this paper we will describe an authentication protocol between two ECUs, equipped with Medium HSMs, communicating on CAN bus. The purpose is to implement the protocol in the practice: we have already started to work on it, but we haven't reached the scope yet.
Anyway, for the communication on the CAN bus, we needed to decide how our CAN packets shall be made:
as we saw before, our packets should have maximum: 29 bits of identifier plus 64 bits of data.
The reduced length of data and the 128 bits blocks of AES-128 seam to be a bad combination to manage. That is true, but, to minimize the gap, we decided to use the ***'Transport Protocol'*** on CAN bus, proposed in the EVITA project.

C.1 EVITA Transport Protocol

EVITA transport protocol defines a way to use the bits of the CAN packets in order to transmit all the information necessary for the communication. Furthermore it defines how to divide messages longer than 64 bits, in order to recompose them univocally.
Let's see it more in details.

Generally in the CAN communication the same CAN ID could be used by more then one ECU, but, in this case, it's necessary to suppose that every ECU has a different ID of 15 bits, that identifies it univocally.

We divide the messages in two categories:

1. Single: if the content of the message is $\leq$ 6 byte (48 bits)
2. Multiple: if the content of the message is $>$ 6 byte

Single messages are sent with a single CAN packet, multiple messages shall be divided

in a sequence of CAN packets.
The CAN packets considered have 29 + 64 bits, that means:

- 29 bits for the identifier (ID)
- 64 bits for data

The 29 bits of fixed CAN ID are divided as follow:

- 7 bits for ‘Security Features’: they define if the content of the message is encrypted and in which way
- 15 bits for ‘Source Address’ : they report the unique CAN address of the ECU that is sending the packet
- 7 bits for ‘Receiver ID 1’: they report the first part of receiver’s ID

29 bits of ID		
7 bits	**15 bits**	**7 bits**
Security Features	Source Address	Receiver ID 1

The 64 bits of variable Data are used as follow:

- For SINGLE messages:
  – 8 bits for ‘Receiver ID 2’: they report the second part of receiver’s ID
  – 8 bits for ’Length’: they report the total length of the content of the message
  – 48 bits for ’Message’: they report the content of the message

64 bits of Data (8 byte)		
8 bits (1 byte)	**8 bits (1 byte)**	**48 bits (6 byte)**
Receiver ID 2	Length	Message

- For MULTIPLE messages we should differentiate two cases:
  1. the packet is the FIRST packet of a sequence:
     – 8 bits for ‘Receiver ID 2’: they report the second part of receiver’s ID

- 8 bits for ‘Length’: they report the total length of the content of the message
- 48 bits for ‘Message’: they report the first part of the message

64 bits of Data (8 byte)		
8 bits (1 byte)	**8 bits (1 byte)**	**48 bits (6 byte)**
Receiver ID 2	Length	Message

2. the packet is a FOLLOWING (i.e. not the first one) packet of a sequence: It’s the same as for a single message
   - 8 bits for the ‘Counter’: they count the packets of the sequence (we shall enumerate them in order to recompose them in the right order)
   - 56 bits for ‘Message’: they report the following parts of the message

64 bits of Data (8 byte)	
8 bits (1 byte)	**56 bits (7 byte)**
Counter	Message

The scope of this transport protocol is to exploit for the better, the reduced length of the packets maintaining the transmission clear for the receiver.
For this reason we decided to use it in our practical implementation of the authentication between ECUs.

Appendix D

Code Signing Authentication

Code Signing is a technique used to authenticate an important part of the code (e.g. engine maps, secret information, the flag that identifies the state of a car's option with fee) set in an insecure memory (e.g. flash), in order to prevent a modification without the legitimate permission.
A trustworthy processor with a secure memory is necessary to verify the authentication: the HSM module, with his secure flash, could be a valid option.
The implementations are various, one way could be to use an hash function, like SHA or MD5.
Supposing that our scope is to validate an engine map M stored in the flash; if the HSM is integrated in the same ECU that has the map, a possible solution is described by the following procedure, where h is a hash cryptographic function:

HSM has stored in his memory the hash value of the original map $= H_M$*, called* finger print

1. At action X (e.g. car is switched on), the HSM reads M from the flash and computes $h(M)$
2. HSM compares $h(M)$ and H_M:
   - if they are equal, the authentication is done
   - if they are different, a recovery solution is implemented, such as:
     - restore the map to a default value, memorized in the secure flash

- only a limited use of the vehicle is permitted
- a Denial of Service (DoS) state is recognized
- etc.

Signing all the code could be too costly, so we should chose to validate only some selected parts of it. Then, since HSM should store their hash value in his memory from the start, they should be fixed a priori by the manufacturer; if they could be modify or update after, depends on the way finger prints are memorized (OTP or not).
An attacker can't make the HSM accept his own code if it's not able to modify the finger print in the secure flash.
Supposing the HSM embedded in the ECU we want to test, allows us to assume the exchange of information secure.
However, probably, not every ECU could have a HSM module integrated, so what if the HSM has to validate the map of a not secure ECU?
The difficult increases considerably, because HSM can't trust anymore the map's value received; in fact a hacker could modify the map, but send always the original value in order to be authenticated.
A solution could be to ask to the ECU not the hash value of the total map (that is always the same), but the hash value of a random walk on map's code, generated with a seed choosen by the HSM. The difference is that the ECU should calculate the hash value only of some random parts of the code and not of all. But a problem is still present: to succeed, the HSM should have in his memory the hash value of the random walk associated to the seed, that means, known the seed, it exists a way to determine which path the random walk will do: it could be follow a precise algorithm (like the one to generate pseudo random numbers), a pre-determined sequence or even something not directly related to the seed. In any case, the way will be stored somewhere in the ECU, and we can't prevent a corrupt ECU to have access to it.
If the attacker has access to the original map and the 'random' algorithm/information, it could calculate all the hash values the HSM asks for.

The problem is still opened.

Part III

A Practical Implementation of a Challenge-Response Authentication Protocol

Why an authentication protocol?

As practical activity, we decided to implement an authentication protocol between ECUs.
The introduction of an authentication protocol between car's ECUs can prevent the substitution of one of them with an unauthorized one in order to:

- avoid some dangerous consequences, like the installation of a counterfeit airbag (seen at page 39)
- restrain the black-market of auto parts
- permit the 'marriage' between ECUs that have to exchange sensitive information, that can damage the car if modified
- avoid the installation of ECUs that have been created and calibrated for a different car: for example the ECM with its engine maps, or the CDCM with its type of transmission (manual, automatic or semi-automatic)

In our design an ECU, called Master, shall test the authenticity of some other ECUs, called Slaves, when the car is started. We suppose the presence of a cryptographic module, with certain characteristics, later explained, in every ECU considered.
For the moment, the authentication we started to implement is one-directional, mainly for simplicity reasons, but it's reasonable to think that in a future application it will become mutual.
A negative result of the authentication process can be used later to trigger some consequences, like preventing the car from running or reducing its performances and forcing the driver to go to an authorized dealer or workshop in order to detect and solve the problem.

Chapter 5

The Authentication Protocol's Requirements

Before starting to model a project it's necessary to write its Functional Requirements Specifications, that means one or more papers where it's reported the detailed description of the model.
The objective is to create into the reader, a unique idea of how the model shall be done and how it should work, setting the main guidelines of the implementation, without any detail of the code.

5.1 ECUs Authentication Protocol Functional Requirements Specifications

Fist, we're going to describe the **affected systems**, then we will give the **general overview** of the functionality and finally an explicative **diagram**.
Because of enterprise's security and secrecy reasons we will omit the detailed description and the Input/Output/NVM Parameters explanation.

5.1.1 Affected Systems

This Functional Requirements Specification is intended for ECUs with a **dedicated module to do hardware cryptography, integrated in the microprocessor**.
In particular, we consider a network with a MASTER ECU and a set of SLAVE ECUs associated.
The purpose is that MASTER ECU verifies the identity of its associates in order to prevent their unauthorized replacement.
As soon as the car is started, MASTER ECU shall run the authentication protocol.
Then depending on the results of the authentication it should trigger some consequences, like the permission of running the car or not.
For this reason, only an ECU that controls critical components of the car is a good choice of MASTER ECU.

MASTER ECU's cryptographic module shall have:

- a dedicated core
- a dedicated secure RAM
- a secure debug interface
- a unique ID
- a secure part of NV(*Non-Volatile*) memory where:
  - its own ID
  - a vector that contains all SLAVE ECUs IDs
  - a vector that contains all the keys shared with SLAVE ECUs,
  - a vector that contains the estimated time of answering for each SLAVE ECU

  are stored
- a cryptographic HW engine (symmetric cipher)
- a cryptographically secure Random Number Generator

A SLAVE ECU's cryptographic module shall have:

- a dedicated core
- a dedicated secure RAM
- a secure debug interface
- a unique ID

- a secure part of NV memory where:
  - the key shared with MASTER ECU
  - and its own ID

  are stored
- a Cryptographic HW engine (symmetric cipher)

The symmetric cipher of MASTER ECU and SLAVE ECU shall be the same. Otherwise no encrypted communication is possible between them.

5.1.2 Functionality Overview

The aim of this functionality is to implement an Authentication Protocol that permits MASTER ECU to verify the identity of one or more SLAVE ECUs, one at a time, communicating with them on an **insecure channel** (e.g. CAN bus).
One of the purposes of this authentication is to prevent the unauthorized replacement of ECU.

By default configuration, when the car is started, MASTER ECU tests all its associated SLAVE ECUs, but the option to test only one or more chosen ECUs shall be available.

MASTER ECU implements a 'challenge- response' protocol in order to verify a SLAVE ECU ID.

Protocol overview:

1. MASTER ECU sends a challenge to a SLAVE ECU
2. SLAVE ECU encrypts it with its own ID, using the secret key, and sends back the response,
3. MASTER ECU verifies if the answer is the correct one. If the response is the right one, SLAVE ECU's identity is proven.

With this process MASTER ECU could verify the identity of all SLAVE ECUs, taken one by one, and then use the information acquired to implement the related consequences.

The ***main outputs*** of the protocol are:

1. The global state of the authentication protocol (completed/interrupted/not performed/...)
2. The state of each SLAVE ECU authentication test (successful/failed/...)

5.1.3 Functionality Description

[· · ·] We will call SLAVE#x a generic SLAVE ECU associated to the MASTER.
To guarantee proper functionalities of the transmission on the CAN bus, SLAVEs authentication shall be done considering the SLAVE ECUs ONE AT A TIME. And when the last ECUs that shall be tested completes the protocol, the authentication process shall be considered concluded. To end an authentication process, it is not necessary to have a successful authentication result in all cases.

[· · ·]

Challenge-Response Authentication Protocol Description

The protocol is the one described at page 52. See that page for all details.

Protocol Activation/Deactivation Criteria

1. NECESSARY CRITERIA
   These criteria shall be controlled and satisfied, before allowing the implementation of the protocol:
   (a) Integrity of MASTER ECU's cryptographic module:
   The integrity of MASTER's cryptographic module (keys, secure memory,...) shall be checked to prevent possible modifications
   (b) MASTER ECU's communication is active:
   MASTER ECU shall be able to send and receive messages via CAN bus

2. TRIGGERING CRITERIA
   (a) MASTER ECU shall start the authentication process as soon as the driver turns the ignition keys to the on position

   (b) later, MASTER ECU shall repeat the test for those SLAVE ECUs that failed the authentication, for a maximum of MaxNumAuthFail times.

3. TERMINATION CRITERIA
   The protocol shall be stopped before the end if:
   (a) For some reasons CAN communication is not working anymore
   (b) The debug port of the cryptographic module of MASTER ECU is opened

Main Outputs of the functionality

The main outputs of the functionality are two processes:

1. The **state of the authentication**; the possible states are:
   - Not Performed: before the start of the authentication protocol (default state)
   - Running: while the authentication process is performing
   - Performed: if the authentication process ends. This means that the authentication of all ECUs has been run, but not that all the ECUs passed successfully the authentication test.
   - Interrupted: if the authentication process is interrupted before completion (i.e. because of one of the Termination Criteria)
2. The **results of the authentication protocol** for each SLAVE ECU.
   For each ECU the possible states are:
   - No Challenge Reception: SLAVE ECU didn't received the challenge
   - Successful: SLAVE ECU passed correctly the test, that means its authentication is succeed
   - Failed: SLAVE ECU sent the wrong answer that means its authentication has failed
   - Expired Time: SLAVE ECU took too much time to answer or didn't answer at all

MASTER ECU Functionality

MASTER ECU functionality is partitioned in the following sub-functionalities:

1. **Authentication Manager**
2. **Crypto**
3. **Communication**

1. Authentication Manager, is the brain of the authentication process: following the Necessary Criteria and the Triggering Criteria, it sets the start of the protocol's implementation and which SLAVE ECUs have to be tested.
   Then it uses Crypto and Communication to implement the protocol.
   Furthermore it manages the outputs that describe the state of the authentication and the authentication results.

2. Crypto is implemented by the cryptographic module, and it carries out the cryptographic functionalities.

3. Communication manages the communication between MASTER ECU and SLAVE ECUs over the CAN bus.

SLAVE#x ECU Functionality

SLAVE#x ECU functionality is partitioned in the following sub-functionalities:

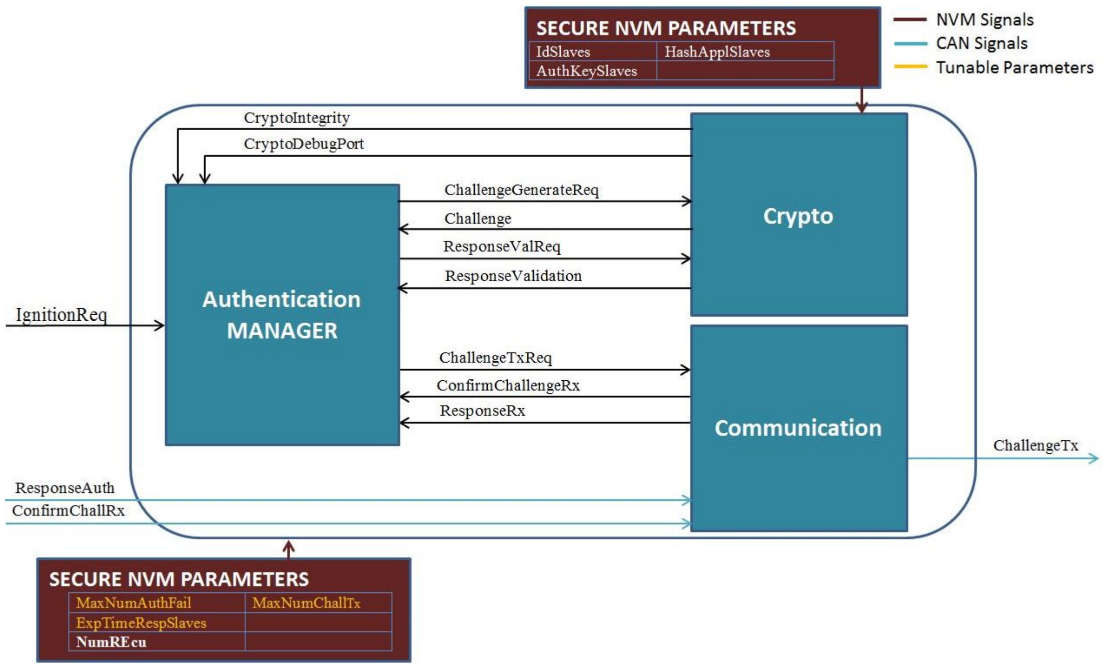
1. **Application**
2. **Crypto**
3. **Communication**

1. Application uses the other modules (SLAVE Crypto and SLAVE Communication) to generate and send the response to MASTER ECU.
   Furthermore: since a SLAVE can answer only to a limited number of authentication challenges in a minute, it shall control that the request didn't overshoot the limit.

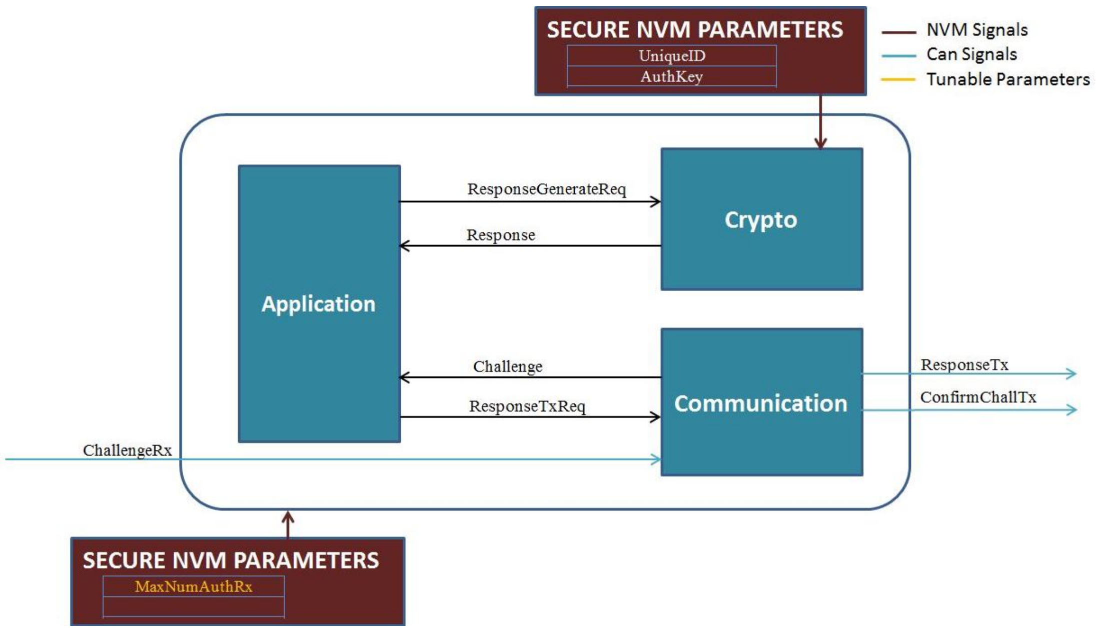
2. Crypto is implemented by the cryptographic module; in particular it generates the response to the authentication test.

3. Communication manages the communication between the ECU and MASTER ECU over the CAN bus.

5.1.4 Interface Diagram



I/O interface of MASTER ECU



I/O interface of an SLAVE#x ECU

5.1.5 I/O Interfaces, NVM Parameters

[···]

Despite the gaps, now we should have a general idea of which kind of model we are going to create.
Now, all the specifications of the Master and the Slave sub-functionalities shall be written, and only after, we can start to program the model. The process is long and actually we haven't finished yet.
Up to now, we modeled in Simulink/MATLAB only the Master Authentication Manager, so in the following chapters we will describe only its specifications and some parts of its model, keeping private most details, for obvious enterprise's secrecy reasons.

5.2 MASTER Authentication Manager Functional Requirements Specifications

5.2.1 Functionality Overview

The aim of this functionality is to manage the authentication protocol, using the other sub-functionalities, in particular, it shall:

- decide which ECUs SLAVE shall be tested
- start the Authentication process and set its progress
- ask to Crypto to generate the challenges
- authenticate the requested ECU, managing the expiration time of the response
- verify which ECUs passed the test and eventually trigger some consequences (i.e. repeat the authentication)

Since the authentication is done considering one SLAVE at a time, the challenge shall be different for each ECU tested. This choice prevents a SLAVE to have more time than is necessary to generate the answer.

Otherwise, a SLAVE that knows in advance which challenge the MASTER will send it, makes useless the control of the expiration time of its response. And the limited expiration time of the response is important because it shall partially prevent this kind of 'man in the middle' attacks:

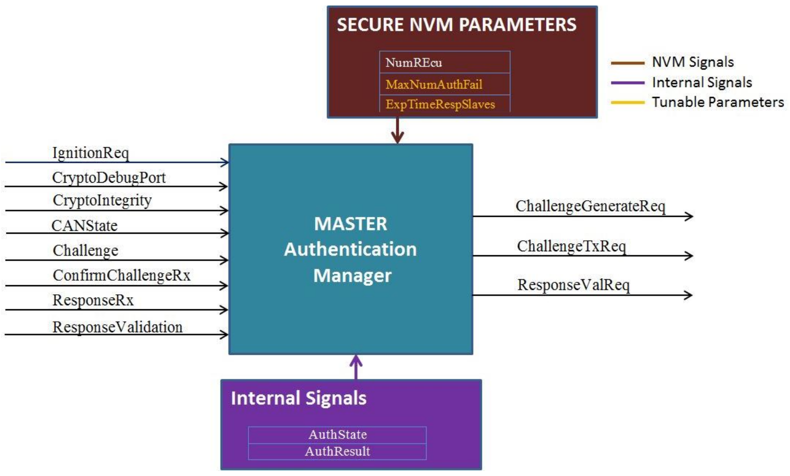
- a malicious ECU uses the extra-time and the challenge to ask in advance to the SLAVE it wants to impersonate, the correct authentication response.
  Then it replays the answer with the MASTER during its authentication test. At the end it will be authenticated even without knowing any secret information.

Of course this attack is still possible, but at least, not the extra-time.

5.2.2 Functionality Description

[· · ·]

5.2.3 Interface Diagram



I/O interface of Master Authentication Manager

5.2.4 I/O Interfaces, NVM Parameters

[· · ·]

Chapter 6

A model in Simulink and Stateflow

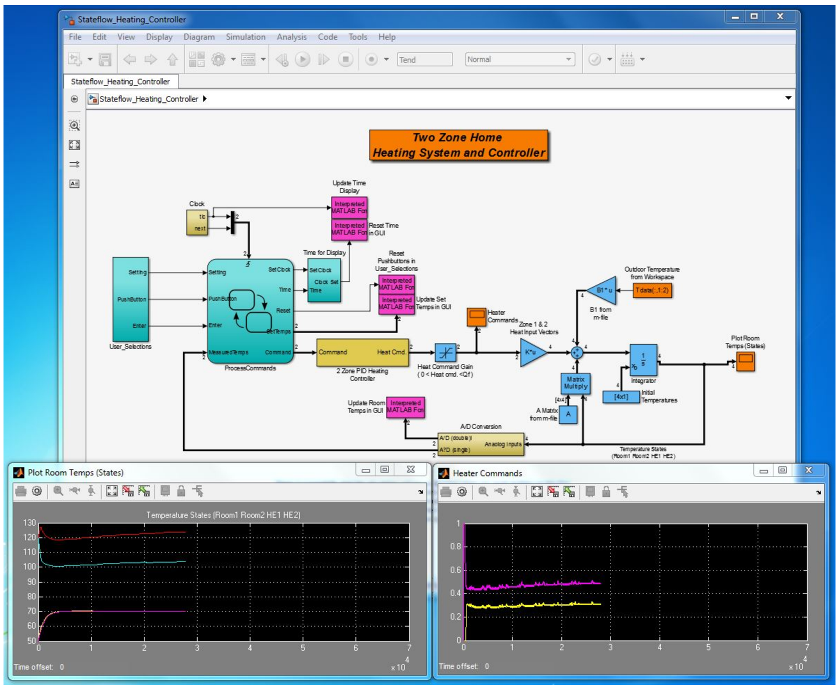
To create our model we used '*Simulink*' and '*Stateflow*'.

> "*Simulink is a block diagram environment for multidomain simulation and Model-Based Design. It supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with MATLAB, enabling you to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.*"[1]

> "*Stateflow is an environment for modeling and simulating combinatorial and sequential decision logic based on state machines and flow charts. Stateflow lets you combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, to model how your system reacts to events, time-based conditions, and external input signals.*"[2]

During the modeling, we followed the '***MAAB Rules***':
The MAAB (MathWorks Automotive Advisory Board) is an independent board that

[1] [Web: 32]
[2] [Web: 33]


Stateflow_Heating_Controller
Two Zone Home Heating System and Controller
Clock
User_Selections
ProcessCommands
2 Zone PID Heating Controller
Heat Command Gain
A/D Conversion
Outdoor Temperature from Workspace
Matrix Multiply
Integrator
Initial Temperatures
Plot Room Temps (States)
Heater Commands
Temperature States (Room1 Room2 HE1 HE2)

Example of Model in Simulink/Stateflow

focus on the usage and enhancements of MathWorks controls, simulation, and code generation products and, in particular, develops guidelines for using MATLAB, Simulink, Stateflow and Embedded Coder; it involves many of the major automotive OEMs and suppliers.[3]

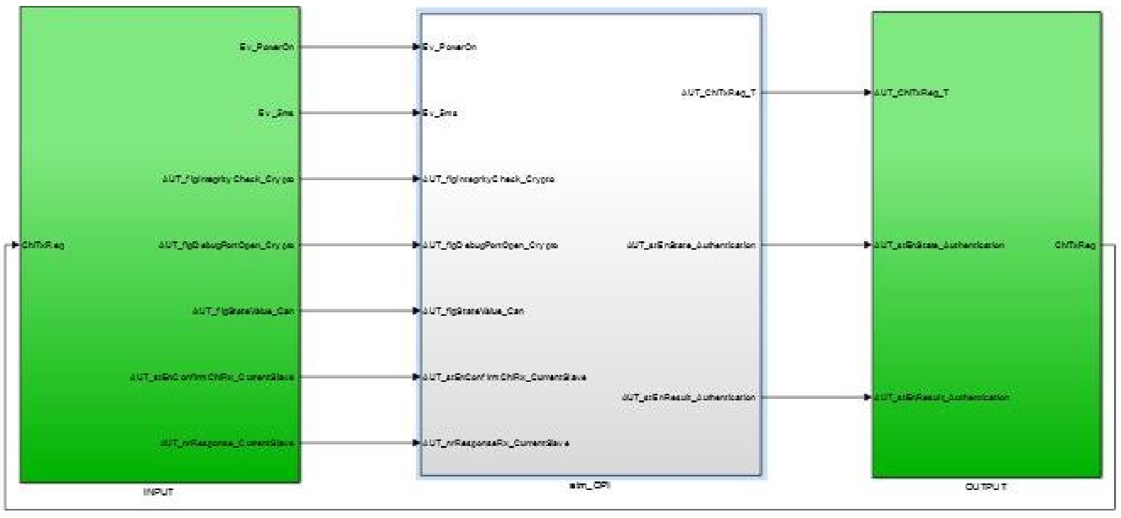
Some of the benefits of following these rules are:[4]

- System integration without problems
- Uniform appearance of models, code and documentation
- Reusable models
- Readable models
- A simple, effective process
- Fast software changes

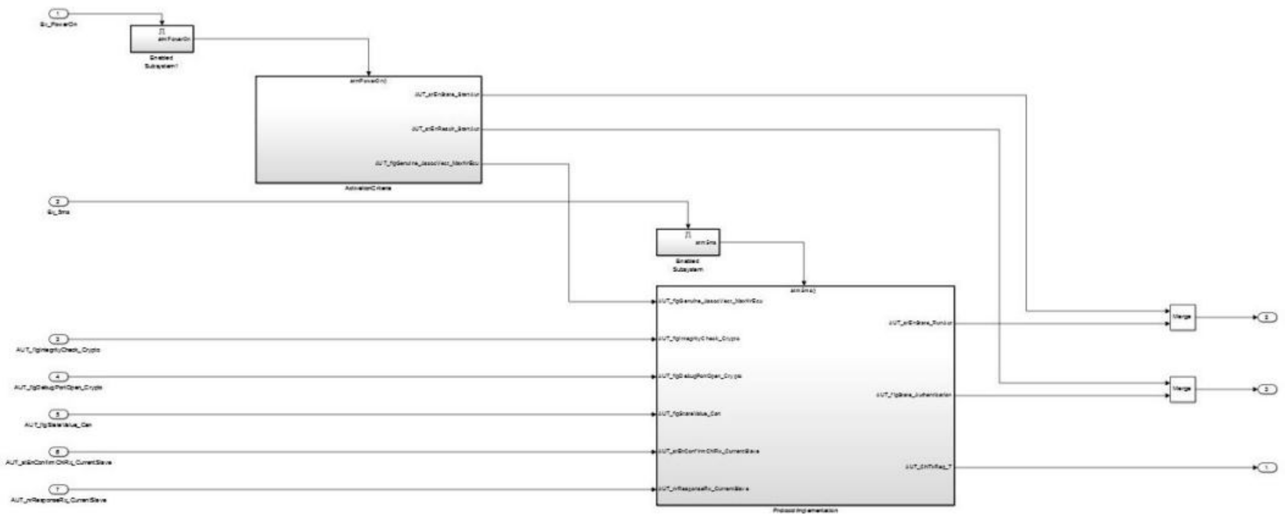
We can't specify now the details of Manager Authentication's model, or the strategies we adopted, I'll just say that it was an interesting challenge, and we rode it out.

[3][Web: 34]
[4][19]

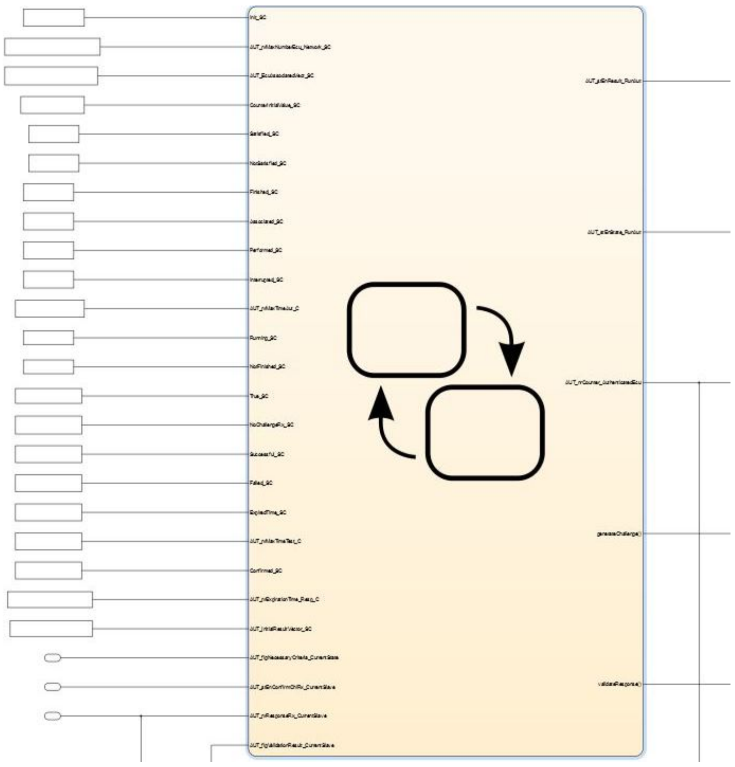
Here are reported some pictures of our model:


Ev_PowerOn
Ev_Sms
AUT_flgIntegrityCheck_Crypto
AUT_flgDebugPortOpen_Crypto
AUT_flgStateValue_Can
AUT_stEnConfirmChRx_CurrentSlave
AUT_nrResponseRx_CurrentSlave
ChTxReq
INPUT
sim_CPI
AUT_ChTxReq_T
AUT_stEnState_Authentication
AUT_stEnResult_Authentication
OUTPUT

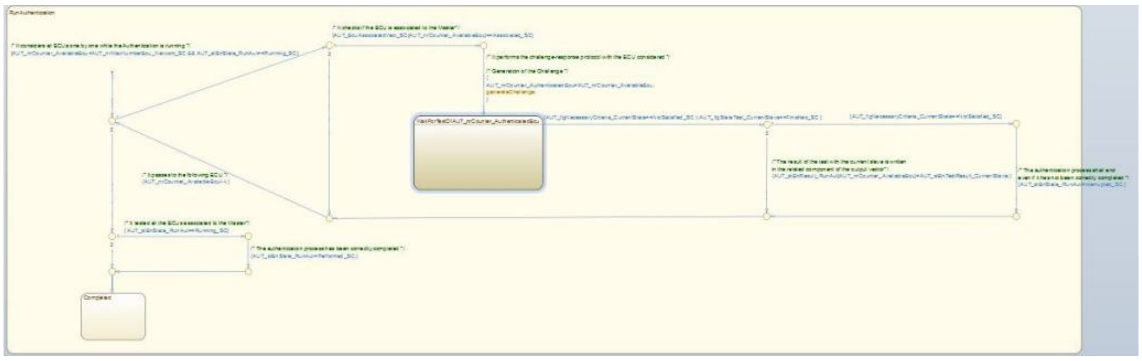
Simulink: 3 subsystems; the gray rectangle contains our model, the left one the simulation of the external Inputs and the right one the model's Outputs.



Simulink: 2 subsystems, triggered by certain events



Stateflow: a finite state machine; on the left the Inputs that will be processed inside and on the right the Outputs



Stateflow: inside the previous state machine; the triangle is a for cycle and the horizontal and vertical lines are $if/else$ conditions.

After, we created 14 **test patterns**: a test pattern is a MATLAB script where the values of model's inputs are specified in every instant of time of model's running.

Nine of them were done considering one Slave to test, the others, two.
Tests patterns are useful to test the running and the accuracy of the model in an initial stage.
Furthermore, thanks to them, we computed also the ***decision coverage***.
Decision coverage analyzes elements that represent decision points in a model, such as a Switch block or Stateflow states. For each item, decision coverage determines the percentage of the total number of simulation paths through the item that the simulation actually traversed.[5]
The model's decision coverage of the union of the five tests patterns with two Slaves to test, is 98%. (The minimum admitted for a model is 80%.)

[5][Web: 35]

Chapter 7

How to Prove the Security of an Authentication Protocol

All the work we did until now, implementing and optimizing our authentication protocol, will be totally useless if we don't try to give a proof of the security of the protocol.

So how we can prove the security of an authentication protocol?
The answer is not trivial.
For centuries, when a new scheme was proposed, people tried to break it. After a while, if they didn't succeed, then the scheme was assumed to be appropriate.
History teaches that this is not the best way to prove security, in fact, in most part of cases protocols were broken, maybe some years after their birth.
Hence the necessity of a '**provable security**', that means: "*prove the security under 'standard' and well-believed complexity theoretic assumptions (e.g. the assumed intractability of factoring)*" [1].
In 1993, Mihir Bellare and Phillip Rogaway, with their article '*Entity Authentication and Key Distribution*', gave a formal security proof of a Mutual Authentication Protocol (MAP1).
In this chapter we will reminisce their reasoning and then we will observe the linking

[1] [20]

with their and our authentication protocol of page 52.

7.1 MAP1 security proof

7.1.1 Notations and previous definitions

- $\{0,1\}^*$ will denote the set of finite binary strings
  $\{0,1\}^\infty$ will denote the set of infinite ones
  $\{0,1\}^{\leq L}$ the set of binary strings of length at most L
  λ the empty string.

- if a, b are strings, we denote $a||b$ the concatenation of a and b.

Now, we consider a set of *identities* I, which defines the *players* who can legally participate in the protocol (i.e. the adversary $E \notin I$ and we won't refer to her as a *player*).
For simplicity, our authentication protocol will involve only 2 parties ($I = \{A, B\}$), but, in general, the set of players could be larger.

The protocol we consider is formally specified by a computable function Π on the following inputs:
- 1^k , $k \in \mathbb{N}$: the security parameter.
- A , $A \in I \subseteq \{0,1\}^k$: the identity of the sender.
- B, $B \in I \subseteq \{0,1\}^k$: the identity of the legitimate partner.
- a , $a \in \{0,1\}^*$: the secret information of the sender. It's its secret key, also called *long-lived key.* In our protocol all players $i \in I$ will get the same a.
- k , $k \in \{0,1\}^*$: the conversation had until that moment
- r , $r \in 0,1^\infty$: the random coin flips of the sender

The value of $\Pi(1^k, i, j, a, k, r) = (m, \delta)$ is:
- m , $m \in \{0,1\}^* \cup \{*\}$: the next message to send out.
  In this case $\{*\}$ means the player sends no message.

- δ , $\delta \in \{\mathcal{A}, \mathcal{R}, *\}$: the 'decision'.
  $\mathcal{A}$ means *Accept*, $\mathcal{R}$ *Reject*, and $*$ means '*the player has not reached a decision yet*'.

Definition 7.1.1.1. ***(efficiently computable)***
A function is *efficiently computable* if it can be computed in time polynomial in its first argument.

Definition 7.1.1.2. ***(negligible)***
A real-valued function $f(k)$ is *negligible* if:

$$\forall\ c \in \mathbb{R},\ c > 0, \quad \exists\ k_c > 0 \text{ such that } f(k) < \frac{1}{k^c}\ , \quad \forall\ k > k_c$$

Definition 7.1.1.3. ***(LL-key Generator)***
The LL-key generator (*long-lived key generator*) $\mathcal{G}(1^k, i, r_G)$ is a polynomial time algorithm which takes as input: the security parameter 1^k, the identity of a party $i \in I \cup E$ and an infinite string $r_G \in \{0,1\}^\infty$ (coin flips of the generator).
Its output is the secret key a, also called *long-lived key.*
In the protocol considered, the LL-key generator is a symmetric one, that means:
$\mathcal{G}(1^k, i, r_G) = \mathcal{G}(1^k, j, r_G) \quad \forall i, j \in I$ and $\mathcal{G}(1^k, E, r_G) = \lambda$.

7.1.2 Adversary's resources

Let's make now some hypothesis of what an adversary can do to annoy the authentication process: we assume that any communication among the parties is under the adversary's control. In particular it can:

- read the messages produces by the parties
- provide messages of his own to them
- modify messages before they reach their destination
- delay messages
- replay messages
- engage many session at once with all the parties (e.g. like in a parallel attack, see page 7)

7.1.3 The mathematical model

Formally each player is modeled by an infinite collection of **oracles** which the adversary may run. So, the oracle $\Pi^s_{A,B}$ *models player A attempting to authenticate player B in session s.*
The adversary E, instead, is a **probabilistic machine** $E(1^k, a_E, r_E)$ equipped with an infinite collection of oracles (the players) $\Pi^s_{i,j}$ $\forall\ i, j \in \{A, B\}$ and $s \in \mathbb{N}$.
In few words, we can imagine a *probabilistic machine* as a virtual computer, with the ability to make random decisions, that means: at each point it randomly chooses between the available transitions, according to some probability distributions.
E communicates with the oracles via **queries** of the form (i, j, s, x), $i, j \in \{A, B\}, s \in \mathbb{N}$.
The query shall be red as follow: *'E is sending the message x to A, claiming that is from B in section s'.*
In our model, a query will be answered by $\Pi^s_{A,B}$; the response will be generate according to the following protocol:

Running of the protocol Π*, with LL-key generator, in the presence of an adversary E, using the security parameter k:*

(1) Choose a random string $r_G \in \{0,1\}^\infty$, and set $a = \mathcal{G}(1^k, A, r_G) = a_A = a_B$ (the secret key), and $a_E = \mathcal{G}(1^k, E, r_G)$

(2) Choose a random string $r_E \in \{0,1\}^\infty$, and a random string $r^s_{i,j} \in \{0,1\}^\infty$ for each $i, j \in \{A, B\}, s \in \mathbb{N}$.

(3) Set $k^s_{i,j} = \lambda \quad \forall i, j \in \{A, B\}, s \in \mathbb{N}$.

(4) Run adversary E on input $(1^k, a_E, r_E)$:
when E asks a query (i, j, s, x), oracle $\Pi^s_{i,j}$ computes $(m, \delta) = \Pi(1^k, i, j, a, k^s_{i,j}||x, r^s_{i,j})$ and answers with (m, δ).
Then $k^s_{i,j}$ gets replaces by $k^s_{i,j}||x$

So, from an oracle's answer, E learns out the outgoing message m and whether or not the oracles has accepted or rejected.
To have a **time notion**, we can say that the adversary's l-th query to an oracle occurs

at time $\tau_l \in \mathbb{R}$.
More notions of time are available, like: $\tau_l = l$ *'absolute time'*, $\tau_l =$ the l-th step in E's computation: *'Turing machine time'*, or $\tau_l =$ the exact time when the l-th query is made: '*real time*'.
We can choose the one we prefer under the condition $\tau_l < \tau_p \quad when \quad l < p$.

7.1.4 MAP1 protocol

We will define mutual authentication (MA) by an experiment involving the running of adversary E with security parameter k. When E terminates, each oracle $\Pi^s_{i,j}$ has had a certain conversation $k^s_{i,j}$ with E, and it has reached a certain decision $\delta \in \{\mathcal{A}, \mathcal{R}, *\}$.

Definition 7.1.4.1. ***(initiator and responder oracle)***
Fixed an execution of an adversary E (i.e. fixed the coins of the LL-key generator, the oracles and the adversary), for any oracles $\Pi^s_{i,j}$ we can capture its *conversation* by a sequence

$$K = (\tau_1, \alpha_1, \beta_1), (\tau_2, \alpha_2, \beta_2), \cdots (\tau_m, \alpha_m, \beta_m)$$

This sequence encodes that at time τ_l oracle $\Pi^s_{i,j}$ was asked α_l and responded with β_l.
Suppose oracle $\Pi^s_{i,j}$ has conversation prefixed by $(\tau_1, \alpha_1, \beta_1)$.
Then, if $\alpha_1 = \lambda$ we call $\Pi^s_{i,j}$ an *initiator oracle.*
If α_1 is any other string we call $\Pi^s_{i,j}$ a *responder oracle.*

Definition 7.1.4.2. ***(matching conversation)***
Fixed a number of moves $R = 2p - 1$ (we suppose R odd, the case with R even is analogous) and a R-move protocol Π, given two oracles $\Pi^s_{A,B}$ and $\Pi^t_{B,A}$, we run Π in the presence of an adversary E.
Let K and K' be the corresponding conversations engaged.

(1) We say that K' *is a matching conversation to* K if there exist $\tau_0 < \tau_1 < \cdots < \tau_R$ and $\alpha_1, \beta_1, ..., \alpha_p, \beta_p$, such that K is prefixed by:

$$(\tau_0, \lambda, \alpha_1), (\tau_2, \beta_1, \alpha_2), (\tau_4, \beta_2, \alpha_3), \cdots, (\tau_{2p-4}, \beta_{p-2}, \alpha_{p-1}), (\tau_{2p-2}, \beta_{p-1}, \alpha_p)$$

and K' is prefixed by:

$$(\tau_1, \alpha_1, \beta_1), (\tau_3, \alpha_2, \beta_2), (\tau_5, \alpha_3, \beta_3), \cdots, (\tau_{2p-3}, \alpha_{p-1}, \beta_{p-1})$$

(2) We say that K *is a matching conversation to* K' if there exist $\tau_0 < \tau_1 < \cdots < \tau_R$ and $\alpha_1, \beta_1, ..., \alpha_p, \beta_p$, such that K' is prefixed by:

$$(\tau_1, \alpha_1, \beta_1), (\tau_3, \alpha_2, \beta_2), (\tau_5, \alpha_3, \beta_3), \cdots, (\tau_{2p-3}, \alpha_{p-1}, \beta_{p-1}), (\tau_{2p-1}, \alpha_p, *)$$

and K is prefixed by:

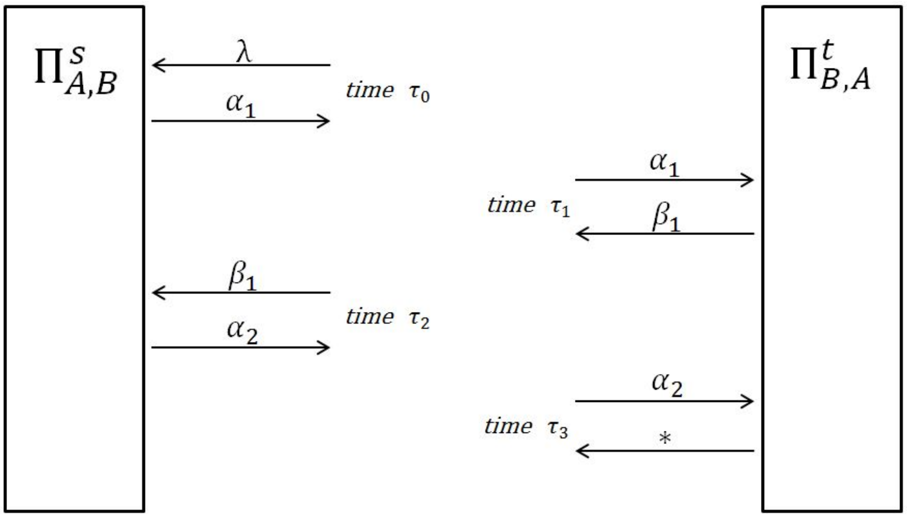
$$(\tau_0, \lambda, \alpha_1), (\tau_2, \beta_1, \alpha_2), (\tau_4, \beta_2, \alpha_3), \cdots, (\tau_{2p-4}, \beta_{p-2}, \alpha_{p-1}), (\tau_{2p-2}, \beta_{p-1}, \alpha_p)$$

Observation 1. Case (1) defines when the conversation of a responder oracle matches the conversation of an initiator oracle. Case (2) defines when the conversation of an initiator oracle matches the conversation of a responder oracle.

Observation 2. To explain our definition we can consider the execution in which $\Pi^s_{A,B}$ is the initiator oracle and $\Pi^t_{B,A}$ is the responder oracle.
If every message that $\Pi^s_{A,B}$ sends out, except maybe the last one, is subsequently delivered to $\Pi^t_{B,A}$, and the response to this message is returned to $\Pi^s_{A,B}$ as its own next message, then we can say that the conversation of $\Pi^t_{B,A}$ matches that of $\Pi^s_{A,B}$.
Similarly, if every message that $\Pi^t_{B,A}$ receives was previously generated by $\Pi^s_{A,B}$, and each message that $\Pi^t_{B,A}$ sends out is subsequently delivered to $\Pi^s_{A,B}$, and the response this message generates is returned to $\Pi^t_{B,A}$ as its own next message, then we can say that the conversation of $\Pi^s_{A,B}$ matches the one of $\Pi^t_{B,A}$.

Example 1. *The following figure reports a matching conversation for a 3-move protocol: the left-hand conversation matches the right-hand one and, omitting arrows associated to* τ_3 *, the right-hand conversation matches the left-hand one.*

Now, let f be a pseudorandom function family.
A pseudorandom function family (PRF) is a collection of efficiently-computable functions which emulate a random oracle in the following way: no probabilistic algorithm that runs in polynomial time can distinguish between a function chosen randomly from the PRF family and a random oracle (a function whose outputs are fixed completely at random).
We denote $f_a : \{0,1\}^{\leq L(k)} \to \{0,1\}^{l(k)}$ the function specified by key a.
In general the length of the key a, the length L of the input of f_a and the length l of the



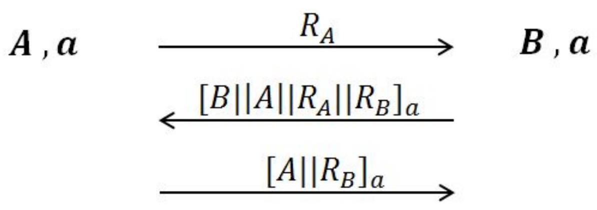
output, are all functions of the security parameter 1^k. Here we assume that the length of the key is k, $L(k) = 4k$ and $l(k) = k$.
For any string $x \in \{0,1\}^{\leq L(k)}$ we define $[x]_a = (x, f_a(x))$: this will serve as an authentication of message x, and for any $i \in A, B$, $[i||x]_a$ will serve as i's authentication of message x.

Definition 7.1.4.3. ***(MAP1)***
We define the following mutual authentication protocol MAP1:

1. A sends to B a random challenge R_A of length k
2. B responds making up a different random challenge R_B and returning $[B||A||R_A||R_B]_a$
3. A checks if this message is of the right form and is correctly tagged as coming from B. This includes checking that the nonce present in the message is the same nonce sent in the first message.
   If it is, the identity of B is proven and A **accepts**. Then A sends B the message $[A||R_B]_a$
4. B checks if this message is of the right form and is correctly tagged as coming from A.
   If it is, the identity of A is proven and he **accepts**

7.1.5 Security proof

Definition 7.1.5.1. ***(secure mutual authentication)***



MAP1 protocol

Π *is a secure mutual authentication protocol if for any polynomial time adversary E,*

1. **(Matching Conversations ⇒ acceptance)**
   If oracles $\Pi^s_{A,B}$ and $\Pi^t_{B,A}$ have matching conversations, then both oracles accept
2. **(Acceptance ⇒ matching conversation)**
   The probability of **No-Matching**$^E(k)$ is negligible.

   No-Matching$^E(k)$ is the event that there exist i, j, s such that $\Pi^s_{i,j}$ accepts but there is no oracle $\Pi^t_{j,i}$ which engaged in a matching conversation.

Definition 7.1.5.2. ***(MAP1^g protocol)***
Let g be a random function from $\{0,1\}^{\leq L(k)}$ to $\{0,1\}^k$, we denote $[x]_g = (x, g(x))$.
We call MAP1^g a protocol that works like MAP1, but in which the parties, instead of sharing a secret key a, share an oracle for g, and they ask him to compute $[x]_g$ every time that MAP1 asks for $[x]_a$.

Observation 3. E doesn't have access to the g oracle.
When $g = f_a$, for randomly chosen a, then, running E for MAP1 coincides to running E for MAP1^g.

Lemma 7.1.5.1. *The probability that an adversary E is successful in running MAP1^g is at most $T_E(k)^2 \cdot 2^{-k}$, where $T_E(k)$ is a polynomial bound on the number of oracle calls made by E; we assume, without loss of generality, that this is at least two.*

Proof. *(of Lemma* (7.1.5.1)*)*

Case 1:

We demonstrate now that: fixed A, B, s, and given an initiator oracle (suppose $\Pi^s_{A,B}$), the probability that it accepts without a matching conversation is at most

$T_E(k) \cdot 2^{-k}$.

We suppose that at time τ_0 oracle $\Pi^s_{A,B}$ sent the challenge R_A. We set:

$$\mathcal{R}(\tau_0) = \{R'_A \in \{0,1\}^k : \exists \tau, t \text{ such that } \Pi^t_{B,A} \text{ was given } R'_A \text{ as first flow at a time } \tau < \tau_0\}$$

If $\Pi^s_{A,B}$ is to accept, then at some time $\tau_2 > \tau_0$ it must receive $[B||A||R_A||R_B]_g$ for some R_B. If no oracle previously output this message, the probability that the adversary can compute it correctly is at most 2^{-k}.

So, consider the case where some oracle did output this flow. The form of the message implies that the oracle which output it must be $\Pi^t_{B,A}$ which earlier received R_A .

The probability of this event happening before time τ_0 is bounded by the probability that $R_A \in \mathcal{R}(\tau_0)$, and this probability is at most $[T_E(k) - 1] \cdot 2^{-k}$. If it happened after time τ_0 then we would have a matching conversation.

We conclude that the probability that $\Pi^s_{A,B}$ accepts but there is no matching conversation is at most $T_E(k) \cdot 2^{-k}$.

Case 2:

We demonstrate that: fixed B, A, t, and given a responder oracle (suppose $\Pi^t_{B,A}$), the probability that it accepts without a matching conversation is at most $T_E(k) \cdot 2^{-k}$.

Suppose at time τ_1 oracle $\Pi^t_{B,A}$ received R_A and responded with $[B||A||R_A||R_B]_g$. If $\Pi^t_{B,A}$ is to accept, then at some time $\tau_3 > \tau_1$ it must receive $[A||R_B]_g$. If no oracle previously output this message, the probability that the adversary can compute it correctly is at most 2^{-k}.

We consider now the case where some oracle did output this message. The form of the flow implies that the oracle which output it must be a $\Pi^s_{A,C}$ oracle.

The interaction of a $\Pi^s_{A,C}$ oracle with E has in general the form:

$$(\tau_0, \lambda, R'_A), (\tau_2, [C||A||R'_A||R'_B]_g, [A||R'_B]_g) \text{ for some } \tau_0 < \tau_2$$

For any such interaction, with probability $1 - 2^{-k}$, there is a $\Pi^u_{C,A}$ oracle which output $[C||A||R'_A||R'_B]_g$ at some time.

Now, if $(u, C) \neq (t, B)$, then the probability that $R'_B = R_B$ is at most

$[T_E(k) - 2] \cdot 2^{-k}$, and so, the probability that the message $[A||R'_B]_g$ leads $\Pi^t_{B,A}$ to accept is at most $[T_E(k) - 2] \cdot 2^{-k}$.
On the contrary, if $(u, C) = (t, B)$, then $R'_A = R_A$, $R'_B = R_B$ and $\tau_0 < \tau_1 < \tau_2 < \tau_3$, that means: the conversations match.
Finally the probability that there is no matching conversation and $\Pi^t_{B,A}$ accepts is at most $T_E(k) \cdot 2^{-k}$

We can conclude that: the probability that exists an oracle which accepts without a matching conversation is at most $T_E(k)$ times the bound obtained before, so:
$T_E(k)^2 \cdot 2^{-k}$. □

Theorem 7.1.5.2. (MAP1 is a secure MA)
Suppose f is a pseudorandom (PRF) function family; then, the protocol MAP1, described above and based on f, is a secure mutual authentication.

Proposition 7.1.5.3. (uniqueness)
Suppose Π *is a secure MA protocol. Let E be any polynomial time adversary.*
The probability of ***Multiple-Match***$^E(k)$ *is negligible.*

Multiple-Match$^E(k)$ *is the event that* $\Pi^s_{i,j}$ *accepts and there are at least two distinct oracles* $\Pi^t_{j,i}$ *and* $\Pi^{t'}_{j,i}$ *which have had matching conversations with* $\Pi^s_{i,j}$.

Proof. *(of Theorem* (7.1.5.2)*)*
We shall prove that the two conditions of the definition (7.1.5.1) are verified:

1. *(Matching Conversations ⇒ acceptance)* is satisfied because it says that when the messages between A and B are faithfully relayed to one another, each party accepts, and this is true for our protocol.

2. *(Acceptance ⇒ matching conversation)*, proof by contradiction:
Suppose that the probability that an adversary E is successful (that is the probability that the event **No-Matching**$^E(k)$ happens) is not negligible. This in particular means:
there is an infinite set K and a constant $c > 0$ such that: $\forall k \in K$ the probability of **No-Matching**(k) is at least k^{-c}.

Now, we consider a polynomial time test T which distinguished random functions from pseudo-random functions.
As an oracle, T receives a function:

$$g : \{0,1\}^{\leq L(k)} \rightarrow \{0,1\}^k$$

chosen according to the following rule: flip a coin C,

$$g = \begin{cases} a\ random\ function & if \quad C = 1 \\ f_a,\ with\ a\ random & otherwise \end{cases}$$

T's job is to predict C with some advantages, that means with a probability $> \frac{1}{2}$;
T's strategy is to run E for the protocol MAP1^g. In this experiment T it self simulates all oracles and answers to E's queries.
If E is successful, then T predicts 0 (i.e. g is pseudo-random), else T predicts 1 (g is random).
Now, when $g = f_a$, running E for MAP1^g coincides to running E for MAP1, but, since we supposed E successful in MAP1 with probability at least k^{-c}, we obtain that T gains an advantage k^{-d}, for some $d > 0$ and $\forall k \in K$. And this implies that: the probability that T can distinguish between a random function and a pseudo-random one is not negligible.
But this contradicts the hypothesis of pseudo-randomness of f.
It follows: the probability of **No-Matching**$^E(k)$ is negligible.

□

7.2 Observations and Conclusions

Observation 4. ***(explanation of the proofs)***
Forgetting the formalism for a moment, we can observe, now, what we really demonstrated before; in few simple words:

- With the Lemma (7.1.5.1) we proved that: if we consider the protocol with a real random function, the probability that the adversary gains a significant advantage in the protocol execution (and the consequential victory) is negligible ($O(2^{-k})$). That means that we can consider this protocol secure.

- In the practice, on the contrary, we can't suppose that the players share a real random function because it would be like sharing an infinite table of random values, so we shall consider pseudo-random functions.
  The idea of theorem's demonstration is that: if the adversary E wins with a significant advantage, then, it means that, in the implementation, a pseudo-random function has been used (the advantage, in fact, can't be significant in the presence of a real one because of the Lemma), and so that it's possible to create a test that differentiates the pseudo-random functions from the real-random ones, using the results of E's authentication attempts. But this is contradicting because a pseudo-random function is not, for definition, polynomial distinguishable from a real-random one.
  It follows: E can't win with a significant advantage, and so the protocol is secure.

Previous proofs concern a mutual authentication protocol, different from the one we implemented. In fact, ours is just a 2-moves one way authentication protocol, where only one player, A for example, wants to prove an other identity (B). In this case, we can model A with an initiator oracle and B with a responder one.
Our protocol will have only 2 moves and we can represent it as follow:

$$\begin{tikzcd}
\boldsymbol{A, a} \arrow[r, shift left, "R_A"] & \boldsymbol{B, a} \arrow[l, shift left, "{[R_A \| B]_a}"]
\end{tikzcd}$$

One-way authentication protocol

Furthermore, we can modify the definition of secure authentication into:

Definition 7.2.0.3. (***secure one way authentication)***
Π is a one way secure authentication protocol if for any polynomial time adversary E:

1. **(Matching Conversations $\Rightarrow$ acceptance)**
   If oracles $\Pi^s_{A,B}$ and $\Pi^t_{B,A}$ have matching conversations, then the initiator oracle

accepts

2. **(Acceptance $\Rightarrow$ matching conversation)**
   The probability of **No-Matching**$^E(k)$ is negligible.

   No-Matching$^E(k)$ is the event where the initiator oracle accepts but there is no responder oracle which engaged in a matching conversation.

And, following this new definition, we can say that our protocol is a **secure one-way authentication**.
That means that, an adversary E can't win in this authentication process with a significant advantage, and the only way for her to enter in the conversation between the two players and makes the initiator oracle accept, is simply to read the messages sent and replay them to the right receiver without changing their order or their content, as shown in the following figure:

$$\begin{tikzcd}
\boldsymbol{A, a} \arrow[r, shift left, "R_A"] & \boldsymbol{E} \arrow[l, shift left, "{[R_A \,\|\, B]_a}"] \arrow[r, shift left, "R_A"] & \boldsymbol{B, a} \arrow[l, shift left, "{[R_A \,\|\, B]_a}"]
\end{tikzcd}$$

E's eavesdropping permitted

Final Conclusions

Here we arrive at the end of this report; we should have acquired now a general view of automotive security problems and some glimpses of the solutions that cars world is developing.
We know perfectly that our work is incomplete and the subject is more ample than this, in fact, the research of solutions has just started. What we hope is that this paper showed how serious are the threat and the necessity of improving in this field for the good of all.
Lastly it's always interesting to discover unexpected environments where Math in general, and Cryptography in particular, can be applied in our practical life.

Glossary

City Option increases the steering effort of the steering wheel, helping the driver in certain situations, usually in a city context.

LKA Lane Keep Assistant detects, under certain conditions, if the vehicle is veering off the road. If this is the case it will adjust the steering wheel to correct the automobile's course.

PCS Pre-Collision System determines if the car is going to collide with something, if this is the case it will alert the driver and apply the brakes, **regardless of the state of the acceleration pedal**.

RDS Radio Data System is a communications protocol standard for embedding small amounts of digital information in conventional FM radio broadcasts.

boot the totality of the process that are executed during the starting phase in order to turn on a computer.

engine maps To control the engine's Air/Fuel ratio, ignition timing, idle speed, electronic valves, etc. the ECM has to calculate specific functions, that depend on more values like RPM, gas pedal,... To save time and decrease difficulty, the ECM doesn't calculate these functions every time, but, considering the input parameters, it chooses the output configuration using multidimensional performance maps.

worm is a malware computer program that replicates itself in order to spread to other computers (`http://en.wikipedia.org/wiki/Computer_worm`). Here the worm passes from one device to another.

Bibliography

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, '*Experimental Security Analysis of a Modern Automobile*', In D. Evans and G. Vigna, editors, IEEE Symposium on Security and Privacy. IEEE Computer Society, May 2010.

[2] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno, '*Comprehensive Experimental Analyses of Automotive Attack Surfaces*', In USENIX Security Symposium, 2011

[3] Dr. Charlie Miller, Chris Valasek, '*Adventures in Automotive Networks and Control Units*', 2013

[4] OpenGarages, '*Car Hacker's Handbook*', `http://opengarages.org/handbook/`

[5] Todd E. Humphreys, Brent M. Ledvina, Mark L. Psiaki, Brady W. OHanlon, and Paul M. Kintner, '*Assessing the Spoofing Threat: Development of a Portable GPS Civilian Spoofer*', in Proceedings of the ION GNSS Meeting, (Savannah, GA), Institute of Navigation, 2008

[6] Stephen C. Bono, Matthew Green, Adam Stubblefield, Ari Juels, Aviel D. Rubin, Michael Szydlo, '*Security Analysis of a Cryptographically-Enabled RFID Device*', In USENIX Security Symposium, 2005

[7] Aurelien Francillon, Boris Danev, Srdjan Capkun, '*Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars*', In A. Perrig, editor, NDSS 2011. ISOC, Feb. 2011

[8] Freescale Semiconductor: Geoff Emerson, Jurgen Frank, Stefan Luellmann Applications Engineering, Microcontroller Solutions Group, '*Using the Cryptographic Service Engine (CSE)*', 2011

[9] W.Trappe, L.C.Washington, '*Introduction to Cryptography (with Coding Theory)*', Pearson-Pentice Hall, 2009

[10] Niels Ferguson, Bruce Schneier, '*Practical Cryptography*', John Wiley & Sons, 2003

[11] Daniel J. Bernstein, '*Cache-timing attacks on AES*', Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, 2005

[12] Thomas Eisenbarth1, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani, '*On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme*', D. Wagner (Ed.): CRYPTO 2008, LNCS 5157, pp. 203220, 2008

[13] Amir Moradi and Timo Kasper, '*A New Remote Keyless Entry System Resistant to Power Analysis Attacks*', 7th International Conference on Information, Communications and Signal Processing, ICICS 2009

[14] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar, '*Breaking KeeLoq in a Flash*', AFRICACRYPT 2009

[15] Andrey Bogdanov, '*Cryptanalysis of the KeeLoq block cipher*', `https://eprint.iacr.org/2007/055.pdf`

[16] E-safety vehicle intrusion protected applications, '*Project Summary*', April 2012, `http://www.evita-project.org/Publications/EVITAD0.pdf`

[17] A.Menezes, P.C. van Oorschot, S.A. Vanstone, '*Handbook of Applied Cryptography*', CRC Press, 1997

[18] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger, '*Biclique Cryptanalysis of the Full AES*', ASIACRYPT 2011

[19] MathWorks Automotive Advisory Board (MAAB), '*Control Algorithm Modeling Guidelines using MATLAB, Simulink, and Stateflow*', Version 3.0

[20] Mihir Bellare, Phillip Rogaway, '*Entity Authentication and Key Distribution*', August 1993

[21] Yang, Song Y., '*An introduction to: Formal Languages and Machine Computation*', World Scientific, 1998

Webography

[Web: 1] http://www.digitaltrends.com/cars/can-your-car-be-hacked-car-\
hacking-threats-analyzed/

[Web: 2] http://resources.infosecinstitute.com/car-hacking-safety-without-\
security/

[Web: 3] www.autosec.org

[Web: 4] www.escar.info

[Web: 5] http://www.forbes.com/sites/andygreenberg/2014/02/05/
this-iphone-\sized-device-can-hack-a-car-researchers-plan-to-demonstrate/

[Web: 6] http://securityaffairs.co/wordpress/22070/hacking/
can-hacking-tools.html

[Web: 7] http://cmu95752.wordpress.com/2012/07/21/automotive-telematics\
infotainment-systems-security-vulnerabilities-and-risks/

[Web: 8] http://en.wikipedia.org/

[Web: 9] http://www.cnn.com/2007/US/09/26/power.at.risk/index.html

[Web: 10] http://www.freescale.com/

[Web: 11] http://www.havocscope.com/number-of-counterfeit-airbags-available\
-on-market/

[Web: 12] http://media.gm.com/media/me/en/gm/news.detail.html/content/Pages/news/me/en/2013/gm/General-Motors-Highlights-Issue-of-Counterfeit\-Parts-in-the-Middle-East.html

[Web: 13] https://techinfo.toyota.com/techInfoPortal/appmanager/t3/ti?_pageLabel=ti_j2534_device&_nfpb=true

[Web: 14] http://electronics.howstuffworks.com/gadgets/automotive/rfid-ignition-system.htm

[Web: 15] https://techinfo.toyota.com

[Web: 16] http://timesofindia.indiatimes.com/city/patna/Fake-spares-cause-20-of-mishaps-Reports/articleshow/22174552.cms

[Web: 17] http://www.eetimes.com/document.asp?doc_id=1279619

[Web: 18] http://www.bmw.com/com/en/insights/technology/technology_guide/articles/electronic_immobiliser.html

[Web: 19] http://www.e38.org/EWS.pdf

[Web: 20] http://www.ecudoctors.com/bmw-ecu-dme-reprogrammed-and-rebuilt.html

[Web: 21] http://www.locksmithsacramento.net/Transponder_Keys.html

[Web: 22] http://advanced-keys.co.uk/product_ref.php

[Web: 23] http://electronics.stackexchange.com/questions/85657/rolling-code-explanation

[Web: 24] http://computer.howstuffworks.com/rom3.htmhttp://computer.howstuffworks.com/rom3.htm

[Web: 25] http://en.wikipedia.org/wiki/Programmable_read-only_memory

[Web: 26] http://en.wikipedia.org/wiki/Fuse_(electrical)

[Web: 27] http://www.evita-project.org/index.html

[Web: 28] http://www.top500.org/lists/2014/11/

[Web: 29] http://argus-sec.com/

[Web: 30] http://www.tower-sec.com/

[Web: 31] http://www.cisco.com/

[Web: 32] http://it.mathworks.com/help/simulink/gs/product-description.html

[Web: 33] http://www.mathworks.com/products/stateflow/

[Web: 34] http://www.mathworks.com/solutions/automotive/standards/maab.html

[Web: 35] http://www.mathworks.com/help/slvnv/ug/types-of-model-coverage.html#bqi9xaj