



## Assignment of master's thesis

<b>Title:</b>	Authentication Effectiveness in Vehicle Unified Diagnostic Services
<b>Student:</b>	Bc. Jakub Weisl
<b>Supervisor:</b>	Ing. Jiří Dostál, Ph.D.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Computer Security
<b>Department:</b>	Department of Information Security
<b>Validity:</b>	until the end of summer semester 2023/2024

### Instructions

Modern car design typically includes various Electronic Computing Units (ECUs) connected through communication buses and computer networks. ECUs also oversee and report the technical status of different car units, ensuring proper communication between them. They process the incoming signals and deploy actions based on gathered information, e.g., powertrain control or brake assist.

There is also a diagnostic unit present. It is responsible for reporting the health status of the car and communicating with service devices. It employs a server-client communication model using the CAN bus. As an overlay layer over the CAN bus, there is a Unified Diagnostic System (UDS) protocol, defined by ISO 14229-2020. All the functions of the UDS protocol are grouped into services. The main focus of this thesis is the authentication service defined in chapter 10.6 of the previously mentioned ISO standard.

The authentication service offers two authentication schemes. The first is authentication using PKI and the second is challenge-response authentication using symmetric cryptography. The goals of this diploma thesis are:

1. Design a system for measuring the effectiveness of different authentication schemes.
2. Create a testing environment emulating communication between the diagnostic unit and client station based on ISO 14229-2020 standard.
3. Implement different authentication mechanisms. Implementations must cover both



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

symmetric and asymmetric cryptography.

4. Assess the effectiveness of the implemented authentication mechanisms using the designed system and consider further strengths and weaknesses of implemented options.



Master's thesis

**AUTHENTICATION  
EFFECTIVENESS IN  
VEHICLE UNIFIED  
DIAGNOSTIC SERVICES**

**Bc. Jakub Weisl**

Faculty of Information Technology  
Department of Information Security  
Supervisor: Ing. Jiří Dostál, Ph.D.  
January 11, 2024

Czech Technical University in Prague  
Faculty of Information Technology

© 2023 Bc. Jakub Weisl. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Weisl Jakub. *Authentication Effectiveness in Vehicle Unified Diagnostic Services*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

# Contents

Acknowledgments	v
Declaration	vi
Abstract	vii
Introduction	1
<b>1 Analysis</b>	<b>3</b>
1.1 CAN bus	3
1.1.1 History	3
1.1.2 CAN 2.0	4
1.1.3 CAN-FD	5
1.2 ISO-TP protocol	6
1.3 UDS	6
1.4 Cryptography	6
1.4.1 RSA	6
1.4.2 ECC	7
1.4.3 HMAC	7
<b>2 Implementation</b>	<b>9</b>
2.1 Infrastructure	9
2.2 Source code description	10
2.2.1 Class structure	10
2.2.2 Implemented features	11
2.3 Methodology of Experiment	12
<b>3 Evaluation</b>	<b>15</b>
3.1 measurement interpretation	15
3.2 unmeasurable variables	15
<b>4 Conclusion</b>	<b>17</b>
<b>A Attachments</b>	<b>19</b>
Content of attached memory storage	23

## List of Figures

1.1	CAN bus protocol on ISO-OSI model and its ties to ISO 11899 series . . . . .	4
1.2	Structure of CAN2.0 A frame . . . . .	4
1.3	Start of a CAN-FD frame . . . . .	6
1.4	End of a CAN-FD frame . . . . .	6

## List of Tables

## List of code listings

2.1	Interface for calculating proof of ownership . . . . .	11
2.2	Interface for checking proof of ownership . . . . .	11
2.3	Declaration of setsocketopt function . . . . .	11

*Chtěl bych poděkovat především sit amet, consectetur adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on January 11, 2024

.....



## Abstract

Fill in abstract of this thesis in English language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

**Keywords** enter, comma, separated, list, of, keywords, in, ENGLISH

## Abstrakt

Fill in abstract of this thesis in Czech language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

**Klíčová slova** enter, comma, separated, list, of, keywords, in, CZECH


## List of abbreviations

- BRS** Bit Rate Switch. 6
- CAN** Control Area Network. 3-6, 9
- CRC** Cyclic Redundancy Check. 5
- DLC** Data length code. 5, 6
- ECU** Electronic computing unit. 3, 9
- EOF** End of Frame. 5
- ESI** Error Status Indicator. 6
- HMAC** Hash-based message authentication code. 11
- LAN** Local area network. 9
- NAT** Network address translation. 9
- OBD** On Board diagnostic. 3
- PKI** Public key infrastructure. 11
- POW** Proof of ownership. 11
- RRS** Remote Request Substitution. 5
- RTR** Remote Transmission Request. 5
- SBC** Stuff Bit Count. 6
- SOF** Start of Frame. 5
- UDS** Unified diagnostic service. 3, 11, 12

# Introduction

*test test test test*





# Chapter 1

## Analysis

Focuses of this chapter is mainly on defining key topics of this thesis and bringing up the necessary context to them. Namely this chapter will briefly shows, how the CAN protocol and its overlay ISO-TP protocol work and how these two are connected. How they are used in modern cars and what UDS is. Regarding UDS the main focus is to explain details of *service 29* which is the core topic of this thesis. At last the choice of particular cryptographic algorithms is reasoned.

Every modern car or other vehicle is equipped with many digital systems which have various purposes from timing fuel injection into the engine across breaking assistants to keeping the the temperature inside a cockpit. These various systems are run by various ECUs in the vehicle. Modern car can have more than 150 of ECUs[1]. In order to work properly the ECUs must gather lot of information from various sensors and from each other. For that purpose the CAN protocol is used. The information exchange is done through various proprietary protocols, however in case of communication with the outside word (e.g. diagnostic station, emission control) there are standardized protocols (e.g. OBD 2, UDS). For all of the protocols CAN protocol is used as physical and data link layer as defined on ISO-OSI model.

### 1.1 CAN bus

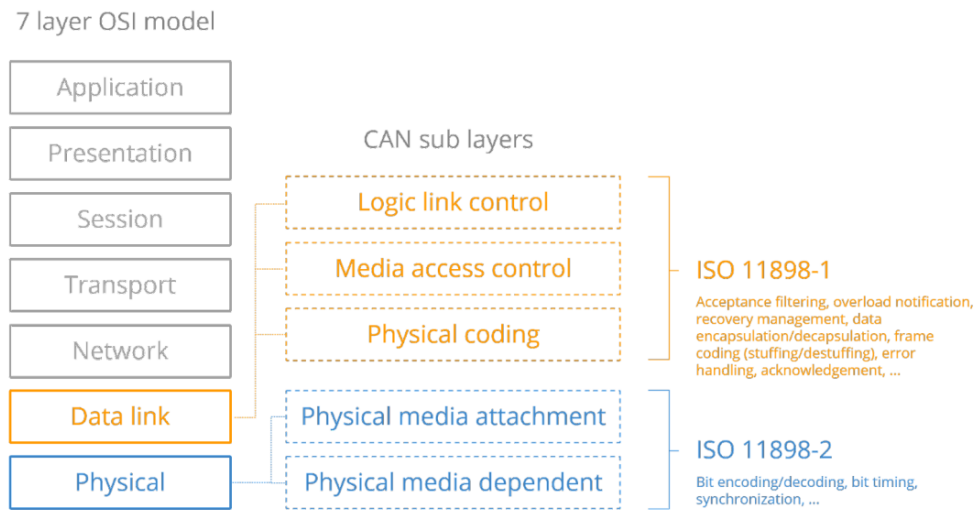
CAN protocol is a real time communication protocol, communicating over central bus. Every node in a network is broadcasting frames to all the other nodes. CAN protocol is prioritising messages based on CAN ID 1.1.2.

All the messages are separated by 3 zero bits.[2]

#### 1.1.1 History

CAN bus is a link layer protocol first developed by Bosch company in 1986. Its later version was introduced in 1993 under name CAN2.0. Since then CAN bus became standard communication protocol in automotive industry. In 1993 CAN was adopted as international standard in ISO 11898. Later the standard was broaden into series where standard ISO 11898-1 defines the data link layer and ISO 11898-2 defines the parameters for the physical layer.

In 2012 Bosch released CAN-FD, where FD stands for flexible data rate. This upgrade of CAN protocol was standardised in 2015 into previously mentioned ISO norm.[3]



■ **Figure 1.1** CAN bus protocol on ISO-OSI model and its ties to ISO 11899 series

### 1.1.2 CAN 2.0

CAN2.0 is currently most used version of CAN bus. It is used in various types of vehicles for internal communication. There are 2 types of CAN protocols CAN2.0 A and CAN2.0 B, which differ in length of CAN ID (CAN2.0 A 11 bits, CAN2.0 B accepts both 11 bit and 29 bits IDs).[2] The CAN2.0 B is usually used in heavy-duty vehicles, which use J1939 protocol in higher levels of communication.[3]

Standard CAN communication has 4 types of frames:

- Data Frame - Standard CAN frame carrying data.
- Remote Frame - CAN frame requesting data from another node.
- Error Frame - Error frame is transmitted by any node which encounters an error on a common bus.
- Overload Frame - This frame indicates additional delays in communication due to node overload.

CAN frames can carry up to 8 B of payload and can achieve speed up to 1 Mb/s. Maximal speed can be achieved only with cables length less then 40 meters than the speed decreases. Can frames have defined structure that can be seen on Figure 1.2.



■ **Figure 1.2** Structure of CAN2.0 A frame

[3]

The frame carries another 44 bits of information. Their meaning is following:

- SOF: The Start of Frame is set to 0, to indicate start of a message
- ID: Frame identifier, as mentioned earlier CAN2.0 B has option of both 11 bits or 29 bits ID. Lower IDs have higher priority.
- RTR: The Remote Transmission Request this flag indicates if a frame requests data (remote frame set to 1) or carries data (data frame set to 0).
- Control: Control has length of 6 bits where first 2 bits are reserved and must be set to 0 and next 4 bits contains DLC indicating the length of a payload.
- Data: Payload.
- CRC: The Cyclic Redundancy Check ensures the payload integrity.
- ACK: The ACK slot indicates if the node has acknowledged and received the data correctly. If everything is correct receiving node sends back same message, but with ACK bit set to 0.
- EOF: The End of Frame.

As mentioned before communication on common bus is broadcasted from transmitter to all other nodes this means that only one node can transmit at the time. In the case that bus is idle any can transmit its data as long as node with message with higher priority needs to start transmitting. In case more nodes needs to broadcast some information in same the priority of the message is considered. The arbitration process between the nodes is called *carrier sense multiple access with collision detection*. The arbitration field is put together from the SOF bit and ID bits. Since the SOF of the same for all the messages the ID field is what decides. The arbitration scheme is following:

1. All nodes start broadcasting at once.
2. The bus acts as bitwise AND, so if any node is transmitting 0 bit and some other nodes are trying to transmit 1 at the same position. The bit is zeroed.
3. Node reads back written bit from a bus if it is the same as last transmitted bit it keeps transmitting if not the node stops transmitting a waits for the bus to be idle, or for the the start of new message.

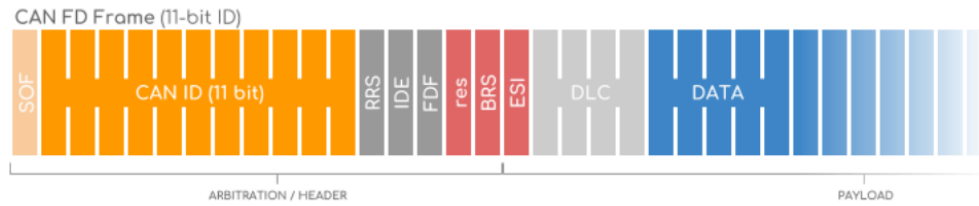
[2]

### 1.1.3 CAN-FD

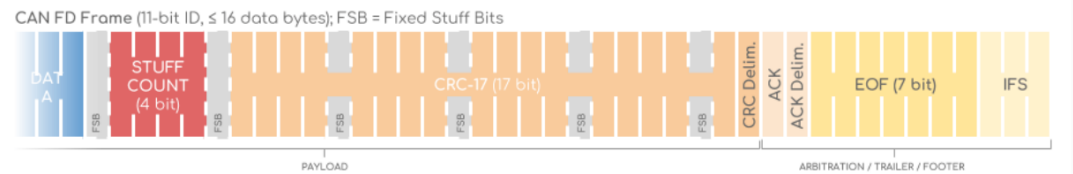
CAN-FD is a upgrade of classical CAN2.0 in several ways. It allows variable payload size between 8 and 64 bytes, although the header is bigger than classic CAN, so the in case of 8 byte payload the overhead is bigger than with standard CAN. Another improvement is that CAN-FD is capable of higher speed, theoretically up to 12 Mb/s when sending data. However technical specification in 11898-2:2016 sets standards for 5 Mb/s. CAN-FD frame has the following structure:

As can be seen at 1.3 the start of the frame is the same as classical CAN so the arbitration process does not change for the CAN-FD. However after CAN ID field there are some major changes. The frame after ID is composed of following parts:

- RRS replaces 1.1.2 in classical CAN. CAN-FD si not supporting remote requests and Remote Request Substitution (RRS) is always set to 0.
- IDE: IDE is reserved bit as in CAN frames and is set 0.
- FDF: FDF is also reserved bit but compare to classical CAN is set to 1.



■ **Figure 1.3** Start of a CAN-FD frame



■ **Figure 1.4** End of a CAN-FD frame

- 3 bits newly introduced in CAN-FD:
  - res: Res reserved bit in CAN-FD protocol and it is set to 0.
  - BRS: The Bit Rate Switch (BRS) indicates mode in which payload will be sent. If set to 0 data section of frame are sent in same rate as header of frame. If BRS is set to 1 then payload is sent at higher bitrate.
  - ESI: The Error Status Indicator (ESI) is set to 0 and indicates that transmitter is in error active mode, otherwise it is in error passive mode.
- DLC: 4 bits of DLC has the same meaning as in classical CAN (1.1.2)
- Data: Payload carried by a CAN frame. Payload can be 1-8 and then 12, 16, 20, 24, 32, 48 or 64 bytes long.
- SBC: The Stuff Bit Count (SBC)
- CRC
- ACK
- EOF
- IFS

[3]

## 1.2 ISO-TP protocol

## 1.3 UDS

## 1.4 Cryptography

### 1.4.1 RSA



**1.4.2 ECC**

**1.4.3 HMAC**



# Implementation

This chapter has goal to show and explain the structure of the program which was used to carry out the experiment and simulate the authentication process between client endpoint and ECU. It also names other software tools which were used on the experiment and reasons theirs choice.

The whole emulation of authentication process to ECU is implemented in one piece of software which can simulate either a client side or a server side of the authentication process, based on initial configuration. The server must be run with configuration for all supported authentication mechanisms and cryptography protocols to be able to answer all the supported requests.

When the software is run in client mode it is necessary to supply only encryption keys required for the chosen authentication mechanism and cryptography protocol.

## 2.1 Infrastructure

Whole experiment was done in virtual environment using Virtual box hypervisor and Canneloni tool <sup>1</sup>, which provides connection using the canbus protocol between virtual client and virtual server. Although CAN bus is a layer 2 protocol on ISO/OSI model Canneloni channels CAN bus protocol over TCP/IP. The original plan was to use virtual serial line between the client and the server. However slcand <sup>2</sup> tool which is used to handle CAN bus protocol over serial line does not support CAN FD version of the protocol. This raised a problem, when variable size of CAN FD packets must be used to simulate different throughput of the channel, because linux kernel module which handles CAN bus communication is not able send CAN frames in shorter intervals than 100  $\mu$ s which is not enough for needs of this experiment. This restriction has its reason because maximal speed which CAN2.0 bus protocol can achieve in reality is 1 Mb/s.

Interconnection of the client and the server over the TCP/IP had and other unexpected advantage in more stable connection than was the connection over the virtual serial line which led to significantly fewer erroneous runs during the measurement. With use of serial line there were tens or even lower hundreds of erroneous runs per measurements which contained 2000 runs of the program. Connection over TCP/IP resulted with up to 10 erroneous runs which were caused by interface buffer to be filled and was easily solved by extending buffer size on a network interface and so it was possible to achieve 100% of successful runs.

Both virtual machines were equipped with two virtual network cards, one connected to the private LAN to establish connection between client and server and to ensure minimal interference in the communication. Second virtual card was connected to the internet through NAT to be able to communicate with git repository which held the latest version of the code. In order

<sup>1</sup><https://github.com/mguentner/cannelloni>

<sup>2</sup><https://github.com/linux-can/can-utils/tree/master>

to avoid any unforeseen problems with portability between development environment a testing virtual machines the code was always compiled on target machines.

Virtual machines used as client as server both run publicly available Linux distribution Ubuntu server version 22.04.02 LTS with `can-utils`<sup>3</sup> package installed. For the compilation of source code it is also necessary to download Linux header files and compiler for C++ source code with C++ standard library and OpenSSL library.

## 2.2 Source code description

The goal of this part is to introduce to the reader the most important parts and some of the interesting features of the implementation of ECU authentication mechanism according to ISO

The source code of the software is completely written in C++ and divided into several class.

### 2.2.1 Class structure

Whole software is build from set of classes which most of them is common for both modes the server and the client. Development of the software went through several stages, and during them was almost completely redesigned. First designs worked implemented the client and the server as two independent pieces software. This design proved inefficient since it duplicated approximately 60% of classes. Later stages worked with one piece of software which incorporated both the client class and the server class which each provides role specific methods, and both inherit grate part of their functionalities from common parent.

The fundamental class is class named *connection*. Its main purpose is to setup CAN socket and provide methods for sending and receiving messages through a socket. The *connection* class mainly uses `ISO-TP` kernel module<sup>4</sup> and `SocketCAN` kernel module, which both are now included in standard kernel since version 5.10. Work with `SocketCAN` kernel module was same as with standard C++ network sockets this module is also well documented in kernel documentation [4]. The work with sockets is not very comfortable in C++, but this kernel module can work only with individual CAN packets.

In order to simplify work with incoming and outgoing traffic `ISO-TP` kernel module was used. This module implements `ISO-TP` protocol, which manages CAN packets into complete messages and creates some sort of sessions. Work with this module is bit more trickier than with the `SocketCAN` kernel module, because of missing documentation. Only clues how to use the module lies in the source code comments and slide presentation from author of the module<sup>5</sup>.

The class which holds most of the functions a makes base for both the client and the server is class called *service\_29*. This class holds message structures as defined in ISO 14229-1:2020. This class holds also methods for cryptography operations done during the communication and some other support functions like loading encryption keys, creating random challenges etc.

Classes representing the client and the server both inherit structures and methods from previously mentioned *service\_29* class. The *client* class overloads *auth*, which starts the whole authentication process, and based passed parameters, it is determined which kind of authentication is used.

The *server* class represents the server and its main purpose is to listen for incoming authentication communication. All the necessary data are passed to *server* class constructor method, and the *auth* method only starts listening for incoming connection. The class is constructed in a way, that it can handle all kinds of authentication mechanism as defined in ISO 14229-1:2020 and all currently implemented encryption algorithms based on information in incoming packets.

<sup>3</sup><https://packages.ubuntu.com/jammy/net/can-utils>

<sup>4</sup><https://github.com/hartkopp/can-isotp>

<sup>5</sup>[https://s3.eu-central-1.amazonaws.com/cancia-de/documents/proceedings/slides/hartkopp\\_slides\\_15icc.pdf](https://s3.eu-central-1.amazonaws.com/cancia-de/documents/proceedings/slides/hartkopp_slides_15icc.pdf)

Both classes the *client* and *server* both hold implementation of methods implementing communication scheme as defined in earlier mentioned ISO norm and shown in 1.3.

## 2.2.2 Implemented features

This implementation of UDS service 29 communication holds few options and set ups which are worth mentioning.

Implementation of cryptographic function uses openssl crypto library. There are two types of cryptographic operations implemented first one calculating POW and second checking incoming POW. Method calculating POW must have interface in following:

```

1  static const std::vector<uint8_t> calculate_POW (const std::vector<uint8_t>
2  & key,
3  const std::vector<uint8_t>
4  & pow_base);

```

■ **Code listing 2.1** Interface for calculating proof of ownership

Method for checking received POW must have interface in this form:

```

1  static const std::vector<uint8_t> check_POW (const std::vector<uint8_t> &
2  key,
3  const std::vector<uint8_t> &
4  rcv_pow,
5  const std::vector<uint8_t> &
6  pow_base);

```

■ **Code listing 2.2** Interface for checking proof of ownership

The static keyword is only necessary if implemented functions are also methods of *service\_29* class. They can be also implemented as stand alone functions outside of the class.

Use of these interfaces provides possibility adding various encryption algorithms and their easy incorporation into this project. When adding new cryptographic algorithm into project, it must be add into switch statement, which is located in the method *service\_29::\_match\_alg*, where addresses of new methods are assigned to the class variables. New methods must be also added into main function which handles input and parameters from command line.

The current implementation allows the use of HMAC cryptographic protocol and all asymmetric ciphers supported by OpenSSL library in both scenarios challenge response or with use of PKI. The broad spectrum of asymmetric ciphers available for use is achieved by using OpenSSL *d2i* functions which can determine correct cipher based on provided DER encoded public key or from certificate which directly holds desired information.

Another note worthy thing is set up of CAN socket when using ISO-TP kernel module. The only available documentation is in the source code of the module. In this project the set up is done in the constructor of *connection* class. The set up is done through function *setsockoptopt* and structures defined in the kernel module. The function has the following declaration:

```

1  int setsockoptopt(int socket,
2  int level,
3  int option_name,
4  const void *option_value,
5  socklen_t option_len);
6

```

■ **Code listing 2.3** Declaration of setsockoptopt function

Except the *socket* parameter, all the other parameters are defined in the kernel module. Parameters *level* and *option\_name* are defined as macros at the start of header of file of the

module. According to option name proper structure must be passed as the *option\_value* and also its length as *option\_len* parameter, which are defined in the same file. There are three structures defined which each is responsible for setting up different part of ISOTP protocol. There is *can\_isotp\_options* structure, which handles all local setting of ISO-TP communication. There are 2 parts of this structure worth mentioning the first one is *can\_isotp\_options.flags*. This is unsigned 32 bits data type which holds values of all setup flags defined in the header file. These flags are key to set up correct behavior of the communication. The second one is *can\_isotp\_options.frame\_txtime*, which holds time interval between individual frames. However this time interval is used only when *CAN\_ISOTP\_FORCE\_TXSTMIN* flag is set up, or there is no Flow Control Frame received. In other cases the value in *can\_isotp\_options.frame\_txtime* is overwritten by value received in Flow Control Frame. The structure *can\_isotp\_fc\_options* sets up the values received in already mentioned Flow Control Frame specially the time interval between frames, block size and maximum number of wait frames. The last structure named *can\_isotp\_ll\_options* enables use of CAN-FD packets and also sets up a size of a payload.

The last thing to point out from the implementation of the UDS service 29 is the way encryption keys and certificates are handled in the client and the server. Both instances must handle those differently because server must be able to handle any incoming implemented cipher suit and so needs list of all cryptographic keys available. But the client only needs a chosen encryption and correct decryption key. This need resulted in different requirements for passed arguments to some parameters.

The difference is with parameters *-k* and *-p*. With parameter *-k* the the client expects path to client's private key, similarly with parameter *-p* the client expects path to server's public key. However the server with these parameters expects list of all keys available. The server's *-p* parameter expects path to file which holds structured information about private keys available to the server, this includes keys to symmetric ciphers. The file delimiter is `:` character and the structure of the file is following:

```
server_ID : cipher_ID : path_to_private_key
```

*Server\_ID* is defined with parameter *-i* and must match with at least one *server\_ID* in the configuration file. *Cipher\_ID* is defined in program and shown in help of the program. The *-k* parameter requires also path to the file but this file holds structured information about public keys of the clients. This file is used only for authentication with asymmetric encryption but for simplification and possible implementation of cryptographic algorithms not supported by OpenSSL library it holds same structure as file for storing paths to private keys, except instead of holding *server\_IDs* it holds *client\_IDs*. If in an incoming connection the asymmetric cryptography is identified a path to particular public key is matched based on *client\_ID* in the incoming message.

## 2.3 Methodology of Experiment

The experiment was carried on the same virtual infrastructure, it was developed. In order to minimize influence of guest operating system load and the load of host operating system as well the client was newly started and waited to properly finish in each run. The server on other virtual machine runs all the time and waits for all incoming connections.

The time was measured only on the client which initiated the communication and the stop watch was started with the call to method to start the communication and stopped with the return from this method. This way the results were not influenced by the parameters and arguments ingestion and checks.

In the experiment the main focus was on time complexity of individual authentication mechanisms. The secondary focus is an amount of transferred data, but this dimension of experiments is constant through all runs.

In order to get reliable data of the time complexity of the individual authentication mechanism the client was run 2000 times for bidirectional and also unidirectional authentication. All the runs

were done with the same arguments. In order to get closer to real environment all the encryption keys were generated using OpenSSL command line tool with the standard recommended length of the keys. For the RSA the key length was chosen to 2048 bits, ECC used key of length 256 bits and for HMAC crypto algorithm was used 256 bits of key. All the runs were made with the same cryptographic keys.







## Chapter 3

# Evaluation

**3.1** measurement interpretation

**3.2** unmeasurable variables





**Chapter 4**

**Conclusion**



..... Appendix A

# Attachments

Sem přijde to, co nepatří do hlavní části.



# Bibliography

1. CHARETTE, Robert N. *HOW SOFTWARE IS EATING THE CAR* [online]. IEEE, [n.d.]. Available also from: <https://spectrum.ieee.org/software-eating-car>.
2. J. A. COOK, J. S. Freudenberg. *Control Area Network (CAN)* [online]. 2008. Tech. rep. Michigan University. Available also from: [https://www.eecs.umich.edu/courses/eecs461/doc/CAN\\_notes.pdf](https://www.eecs.umich.edu/courses/eecs461/doc/CAN_notes.pdf).
3. *CAN bus the ultimate guide* [online]. CSS Electronics, 2023. Tech. rep.
4. *The Linux Kernel* [online]. Available also from: <https://www.kernel.org/doc/html/latest/networking/can.html>.





# Content of attached memory storage

Makefile .....	Makefile for setup Cannelloni tool
└─ exe .....	adresář se spustitelnou formou implementace
└─ src	
└─ impl .....	zdrojové kódy implementace
└─ thesis .....	zdrojová forma práce ve formátu L <sup>A</sup> T <sub>E</sub> X
└─ text .....	text práce
└─ thesis.pdf .....	text práce ve formátu PDF