Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Computer Engineering

a.y 2023/2024

Graduation Session October 2024

Implementation of Non-Standard Digital Signature Method in Vehicle-to-Everything Environments

Supervisors

Prof. Fulvio VALENZA

Eng. Leonardo DE CANDIA

Candidate

Matteo PEDONE

A Luciano,
colui che da professore mi ha stimato e da amico mi ha sostenuto,
al suo amore sincero e alla sua bontà d'animo

"Sei sempre il mio Matteo"(8 Luglio 2019)

Index

Acronyms

3GPP 3rd Generation Partnership Project

AA Authorization Authority

ADAS Automotive advanced driver-assistance systems

ASIL Automotive Safety Integrity Level

AT Authorization Ticket

BS Base Station

C-ITS Cooperative Intelligent Transport Systems

C-V2X Cellular Vehicular to Everything

CAM Cooperative Awareness Message

CRL Certificate Revocation List

CTL Certificate Trusted List

DC Distribution Center

DENM Decentralized Environmental Notification Message

DSRC Dedicated Short Range Communication

EA Enrollment Authority

ECDA Enhanced Distributed Channel Access

ECDSA Elliptic Curve Digital Signature Algorithm

ECIES Elliptic Curve Integrated Encryption Scheme

EDR Event Data Recorder

GNSS Global Navigation Satellite System

GPS Global Positioning System

I2I Infrastructure-To-Infrastructure

ITS Intelligent Transportation Systems

LTE V2X Long-Term Evolution Vehicular to Everything

NIST National Institute of Standards and Technology

OBU On Board Unit

Post Quantum Criptography Post Quantum Criptography

RCA Root Certification Authority

RSU Road Side Unit

TA Trusted Authority

TLM Trust List Manager

ToC Transition of Control

TPD Tamper-Proof-Device

UE User Equipment

V2I Vehicle-To-Infrastructure

V2V Vehicle-To-Vehicle

V2X Vehicle-To-Everithing

VANET Vehicular Ad-hoc Network

WAVE Wireless Access in Vehicular Environments

Chapter 1

Introduction

This thesis work focuses on a framework that enables the testing and evaluation of V2X communication integrated with Post-Quantum-Cryptography algorithms, aiming to identify a point of confluence between these two largely relevant areas of study.

During the thesis study, an analysis is conducted on how V2X communication aims to enhance road safety by increasing road users' awareness of the surrounding environment. However, since V2X relies mainly on public-key-based systems, the advent of quantum attacks could pose serious risks to the security of the whole V2X communication environment.

In the development of this thesis, the process that led to choosing the framework used to integrate PQC algorithms into the V2X standards is highlighted. The challenges encountered and the decisions made to overcome them are described. The thesis work presents two main focuses: generating digital certificates supporting PQC keys and signatures for V2X entities and integrating these certificates into the V2X framework to produce new non-standard CAM, appending the new PQC signatures.

The objective of replacing the current V2X standard algorithms for the digital signature process, based on elliptic curves, with a Post-Quantum algorithm is examined through the results of the developed implementations. A relative analysis is performed to assess whether the chosen approach in the thesis case study is effective and has the potential for future development.

This thesis work is inserted in a set of researches conducted by **Nardò Technical Center - Porsche Engineering**, the Italian subsidiary of Porsche Engineering, which is active in various automotive domains for software development.

Chapter 2

V2X communication

2.1 VANET and ITS

Vehicular Ad-hoc Network (VANET) is an important area of research in the automotive domain. The targets are:

- improve the safety of vehicles on the road
- traffic management
- congestion monitoring

through communication between vehicles and roadside units. The entities involved in VANET are:

- **OBU**: On Board Unit. Each vehicle is equipped with an OBU to provide wireless communication with other neighboring vehicles. It permits the exchange of traffic information and road conditions to ensure global safety.
- **RSU**: Road Side Unit. It is a device placed along roads that defines an area around itself. It is subordinated by the Trusted Authority (TA) and is responsible for exchanging information with the TA and OBUs inside the area managed by itself.
- **TA**: Trusted Authority. It registers the RSUs and OBUs. It needs sufficient storage and computation capabilities to issue the main keys of the network.

The smart vehicles are not only equipped with OBUs, but they contain other components useful for a good driving experience. For example, GPS (Global Positioning System) used for navigation, sensors used to detect objects around the vehicle or at a certain distance, EDR (Event Data Recorder) has a computing unit to ensure the process and storage of data, a wireless transceiver that provides V2X communication according to a standard, a TPD (Tamper-Proof-Device) to store the secret information (secret key) and is responsible for signing outgoing messages.

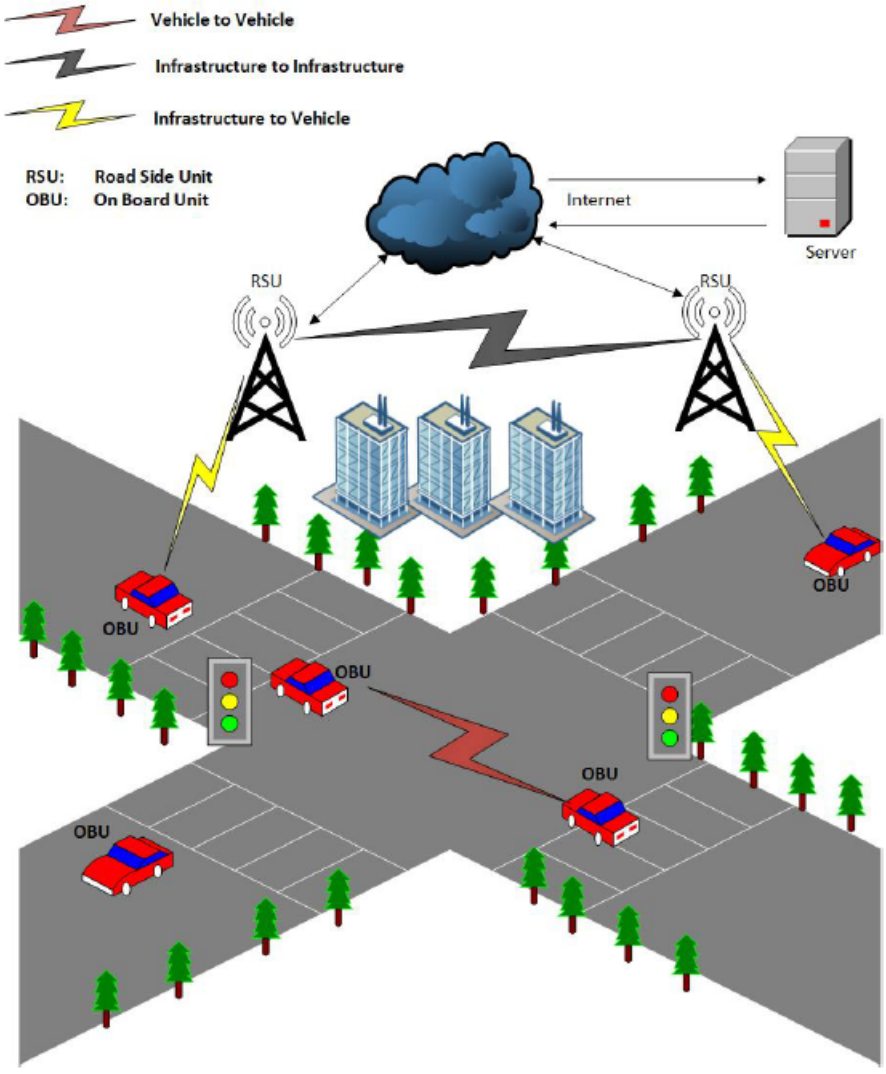


Figure 2.1: System architecture in VANET [1]

The **V2X communication** is a node-to-node communication, where each node exchanges information with the others in a short timeframe. There are three kinds of communication methods [1] (as described in Fig.2.1):

- **V2V**: vehicle to vehicle
- **V2I**: vehicle to infrastructure
- **I2I**: infrastructure to infrastructure

VANETs are considered the appropriate solution to increase road safety levels and are a fundamental part of the Intelligent Transportation Systems (ITS) structure.

An **ITS** is an innovative application designed to provide advanced services related to various modes of transportation and traffic management. It enables drivers to benefit from increased safety, better real-time information about road conditions, and improved coordination with one another through the use of communication networks.

The main standards to regulate ITS systems are [2]:

- **ITS-G5**, by the European Telecommunications Standards Institute (ETSI) in Europe;
- **WAVE**, by the Institute of Electrical and Electronics Engineers (IEEE) in the United States;
- **STD-T109**, by ARIB, that means the Association of Radio Industries and Businesses in Japan.

ITS is distinguished by an appropriate **stack**. The ITS stacks of various ITS standards are mostly built on a common reference design that adheres to the layered communication protocols of the OSI model. The scheme is described in Fig.2.2):

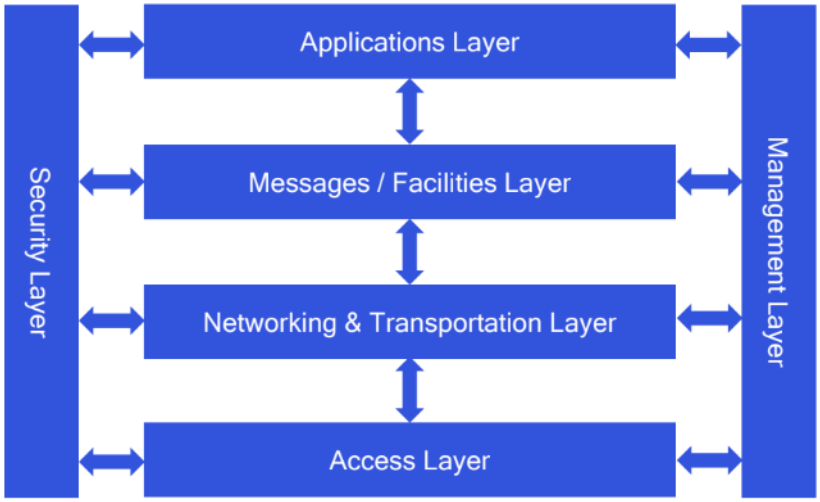


Figure 2.2: ITS stack: reference architecture [3]

Different communication media and protocols, that are related to the physical and data connection layers (OSI layers 1 and 2), are included in the ITS access technologies layer. These technologies make it easier for internal and external communication components of an ITS station to communicate with one another and with other ITS stations. For external communication, some ITS access technologies utilize complete, non-ITS-specific systems (such as GPRS, UMTS, and WiMAX), which serve as 'logical links' to transparently transport ITS data.

The ITS network and transportation layer corresponds to OSI layers 3 and 4. They use protocols for data delivery among ITS stations and from ITS stations to other network nodes. The ITS network protocols effectively distribute data in geographic regions and route data from source to destination through intermediary nodes. The ITS transport protocols are the end-to-end delivery of data, reliable data transfer, flow control, and congestion avoidance.

The ITS facilities layer is useful to support the ITS applications and corresponds to OSI layers 5, 6, and 7. Data structures provided by facilities are useful to store, aggregate, and maintain data of different types and sources (from sensors or from data received by means of communication). Regarding communication, ITS facilities support various types of application addressing, handle ITS-specific messaging, and manage the establishment and maintenance of communication sessions. A key feature is the management of services, which includes the discovery,

download, and also administration of software modules within the ITS station.

The stack is extended to include the ITS applications layer, which deals with applications and use cases focused on road safety, traffic efficiency, infotainment, and business.

The two vertical protocol entities are:

The ITS management entity, which oversees the configuration of an ITS station, facilitates cross-layer information exchange between various layers and performs other related tasks.

The ITS security entity, which delivers security and privacy services, including secure messaging across different layers of the communication stack, identity and security credential management, and ensuring platform security (e.g., firewalls, security gateways, tamper-proof hardware) [3].

2.2 V2X Standards and Protocols

V2X is an umbrella term that is used to refer to a vehicle's communication with all other entities, such as other moving or parked cars, pedestrians, traffic signals, road signs, construction sites, ext.

Vehicle communication can operate at two different levels [2]:

- Short-range communications, where the main standards are **DSRC** (Dedicated Short-Range Communications) in the USA and **C-ITS** (Cooperative Intelligent Transport Systems) in Europe.
- Long-range communications, where the primary standard is LTE V2X (Long-Term Evolution Vehicular to Everything), later renamed **C-V2X** (Cellular Vehicular to Everything), which utilizes the 3GPP standard.

2.2.1 DSRC

DSRC is a V2X technology based on the IEEE 802.11p standard. This latter standard is a vehicular communication protocol designed to provide wireless access in vehicular environments (WAVE).

The facilitating of wireless connectivity between nodes is possible thanks to the IEEE 802.11 standard specifies multiple physical layers including a Medium Access

Control (MAC) sublayer. The **802.11p** protocol, an expansion of IEEE 802.11a, is used by the MAC layer in vehicle networks to improve the speed at which data packets flow. The MAC protocol in IEEE 802.11p is built on Enhanced Distributed Channel Access (EDCA), a system that facilitates communication between vehicles and infrastructure by transmitting packets [2].

WAVE enables dedicated short-range communication (DSRC) channels using WLAN technology, allowing vehicles to communicate directly with other entities over short to medium distances, typically up to 300 meters.

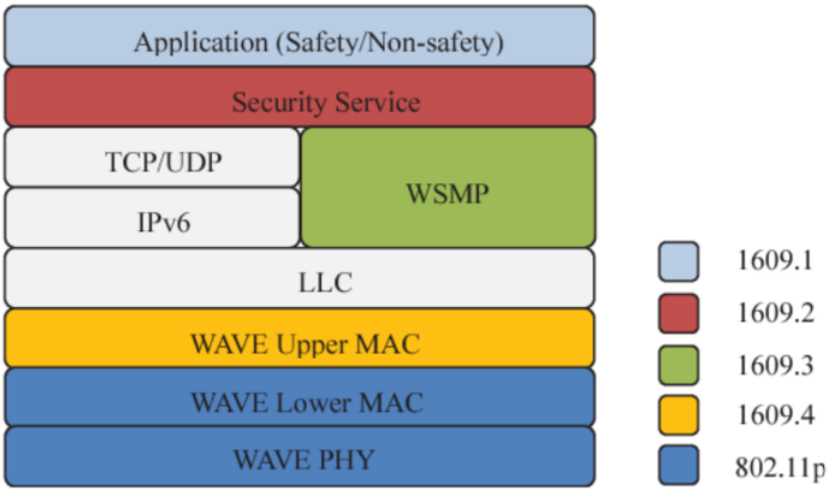


Figure 2.3: Stack architecture of DSRC/WAVE model [2]

The IEEE 1609 WAVE (Wireless Access in Vehicular Environments) working group developed the first version of the architecture for the lower layers of the protocol stack, specifically adapted for VANETs. This protocol was developed to operate above the 802.11p layer but remains independent of it and can also be used as a reference protocol with different wireless technologies. This flexibility permits the maintenance of the same WAVE standard, independent of the changes in the wireless technology based on DSRC. This advantage leads to the migration from V2X systems based on 802.11p to GSM or LTE technologies.

The IEEE WAVE standards are illustrated in Fig.2.3, that represents the protocol stack, and it is described below:

The 1609.1 WAVE Resource Manager (RM) outlines the methods for remote site applications to communicate with OBUs via RSUs, specifying the format

for command messages and data storage. The 1609.2 standard covers security services, including confidentiality, authentication, authorization, and integrity. The 1609.3 standard defines network and transport services, addressing, and routing mechanisms, as well as control of the Logical Link Layer for both IP traffic and WSMP (WAVE Short Message Protocol). Finally, the 1609.4 standard governs multichannel access operations[2].

DSRC is essentially a modification of Wi-Fi [4]. The advantages are:

- enabling data to be exchanged directly between two devices without the need for intermediaries. This is very useful in case of the absence of any telecommunication infrastructure;
- the elimination of intermediaries leads to a very low latency;

DSRC has to face some challenges [5]:

- Channel congestion in dense vehicular environments
- Lack of handshake/ACK in delivering broadcast frames
- Self-interference due to inadequate spectrum mask
- No QoS support
- Restricted network connectivity — no internet access
- Lack of ability to receive broadcast messages

2.2.2 C-ITS

Europe is at the forefront of deploying short-range direct V2X services, with 1.5 million vehicles already in operation and over 20,000 kilometers of motorways equipped with the necessary infrastructure. This is the largest interoperable V2X operation globally. Europe's success is built on a robust regulatory C-ITS ecosystem that helps cooperation and trust among all V2X stakeholders [6].

Cooperative Intelligent Transport Systems (C-ITS) are transport systems where collaboration between multiple ITS sub-systems (such as personal, vehicle, roadside, and central) enhances the quality and service level of an ITS service,

providing superior performance compared to when only a single sub-system is involved. [7].

In EU law, V2X services are defined as Cooperative Intelligent Transport Systems (C-ITS), prioritizing transport safety, sustainability, and also traffic management. The European radio regulation has allocated 50 MHz of spectrum in the 5.9 GHz frequency band for both basic and advanced C-ITS services, supporting safety and automation across all EU Member States [6].

As outlined by the C-ITS Deployment Platform organized by the European Commission, Cooperative Intelligent Transport Systems (C-ITS) utilize well-established ad-hoc short-range technologies (such as ETSI ITS G5) alongside complementary wide-area communication technologies (such as 3G, 4G, and future 5G). **ETSI ITS G5** is a cooperative V2X communication technology based on the US market IEEE 802.11p WLAN standard specially designed for automotive applications. The successor standard, IEEE 802.11bd, enhances performance and provides a smooth evolution of radio technology. It ensures optimal utilization of the allocated spectrum, guarantees uninterrupted operation of existing services, and protects prior investments.

Compared to other communication technologies, ETSI ITS G5 offers features particularly suited for safety-related applications, including locally self-organizing ad-hoc networks, operation at any time and place, free data transmission (without subscription), robustness, compliance with functional safety requirements, and independence from third-party commercial decisions and other commercial communication networks. Additionally, the limited communication range of ETSI ITS G5 supports design features that align with privacy requirements under the European GDPR.

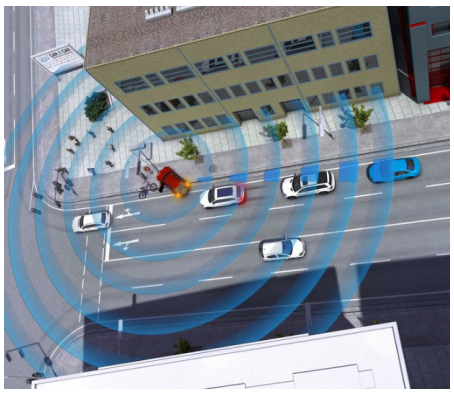


Figure 2.4: Awareness Driving[7]

Cooperative V2X services are characterized by various phases of innovation: [7]:

- Awareness Driving: The exchange of status data through cooperative V2X

communication (such as information on position, speed, driving direction, or specific incidents like vehicle malfunctions) facilitates a range of information and warning services. These services help road users drive more proactively and become aware of potential risks that may not yet be visible. Examples are Intersection Collision Warning, Emergency Vehicle Warning, Dangerous Situation Warning, Stationary Vehicle Warning, Traffic Jam Warning, and Pre-/Post-crash Warning (Fig.2.4).

- Sensing Driving: Road users equipped with cooperative V2X technology can share sensor observations and advanced environmental information. This described approach has more functionalities: not only alerts other traffic participants to dangers they might not yet perceive but also considers and protects non-communicating road users in various traffic scenarios. Examples are Extended Intersection Collision Warnings, Vulnerable Road User Warnings, Overtaking Warnings, Cooperative Adaptive Cruise Control, Long-term Road Works Warning, and Special Vehicle Prioritisation (Fig.2.5).

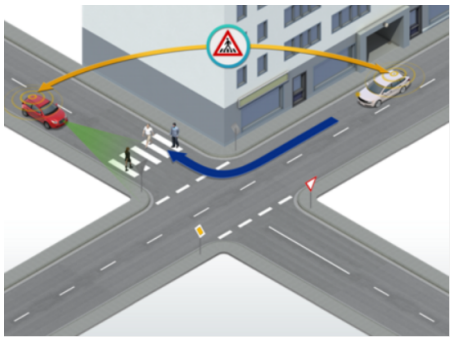


Figure 2.5: Sensing Driving[7]

- Cooperative Driving: Cooperative V2X road users can share intention data, enabling intelligent interaction and coordination of behavior even in complex traffic scenarios. Achieving the long-term objective of highly automated and autonomous driving requires the ability to predict the expected behavior of all users of the road.. Examples are (Static or dynamic) Platooning, Area reservation, Cooperative Merging, Cooperative Lane Change, and Cooperative Overtaking Fig.2.6).

Figure 2.6: Cooperative Driving[7]

2.2.3 C-ITS message types

The message set adopted in the vehicle-to-vehicle and vehicle-to-infrastracture communication comprises:

- ETSI ITS Cooperative Awareness Messages used by vehicles to inform the other entities around them about their status, position, distance to preceding and following vehicles, ext.
- ETSI ITS Decentralized Environmental Notification Messages used to notify road hazards, to indicate an area where automated driving is not permitted or with speed limits due to an accident, highlight a lane closure, ext.
- ETSI ITS Collective Perception Messages used to inform about any detected objects.
- ETSI ITS MAPEM used to transmit road segment topologies or/and intersections. Another usage is to prevent or limit the negative effects of a possible Transition of Control (ToC).
- ETSI ITS SPATEM Messages are used to communicate dynamic information about the status of a signalized intersection, including phase and timing details for its input-output connections.
- ETSI ITS In-Vehicle Information Messages are used to transmit both static and dynamic road sign and message sign information on highways.
- ETSI ITS Maneuver Coordination Message facilitates the coordination of maneuvers among cooperative automated vehicles and includes a proposed extension to allow infrastructure to send suggestions to these vehicles, aiming to enhance overall traffic safety and efficiency.

In this paper the CAM Structure is deeply explained because it will be valuable for the work. The **Cooperative Awareness Message (CAM)** is generated by the Cooperative Awareness (CA) service, which operates at the Facilities layer of the ETSI ITS communication architecture stack(Fig. 2.2). CAMs are exchanged between C-ITS stations equipped with V2X technology, like vehicles and infrastructure stations, to establish and maintain awareness of each other. These

messages convey information about the presence, position, dynamics, and basic attributes of the originating station. The received data can be utilized by various C-ITS applications; for instance, by comparing the position and dynamics of the originating station with its own, a receiving station can assess the risk of a collision.

The structure of CAM is described in Fig.2.7.

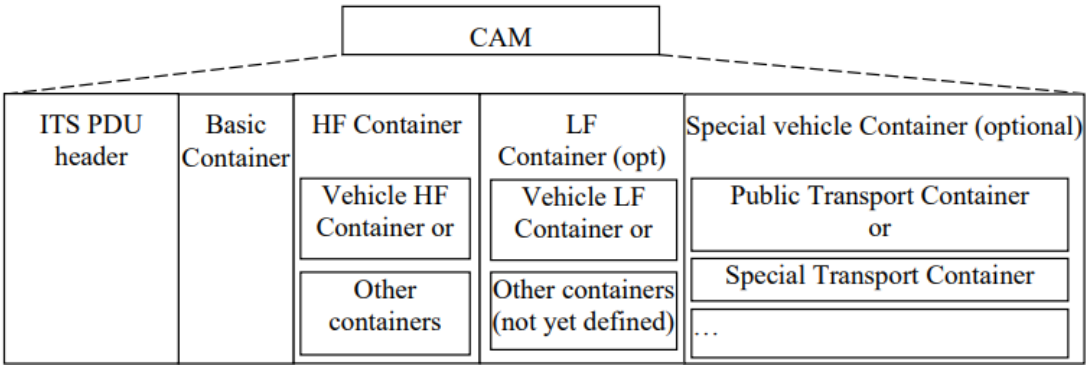


Figure 2.7: General structure of a CAM for vehicles [1]

A CAM originated by an OBU is composed of a PDU header and a set of containers. The number of container is: one basic container, one high frequency container, an optional low frequency container and a variable number of special containers:

- The basic container has information like position and type of originating station

- The high-frequency container is designed to transmit information that requires frequent updates. The standard provides options for selecting the appropriate transmission method. For vehicles, the current standard specifies the BasicVehicleContainerHighFrequency, which includes dynamic data such as vehicle speed and acceleration.

- The low-frequency container is used for information that does not require frequent updates. The standard offers choices for selecting the best transmission technique. Actually, the standard defines the BasicVehicleContainerLowFrequency for vehicles, which comprises static or slowly changing data like the

condition of the outside lights or the function of the vehicle (such as emergency vehicle or public transportation).

- The special vehicle container contains information specific to the vehicle's role.

Similarly, a CAM transmitted by an RSA has a basic container and one high-frequency container including the position of RSU and the protected communication zones.

The CAMs are sent by every ITS station every 100 ms on the control channel (G5-CCH). CAM generation frequency could be modified only for: the channel congestion and the vehicle dynamics. The Decentralized Congestion Control (DCC) system monitors the current radio channel usage and determines the maximum frequency for generating CAMs. However, if there is a significant change in the vehicle's dynamics, a CAM must be triggered, provided that DCC conditions are met. A CAM will be triggered under any of the following conditions:

- The absolute difference between the current vehicle heading and the heading in the last CAM exceeds 4°.

- The distance between the current vehicle position and the position in the last CAM exceeds 4 meters.

- The absolute difference between the current vehicle speed and the speed in the last CAM exceeds 0.5 m/s.

When a CAM is generated, the CA basic service must assemble the CAM PDU with the required containers, specifically the basic container and the vehicle high-frequency container. Optionally, a CAM may also include additional containers, such as the vehicle low-frequency container or the special vehicle container. However, these optional containers have specific generation requirements and should only be included if at least 500 ms have passed since their last transmission. For RSU stations, the CAM generation frequency must be configured to ensure that at least one CAM is transmitted while a vehicle is within the RSU's communication range, with a maximum generation frequency of 1 Hz[8].

2.2.4 CV2X

Cellular V2X (C-V2X) is an alternative vehicular communication protocol specifically designed for V2X applications. C-V2X, created by the 3rd Generation Partnership Project (3GPP), uses the same infrastructure as mobile phones but makes use of cellular radio technology rather than WLAN.

The key distinction between C-V2X and DSRC is its ability to support both direct and indirect communication. In direct C-V2X, vehicles communicate directly with other vehicles (V2V) and roadside units (V2I), similar to DSRC by means of PC5 interface. On the other side, in indirect C-V2X, vehicles connect to other entities via the cellular network (V2N), a capability that DSRC lacks, using Uu interface[4].

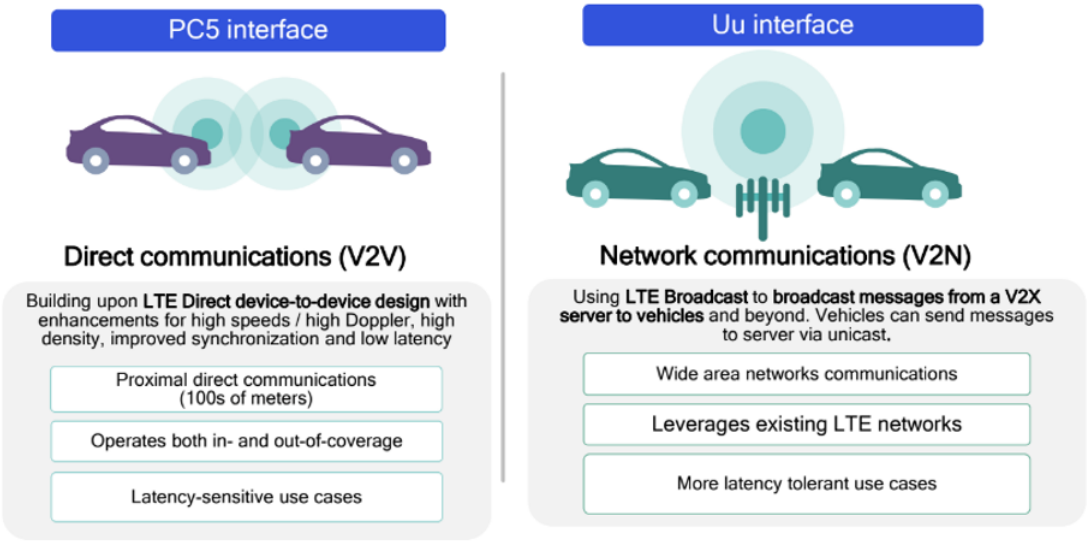


Figure 2.8: PC5 and Uu interfaces [5]

Direct communication between vehicles and other devices utilizes the **PC5 interface**. PC5 refers to a reference point where UE (User Equipment), like a mobile phone, directly communicates with another UE over a direct channel, bypassing the need for a Base Station (BS). This direct communication reduces latency by allowing data traffic to be managed locally without involving third parties. In order to ensure that law enforcement and emergency responders could stay in contact in the event that network infrastructure was down, as would happen

following a natural disaster, PC5 was first developed to meet the communication needs of public safety.

On the other hand, network communication follows the traditional cellular network path using the **Uu interface**. In particular, the Uu interface represents the logical link between UE and the BS and is mainly used for V2N (Vehicle-to-Network) communication. V2N is a unique feature of C-V2X that does not exist in 802.11p-based V2X, which only supports direct broadcast communication. To use the Uu interface, a vehicle must register with a cell, authenticate, and connect to the network. In V2N, the vehicle transmits its status information (such as speed, direction, and hazard messages) to a server accessible via the internet, which then relays this information to other vehicles or devices. These entities use the data for purposes like vehicle tracking or traffic management [2](Fig. 2.8).

The advantages of CV2X are[9]:

- Higher Spectral Efficiency: C-V2X supports a larger number of users on the road due to its efficient use of spectrum.
- Increased Security: It offers a higher degree of security compared to 802.11p technology.
- Superior Direct Communication: the protocol facilitates more effective direct communication between vehicles.
- Extended Range and Reliability: C-V2X delivers longer range and more reliable performance for V2X safety applications.
- Benefits of 5G NR C-V2X: For autonomous vehicles, 5G NR C-V2X offers high spectral efficiency, greater throughput, reduced latency, improved reliability, and supports high vehicle speeds.
- Backward Compatibility: 5G NR C-V2X Release 16 is compatible with Release 14, ensuring continuity in safety features.
- Comprehensive Awareness for Autonomous Driving: It works well in inclement weather and at night, offering 360-degree non-line-of-sight (NLOS) awareness.

The disadvantages are[9]:

- Privacy Risks: The system collects vehicle locations and other details, putting user privacy at risk.
- Autonomous Driving Risks: Failures in autonomous driving systems can lead to serious accidents. These systems must be rigorously tested before being deployed for real-world driving.
- Sensor Malfunctions: Defective sensors or auto parts may lead to communication problems.
- Vehicles in C-V2X are connected with internet and hence it is prone to hacking.

2.3 V2X helps ADAS

Automotive advanced driver-assistance systems (ADAS) and autonomous technologies commonly use sensors like cameras, radar, and lidar to monitor the vehicle's surroundings. These sensors send data to in-vehicle systems, which process and respond to it instantly. On one side, these traditional perception sensors can provide a 360-degree view around the vehicle, on the other side they have also certain limitations.

For instance, they cannot perform some actions, as see around corners, detect obstacles or traffic conditions beyond their immediate vicinity, or warn drivers about potential hazards or slowdowns before they are encountered on the road. Additionally, these systems depend on artificial intelligence (AI) to interpret visual data, which can sometimes result in the misidentification of objects in specific edge cases.

C-V2X communication solves the shortcomings of perception sensor-based systems and gives cars a **"sixth sense"**. The vice president of research and development at AutoTalks, a leading supplier of C-V2X connectivity solutions, Amos Freund explains: *"C-V2X is a sensor like lidar, like radar, like cameras, like ultrasonic. C-V2X has a counterpart on the 'other side' to communicate with: V2V (vehicle) or V2I (infrastructure.) And C-V2X is the only sensor with non-line-of-sight capability; it can see behind obstacles. In addition, it can operate and 'see' in any weather or lighting conditions such as fog, night and direct sunlight"*[10].

An example of the integration of V2X with ADAS is described by the **flow of automatic braking based on an ASIL V2X receiver**.[11]

A V2X system comprises GNSS, V2X radio, and a processor. For a system to be classified as ASIL, all of its components must meet ASIL standards. **ASIL** stands for Automotive Safety Integrity Level. The worldwide ISO 26262 standard, which focuses on the functional safety of electrical and electronic systems in motor vehicles, defines the ASIL classification. ASIL assesses a risk associated with a system and determines the required safety measures in light of that evaluation.

V2X technology enables the early detection of a target vehicle before it reaches an intersection. Even when other sensors have successfully identified the target, integrating V2X remains also advantageous due to its ability to improve both the accuracy and timing of future path predictions. Similar to other sensors, V2X uses map matching to confirm and put data in context before initiating automated braking, which usually necessitates risk assessment by two or more sensors.

When a V2X target poses a risk, a non-ASIL V2X system, it is limited to issuing warnings, initiating pre-braking, and adjusting ADAS detection thresholds. On the other side, a V2X ASIL-compliant vehicle can take one of four actions, as detailed below:

- Ignore: The target is deemed implausible due to unvalidated positioning or other parameters.

- Issue a warning (light or sound): Alert the driver to an impending occurrence that could happen in a matter of seconds. A warning light can be utilized more freely than a warning sound, which is only engaged once the incident has been verified.

- Initiate moderate braking: V2X is the only system that can identify this hazard, which calls for an instant reaction. Braking deceleration is calibrated according to ASIL B standards.

- Initiate hard braking: The hazard demands an immediate response and is confirmed by multiple sensors. Braking deceleration is set in accordance with ASIL D standards.

2.4 Car Makers

V2X is a technology studied and approached by car makers in different ways. The car makers who first entered the V2X sector based their studies on DSRC. Following its initial approval in 2010, DSRC was subject of testing before being implemented in select Toyota vehicles produced in Japan in 2015, and later in certain Cadillac models in the U.S. in 2017. In 2019, the Volkswagen Golf 8, one of Europe's most popular cars, became the top-selling V2X-enabled vehicle on the market.[4]

Volkswagen's activism in V2X technology reopened a debate that seemed settled between two camps: those who have more confidence in the emerging 5G cellular network and those who have relied on the more established DSRC, which in its European version is essentially WLANp/ITS G5. General Motors, NXP, Toyota, Volkswagen, and Volvo have so far expressed support for WiFi. On the other side supporters of 5G include BMW, Daimler (Mercedes-Benz Group AG), Ford, Huawei, Intel, Qualcomm, and Samsung[12].

Chapter 3

State of art of CyberSecurity in V2X: present and future

3.1 Security requirements of ITS

ETSI TS 102 940 [13] shows that the ITS applications, discussed in the previous chapter, can be categorized, and each category needs security:

- Cooperative Awareness. The application of this type is valuable to enhance traffic safety, giving information about the vehicle status and environment. Aside from ensuring authenticity and integrity, authorization is a necessary key point to restrict access to legitimate users. Different levels of authentication may be required, depending on the application and participation requirements. Authorization may be based on factors such as status (e.g., vehicle priority), properties (e.g., sensor equipment, implementation, vehicle type), or subscription to a paid service (e.g., personalized route guidance). In general, authentication is necessary for applications that aim to transmit messages over the network.

  There are several categories of CAM authorization:

  - Basic CAM Authorization:
    * Associated with fundamental data such as vehicle length, width, speed, heading, acceleration, and brake status.

    - Granted to all enrolled ITS stations, allowing them to participate in basic ITS functions.
  - Advanced CAM Authorization:
    - Includes additional information needed for tasks like turning across traffic, merging assistance, and collision warnings.
    - Depends on the capabilities of the sending station, including implemented cryptographic algorithms, sensor equipment, and its perceived trustworthiness.
  - Authorization for Emergency Vehicle Priority:
    - Granted exclusively to specially authorized emergency or public transport vehicles, as per national legislation. Multiple priority levels may be defined, such as priority for emergency vehicles and, at a lower level, authorization to use dedicated lanes for public transportation.
    - Issued by a governmental organization or its designated proxy.
    - Priority rights are asserted during operation rather than at the time of authorization.
  - Authorization to Issue Regulatory Orders (e.g., speed limits, road closures):
    - Granted only to specially authorized ITS stations, such as RSUs and police vehicles.
    - Issued by a governmental organization or its designated proxy.

  There are no confidentiality requirements since CAMs are broadcast to any potential receiver.

  CAMs are transmitted periodically, up to several times per second, by each ITS-S and contain significant status information, such as the location of the sending ITS-S. Therefore, ensuring that this data cannot be associated with any individual is crucial, to preventing the leakage of personally identifiable information through the CAM service. Thus, privacy is fundamental.

- Static Local Hazard Warnings. They share many similarities with the CAM service, except that RSUs transmit them. The authentication and authorization requirements are similar to those for CAM, but there is an additional

requirement about the authorization that should be restricted to the specific functionality, purpose, and location of the RSU in question. Since the service involves broadcasting from a static RSU, there are no confidentiality or privacy requirements.

- Interactive Local Hazard Warnings. In general, the authorization and authentication requirements are similar to those for CAM. However, in the subsequent unicast session, additional authorization and/or authentication may be required based on the local policies of the participating partners, which are beyond the scope of this document. Confidentiality and privacy requirements depend largely on the specific application and the nature of the information exchanged. Confidentiality could become a concern, since unicast sessions may involve more privacy-sensitive and personally identifiable information. It is typically addressed within the application or through agreements between the involved parties

- Area Hazard Warnings. Authorization for area hazard warnings, such as Decentralized Environment Notification Messages (DENM), may be granted at different levels based on factors like sensor equipment, sensor quality, and the algorithmic and processing capabilities of the ITS-S. Aside from these differences, the requirements are generally similar to those for CAM. Since the service is event-driven and thus sporadic, and because the sender's properties or identity are not crucial for the area warning, privacy concerns are minimized compared to the CAM service. As a result, confidentiality services are not required.

- Advertised Services. A service provider unit uses the Service Advertisement Message (SAM/WSA) to notify ITS stations about local services that are available or services that can be accessed through a remote server. To protect ITS stations from threats such as attacks by fake or malicious service providers or impersonation of Internet servers, the Service Advertisement Message must be secured for integrity and authenticity, requiring it to be signed by the ITS-S (either roadside or vehicle) providing the advertisement to neighboring ITS stations. Authentication is generally required for applications that send messages over the network. Additionally, special authorization must

be promised to the ITS-S or the specific purpose of the service advertisement application. Since the service is broadcast to any potential receivers and the sender is either a static RSU or a mobile vehicle playing a distinguishable role (e.g., leader vehicle in a platoon), there are no confidentiality or privacy requirements.

- Local High-Speed Unicast Services
- Local Multicast Services
- Low-Speed Unicast Services
- Distributed (networked) Services

3.2 ITS station communications security architecture

ETSI EN 302 665 [14] reports the security layer as a vertical layer adjacent to each ITS layer. However, in practice, security services are implemented to be divided into the four fundamental ITS processing layers, as illustrated in Figure 3.1.

Security services operate within one or more ITS architectural layers, or within the Security Management layer, as shown in the figure. Additionally, the Security Entity in the ITS communications architecture, as outlined in ETSI EN 302 665 [14], includes a third sublayer: the security defense layer. This layer is responsible for safeguarding critical system assets and data from direct attacks and enhancing the likelihood of detecting intrusions (e.g., through firewalls and intrusion detection or prevention systems).

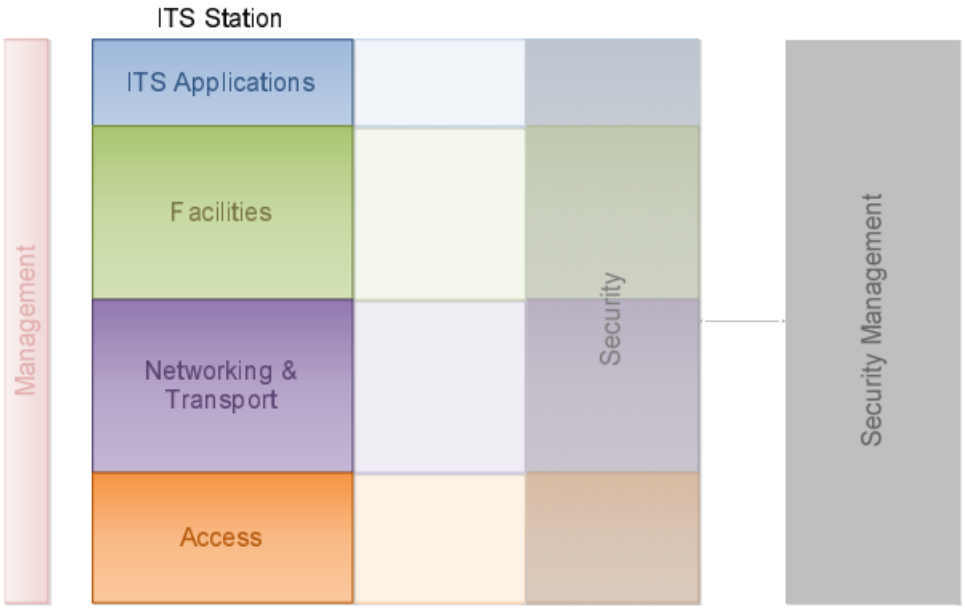


Figure 3.1: Architectural ITS security layers [13]

3.3 Range of security services

The various security services that an ITS station may support to ensure secure communication with other stations are illustrated in ETSI TS 102 731 [15]. Here these services are listed with a brief description for each one:

- Security Associations management
  - services: Establish Security Association, Update security association, Send Secured Message, Receive, Secured Message, Remove security association
  - description: Establishment of a secure communication channel between two ITS stations to enable secure message exchange. Both ITS stations must use this service, to set up a bi-directional secure communication.
- Single message services
  - services: Authorize Single Message, Validate Authorization on Single Message, Encrypt Single Message, Decrypt Single Message

  - description: This service, like for CAM or DENM, guarantees the security of the injection or reception of a single message..

- Integrity services
  - services: Calculate Check Value, Validate Check Value, Insert Check Value
  - description: Calculation of a check value to integrate in an outgoing message and verify that an incoming message has not been altered by using its check value.

- Plausibility validation
  - service: Validate Data Plausibility
  - description: Based on the message plausibility, verifying that the information extracted from the incoming message can be trusted.

- Replay Protection services
  - services: Replay Protection Based on Timestamp, Replay Protection Based on Sequence Number
  - description: Ensuring that messages are sent and received consistently by including a timestamp or sequence number in outgoing messages and verifying the timestamp or sequence number of incoming messages.

- Enrolment
  - services: Obtain, Update and Remove Enrolment Credentials
  - description: The Enrollment Authority (EA) is responsible for managing the enrollment credentials that are used for this service. With the use of these credentials, the ITS-Station gains credibility and proves its authenticity to other ITS-S.

- Authorization
  - services: Obtain and Update Authorization Tickets, Publish Authorization Status, and Update Local Authorization Status Repository

  - description: The Authorization Authority (AA) is responsible for issuing the AuthZ Ticket (AT) that is used for this service. With this ticket, the requesting ITS station obtains specific authorizations and permissions.

- Accountability services
  - services: Record Incoming and Outgoing Messages in the Audit Log
  - description: The incoming and outgoing messages are registered because each ITS station must be accountable for the actions performed during communication.

- Remote management
  - services: Remote Activate Transmission, Deactivate ITS transmission
  - description: Allows the ITS infrastructure to remotely manage a malfunctioning ITS station. Specifically, this service enables the ITS infrastructure to activate or deactivate message transmission on a particular ITS station from a distance.

- Report Misbehaving ITS-S
  - service: Report misbehavior
  - description: Allows ITS stations to report suspicious activities, such as the behavior of a malfunctioning ITS station, to the ITS infrastructure.

- Identity Management
  - service: Subscribe ID Change Notification, Unsubscribe ID Change Notification, ID Change Notification, Trigger ID Change, Lock ID Change, Unlock ID Change
  - description: Offers services that support the simultaneous update of communication identifiers (such as station ID, network ID, MAC address) and credentials used for secure communications within the ITS station. Additionally, it provides options to disable changes to these identifiers.

3.4 PKI infrastructure

ETSI TS 102 940 [13] describes that communications security services inherently involve multiple components within their functional model. The key functional elements and their reference points can be identified by examining a straightforward ITS communications scenario. This scenario is represented by the global architecture of the Cooperative-ITS Security Certificate Management System (commonly C-ITS Trust Model), the involved functional entities, the list of interfaces and services useful to support the life-cylcle management of Trusted C-ITS Stations. This architecture is displayed in Figure 3.2

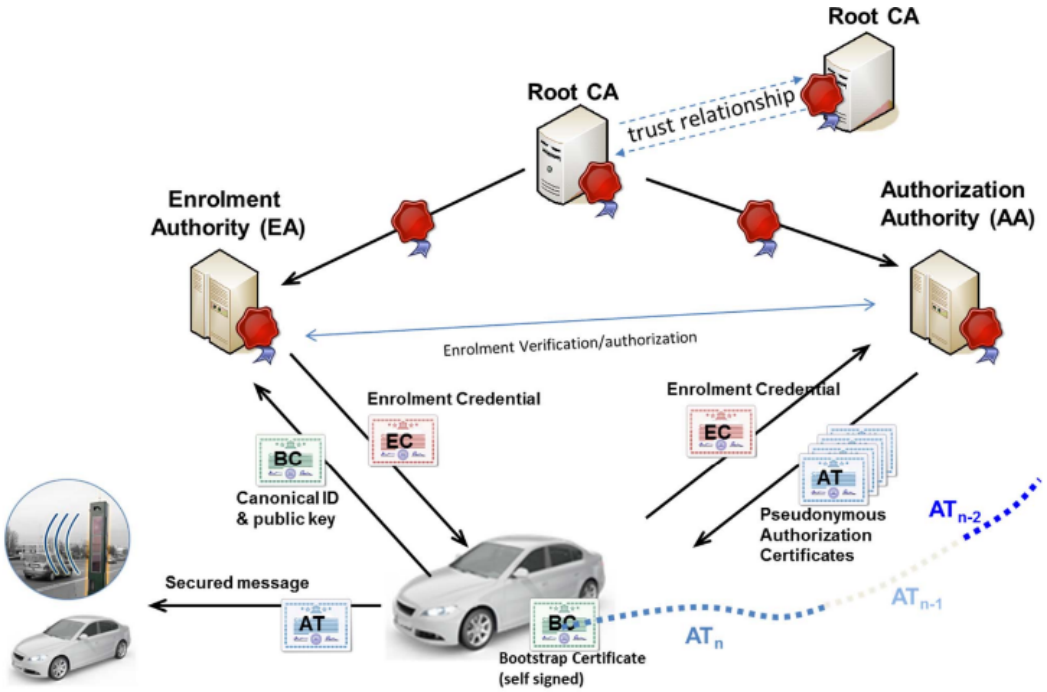


Figure 3.2: PKI architecture [13]

The following is a list of the roles that have been defined for each functional element present in the PKI infrastructure:

- **Root Certification Authority (Root CA)**: RCA is the CA at the top of the certification hierarchy, that is in charge of granting certificates to the Authorization Authority (AA) and Enrollment Authority (EA). The RCA can

create, renew, distribute, or revoke new trusted sub CA certificates. Another function is the generation and issuance of a Certificate Revocation List or Certificate Trust List, both signed by itself. For auditing purposes, RCA must generate log files of all its operations.

When a Root Certification Authority (RCA) is used as a Trust Anchor for an ITS station (ITS-S), the RCA certificate must be securely transmitted to the ITS-S for initialization, typically during the manufacturing process. To validate an incoming message, the ITS-S must have secure access to the root certificate at the top of the hierarchy for the authorization ticket attached to the message. Root certificates may be obtained during the manufacture or maintenance stages.

Root certificate information can be distributed over the air via cross-certification, a bridge CA, or a trusted certificates list (CTL). The Root CA should store its certificates and trust list information (such as Certificate Revocation Lists (CRLs) and CTLs) in a local repository. This repository must limit WRITE access to approved, authenticated users and grant unrestricted READ access to all local users and apps. If the Distribution Center (DC) is available, also additional information about its URL could be included in the repository. A dedicated communication channel should be used to grant access to the local repository. The aim is to guarantee the integrity and validity of the data.

- **The Enrolment Authority (EA)** authenticates the requesting ITS Station (ITS-S) and issues an Enrollment Credential. It is like a proof of identity and, in particular, serves as evidence of the ITS-S's identity without revealing its canonical identifier to third parties. The ITS-S uses this credential to request authorization for services from an Authorization Authority (AA). The EA is responsible for several key functions, such as including the registration of ITS-S, management of their status and permissions, and also the administration of their cryptographic certificates, which specify the applications, services, and capabilities the ITS-S is permitted to use. The EA can not only be responsible for the issuing of Enrolment Certificates to ITS-S after successful authentication, but it can also revoke these credentials at the end of the ITS-S's life or in cases of exceptions, such as the compromise of private keys.

Additionally, the EA handles the creation and renewal of EA certificates, which may involve generating key pairs, certificate requests, and submitting these to the Root Certification Authority (RCA). It also verifies that the ITS-S has the necessary permissions and is trusted when it requests authorization tickets. The EA is tasked with updating trust information lists issued by RCAs through the Distribution Center (DC) or local repositories and updating the Extended Certificate Trust List (ECTL) if a Trust List Manager (TLM) is set up. Furthermore, the EA generates log files of its operational activities for auditing purposes.

- **Authorization Authority (AA)**: An ITS Station (ITS-S) that has been enrolled with and authenticated by an Enrolment Authority (EA) must apply to an Authorization Authority (AA) for specific permissions within the EA's domain and the AA's authorization context. These permissions are granted through **Authorization Tickets (AT)**, each of which specifies a particular authorization context that includes a set of permissions. For example, an authorization ticket might grant an ITS-S the ability to broadcast messages from a particular message set or to claim certain privileges. To protect the privacy of the ITS-S, the AA is designed to operate without knowledge of the ITS-S's long-term identity (Enrolment Credential, or EC), and the EA cannot link the short-term identity (Authorization Ticket, or AT) to the long-term identity of the ITS-S.

  The AA is in charge for several functions: not only the issuing of Authorization Tickets to the ITS-S after authenticating and authorizing the AT request forwarded by the EA, but AA also handles the creation and renewal of AA certificates, which may include generating key pairs, creating certificate requests, and submitting them to the Root Certification Authority (RCA). Another function is the verification that the ITS-S has the necessary permissions and is trustworthy when requesting authorization tickets. Furthermore, the AA manages the updating of trust information lists issued by RCAs, either through the Distribution Center (DC) or local repositories, and ensures the update of the Extended Certificate Trust List (ECTL) if a Trust List Manager (TLM) is in place. The last function is about auditing: the AA is responsible

for generating log files of its operational activities.

- **Distribution Center (DC)** must ensure that all entities within the PKI hierarchy managed by the Root Certification Authority (Root CA) have access to trust list information, such as Certificate Trust Lists (CTLs) and Certificate Revocation Lists (CRLs). The DC must allow unrestricted read access to the trust lists, while write access should be granted only to authorized and authenticated users. Additionally, if multiple Root CAs operate within the Cooperative Intelligent Transport Systems (C-ITS) Trust model, the DC may provide access to Extended Certificate Trust Lists (ECTLs) and to the CRLs of other Root CAs.

- **Trust List Manager (TLM)** is responsible for several critical functions. It handles the creation and renewal of its certificates and, also, ensures their distribution through either a local repository or the Central Point of Contact (CPOC). The TLM also manages the insertion or removal of Root CA certificates in the Extended Certificate Trust List (ECTL), subject to approval by the Policy Authority, which is particularly important when a Root CA certificate is created, renewed, or revoked. Additionally, the TLM is tasked with creating the signed list of Root CA certificates and TLM certificates, known as the ECTL, and distributing this list to all participants within the Cooperative ITS trust domain, either via a local repository or the CPOC as well. Furthermore, the TLM generates log files documenting its operational activities to facilitate auditing processes.

- **Sending ITS-S**: Obtains access rights to ITS communications from the Enrollment Authority, negotiates service rights with the Authorization Authority, and sends both single-hop and relayed broadcast messages.

- **Sending ITS-S**: Obtains access rights to ITS communications from the Enrollment Authority, negotiates service rights with the Authorization Authority, and sends both single-hop and relayed broadcast messages.

- **Relaying ITS-S**: Forwards broadcast messages from the sending ITS-S to the receiving ITS-S, if necessary.

- **Receiving ITS-S**: Receives broadcast messages from either the sending ITS-S or the relaying ITS-S.
- **Manufacturer**: Installs the necessary security management information in ITS-S during production.
- **Operator**: Installs and updates the necessary security management information in ITS-S during operation.

3.5 Secure Data Structures

ETSI TS 103 097 [16] specifies the secure data structure for messages exchanged in V2X environment. This structure is of type EtsiTs103097Data, that corresponds to a Iee1609Dot2Data.

The Iee1609Dot2Data content could be one of these options:

- unsecuredData, used to encapsulate an unsecured data structure
- signedData, used to transmit a data structure with signature
- encryptedData, used to send an encrypted data structure

The EtsiTs103097Data profiles are:

- EtsiTs103097Data-Unsecured: Uses the Ieee1609Dot2Data option unsecuredData.
- EtsiTs103097Data-Signed: Uses the Ieee1609Dot2Data option signedData, containing the data structure in the component tbsData.payload.data.
- EtsiTs103097Data-SignedExternalPayload: Uses the Ieee1609Dot2Data option signedData, containing the digest of the data structure in the component tbsData.payload.extDataHash.
- EtsiTs103097Data-Encrypted: Component ciphertext.aes128ccm.ccmCiphertext contains the encrypted data structure. It uses the Ieee1609Dot2Data option encryptedData.

- EtsiTs103097Data-SignedAndEncrypted: Uses the parameterized type EtsiTs103097Data-Encrypted, containing an encrypted EtsiTs103097Data-Signed.
- EtsiTs103097Data-Encrypted-Unicast: Uses the parameterized type EtsiTs103097Data-Encrypted, further constrained to have one entry in the component recipients.
- EtsiTs103097Data-SignedAndEncrypted-Unicast: Uses the parameterized type EtsiTs103097Data-Encrypted, containing an encrypted EtsiTs103097Data-Signed, and further constrained to have one entry in the component recipients.

3.5.1 Signed Data

The type SignedData has the following constraints:

- hashId: Indicates the hash algorithm used to generate the message hash, in accordance with IEEE Std 1609.2[17], clauses 6.3.5 and 5.3.3.
- tbsData: Must be of type ToBeSignedData as defined in IEEE Std 1609.2[17], clause 6.3.6. This structure includes:
  - payload: the type is SignedDataPayload (as defined in clause 6.3.7) and contains either:
    - data: is the payload to be signed, of type Ieee1609Dot2Data.
    - extDataHash: The hash of data not explicitly transported within the structure.
  - headerInfo: Of type HeaderInfo (as defined in clause 6.3.9), with the following security headers:
    - psid: represents the ITS-AID corresponding to the message. generationTime: Always present.
    - expiryTime: May be present or absent, depending on message profiles specified in clause 7.
    - generationLocation: present or absent, as specified in clause 7.
    - p2pcdLearningRequest: Always absent.
    - missingCrlIdentifier: Always absent.
    - encryptionKey: May be present or absent, as specified in clause 7.

    * inlineP2pcdRequest: May be present or absent, as specified in clause 7.
    * requestedCertificate: May be present or absent, as specified in clause 7.

- signer: Must be of type SignerIdentifier (as defined in clause 6.3.24), constrained to one of the following:
  - certificate: Contains only one entry in the SequenceOfCertificate list of type TS103097Certificate, representing the signing certificate as defined in clause 6. The signer is represented by its certificate one second after the last inclusion of the certificate inside the message.
  - digest: Contains the digest of the signing certificate (as defined in clause 6.3.26). A certificate digest is the sequence of the least significant 8 bytes of the output produced by a hash function over the certificate, (i.e. digest = LSB8(Hash(certificate))). Once each second, the message must include the whole certificate. In the other messages, the certificate digest is present.

- signature: Must be of type Signature (as defined in clause 6.3.28) and must contain an ECDSA signature, following the specifications in clauses 6.3.29, 6.3.29a, and 5.3.1 of IEEE Std 1609.2[17].

For reaching message integrity and protection, the ECDSA is required to generate digital signatures. **ECDSA (Elliptic Curve Digital Signature Algorithm)**, as explained in the book Practical Cryptography for Developers [18], is a cryptographic digital signature scheme based on elliptic-curve cryptography (ECC). It is founded on the mathematics of cyclic groups of elliptic curves over finite fields and the challenge of solving the elliptic-curve discrete logarithm problem (ECDLP). The ECDSA signing and verification processes are based on elliptic curve point multiplication.

Compared to RSA, ECDSA offers shorter key sizes and signatures while providing equivalent security levels. Usually, the public key (PK) and the signature length are 32 and 64 bytes longer, respectively.

In ECDSA (Elliptic Curve Digital Signature Algorithm), the key pair generation produce:

- Private Key: An integer named privKey.
- Public Key: An elliptic curve point denoted as pubKey, which is calculated as pubKey = privKey * G, where G is the generator point of the elliptic curve.

The private key is a randomly generated integer within the range [0, n-1], where n is the order of the elliptic curve. Instead, the public key is derived by multiplying the private key with the generator point G on the elliptic curve.

The public key point x, y can be compressed into a more compact form, requiring only one of the coordinates and an additional bit for parity. For the secp256k1 curve, the private key is an integer 256-bit (32 bytes) longer, and the compressed public key is a 257-bit integer (approximately 33 bytes).

The ECDSA signing process involves the following steps:

- Hash the Message: The hash of the message msg is computed by means of a cryptographic hash function, such as SHA-256: h=hash(msg)
- Generate Random Number: Securely generate a random integer k within the range [1, n-1]. For deterministic ECDSA, derive k using HMAC with the hash h and the private key privKey.
- Calculate Random Point: The process of calculating a random point involves two steps: elliptic curve point $R = k \cdot G$ is computed, and its x-coordinate is extracted as $r = R_x$.
- Compute Signature Proof: Apply the following formula to determine the signature proof s:

$$s = k^{-1} \cdot (h + r \cdot \text{privKey}) \mod n$$

  where k^{-1} is the modular inverse of k mod n, satisfying $k \cdot k^{-1} = 1 \mod n$.

- Return the signature r, s. This pair is composed of integers, each one is in the range [1...n-1]. It encodes the random point $R = k \cdot G$ and includes a proof s that confirms whether the signer knows of the private key privKey and the

message h. This proof s can be validated using the corresponding public key pubKey.

3.5.2 EncryptedData

The EncryptedData type is subject to the following constraints:

- Recipients Component:
  - The recipients component of EncryptedData must be of the type SequenceOfRecipientInfo as specified in IEEE Std 1609.2 [17], clause 6.3.31.
  - Each entry in the SequenceOfRecipientInfo must be one of the following options:
    - pskRecipInfo as defined in IEEE Std 1609.2 [17], clause 6.3.32.
    - certRecipInfo as defined in IEEE Std 1609.2 [17].
    - signedDataRecipInfo as defined in IEEE Std 1609.2 [17], clause 6.3.34.
- Encryption Scheme:
  - The encryption scheme employed must be ECIES, as outlined in IEEE Std 1609.2 [17], clause 5.3.5.
- Ciphertext Component:
  - The ciphertext component of EncryptedData must be of type SymmetricCiphertext, as specified in IEEE Std 1609.2 [17], clause 6.3.37.
  - This SymmetricCiphertext must contain an EtsiTs103097Data object encrypted in accordance with IEEE Std 1609.2 [17], clauses 6.3.38 and 5.3.8.

3.6 Certificate format

A certificate contained within a secure data structure shall conform to the following specifications:

- Certificate Type:

  - The certificate must be of type EtsiTs103097Certificate.
  - This corresponds to a single ‘ExplicitCertificate‘ as specified in IEEE Std 1609.2 [17], clause 6.4.6, with additional constraints outlined in this clause.

- toBeSigned Component:
  - The ‘toBeSigned‘ component of ‘EtsiTs103097Certificate‘ shall be of type ‘ToBeSignedCertificate‘ as defined in IEEE Std 1609.2 [17], clause 6.4.8, with the following constraints:
    * id: Must be of type ‘CertificateId‘ and constrained to the choice type ‘name‘ or ‘none‘.
    * cracaId: Must be set to ‘000000’H‘.
    * crlSeries: Must be set to ‘0’D‘.
    * validityPeriod: No additional constraints.
    * region: Must be of type ‘GeographicRegion‘ as defined in IEEE Std 1609.2 [17], and may be present or absent according to the certificate profile specifications in clause 7.
    * assuranceLevel: Must be of type ‘SubjectAssurance‘ as defined in IEEE Std 1609.2 [17], and may be present or absent according to the certificate profile specifications in clause 7.
    * appPermissions: Must be of type ‘SequenceOfPsidSsp‘ as defined in IEEE Std 1609.2 [17], and may be present or absent according to the certificate profile specifications in clause 7.
    * certIssuePermissions: Must be of type ‘SequenceOfPsidGroupPermissions‘ as defined in IEEE Std 1609.2 [17], and may be present or absent according to the certificate profile specifications in clause 7.
    * At least one of ‘appPermissions‘ or ‘certIssuePermissions‘ must be present.
    * certRequestPermissions: Must be absent.
    * canRequestRollover: Must be absent.

  - encryptionKey: Must be of type 'PublicEncryptionKey' as defined in IEEE Std 1609.2 [17], and may be present or absent according to the certificate profile specifications in clause 7.
  - verifyKeyIndicator: Must be of type 'VerificationKeyIndicator' as defined in IEEE Std 1609.2 [17], present, and constrained to the choice 'verificationKey'.

- Signature Component:
  - The 'signature' component of 'EtsiTs103097Certificate' must be of type 'Signature' as defined in IEEE Std 1609.2 [17], clause 6.3.28.
  - It shall contain the signature, calculated by the signer identified in the 'issuer' component, as specified in IEEE Std 1609.2 [17], clauses 6.3.29, 6.3.29a, and 5.3.1.

3.7 The advent of quantum computing

Vehicles ensure message integrity through the use of digital certificates and signatures. When a vehicle transmits a message, it generates the signature, it attaches the signature at the end of the message and inserts a certificate during transmission, following the protocols and standards discussed before. The receiving vehicles then verify the sender's authenticity and the message's integrity by checking the certificate chain and the attached signature using the public key in the certificate. Therefore, creating and distributing certificates within the PKI system are crucial components of the security framework in V2X communication.

However, the growing capabilities of **quantum computers** in recent years present serious threats to the security of current computer systems. Public-key-based algorithms, in particular (as V2X), are at risk from quantum attacks. This has prompted governments, industries, and academia to explore and develop alternative algorithms to resist quantum-based threats and attacks [19].

Using the principles of quantum physics, fully developed quantum computers could solve highly complex problems at speeds vastly greater than today's machines. A quantum computer could complete tasks that would take classical computers thousands of years in mere minutes.

The quantum mechanics, that is the study of subatomic particles, uncovers the unique and fundamental laws of nature. Quantum computers exploit these principles, allowing them to perform calculations using probabilistic and quantum mechanical methods [20].

Since 2012, the **National Institute of Standards and Technology (NIST)** has been leading a standardization effort to select a set of post-quantum cryptography (PQC) algorithms to contrast the attacks from quantum computers. This initiative includes both Key-Encapsulation Mechanisms (KEM) and digital signature algorithms. In June 2022, the results of the third round of evaluations were finalized and published. Among the signature algorithms advancing to the fourth round are **CRYSTALS-Dilithium**, **Falcon**, and **SPHINCS+**. The first two are lattice-based, while the third is hash-based.

A key characteristic of these PQC algorithms is that their public keys and signatures are significantly larger than those used in conventional algorithms, like ECDSA, with sizes varying across different algorithms. In extreme cases, such as with the Rainbow-V signature algorithm, the public key size can approach 2 megabytes.

From the point of view of V2X communication, this means that PQC certificates and their associated digital signatures will result in messages that are considerably larger than those using traditional ECDSA. This increase in message size imposes additional processing demands and requires more storage in the vehicle's On-Board Unit (OBU). Moreover, a greater load on the communication channels is possible, which is a critical issue for real-time systems like V2X communication, where messages between moving vehicles must be processed with the order of milliseconds to prevent accidents and enhance road safety [19].

3.8 Analysis of new PQC algorithms

When evaluating a digital signature algorithm, multiple metrics should be assessed, like communication costs, performance, efficiency, and implementation features. NIST has not set a priority, recognizing that different real-world scenarios may necessitate various trade-offs. Ideally, a strong candidate should stand out in at least one of the following areas: public key and signature sizes, or computational

costs.

NIST has established five security levels. Each level is defined by comparing the algorithm's security to that of a standard symmetric primitive: levels I, III, and V correspond to AES-128, AES-192, and AES-256, respectively, while levels II and IV correspond to SHA-256 and SHA-384, respectively. For instance, for an algorithm to qualify for level I, any known attack capable of compromising its security must have a computational cost (either classical or quantum) that is equal to or greater than the cost of a brute-force attack on AES with 128-bit keys. Therefore, the submitted schemes to NIST competition must include various parameter settings to accommodate the different security levels.

The figure Fig.3.3) shows a graph comparing the sizes of public keys and signatures of the new submitted proposals, the winning signature schemes from the fourth round post-quantum competition and pre-quantum standards. To make the comparison easier, only level I and II parameterizations are displayed. A red border around the new proposals indicates the presence of identified security vulnerabilities[21].

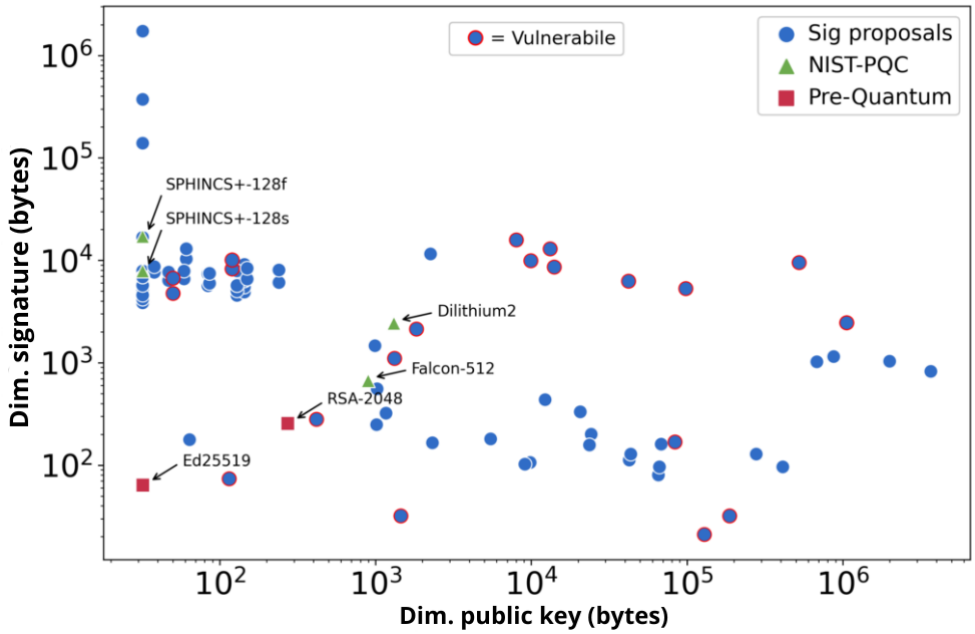


Figure 3.3: Comparison NIST-PQC algorithms[21]

An in-depth analysis of the third-round PQC standardization finalist digital

signature schemes: Dilithium, Falcon, and Rainbow. is conducted in the conference paper *"Security Comparisons and Performance Analyses of Post-quantum Signature Algorithms"* [22]. Even though this paper examines an earlier step in the standardization process compared to the current stage at the time of writing this document, it is very valuable for understanding the characteristics and differences of the digital signature algorithms, particularly those based on lattice methods.

Algorithm	Scheme	NIST Status	Security Level
Dilithium	Lattice-based	Finalist	1 (AES128) 2 (SHA256) 3 (AES192)
Falcon	Lattice-based	Finalist	1 (AES128) 5 (AES256)
Rainbow	Multivariate	Finalist	1 (AES128) 3 (AES192) 5 (AES256)

Figure 3.4: Third-Round PQC standardization algorithms[22]

The lattice-based signature schemes are Dilithium and Falcon, and the multivariate-based signature scheme is Rainbow (Fig.3.4). Within each family of algorithms (Dilithium, Falcon, and Rainbow), there are various versions of the algorithm based on different parameter choices for controlling the security level. These parameters influence the lengths of the public key, private key, and signature for the PQC signature algorithms. As these lengths increase, the security strength also improves.

The core of **lattice-based signature systems** is a collection of n-dimensional points organized in a periodic pattern. Lattice-based cryptography is secure because it solves multiple NP-hard problems:

- Short Vector Problem (SVP): Finding the shortest non-zero vector in a lattice.
- Closest Vector Problem (CVP): Identifying the shortest vector closest to a given point.
- Learning With Errors (LWE): This problem involves solving a linear function over a finite ring given certain samples, making it computationally intensive.
- Short Integer Solution (SIS): According to Ajtai's theorem, there exists another algorithm B that can solve the Short Vector Integer issue (SVIP) if a polynomial-time algorithm A can solve the SIS issue.
- Learning With Rounding (LWR): A variant of LWE that avoids the complex randomization elements of LWE, making it more efficient.

Dilithium is one of the two lattice-based signature algorithms in the third round of the NIST PQC standardization process. It is based on the Fiat-Shamir transform and the Short Vector Problem (SVP) for its security. Dilithium offers three versions: Dilithium 2, Dilithium 3, and Dilithium 4, which correspond to security levels 1, 2, and 3, respectively, according to NIST's post-quantum security categories.

Falcon, which stands for Fast Fourier lattice-based compact signatures over N-th Degree Truncated Polynomial Ring (NTRU), is the other lattice-based signature algorithm in the third round. It relies on the NTRU scheme for key generation, encryption, and decryption, and uses Short Integer Problems (SIS), floating-point arithmetic, and Gaussian sampling for its security. The Falcon 512 and Falcon 1024 algorithms correspond to security levels 1 and 5, respectively, according to NIST's post-quantum security strengths.

Multivariate-based signature schemes, such as the unbalanced Oil-Vinegar (UOV) system, involve hiding quadratic equations in n unknowns, where "oil" represents the quadratic equations and "vinegar" refers to the n unknowns, within a finite field. This approach relies on solving quadratic equations over finite fields, an NP-hard problem. The security of these schemes depends on the number of variables and the size of the field, which results in large key sizes. Rainbow is the sole finalist among the multivariate-based signature schemes.

Rainbow, a multivariate signature scheme, is the only finalist from the multivariate category in the third round. It relies on a binary field L based on the Unbalanced Oil and Vinegar (LUOV) problem, which is then extended to a larger field K for its security. The Rainbow family includes several algorithms: Rainbow Ia, Rainbow IIIc, and Rainbow Vc, with I, III, and V corresponding to security levels 1, 3, and 5, respectively, according to NIST's post-quantum security strength criteria. Rainbow also features variants that use cyclic and compressed algorithms.

Lattice-based algorithms assess the security strength of ciphers against quantum attackers by evaluating the cost in terms of qubits needed to break them. On the other side, multivariate-based algorithms measure security strength based on the cost in quantum gates.

In classical computing, the state of a bit is always known, either 0 or 1. However, in quantum computing, the state of a qubit remains unknown until it is measured.

While a qubit behaves like a classical bit after measurement, taking on one of two possible values, it can also exist in a superposition of both states before measurement, governed by a wave function. This allows qubits to exploit quantum interference effects. Classical brute-force attacks, on the other hand, are blind searches; the attacker has no way to know if a guessed value is closer or farther from the key, and can only receive a binary outcome of "true" or "false" when checking if the key is correct.

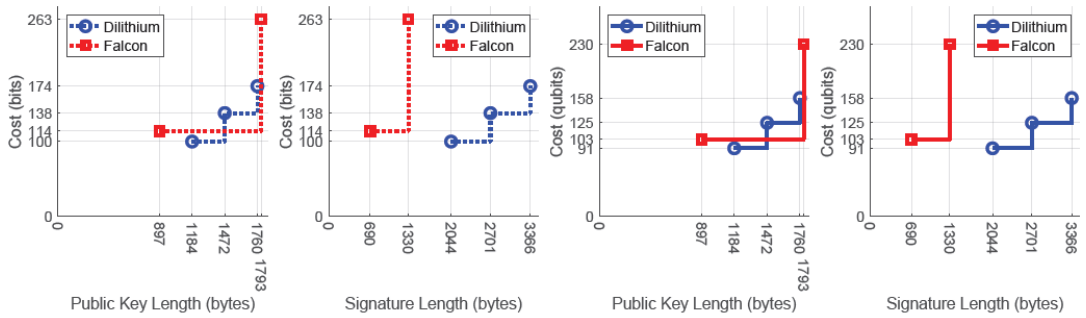
Qubit Cost: Refers to the number of qubits an attacker would need to break the cipher, assuming they have enough quantum gates.

Quantum Gate Cost: Refers to the minimum number of logical quantum gates required for a successful attack, assuming sufficient qubits are available.

A metric used to enable the inter-scheme comparison is the depth-width cost (**DW cost**). The number of RAM operations required to run a quantum circuit is directly related to the DW cost metric, which takes into account both qubits and quantum gates as key factors in the cryptanalyst's cost, assuming active error correction.

3.8.1 In-deep comparison base on qubit cost

For lattice-based algorithms, Fig 3.5 represents the classical and quantum security cost trade-off analyzing the public key length and signature length. On the one hand, reducing communication costs is achieved by using low signature and public key sizes. On the other hand, in Fig.3.5, not only an ideal algorithm should be on the left side of the plot (small communication cost), but it should be also on the top (strong security).

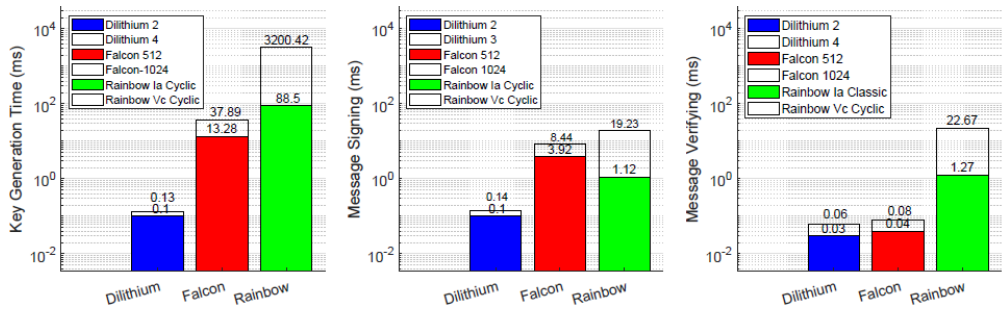


(a) Security in bits vs. public key length (b) Security in bits vs. signature length (c) Security in qubits vs. public key length (d) Security cost vs. signature length

Figure 3.5: Security costs for lattice-based signature candidates are depicted in the figures. The dotted line represents the classical bit security level (shown in the two figures on the left), while the solid line represents the quantum bit security level (shown in the two figures on the right). The vertical scales remain consistent across all the figures.[22]

In all the reported cases, Falcon reaches higher bits or qubits cost than Dilithium. Thus, it provides better security with smaller public key and signature sizes. For instance, in Fig. 3.5.c, when the public key length of Dilithium is 1760 Bytes, Falcon has only 33 bytes more, but it offers 72 qubits more quantum security cost. In Fig. 3.5.d, Falcon is close to the ideal position on the top-left corner of the plot, when its signature is less than half the signature size of Dilithium.

3.8.2 In-deep comparison base on execution time



(a) Key generation time (b) Message signing time (c) Message verifying time

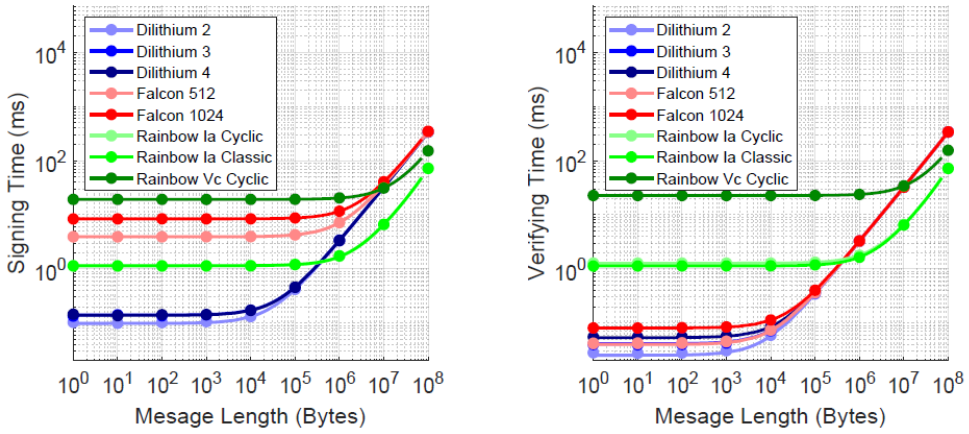
Figure 3.6: A performance analysis of the fastest (represented by colored bars) and slowest (represented by outlined bars) signature candidates from each algorithm family, measured across the three signature phases key-pair generation, signing, and verification using a message length of 100 bytes.[22]

Here is examined the execution times of each finalist algorithm for key-pair generation, signature creation (signing), and signature verification. In *"Security Comparisons and Performance Analyses of Post-quantum Signature Algorithms"* [22] the *liboqs* library is used to evaluate the performance of these algorithms. Using their benchmarking software, the researchers sample each algorithm to determine the average time required for key-pair generation, signing a message, and verifying messages. Additionally, they use different message lengths with random data to assess how the algorithms perform in relation to message scalability.

Fig. 3.6 shows the performance of algorithms within each PQC family, which is analyzed to compare their execution times once a specific scheme or family is chosen. The intra-family performance ratio, R_i, is introduced, where i represents the algorithm family ($i \in \{D, F, R\}$). For example, R_D refers to the ratio between the execution time of the slowest and fastest Dilithium algorithms. By definition, $R_i > 1$ for all families.

Dividing the execution time of the slowest algorithm by the fastest for each family reveals the following key generation ratios: $R_D = 1.3$, $R_F = 2.85$, and $R_R = 36.16$ (Fig. 3.6.a). For the message signing phase (Fig. 3.6.b), the ratios are $R_D = 1.4$, $R_F = 2.15$, and $R_R = 17.17$. In the message verification phase (Fig. 3.6.c), the results are $R_D = 2$, $R_F = 2$, and $R_R = 17.85$. These results indicate that

Dilithium demonstrates more consistent performance across its algorithms when compared to Falcon and Rainbow, especially regarding execution times within each family.



(a) Signing time, varying message lengths (b) Verifying time, varying message lengths

Figure 3.7: A performance analysis of the fastest and slowest candidates from each algorithm family. The horizontal axis is presented on a logarithmic scale, with dots indicating the discrete experimental data points.[22]

In the key generation phase, Dilithium overcomes the performances of the other finalist algorithms by a factor ranging from 102.15 to 32004.2. During message signing, Dilithium is between 8 and 192.3 times faster than the other algorithm families. For message verification, Dilithium can be up to 755.67 times faster than its counterparts, though only 0.01 milliseconds quicker than Falcon. Based on both intra-family and inter-family comparisons, Dilithium emerges as the fastest algorithm.

The analysis is extended to assess algorithm scalability by varying message lengths, comparing how performance is affected as input size increases, as is represented in Fig. 3.7.

The data in Fig. 3.7.a indicates that Dilithium 2 is the most efficient algorithm for signing until the message size reaches 387,578 bytes, at which point Rainbow Ia Cyclic becomes the fastest option. In terms of verification (Fig 3.7).b, Dilithium 2 remains the quickest algorithm until the message size exceeds 461,478 bytes, after which Rainbow Ia Classic takes over as the fastest algorithm for verifying.

3.8.3 Results emerged from the analysis

This section evaluates PQC ciphers based on their security and performance. Among lattice-based algorithms, Falcon 512/1024 demands the highest computational effort against quantum-equipped cryptanalysts, measured in qubits. The performance analysis shows that Dilithium 2 is the fastest algorithm for signing messages shorter than 390 KB, whereas Rainbow Ia Cyclic excels in signing longer messages. For message verification, Dilithium 2 is the quickest for messages up to 460 KB, while when messages exceed 460 KB, Rainbow Ia Classic performs better [22].

Dilithium can be implemented using only integer arithmetic, whereas Falcon requires floating-point arithmetic. This difference in complexity has led to delays in Falcon's standardization process. Dilithium is expected to reach standardization by the summer of 2024, however the draft Falcon standard is still pending. NIST is proceeding cautiously with Falcon to ensure thorough evaluation [23].

In a contest like V2X, in which digital signatures are used to ensure integrity, verifying efficiency is critical since numerous operations need to verify each message. Given that message sizes could be typically less than 460 KB, Dilithium 2 is recommended for its superior performance in such scenarios.

Chapter 4

Thesis purposes

This chapter outlines the primary objectives of this thesis, which will be crucial and useful in defining the expected outcomes. By clearly defining these objectives, the purpose of the study becomes well delineated, allowing for a precise approach and a comprehensive understanding.

Vehicle-to-Everything (V2X) technology is increasingly gaining traction worldwide and represents an important step towards improving driving safety by creating real inter-vehicle networks. From a security perspective, V2X technology relies on a Public Key Infrastructure (PKI) and follows currently developed methods, particularly public-key algorithms. The security of public-key-based computer systems is particularly at risk from the advent of quantum computing. This presents a serious problem that V2X must confront.

The main objective of this thesis, which has not yet been extensively addressed in the current literature, is to find a point of convergence between V2X and Post-Quantum Cryptography, a solution to counter the potential threats posed by quantum computing.

To achieve this goal, the work has focused on replacing the current standard algorithms for the message digital signature process, based on elliptic curves, with a Post-Quantum algorithm.

The working environment for conducting these studies is a proprietary framework used by *Nardò Technical Center - Porsche Engineering*, where the architecture and code necessary to simulate the transmission of CAM messages in a V2X environment

are implemented. If this proprietary solution fails to meet the required development needs, an open-source alternative will be considered.

The specific objectives to reach the main goal of the thesis are:

- Integrating PQC APIs into the code used, to extend the standard followed for inter-vehicle communication. Specifically, this includes incorporating functions for key creation, message signing, and signature verification, which are crucial in the process of generating and sending CAM in a V2X environment.
- Extending the structure and fields of the certificates of the entities involved in the V2X PKI infrastructure. Specifically, the RCA, AA, and AT use X.509 certificates where the keys are elliptic curve-based, as is the certificate signature. The expected outcome of this work is to generate new certificates that support the use of Post-Quantum Cryptography algorithms for the signing process.
- Sending new non-standard CAM messages over the network interface, containing not only the new certificate inside the appropriate field of the Secured Header but also appending the new digital signature, generated using the specific PQC API and calculated on the unsecured part of the message.
- Conducting a comparative analysis of message size and signing times between the existing standard method of sending CAM and the method implemented in this thesis.

The achievement of these objectives will provide valuable insights into the integration of PQC algorithms in V2X environments, potentially influencing future standards in automotive security and contributing to the broader field of post-quantum cryptography.

Having established the objectives of this thesis, the next chapter will explore the methodology used to achieve these goals.

Chapter 5

Case of study

5.1 First study on a proprietary solution

The work to reach the objectives has evolved from a proprietary solution to an open-source solution. Initially, the work environment was a proprietary solution that *Nardò Technical Center - Porsche Engineering* made available: **Virtual Machines provided by Cohda Wireless**. Cohda Wireless[24] is the leading provider of innovative Connected Vehicle solutions, enabling vehicles to communicate with each other and with intelligent infrastructure in the ITS, Smart City, and mining sectors. In 2023, it became part of the Danlaw Group, with the aim of accelerating the development of advanced connected vehicle safety, smart city solutions, and safety at mining sites. Accurate vehicle positioning, cooperative collision avoidance, and congestion reduction are made possible by the innovative software that was developed. Vehicles equipped with Cohda Wireless technology may interact in real-time with traffic infrastructure, identify invisible obstructions, and calculate their position without using a GPS. The Cohda Wireless virtual machine simulates the OBU (On Board Unit). Thus, the initial idea was to set up two Virtual Machines to emulate the behavior of a sender and a receiver in a V2X system.

These virtual machines have a Vehicle Simulator (VSIM), an engineering tool intended to help the development and testing of safety applications. Among its primary functions, the reliable one for testing the VMs is drive simulation, obtaining synthesized GNSS data for driving virtual vehicles. The data produced by the

simulator are integrated into the Cohda sender firmware, which is used to fill the CAM structure and, finally, to send the message to the receiver.

Wireshark, a tool to capture and display the messages sent back and forth on the network, is used to intercept the CAM messages. An example is reported in Fig 5.1. Wireshark permits the analysis of the content of the message, in particular the structure of CAM and its headers.

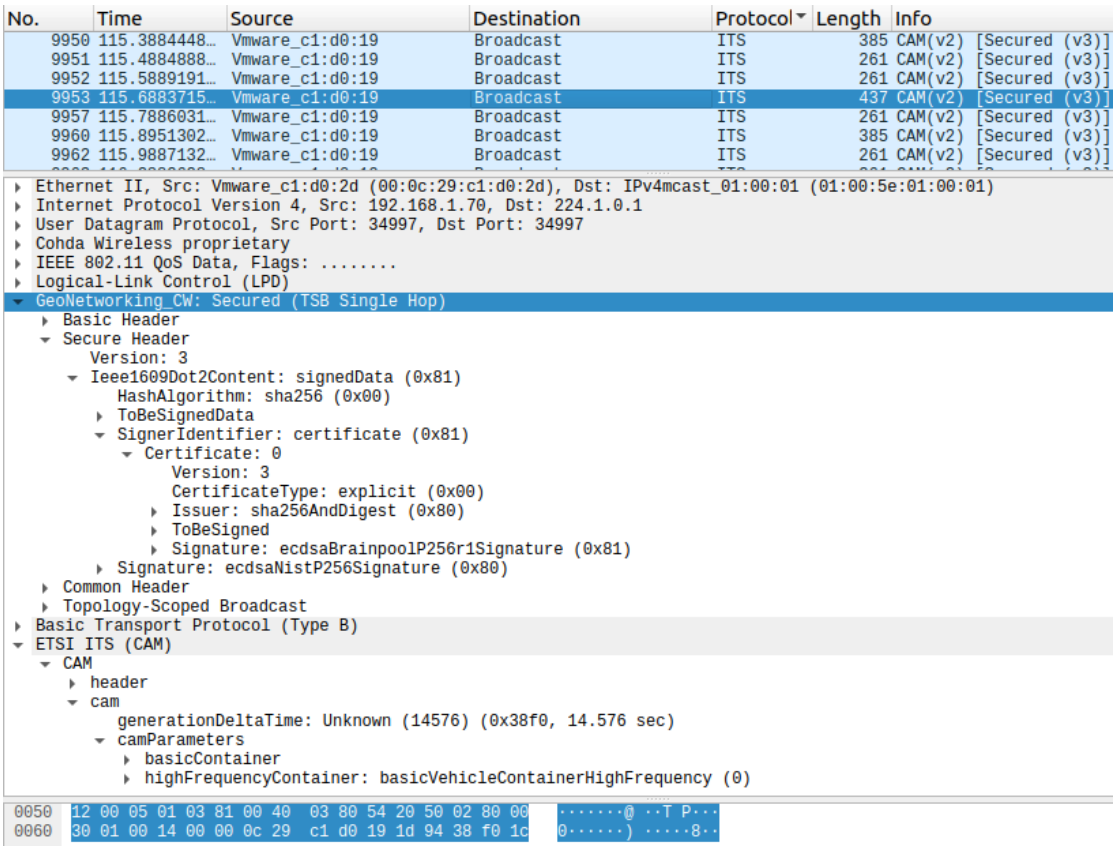


Figure 5.1: Example of CAM sniffed with Wireshark into Cohda Wireless VM

Once the entire environment is set up and it has been verified that the virtual machines can communicate with each other and with the host, thanks to Bridged mode, the idea was to send custom CAM from the host to the sender. The purpose is to have the Cohda firmware which depends on the data sent by a proper socket UDP, having control of the content of the message, and not from the simulator.

The challenge with this proprietary solution lies in the Cohda APIs, especially the API tasked with sending messages. This function combines both the generation

of the message signature and the transmission over the network interface. For this project, it is necessary to separate the signing and sending processes, and Cohda does not permit it.

For this reason, the thesis study switches to an open-source solution with APIs suitable for our purpose.

5.2 V2X Open-Source tool

The encountered issue with *Codha Wireless* led the work to an open-source tool called ***fsmsggen*** developed by the company *FilLabs* and available on GitHub.

FSMsgGen is a demonstration tool and V2X message generator. It deals with CAM (as defined in ETSI EN 302 637-2), DENM (as defined in ETSI EN 302 637-3), PKI (as defined in ETSI TS 102 941).

FSMsgGen uses the *FITSec* library, also developed by FilLabs. It is a robust library for real implementations: provides a security engine, implementing messages and certificates processing for ITS communications. This library is also available on GitHub.

FSMsgGen can inject and capture V2X messages directly through the Ethernet interface using the PCAP library. The tool needs a set of certificates to send and verify messages. These certificates can be compliant with the certificate profiles present in ETSI ITS Security Test Suite or with an own set of certificate profiles. To start injecting messages, it is needed:

- one RootCA certificate;
- one AA certificate, signed with RootCA certificate;
- at least one AT certificate, signed with AA certificate and providing CAM permissions.

For generating the needed certificates, a certificate generation tool is used. It is called ItsCertGen, and it is a part of the ETSI ITS Security test suite. It is used for testing purposes and permits to generate XER from profile, convert XER to OER, and generate keys and signatures.

The full portability of the certificate and CAM message generation framework is ensured using containers. Containers use a lightweight virtualization technology that minimizes the impact on performance and resource usage, making them particularly well-suited for software development, testing, release, and deployment in production environments. One of the most widely used container solutions is Docker. A Docker container is created from an image and includes all the necessary tools to start and run the application.

Compared to virtual machines, the advantages offered by containers are:

- Ease of horizontal scalability due to their lightweight nature. It is possible to create/delete container instances to manage the load quickly;
- Ability to run on any compatible environment;
- Sharing kernel with the host operating system, resulting in reduced resource consumption;
- Complete isolation between applications running in separate containers, even though they share the same operating system kernel.

5.3 PQC certificate profiles

ItsCertGen is designed to generate OER certificates starting from XER certificate profiles. In ETSI ITS Security Test Suite the cert profiles respect the ETSI 103 097 standard [16].

The RCA, AA, AT certificate profiles differ only for some fields:

- All of them have version, type, and issuer fields before the ToBeSigned part, but the RCA issuer is itself, instead for AA the issuer is RCA, and for AT the issuer is AA.
- The ToBeSigned part is composed of id, cracaId, crlSeries, validityPeriod, assuranceLevel, and a different number of appPermissions or certIssuePermissions in base of which entity is. In the RCA cert profile, there is only the verificationKey of type ecdsaNistP256, instead both AT and AA have the encryptionKey of type eciesNistP256, and the verificationKey of type ecdsaNistP256.

- The signature field is of type ecdsaNistP256Signature and it is the same in all of them.

For the thesis purpose, these certificate profiles need to be extended to support Post Quantum Cryptography. The chosen algorithm is Dilithium version 2: the reasons are explained in Section 3.8, and in particular, in sub-section 3.8.3, page 46.

The focus of the work is on the Signature and VerificationKey. The latter is useful for verifying the validity of digital signatures. The original state of the fields to change into the certificate profile is:

```
 <verifyKeyIndicator>
    <verificationKey>
      <ecdsaNistP256>
        <compressed-y-0/>
      </ecdsaNistP256>
    </verificationKey>
  </verifyKeyIndicator>
</toBeSigned>
<signature>
  <ecdsaNistP256Signature>
    <rSig>
      <x-only/>
    </rSig>
    <sSig/>
  </ecdsaNistP256Signature>
</signature>
```

The extended version for the Dilithium key and signature is:

```
<verifyKeyIndicator>
  <verificationKey>
      <dilithium2>
        <publicKey/>
```

```
        </dilithium2>
      </verificationKey>
    </verifyKeyIndicator>
  </toBeSigned>
  <signature>
    <dilithiumSignature>
      <signatureValue/>
    </dilithiumSignature>
  </signature>
```

As noticed, the type of algorithm used for key and signature is changed to Dilithium version 2, and, the new signature has only one component, rather than two components (rSig and sSig) like in ecdsa form.

The whole code of the Dilithium AT certificate profile is shown in Appendix A.1.

5.4 Dilithium certificate generation

5.4.1 Changes into ASN1C module files

The initial step for enabling ItsCertGen to support the Dilithium version of certificate generation involves extending the ASN.1 module files. **ASN.1 compiler** permits to generate the C source code starting from the ASN.1 module files. The obtained code can be used to serialize and deserialize the native C structures into compact BER/OER/PER/XER-base data files. The compiler used in this work is not the original one, because it is not compatible with some complex asn1 features, but is developed by *FilLabs*, available on GitHub.

The extensions in ASN.1 module file are in *Iee1609Dot2BaseTypes.asn*. In the section of crypto structures, the Dilithium signature structure has been defined and the Signature structure, containing the list of supported public key algorithms, is extended to include the Dilithium Signature. The extended version of the Signature structure and the new DilithiumSignature struct in asn file and in the produced C code is shown in Appendix A.2. An example of the changes made in asn1c is:

```
Signature ::= CHOICE {
    ecdsaNistP256Signature        EcdsaP256Signature,
    ecdsaBrainpoolP256r1Signature EcdsaP256Signature,
    ...,
    ecdsaBrainpoolP384r1Signature EcdsaP384Signature,
    ecdsaNistP384Signature        EcdsaP384Signature,
    sm2Signature                  EcsigP256Signature,
    dilithiumsignature            DilithiumSignature
}

/* added Dilithium Sig struct */
DilithiumSignature ::= SEQUENCE {
    signature   OCTET STRING(SIZE(2420))
}
```

The process of extending the Public Verification Key to support the Dilithium Key closely follows the procedure used for signature. The definitions for PublicVerificationKey and DilithiumKey in asn file, along with the generated source code in C, is displayed in Appendix A.3

5.4.2 Extensions in ItsCertGen code

Once the ASN.1 specification has been compiled with the asn1c command, the ItsCertGen code has all the necessary structures for extending the certificate format.

In the process of certificate generation, functions to produce key pairs and signatures are needed. ItsCertGen only provides the API for elliptic curve algorithms. To integrate the Dilithium parameters and functions the open-source library ***liboqs*** has been used. Liboqs is a C library for prototyping and experimenting with quantum-resistant cryptography, and it is available on GitHub. Liboqs offers:

- A collection of open-source implementations regarding the quantum-safe key encapsulation mechanisms (KEM) and digital signature algorithms;
- A unified API for interacting with these algorithms;

- A test framework and benchmarking tools.

Liboqs is part of the Open Quantum Safe (OQS) project, which focuses on developing and integrating quantum-safe cryptography into applications to support real-world deployment and testing.

The constants and APIs defined in liboqs and useful for the thesis work are:

```
#define OQS_SIG_dilithium_2_length_public_key 1312
#define OQS_SIG_dilithium_2_length_secret_key 2528
#define OQS_SIG_dilithium_2_length_signature 2420

OQS_SIG *OQS_SIG_dilithium_2_new(void);
OQS_API OQS_STATUS OQS_SIG_dilithium_2_keypair(uint8_t *public_key,
 ↪  uint8_t *secret_key);
OQS_API OQS_STATUS OQS_SIG_dilithium_2_sign(uint8_t *signature,
 ↪  size_t *signature_len, const uint8_t *message, size_t
 ↪  message_len, const uint8_t *secret_key);
OQS_API OQS_STATUS OQS_SIG_dilithium_2_verify(const uint8_t
 ↪  *message, size_t message_len, const uint8_t *signature, size_t
 ↪  signature_len, const uint8_t *public_key);
```

The main function of ItsCertGen is into the file *certgen.c*. Here the main steps followed are:

- Decoding the certificate profile and memorizing the result into a certificate struct.
- Check the signer. It could be a higher entity in the PKI hierarchy or itself in the case of RCA.
- Generation of encryption and verification keys.
- Generation of signature.

The significant changes compared to the original version have primarily been in the signer verification, generation of key pairs and signature.

Check the signer

The extensions in the code segment responsible for checking the signer and assigning the signer hash to the certificate issuer field follow the same steps for standard certificates.

The distinction is between RCA and the pair of entities AA and AT: on the one hand, RCA is self-signed, on the other hand, AA and AT have their signer entity. In both cases, once the type of hash is checked, the signer hash is computed.

The steps applied for the Dilithium version are the same as ecdsaNistP256: in both cases, the hash method is sha256. For RCA, a pre-computed string is assigned to signerHash. For AT and AA, the signer certificate and its length are passed to a function called *sha256_calculate*, and the signerHash is filled with the obtained digest. At the end, the *OCTET_STRING_frombuf* function clears and allocates new memory for the certificate issuer field to store the eight least significant bits of the signerHash.

The code for the computation of signer digest in AA and AT is shown below:

```
1  switch (hashType)
2     {
3      ...
4      case PublicVerificationKey_PR_dilithiumKey:
5          if (cert->issuer.present == IssuerIdentifier_PR_NOTHING)
6                  cert->issuer.present =
                    ↪  IssuerIdentifier_PR_sha256AndDigest;
7          sha256_calculate(_signerHashBuf, buf, ebuf - buf);
8          _signerHash = &_signerHashBuf[0];
9          _signerHashLength = sha256_hash_size;
10         OCTET_STRING_fromBuf(&cert->issuer.choice.sha256AndDigest,
           ↪  &_signerHash[sha256_hash_size - 8], 8);
11         break;
12     ...
```

Generation of Dilithium key pair

The ItsCertGen project includes not only the generation of the OER certificate but also the generation of some files that contain encryption and verification keys. For instance, starting from the certificate profile CERT_IUT_A_AT_Dilithium.xer, the files in which the keys are stored are:

- CERT_IUT_A_AT_Dilithium.ekey. It contains the private encryptionKey will be used in the decryption phase.
- CERT_IUT_A_AT_Dilithium.ekey_pub. It contains the public encryptionKey will be used in the encryption phase.
- CERT_IUT_A_AT_Dilithium.vkey. It contains the private verificationKey will be used for signing messages.
- CERT_IUT_A_AT_Dilithium.vkey_pub. It contains the public verificationKey will be used for verifying the signature.

Only for RCA, the encryption key is not generated. In the PKI infrastructure, the RCA is a trusted entity with the primary task of digitally signing certificates. The verification key is used to ensure the authenticity and integrity of digital signatures. On the other hand, AA and AT have both verification and encryption key pairs.

The thesis purpose focuses on the Public Verification Key because it is responsible for the signature phase, which is the main topic of the study. The encryption Key is not changed, it is still an elliptic curve key. Thus, the obtained certificates are hybrid, with keys that use different approaches.

To support the Dilithium verification key pairs, a well-designed function has been provided for generating the new types of keys. In the piece of code in which the presence and type of verification key are checked, a case branch has been added, as shown below:

```
1 switch (cert->toBeSigned.verifyKeyIndicator.present)
2   {
3     case VerificationKeyIndicator_PR_verificationKey:
```

```
    switch (cert->toBeSigned.verifyKeyIndicator.choice.
            verificationKey.present)
      {
        ...
        case PublicVerificationKey_PR_dilithiumKey:
         rc = fill_Dilithium_keyPair(&cert->toBeSigned.
              verifyKeyIndicator.choice.verificationKey.choice.
              dilithiumKey, 2, buf);
         break;
```

In this case branch, the *fill_Dilithium_keyPair* is called. The parameters of this function are the dilithiumKey field, contained in the verificationKey struct of the certificate, the version of the algorithm used (in this case Dilithium 2), and the buffer containing the path of the file which will store the private verification key.

The implementation of the whole *fill_Dilithium_keyPair* is shown in Appendix A.4. The steps followed in this function are:

- allocating the memory area for public and private keys. The length of the public key of Dilithium version 2 is 1312 bytes, and the length of the private key is 2528 bytes. The constants defined in *liboqs* are used.
- generating the key pair using the *OQS_SIG_dilithium_2_keypair* function. This is defined in *liboqs*, and the parameters passed are the buffer to store the public key in the certificate field, and the allocated variable for the private key. This function returns a flag, which indicates if the operation has success or not. Here is the described function call:

```
OQS_STATUS stat =
 ↪  OQS_SIG_dilithium_2_keypair(dilithium->publicKey.buf,
                                private_key);
```

- storing the key pair in the respective files. In both cases, the files are opened in write-bytes mode and the keys are written. For the private key file, the path of the file is the "path" parameter passed to the function *fill_Dilithium_keyPair*.

On the other side, the path is the previous one, but with the extension ".vkey_pub" instead of ".vkey".

Generation of Dilithium digital signature

The signature function will be applied on a joint hash: the hash of the ToBeSigned part concatenated with the signer hash. Thus, the step before the signature function call is to compute the sha256 hash of ToBeSigned fields. The function called is the *sha256_calculate* with as parameters the buffer to store the obtained hash, the pointer to the toBeSigned segment, and the length of it.

```
1 _tbsHashLength = 32;
2 sha256_calculate(_tbsHash, (const char *)oer, rc.encoded);
```

Once this sha256 hash is computed, the generation of the Dilithium signature is the last step in the certificate creation process.

The function responsible for this purpose is *Signature_oer_encoder*. It was already implemented in ItsCertGen, but it has been extended to support not only the standard digital signature method with elliptic curve algorithms but now is also compliant with the Dilithium signature. The whole code of the new added section of this function is shown in Appendix A.5. The detailed described steps followed are:

- checking the to-be-signed hash type in the if clause to determine if the digital signature is standard (elliptic curve) or not standard (Dilithium);
- obtaining the secret key of the certificate signer with an ad-hoc function written for the Dilithium version, similar to the original one. The function is *search_private_Dilithium_key* and it is shown in Appendix A.5.1. Starting from the path of the file containing the secret key, it permits to opening of the relative file in read-bytes mode, extracts the content of the file, and outputs it.
- calculating the joint hash with the ToBeSigned Hash computed in *ToBeSigned-Certificate_oer_encoder* and the signer hash. It is possible to concatenate the two initial hashes and call the *sha256_calculate* function with as parameters

the buffer to contain the joint hash, the buffer that have the concatenation of the hashes and its length. Here is an extract of the code:

```
memcpy(_tbsHash + _tbsHashLength, _signerHash,
 ↪  _signerHashLength);
switch (hashId)
{
   case sha_256:
   sha256_calculate(h, _tbsHash, _tbsHashLength +
    ↪  _signerHashLength);
   hl = sha256_hash_size;
   break;
```

- allocating the necessary memory space for storing the signature. In the Dilithium version, the signature is 2420 bytes long. The function to sign the certificate is provided by *liboqs*, it is called *OQS_SIG_dilithium_2_sign* and accepts as parameters:
  - the buffer which will store the signature
  - the signature length
  - the message to sign
  - the message length
  - the secret key used for signature

  In this case, the buffer for the signature is the specific field of the certificate, previously allocated with the exact size thanks to the *liboqs* constant, the message is the computed joint hash, and the secret key is the signer private key obtained from the *search_private_Dilithium_key*. The function outputs a flag to indicate if the process ends with success or not. Here is the piece of code responsible for the signature creation:

```
s->choice.dilithiumsignature.signature.buf =
 ↪  malloc(OQS_SIG_dilithium_2_length_signature);
s->choice.dilithiumsignature.signature.size =
 ↪  OQS_SIG_dilithium_2_length_signature;

OQS_STATUS check = OQS_SIG_dilithium_2_sign(
        s->choice.dilithiumsignature.signature.buf,
        &(s->choice.dilithiumsignature.signature.size),
        h, hl, secret_key);
```

5.4.3 Outputs of ItsCertGen

The integration of several libraries has enabled the steps outlined in previous sections. As just demonstrated, the *ItsCertGen* framework alone is insufficient to extend the standard to support new digital signature algorithms. A key role has been played by *liboqs* in facilitating the inclusion of post-quantum cryptography algorithms. The code has been extended beginning with the certificate profile from the *ETSI ITS Secure Test Suite* and making the necessary modifications to achieve the objectives of this work.

The expected results were the generation of the .oer files containing the certificates, and the .vkey and .ekey files for the keys, and they can be considered achieved. The well-designed files produced are:

* RCA entity:
  - CERT_IUT_A_RCA_Dilithium.oer;
  - CERT_IUT_A_RCA_Dilithium.vkey;
  - CERT_IUT_A_RCA_Dilithium.vkey_pub.
* AA entity:
  - CERT_IUT_A_AA_Dilithium.oer;
  - CERT_IUT_A_AA_Dilithium.vkey;

- CERT_IUT_A_AA_Dilithium.vkey_pub;
- CERT_IUT_A_AA_Dilithium.ekey;
- CERT_IUT_A_AA_Dilithium.ekey_pub.

∗ AT entity:

- CERT_IUT_A_AT_Dilithium.oer;
- CERT_IUT_A_AT_Dilithium.vkey;
- CERT_IUT_A_AT_Dilithium.vkey_pub;
- CERT_IUT_A_AT_Dilithium.ekey;
- CERT_IUT_A_AT_Dilithium.ekey_pub.

These produced files are the starting point for generating and injecting new non-standard CAM in the next part of the thesis study.

5.5 PQC-CAM generation

The chosen open-source tool for implementing new non-standard CAM in the thesis study is ***fsmsggen***. It was briefly explained in Section 5.2, and available on GitHub.

This tool permits managing CAM messages, customizing them, and injecting them into a network interface.

FSMsgGen uses the *FITSec* library, a robust library for real implementations: provides a security engine, implementing messages and certificates processing for ITS communications. This library is also available on GitHub. Its main APIs are explained in the following sections. Unfortunately, not all the sources of APIs are open.

This tool needs a set of certificates and keys. The files generated with the implementations described in previous sections in ItsCertGen tool must be integrated into the fsmsggen folder. As the Readme file of fsmsggen suggested, the certificate and keys files generated for this study are put in the default path "./POLL_CAM". The useful files in this tool are *fsmsggen.c*, *load_data.c*, *msggen_cam.c*.

The general command used to launch the program is "./fsmsggen -i eth0 -1 path/to/certs", where path/to/certs is the path to the directory containing the keys and certificates that *ItsCertGen* obtained (in this thesis work, it is ./POOLĊAM), and eth0 is the network interface.

The principal followed steps in *main()* function in *fsmsggen.c* function are:

- initialization of CAM parameters thanks to the *cam_options()* function defined in *msggen_cam.c*
- installation of certificates in fsmsggen framework.
- signing and injection of messages.

The initialization of CAM parameters is not the thesis objective, because the aim is to attach to them new signatures. The content of CAM fields is not as important as the structure of CAM messages. Thus, the thesis study focuses on the last two steps, as analyzed in the next sections.

5.5.1 Installation and verification of new certificates

This is the first key point that permits the extension of fsmsggen to support the Dilithium digital signatures.

The function used to load the certificates is *FitSec_LoadTrustData()* and the parameters are the FitSec engine, the current time and the path passed by command line after the flag "-1". For the thesis purpose, this latter value is the path of the certificates directory. The function checks if this path exists and if it is a directory, and in that case checks the regular files into the directory.

In the original code of fsmsggen tool, in *load_data.c*, the selection of each certificate does not follow a precise order. The load of the entities with a higher position in the PKI hierarchy is important for verifying the single certificates. The code has been changed to permit to load as the first entity the RCA, then AA and AT. This is possible thanks a first part in which all the certificates are stored in a list containing the ".oer" certificates, as shown below:

```
while (NULL != (de = readdir(d)))
{
    pchar_t *ext = pchar_rchr(de->d_name, '.');
    if (!ext || 0 == strcmp(ext, ".oer") ||
                0 == strcmp(ext, ".crl") ||
                0 == strcmp(ext, ".ctl") ||
                0 == strcmp(ext, ".lcr") ||
                0 == strcmp(ext, ".crt"))
    {
        if (file_count < max_num_files) // save all file names
              files[file_count++] = strdup(de->d_name);
    }
}
```

The second part deals with a *qsort* function for ordering the list of files following determined rules of comparison designed in a *compare_files()* function, as displayed in Appendix A.6.

The purpose of the comparison function is to follow the order RCA-AA-AT

because the thesis feasibility study is focused on the easiest case of one RCA, one AA, and one AT. Thus, the RCA signs AA, and AA signs AT during the certificate generation phase.

```
qsort(files, file_count, sizeof(char *), compare_files);
```

This function could be extended to support more complex situations. Once understood that integrating Digital signatures in CAMs is possible for this specific and minimal case, another step is to verify it in a context with more AA and AT. So the order of processed certificates might take into account which entity is the signer of another one.

However, each certificate path is passed to *load_data()* function.

The original aim of this method was to fill a certain FitSec struct with the content of the cert file. Indeed, the sequence of bytes of the certificate is stored in a "data" variable, and the verification and encryption secret keys are also extracted and put in "vkey" and "ekey" variables. These values, their lengths and the FitSec engine are the parameters of the function *FitSec_InstallCertificate()*. This method is used in the original version of fsmsggen tool to install any kind of certificate (root, AA, local AT pseudonyms, or any other).

```
const FSCertificate *c = FitSec_InstallCertificate(e, data,
↪  cert_len, vkey, vkey_len, ekey, ekey_len, &error);
```

Unfortunately, the code of *FitSec_InstallCertificate()* is not available, so it is not possible to know the precise performed steps, as well as for two other used FitSec functions: *FSCertificate_Digest()* and *FSCertificate_Name()*.

It is only possible to deduce the steps followed based on the inputs, outputs, and brief function descriptions. For example, the two functions mentioned above return, respectively, the digest and the name derived from a certificate. Both start with a well-designed certificate, that is the output of *FitSec_InstallCertificate()*. This function installs certificates in a V2X (Vehicle-to-Everything) security engine, verifying and associating the appropriate keys to ensure secure communications. This method does not support the Dilithium version of certificates or keys, so it has been necessary to write a function that emulates its behavior.

This new function has been called ***Emulated_InstallCertificate()***, and it is displayed here:

```
EtsiExtendedCertificate *certif = Emulated_InstallCertificate(data,
↪ &certif_len, cert_len, vkey, vkey_len, ekey, ekey_len, &error,
↪ flag_PQC, entity);
```

The parameters are much similar to those passed to *FitSec_InstallCertificate()*. The added ones are "certif_len" which is the length of the new structure in output, and "flag_PQC" which indicates that the certificate is of Dilithium version, and the entity of the certificate.

The output of *FitSec_InstallCertificate()* is *FSCertificate*, and it is assumed that it is a well-designed structure that respects the standard rules for certificate, in particular those specified in ETSI 103 097[16]. For the *Emulated_InstallCertificate* an *EtsiExtendedCertificate* structure has been written that satisfies the requirements to be compliant with the standard apart from the signature and verification key, which now there is the double option: Dilithium or standard version.

The implementation of this personalized structure has been possible by integrating all the fields described in the RCA, AA, and AT certificate profiles. Indeed, the certificate structures of these different entities are very similar, but not equal. Some fields could not be present in different certificates. Thus, the work tries to be as much compliant asthe standard and the certificate profiles, adding new fields for the Dilithium implementation. The body of this structure is shown here:

```
typedef struct{
    uint8_t version;
    uint8_t type;
    Issuer issuer;
    ToBeSigned toBeSigned;
    union{
      FSSignature FSsignature;
      DilithiumSignature_t DILsignature;
    }sig;
} EtsiExtendedCertificate;
```

As expected, the main sections of the certificate are the version, the type, the issuer, the toBeSigned part (in which there is also the verification key), and the signature. This latter could be a standard or non-standard signature. It means this structure supports both the versions.

```
typedef struct{
    union{
        FSPublicKey FSverificationKey;
        DilithiumKey_t DILverificationKey;
    }val;
} VerifyKeyIndicator;
```

The verification key also supports two typologies. This choice is taken to have a modular scheme, in which the implementations of both versions are well-separated and can be easily swapped or updated without altering other parts of the code.

Where possible, the fields compliant with ETSI 103 097[16] are defined with the reuse of structures defined in *fitsec.h*. Some examples are *FSPublicKey* and *FSSignature*. The whole definition of the *EtsiExtendedCertificate* structure is reported in Appendix A.8.

The designed structure fields are filled into the *Emulated_InstallCertificate* function. This function has more aims:

- working like a parser, to decrypt a row sequence of bytes in a well-designed structure.
- verifying the signature of the certificate to guarantee integrity and authenticity.

Decrypting the certificate sequence of bytes

This operation has been possible by analyzing the content of the ".oer" files of RCA, AA, and AT. The effort is to translate what seems to be a raw sequence of bytes into a meaningful whole of values. Some correspondences come out when analyzing the content of the ".oer" files and comparing them with the ".xer" profiles.

Numerous bytes indicate the beginning point of some fields, and in some cases also the expected length of the introduced field.

Here is an example: In the ".oer" certificate of AA a string of bytes like this " 80 01 8c 82 0a 04 02 ff ff e0 04 ff 00 00 1f 80 02 02 7d 82 04 01 01 01 ff " could be difficult to interpret. Analyzing the whole certificate sequence of bytes, the position of the sequence, and comparing it with the structure defined in the respective certificate profile, an interpretation has been possible. A scheme that shows the correspondence between the bytes and their exact values is attached below:

```
80                      PsidSspRange indicator
01 8c                      len psid + value spid="SREM"
82 0a                      sspRange indicator + len sspRange
04                            len sspValue
02 ff ff e0                   sspValue
04                            len sspBitMask
ff 00 00 1f                   sspBitMask
80                      PsidSspRange indicator
02 02 7d                   len psid + value spid="SSEM"
82 04                      sspRange indicator + len sspRange
01                            len sspValue
01                            sspValue
01                            len sspBitMask
ff                            sspBitMask
```

This sequence of bytes has been chosen because it is possible to notice the indicator of the fields, their repetition associated with different lengths, and the effective values assigned to the fields. Indeed, PsidSspRange could be a structure that contains the psid and optionally the sspRange, which in turn includes the sspValue and sspBitMask. Each psid, sspRange, sspValue, and sspBitMask field is coupled with the proper size.

This process has been applied to every certificate and field. The code performing this operation is compatible with both the standard certificate version and the Dilithium version, thanks to the "flag_PQC" passed to *EtsiExtendedCertificate* that helps this double implementation.

The *EtsiExtendedCertificate*, generated by this new function, was intended to replace the original structure used by the *FitSec* library for storing certificate value. Indeed, in *fitsec.h* a structure called *FSMessageInfo* is defined to store all the information about the message to send. Among all this information, there are also those referred to the signature parameters, and, in particular, the *FSCertificafe* value to store the certificate to be used for signature.

An attempt was made to force the assignment of the *FSMessageInfo* certificate field with the most recently generated AT certificate structure, but this did not affect the format of the message sent. It is assumed that this is because the FitSec engine used in the *FitSec_InstallCertificate()* performs operations and assignments that, without having access to the engine's structure and the function's code, cannot be predicted.

Verifying the signature of certificate

The *Emulated_InstallCertificate* function works not only as a parser but also as a tool for verifying the certificate's signature. This verification ensures both the authenticity and integrity of the analyzed certificate. A successful check confirms that the certificate has not been tampered with and has been issued by the legitimate authority.

A proper function has been designed for this purpose, and it is:

```
1 void verifySignature(uint8_t *data, size_t cert_length,
2                       uint8_t *_toBeSigned, int lenTBS,
3                       uint8_t *sig, char *entity)
```

The parameters are the analyzed certificate and its length, the pointer to the ToBeSigned section and its size, the signature, and the entity of the certificate. The whole code of this function is shown in Appendix A.9. In this function, the method used for the verification of the Dilithium signature is defined in *liboqs* library and it is:

```
1 OQS_API OQS_STATUS OQS_SIG_dilithium_2_verify(const uint8_t
  ↪  *message, size_t message_len, const uint8_t *signature, size_t
  ↪  signature_len, const uint8_t *public_key);
```

The parameters are the message for which the signature is being verified and its length, the signature and its size, and the public key of the signer.

This *verifySignature* function analyzes three different cases:

- entity = RCA: the last byte of the certificate sequence of bytes is set to "0x80" like done in *ItsCertGen* before to perform the signature. Subsequently, the digest is computed with the *sha256_calculate* function and assigned to a *_signerHashRCA* variable.

```
data[cert_length - 1] = 0x80;
sha256_calculate(_signerHashBuf, data, cert_length);
_signerHashRCA = &_signerHashBuf[0];
_signerHashLength = sha256_hash_size;
```

- entity = AA: various steps are performed:
  - computation of ToBeSigned hash calling the *sha256_calculate* with as parameters the buffer which will contain the hash of tbs, the ToBeSigned pointer, and the proper length.
  - concatenating the signerHashRCA to the toBeSigned Hash (computed with the step before), and calculating the joint hash using the *sha256_calculate* with as parameters the output buffer *hash* (it will be the parameter "message" of the liboqs function for the verification of signature), the buffer containing the concatenation of digests and the total length of it.

```
memcpy(_tbsHash + _tbsHashLength, _signerHashRCA,
 ↪  _signerHashLength);
sha256_calculate(hash, _tbsHash, _tbsHashLength +
 ↪  _signerHashLength);
hashlen = sha256_hash_size;
```

  - computation of the digest *signerHashAA*, following the same process described in the previous RCA case.

- extracting the public key of the signer of AA. The signer of AA is the path to the file name of the signer of the current entity, and it is set using the *set_signer* function in Appendix A.7. However, the function used to extract the public key is *search_public_Dilithium_key* function.

```
char *pubKeySigner = search_public_Dilithium_key(signerAA);
```

- verifying the signature passing to the *OQS_SIG_dilithium_2_verify* function as parameters the hash computed before and its length, the signature and its size, and the public key of the signer. The function outputs a flag that indicates if the process ends with success or not.

```
OQS_STATUS result = OQS_SIG_dilithium_2_verify(hash,
↪  hashlen, sig, OQS_SIG_dilithium_2_length_signature,
↪  pubKeySigner);
if (result != OQS_SUCCESS){
    fprintf(stderr, "\nERROR: OQS_SIG_dilithium_2_verify
    ↪  failed!\n");
}
```

- entity = AT: the steps are the same as those followed in AA case, except for the computation of digest, because AT is not the signer of any other entity, this step is useless. Here is a brief recap of the steps:
  - computation of ToBeSigned hash.
  - computation of joint hash.
  - extracting the public key of the signer of AT.
  - verifying the signature.

5.5.2 Signing and injection of message

The operations described in the previous section, as the generation of new certificates with public key and signature of Dilithium type, and the integration of them in fsmsggen tool, are preparatory for the key point of the work: the signing of CAM message with Dilithium algorithm. This is the last but fundamental step before injecting the CAMs into the network interface.

To reach this goal, the function *MsgGenApp_Send()*, in *fsmsggen.c*, has been extended to support also the Dilithium version. This is possible thanks to the *flag_PQC*, which distinguishes the operations performed in standard or non-standard cases. The whole code of this function is inserted in Appendix A.10

Dilithium digital signature of CAM

The scheme of digital signature is the following: [25]

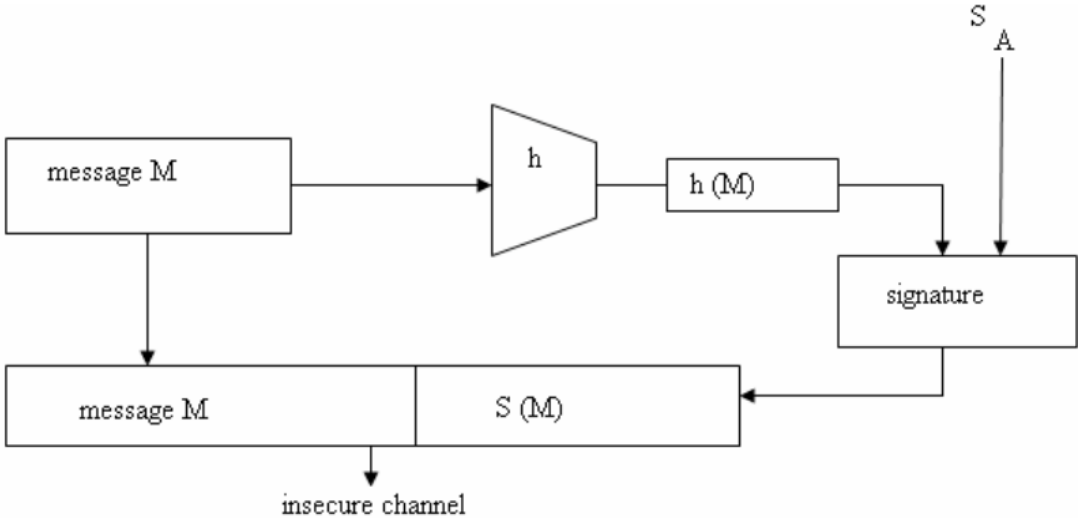


Figure 5.2: Digital Signature scheme[25]

The message M is compressed through a hash function, and its hash (h(M)) is the input of the signature algorithm with the entity's private key (S_A). This algorithm computes a signature value, S(M), attached to the end of the initial message value M. At the end, the concatenation of the message and signature is sent over an insecure channel.

In the thesis study, the message M is that named unsecured data in the screen shown in Fig. 5.3

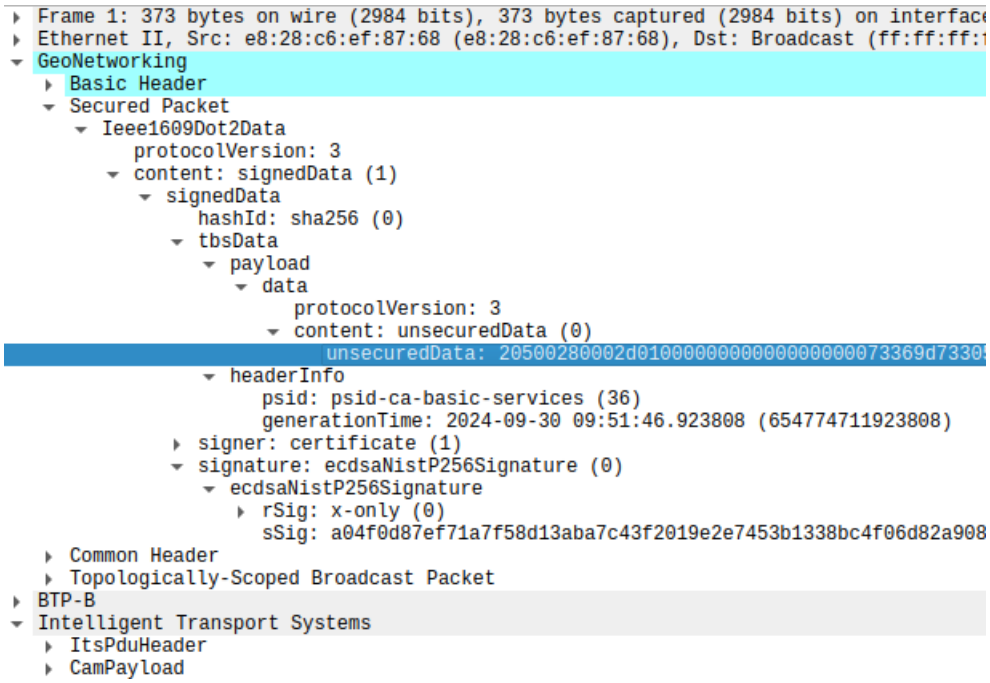


Figure 5.3: unsecured data section into tbsData payload

This is a Wireshark screen of a traditional CAM sent over the network interface. Here is possible to understand the structure of the message, and, in particular, the highlighted unsecuredData section, present in the secured header of GeoNetWorking layer. The GeoNetworking protocol is a network layer protocol designed for packet routing in ad hoc networks, utilizing geographical positions for packet transmission. It enables communication between individual ITS stations and facilitates the distribution of packets within specific geographic areas. GeoNetworking operates over various ITS access technologies, including short-range wireless technologies like ITS-G5 and infrared [26].

The unsecuredData section comprises the sequences of bytes of:

- Common header: it includes common information across message types;
- Topologically-Scoped Broadcast Packet: TSB enables the re-broadcasting of a data packet from a source node to all nodes within a specified number of

hops, effectively reaching all nodes in an n-hop neighborhood [27].

- Basic Transport Protocol - B: BTP offers a connectionless, end-to-end transport service within the ITS ad hoc network. BTP is designed to be lightweight, having a compact 4-byte header, and requiring minimal processing [28].
- Intelligent Transport Systems:
  - ItsPduHeader is a header that contains the protocol version, the message type, and the ITS-Station ID [29];
  - CamPayload is the payload of CAM, containing the basic container, high-frequency container, and low-frequency container [29];

In this case of study, the unsecuredData section is 81 bytes long, from byte 25 to byte 105. It is needed to extract this sequence of bytes and put it in input to the *sha256_calculate* function to obtain the hash of the unsecured data.

```
char unsecuredData[lenUnsecuredData];
int j = 0;
for (int i = 25; i < 106; i++)
    unsecuredData[j++] = buf[i];
int hashLen = 32;
char hashUnsec[32];
sha256_calculate(hashUnsec, unsecuredData, lenUnsecuredData);
```

After obtaining the hash of the message to be signed, the other required value is the entity's private key. This latter is returned by a *search_private_Dilithium_key* function, similar to that used in *ItsCertGen*. The input parameter is the path of the AT certificate and the output is its corresponding secret key.

OQS_SIG_dilithium_2_sign is the function used for the signature computation, that is the *liboqs* API already used in *ItsCertGen* for the signature of the certificate.

```
OQS_STATUS check = OQS_SIG_dilithium_2_sign(signatureMessage,
↪  &(lenSignature), hashUnsec, hashLen, secretKey);
if(check!=OQS_SUCCESS)
   fprintf(stderr, "Error signing message\n");
```

The parameters are the buffer which will contain the signature, its length (2420 bytes), the hash of the unsecured data section computed before and its size, and the secret key of the entity AT. The output flag indicates if the signature operation ends with success or not.

Injection of CAM

The fsmsggen tool uses the PCAP library to read and inject messages on the Ethernet interface. The Packet Capture library offers a high-level interface for packet capture systems, allowing access to all packets on the network, including those intended for other hosts. Additionally, it supports saving captured packets to a "savefile" and reading packets from such a file.

The first step is to set the length of the packet to send. The CAMs are sent by every ITS station every 100 ms, and once each second the message contains the whole certificate. In the other CAMs, the digest of the certificate is present. Thus, it is useful to distinguish the case in which the certificate or the digest is in the structure of the message. The easier implementation to take note of the number of messages is a counter that is incremented every injection of a message. Every ten messages the packet length changes to support the presence of a certificate. Thus, when the remainder of the division between the counter and 10 is 0, the packet length is set to a greater dimension, as is shown in the following piece of code.

```
if (flag_PQC == 1)
{
   if (round_send % 10 == 0)
      ph.caplen = ph.len = (uint32_t)msgLenWithCert_dilithium;
   else
      ph.caplen = ph.len = (uint32_t)msgLenWithDigest_dilithium;
}
```

The *round_send* counter is not only useful for setting the packet length but also for filling the message with digest/certificate and signature.

In the case of a message containing a digest, the code is shown here:

```
// computation digest of certificate
sha256_calculate(h_cert, myCert.buf, myCert.size);
memcpy(lsb, &h_cert[24], 8);

int index = 117;
// type of signer: digest
buf[index++] = 0x80;
// filling digest field
for (int i = 0; i < 8; i++)
     buf[index++] = lsb[i];
// type of signature: Dilithium
buf[index++] = 0x85;
// filling signature field
for (int i = 0; i < 2420; i++)
    buf[index++] = signatureMessage[i];
```

The performed steps are:

- computation of digest of the certificate through the *sha256_calculate*. Here the parameters are the buffer for storing the computed hash of 32 bytes, the sequence of bytes of the certificate, and its size.
- storing only the 8 least significant bytes of the hash.
- setting the byte that indicates "digest" as the signer type. In this case, the value is 0x80.
- filling the message buffer with the content of the digest.
- setting the byte that indicates "Dilithium" as the signature type. In this case, the value is 0x85.

- filling the message buffer with the content of the signature.

In the case of a message containing a certificate, the code is shown below:

```
int index = 120;
// filling certificate field
for (int i = 0; i < myCert.size; i++)
    buf[index++] = myCert.buf[i];
// type of signature: Dilithium
buf[index++] = 0x85;
// filling signature field
for (int i = 0; i < 2420; i++)
    buf[index++] = signatureMessage[i];
```

The performed steps are:

- filling the message buffer with the content of the certificate.
- setting the byte that indicates "Dilithium" as the signature type. In this case, the value is 0x85.
- filling the message buffer with the content of the signature.

For the injection of the message, the PCAP API *pcap_inject* is used, as present in the source code. The only addition is the management of the *round_send* counter. To prevent the counter from becoming too large, it is reset to zero every 100 increments.

```
int result = pcap_inject(h->device, data, ph->len);
round_send++;
if(round_send==100) round_send = 0;
if (result == -1)
    fprintf(stderr, "Error injecting packet: %s\n",
    ↪ pcap_geterr(h->device));
```

5.6 Observations

This thesis study has been conducted mainly using two open-source tools: *ItsCert-Gen* and *fsmsggen*. The first one was made for testing purposes, so it does not make any kind of validation. It just generates certificates. The second one is a demonstration tool to generate secured V2X messages. Thus, the thesis work may lack some controls or management of more complex contexts, involving more entities.

The two tools are been used to have a minimal environment that permits the generation of CAM messages. The aim is not to create and test a real context in which Post Quantum Cryptography is integrated into the V2X standard, but just to try to understand how CAM structure could be extended to support these new types of algorithms.

Comparing the key and signature sizes of the elliptic curve and Dilithium algorithms, the expected results should have highlighted a **significant size difference** between standard and non-standard messages. The confirmation is arrived at the end of implementation.

To demonstrate this analysis, two screens of the capture with Wireshark are attached here:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	0.0.0.00:00:00:00:0…	Broadcast	CAM	373	CAM
2	0.098983003	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
3	0.199087407	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
4	0.298952210	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
5	0.399826314	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
6	0.498674817	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
7	0.599089420	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
8	0.699069224	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
9	0.798584727	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
10	0.899381231	0.0.0.00:00:00:00:0…	Broadcast	CAM	192	CAM
11	0.998622634	0.0.0.00:00:00:00:0…	Broadcast	CAM	373	CAM

Figure 5.4: Standard CAMs captured

In fig.5.4 the size of messages with certificates is 373 bytes and the size of messages with digest is 192 Bytes.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	0.0.0.00:00:00:00:0…	Broadcast	CAM	6368	CAM
2	0.098748489	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
3	0.198729779	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
4	0.299317368	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
5	0.398584157	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
6	0.498041147	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
7	0.599051836	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
8	0.698730325	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
9	0.798642414	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
10	0.899464304	0.0.0.00:00:00:00:0…	Broadcast	CAM	2547	CAM
11	0.999278493	0.0.0.00:00:00:00:0…	Broadcast	CAM	6368	CAM

Figure 5.5: Non-standard CAMs captured

In fig.5.5 the size of messages with certificates is 6368 bytes and the size of messages with digest is 2547 Bytes.

The key observation is that the size of a non-standard message with a certificate is approximately 17 times larger than that of a standard message. In comparison, the size of a non-standard message with a digest is about 13 times larger than the standard one. The increase in message size may require configuring the Maximum Transfer Unit (MTU) to a value higher than the default 1500 bytes.

A comparative analysis has been conducted from the point of view of the **time of signing the messages** on a PC with Intel(R) Core(TM) i7-8565U CPU. A small function has been integrated into the Dilithium signature phase to compare the time taken by the *OQS_SIG_dilithium_2_sign* with the EcdsaNist256, Falcon512, and Sphincs128s signature methods that simulate the computation of the signature starting from the same hash of unsecured data used by the Dilithium function.

The tool generated an arbitrary number of messages to obtain a large sample of measurements. For each message, the signature time was recorded using different algorithms, with 2000 measurements selected as the target sample size. The line graph in Fig.5.6, displayed on a logarithmic scale, illustrates the distribution of signature times across this extensive sampling window.

The orange line represents the standard solution, Ecdsa. The purple line represents the implemented choice, Dilithium. The green and sky-blue lines are Falcon and Sphincs, the other two PQC Nist finalist algorithms.

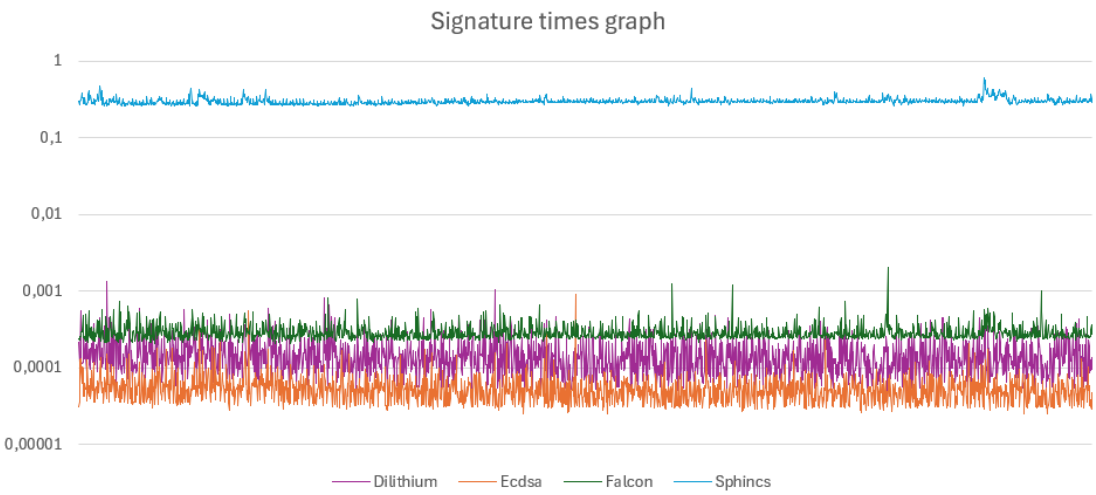


Figure 5.6: Signature times graph

The graph demonstrates what explained in Section 3.8.3. Dilithium is faster than the PQC competitor algorithms but is slightly slower than the Ecdsa solution. For each algorithms the averages of these signature times have been calculated and are presented here:

- Ecdsa: 0.000110 s = 0.110 ms;
- Dilithium: 0.000302 s = 0.302 ms;
- Falcon: 0.000584 s = 0.584 ms;
- Shpincs: 0.599014 s = 599.014 ms.

On average, Dilithium is three times slower than Ecdsa, but it takes about half the time compared to Falcon. Sphincs is significantly slower and, from this perspective, does not represent a performant algorithm.

To accurately measure the **time for verifying the messages**, a standalone program was selected, as opposed to the *fsmsggen* tool, where the verification phase is just performed during certificate installation. In contrast, this smaller program was used to test the average verification times for ECDSA, Dilithium, Falcon, and Sphincs, using a larger sampling window of 2000 measurements per algorithm. The results of averages verification time (conducted with the same PC used for testing the signature times) are presented below:

- Ecdsa: 0.000281 s = 0.281 ms;
- Dilithium: 0.000096 s = 0.096 ms;
- Falcon: 0.000089 s = 0.089 ms;
- Sphincs: 0.000716 s = 0.716 ms.

These values shows Dilithium has lower verification time than Ecdsa, and almost the same with Falcon.

Chapter 6

Conclusions

The thesis work has been completed with the generation of new certificates for V2X entities supporting PQC keys and signatures, the signature phase of CAM messages using a PQC algorithm, and the transmission over a network interface.

The effort of the study started with a first analysis of a proprietary framework. However, once realized that open-source tools could offer greater flexibility in achieving the thesis objectives, the focus shifted to open-source solutions. The chosen tools for the project implementation needed some changes and extensions to integrate new algorithms, which did not follow the actual V2X standards. An open-source library for prototyping and experimenting with quantum-resistant cryptography has been integrated during the generation phase of new certificates and keys, and for the signature phase of CAM messages.

This thesis has aimed to identify a point of convergence between V2X and PQC, to counter the emerging threat of quantum attacks. The final results, along with a comparative analysis, reveal critical challenges or potential starting points for large-scale implementation.

The expected result, that the study confirmed, is the significant difference in size between the standard and the implemented non-standard CAMs. In a real-world context, having vehicles with larger storage in OBU, for certificates and keys, could be necessary. Furthermore, V2X is a real-time system, in which messages are exchanged and processed in the order of milliseconds among vehicles, infrastructures, and road users. The higher load of the communication channel

could avoid the signaling of accidents, failing the main V2X purpose of improving road safety.

Among the PQC algorithms, Dilithium is not the one with the smallest key and signature sizes, but the choice of another PQC method would lead to a significant enlarging of the CAM dimension as well. On the other side, Dilithium offers better times for signing than the other candidates, but a bit higher than Ecdsa. From the point of view of verification time, V2X can take advantage of the Dilithium choice. Thus, the PQC choice in this thesis demonstrates that the new implementation does not introduce significant delays (sender side). It improves the verification time, which is a crucial step for quickly authenticating and verifying entities, and it has a comparable verifying time with the standard solution. If message size expansion is efficiently managed, Dilithium will provide a good balance for securing V2X environments against quantum attacks, offering robust protection without significantly impacting performance

6.1 Future developments

The thesis study has brought significant results. There is still the possibility of introducing improvements and technical innovations. The factors and aspects to consider for potential future development may include:

- **Addressing the larger size of non-standard CAM messages**. Integrating PQC into a V2X framework results in increased message sizes, potentially leading to delays and traffic congestion. Future developments should focus on identifying mechanisms to mitigate these issues, ensuring a gradual transition to a quantum-resistant system without compromising performance. Operations such as fragmentation, segmentation, caching, and compression could be evaluated in a V2X scenario to understand if their integration leads to an improvement.

- **Emulating a real-world scenario**. The study did not focus on simulating a real-world scenario with multiple sub-CAs and end-entities. Instead, it served as a feasibility study to test and extend the protocols that V2X entities might follow. As a result, only a minimal set of entities was involved, specifically one

RCA, one AA, and one AT. Once the implementation and modified structures are validated and tested, the framework can be expanded to support additional AA and AT certificates that comply with the PQC version.

- **Analyzing how the size differs with different PQC algorithms**. In this study, the Dilithium algorithm was employed for both the certificate generation and for signing messages. The impact of the increased signature and key sizes on message and certificate dimensions was analyzed. An additional investigation could explore the use of different algorithms, such as Falcon and Sphincs, by modifying the APIs in the framework. Although the resulting certificates and CAMs would be larger compared to standard ones, it would be valuable to assess the size difference with Dilithium and evaluate the impact on communication performance.

- **Implementing the reception**. In the thesis work, just the CAM generation is managed. The open-source tool used also supports message reception. Thus, future development will extend the framework to process new non-standard CAMs that arrive over the network interface.

- **Studying the network traffic bandwidth**. After the support for receiving new non-standard V2X messages has been designed, it could be valuable to assess the impact of the larger messages on the network performance. In particular, the aspects to analyze could be latency, throughput, and reliability. This study could reveal crucial to understanding the robustness of the V2X network when non-standard messages are exchanged, mainly when there are many entities and traffic congestion is possible.

- **PQC extensions for different types of messages**. The focus of this thesis has been on the CAM (Cooperative Awareness Message), which is the most common message type in V2X scenarios. Future research could expand this work by extending the framework to include other message types, such as DENM (Decentralized Environmental Notification Message).

- **PQC for encryption and decryption of CAM**. In the case of the study, the certificate can have two fields for keys, one for the encryption key and one for the verification key. Just the verification key field has been extended to

support PQC, obtaining a hybrid certificate. A possible development is to change the type of the encryption keys with a proper PQC algorithm.

The whole code was designed to support both the standard version and the custom version described in this thesis. In the same way, it can be easily extended to incorporate new versions while keeping the existing implementations intact.

Appendix A

Code snippets

A.1 Certificate profile

The Dilithium version of the AT certificate profile is displayed.

```
<?xml version="1.0"?>
<EtsiTs103097Certificate>
  <version>3</version>
  <type>
    <explicit/>
  </type>
  <issuer name="CERT_IUT_A_AA_Dilithium">
    <sha256AndDigest/>
  </issuer>
  <toBeSigned>
    <id>
      <none/>
    </id>
    <cracaId>00 00 00</cracaId>
    <crlSeries>0</crlSeries>
    <validityPeriod>
      <start><!--2022-01-01-->568080000</start>
```

```
18      <duration>
19        <hours><!--2023-01-01-->8760</hours>
20      </duration>
21    </validityPeriod>
22    <assuranceLevel>60</assuranceLevel>
23    <!--level=3 confidence=0-->
24    <appPermissions>
25      <PsidSsp>
26        <!--CAM-->
27        <psid>36</psid>
28        <ssp>
29          <bitmapSsp>01 FF FC</bitmapSsp>
30        </ssp>
31      </PsidSsp>
32      <PsidSsp>
33        <!--DENM-->
34        <psid>37</psid>
35        <ssp>
36          <bitmapSsp>01 FF FF FF</bitmapSsp>
37        </ssp>
38      </PsidSsp>
39      <PsidSsp>
40        <!--GN-MGMT-->
41        <psid>141</psid>
42      </PsidSsp>
43    </appPermissions>
44    <encryptionKey>
45      <supportedSymmAlg>
46        <aes128Ccm/>
47      </supportedSymmAlg>
48      <publicKey>
49        <eciesNistP256>
```

```
          <compressed-y-0/>
        </eciesNistP256>
      </publicKey>
    </encryptionKey>
     <verifyKeyIndicator>
      <verificationKey>
        <dilithium2>
          <publicKey/>
        </dilithium2>
      </verificationKey>
    </verifyKeyIndicator>
  </toBeSigned>
  <signature>
    <dilithiumSignature>
      <signatureValue/>
    </dilithiumSignature>
  </signature>
</EtsiTs103097Certificate>
```

A.2 Dilithium signature in asn file and source C code

ASN.1 module file *Iee1609Dot2BaseTypes.asn* has been extended to support Dilithium Signature. The segment of asn file modified and the generated source C code are shown below.

Asn file:

```
Signature ::= CHOICE {
    ecdsaNistP256Signature        EcdsaP256Signature,
    ecdsaBrainpoolP256r1Signature EcdsaP256Signature,
    ...,
```

```
    ecdsaBrainpoolP384r1Signature EcdsaP384Signature,
    ecdsaNistP384Signature        EcdsaP384Signature,
    sm2Signature                  EcsigP256Signature,
    dilithiumsignature            DilithiumSignature
}

/* added Dilithium Sig struct */
DilithiumSignature ::= SEQUENCE {
    signature   OCTET STRING(SIZE(2420))
}
```

Source C code:

```
typedef struct Signature {
  Signature_PR present;
    union Signature_u {
        EcdsaP256Signature_t         ecdsaNistP256Signature;
        EcdsaP256Signature_t
         ↪ ecdsaBrainpoolP256r1Signature;
        EcdsaP384Signature_t
         ↪ ecdsaBrainpoolP384r1Signature;
        EcdsaP384Signature_t         ecdsaNistP384Signature;
        EcsigP256Signature_t         sm2Signature;
        DilithiumSignature_t         dilithiumsignature;
    } choice;

 /* Context for parsing across buffer boundaries */
    asn_struct_ctx_t _asn_ctx;
} Signature_t;

/* DilithiumSignature */
typedef struct DilithiumSignature {
        OCTET_STRING_t         signature;
```

```
19
20          /* Context for parsing across buffer boundaries */
21          asn_struct_ctx_t _asn_ctx;
22  } DilithiumSignature_t;
```

A.3 Dilithium key in asn file and source C code

ASN.1 module file *Iee1609Dot2BaseTypes.asn* has been extended to support Dilithium Public Key. The segment of asn file modified and the generated source C code are shown below.

Asn file:

```
1  Signature ::= CHOICE {
2      ecdsaNistP256Signature        EcdsaP256Signature,
3      ecdsaBrainpoolP256r1Signature EcdsaP256Signature,
4      ...,
5      ecdsaBrainpoolP384r1Signature EcdsaP384Signature,
6      ecdsaNistP384Signature        EcdsaP384Signature,
7      sm2Signature                  EcsigP256Signature,
8      dilithiumsignature            DilithiumSignature
9  }
10
11 /* added Dilithium Sig struct */
12 DilithiumSignature ::= SEQUENCE {
13     signature   OCTET STRING(SIZE(2420))
14 }
```

Source C code:

```
1  typedef struct PublicVerificationKey {
2    PublicVerificationKey_PR present;
3      union PublicVerificationKey_u {
```

```
      EccP256CurvePoint_t          ecdsaNistP256;
      EccP256CurvePoint_t          ecdsaBrainpoolP256r1;
      EccP384CurvePoint_t          ecdsaBrainpoolP384r1;
      EccP384CurvePoint_t          ecdsaNistP384;
      EccP256CurvePoint_t          ecsigSm2;
      DilithiumKey_t          dilithiumKey;
    } choice;

    /* Context for parsing across buffer boundaries */
    asn_struct_ctx_t _asn_ctx;
} PublicVerificationKey_t;

/* DilithiumKey */
typedef struct DilithiumKey {
    OCTET_STRING_t          publicKey;

    /* Context for parsing across buffer boundaries */
    asn_struct_ctx_t _asn_ctx;
} DilithiumKey_t;
```

A.4 Dilithium key pair generation

The function used to generate the Dilithium2 key pair and the assignment of them to the respective files is displayed below:

```
static int fill_Dilithium_keyPair(DilithiumKey_t *dilithium, int
 ↪  algorithmVersion, char *keyPath)
{
        char *private_key;

        dilithium->publicKey.size =
         ↪  OQS_SIG_dilithium_2_length_public_key;
```

```
        dilithium->publicKey.buf =
            malloc(dilithium->publicKey.size);
        private_key =
        ↪  malloc(OQS_SIG_dilithium_2_length_secret_key);

        OQS_STATUS stat =
        ↪  OQS_SIG_dilithium_2_keypair(dilithium->publicKey.buf,
        ↪  private_key);
        if (stat != OQS_SUCCESS)
        {
                printf("\nError generating Dilithium key pair\n");
                return 0;
        }

        /* load Private Key */
        FILE *f = fopen(keyPath, "wb");
        if (f == NULL)
        {
                perror(keyPath);
                return -1;
        }
        fwrite(private_key, 1,
        ↪  OQS_SIG_dilithium_2_length_secret_key, f);
        fclose(f);

        /* load Public Key */
        void *key = NULL;
        keyPath = strcat(keyPath, EXT_PUB);
        f = fopen(keyPath, "wb");
        if (f == NULL)
        {
                perror(keyPath);
```

```
                return -1;
        }
        fwrite(dilithium->publicKey.buf, 1,
        ↪  OQS_SIG_dilithium_2_length_public_key, f);
        fclose(f);

        free(private_key);
        return 6;
}
```

A.5 Dilithium digital signature

The whole process to generate the Dilithium digital signature of the certificate is shown below:

```
static asn_enc_rval_t Signature_oer_encoder(
                const asn_TYPE_descriptor_t *td,
                const asn_oer_constraints_t *constraints,
                const void *sptr,
                asn_app_consume_bytes_f *cb, void *app_key)
{
  Signature_t *s = (Signature_t *)sptr;

  if (is_CurvePoint_empty(&s->choice.ecdsaNistP256Signature.rSig)
  ↪  && _tbsHashType < 6)
  {
    // Standard version
    ...
  }else{
  // added Dilithium Signature
  s->present = Signature_PR_dilithiumsignature;
  // look for signer private key
```

```
17  ecc_hash_id hashId = _pk_type_to_hashid[s->present];
18  const char *sName = _signerName;
19
20  if (sName == NULL && _cert->issuer.present ==
    ↪  IssuerIdentifier_PR_self)
21        sName = _certName;
22
23   uint8_t *secret_key = search_private_Dilithium_key(sName, 2);
24   if (secret_key == NULL)
25   {
26        ASN__ENCODE_FAILED;
27   }
28
29   char h[48];
30   int hl = 0;
31   // calculate joint hash (toBeSigned Hash computed in
     ↪  ToBeSignedCertificate_oer_encoder + signerHash)
32   memcpy(_tbsHash + _tbsHashLength, _signerHash,
     ↪  _signerHashLength);
33   int lenJointHash = _tbsHashLength + _signerHashLength;
34
35   switch (hashId)
36   {
37   case sha_256:
38        sha256_calculate(h, _tbsHash, _tbsHashLength +
      ↪  _signerHashLength);
39        hl = sha256_hash_size;
40        break;
41   case sha_384:
42        sha384_calculate(h, _tbsHash, _tbsHashLength +
      ↪  _signerHashLength);
43        hl = sha384_hash_size;
```

```
44          break;
45      case sm3:
46          sm3_calculate(h, _tbsHash, _tbsHashLength +
            ↪  _signerHashLength);
47          hl = sm3_hash_size;
48          break;
49      }
50
51      if (_debug){
52          char hex[48 * 3 + 1];
53          *_bin2hex(hex, sizeof(hex), _tbsHash, _tbsHashLength) = 0;
54          fprintf(stderr, "DEBUG: ToBeSignedHash[%d]=%s\n",
            ↪  _tbsHashLength, hex);
55          *_bin2hex(hex, sizeof(hex), _signerHash,
            ↪  _signerHashLength) = 0;
56          fprintf(stderr, "DEBUG: SignerHash[%d]=%s\n",
            ↪  _signerHashLength, hex);
57          *_bin2hex(hex, sizeof(hex), h, hl) = 0;
58          fprintf(stderr, "DEBUG: JoinedHash[%d]=%s\n", hl, hex);
59      }
60
61      s->choice.dilithiumsignature.signature.buf =
        ↪  malloc(OQS_SIG_dilithium_2_length_signature);
62
63      s->choice.dilithiumsignature.signature.size =
        ↪  OQS_SIG_dilithium_2_length_signature;
64
65      OQS_STATUS check = OQS_SIG_dilithium_2_sign(
66              s->choice.dilithiumsignature.signature.buf,
67              &(s->choice.dilithiumsignature.signature.size),
68              h, hl, secret_key);
69
```

```
70    if (check != 0){
71       printf("\n Error: during gen Signature phase\n");
72       return;
73     }
74    free(secret_key);
75   }
76   return asn_OP_CHOICE.oer_encoder(td, constraints, sptr, cb,
      ↪  app_key);
77 }
```

A.5.1 Search private Dilithium key

The code to obtain the secret key of the signer of a certificate starting from the path of the ".vkey" file is displayed below:

```
1  static void *search_private_Dilithium_key(const char *sName, int
    ↪  alg)
2  {
3          char *secretKey =
           ↪  malloc(OQS_SIG_dilithium_2_length_secret_key);
4          char *path = cvstrdup(_keyPath, "/", sName, EXT_VKEY,
           ↪  NULL);
5          FILE *f = fopen(path, "rb");
6          if (f == NULL)
7          {
8                  fprintf(stderr, "Error: impossible to open the
                   ↪  file%s\n", path);
9                  free(secretKey);
10                 free(path);
11                 return NULL;
12         }
13         // read content of file
```

```
        size_t bytesRead = fread(secretKey, 1,
        ↪  OQS_SIG_dilithium_2_length_secret_key, f);
        if (bytesRead != OQS_SIG_dilithium_2_length_secret_key)
        {
                fprintf(stderr, "Error: wrong len of bytes read
                ↪  %s\n", path);
                free(secretKey);
                secretKey = NULL;
        }
        fclose(f);
        free(path);
        return secretKey;
}
```

A.6 Ordering the list of processed certificates

This function permits to establish the rules to follow for having a precise order of which certificate must be loaded before the other ones. In the studied case of thesis work the order is RCA-AA-AT, as detailed below:

```
// Comparison function to sort files
int compare_files(const void *a, const void *b)
{
    const char *file_a = *(const char **)a;
    const char *file_b = *(const char **)b;

    // Define the sorting logic based on specific names
    if (strstr(file_a, "RCA") && !strstr(file_b, "RCA"))
        return -1;
    if (strstr(file_b, "RCA") && !strstr(file_a, "RCA"))
        return 1;

```

```
    if (strstr(file_a, "AA") && !strstr(file_b, "AA"))
        return -1;
    if (strstr(file_b, "AA") && !strstr(file_a, "AA"))
        return 1;
    if (strstr(file_b, "AA") && strstr(file_a, "AA"))
        return 1;

    if (strstr(file_a, "AT") && !strstr(file_b, "AT"))
        return -1;
    if (strstr(file_b, "AT") && !strstr(file_a, "AT"))
        return 1;
    if (strstr(file_b, "AT") && strstr(file_a, "AT"))
        return 1;

    // If neither RCA, AA, nor AT, sort alphabetically
    return strcmp(file_a, file_b);
}
```

A.7 Set signer of certificate

A straightforward function that assigns the input path as the signer for the lowest-level entity within the PKI hierarchy, specifically designed for a context with one RCA, one AA, and one AT. However, this function should be extended for scenarios involving multiple AAs or ATs to accommodate more complex PKI structures. An idea could be using hash tables.

```
char *set_signer(char *path)
{
    char *entity;
    if (strstr(path, "RCA") != NULL)
    {
        entity = "RCA";
```

```
7           signerAA = malloc(strlen(path) + 1);
8           strcpy(signerAA, path);
9       }
10      else if (strstr(path, "AA") != NULL)
11      {
12          entity = "AA";
13          signerAT = malloc(strlen(path) + 1);
14          strcpy(signerAT, path);
15      }
16      else
17      {
18          if (strstr(path, "AT") != NULL){
19                  entity = "AT";
20                  char* pathMyCert = malloc(strlen(path)+1);
21                  memcpy(pathMyCert,path,strlen(path)+1);
22          }
23      }
24      return entity;
25  }
```

A.8 Certificate structure for standard and non-standard version

Certificate structure for RCA, AA, AT that supports the ETSI 103 097 standard and the new Dilithium version for verification Key and signature.

```
1  typedef struct DilithiumKey {
2          uint8_t*          publicKey;
3  } DilithiumKey_t;
4  typedef struct DilithiumSignature {
5          uint8_t*          signature;
```

```
} DilithiumSignature_t;
typedef struct{
        uint16_t psid;
        struct
        {
        uint8_t bitmapSspLength;
                uint8_t *bitmapSsp;
        } ssp;
} PsidSsp;
typedef struct{
    uint32_t initialized;
        uint8_t *sspValue;
        uint8_t *sspBitmask;
        size_t sspValueLength;
        size_t sspBitmaskLength;
} BitmapSspRange;

typedef struct{
        uint16_t psid;
        BitmapSspRange sspRange;
} PsidSspRange;
typedef struct{
        PsidSspRange *psidSspRanges;
        size_t numRanges;
} ExplicitPermissions;
typedef struct{
        ExplicitPermissions explicitPermissions;
} SubjectPermissions;
typedef struct{
        SubjectPermissions subjectPermissions;
        uint8_t minChainLength;
        uint8_t chainLengthRange;
```

```
38          uint8_t eeType;
39  } PsidGroupPermissions;
40  typedef struct{
41          PsidGroupPermissions *psidGroupPermissions;
42          size_t numGroupPermissions;
43  } CertIssuePermissions;
44  typedef struct{
45          FSSymmAlg supportedSymmAlg;
46          FSPublicKey publicKey;
47  } EncryptionKey;
48  typedef struct{
49      union{
50          FSPublicKey FSverificationKey;
51          DilithiumKey_t DILverificationKey;
52      }val;
53  } VerifyKeyIndicator;
54  typedef struct{
55          struct {
56          uint8_t idType;
57          union{
58              struct{
59                  uint8_t len;
60                  uint8_t *val;
61              }name
62          }id;
63      }id_Value;
64          uint8_t cracaId[7]; // "00 00 00"
65          uint8_t crlSeries[2];
66          struct
67          {
68                  uint8_t start[4];
69          uint8_t durationType;
```

```
70                  struct
71                  {
72                          uint8_t hours[2];
73                  } duration;
74          } validityPeriod;
75          uint8_t assuranceLevel;
76          PsidSsp *appPermissions;
77          size_t numAppPermissions;
78          union{
79          CertIssuePermissions certIssuePermissions;
80      };
81          EncryptionKey encryptionKey;
82          VerifyKeyIndicator verifyKeyIndicator;
83  } ToBeSigned;
84  typedef struct{
85          uint8_t issuerType;
86      union{
87          uint8_t *Digest;
88      }sha256
89  } Issuer;
90  typedef struct{
91          uint8_t version;
92          uint8_t type;
93          Issuer issuer;
94          ToBeSigned toBeSigned;
95      union{
96          FSSignature FSsignature;
97          DilithiumSignature_t DILsignature;
98      }sig;
99  } EtsiExtendedCertificate;
```

A.9 Verify Dilithium Signature of certificates

The function used to verify the signature attached to the analyzed certificates of RCA, AA, and AT.

```
void verifySignature(uint8_t *data, size_t cert_length, uint8_t
 ↪  *_toBeSigned, int lenTBS, uint8_t *sig, char *entity)
{
    if (strcmp(entity, "RCA") == 0)
    {
        // computation digest of RCA
        data[cert_length - 1] = 0x80;
        sha256_calculate(_signerHashBuf, data, cert_length);
        _signerHashRCA = &_signerHashBuf[0];
        _signerHashLength = sha256_hash_size;
    }
    else if (strcmp(entity, "AA") == 0)
    {
        // TO BE SIGNED HASH
        _tbsHashLength = 32;
        sha256_calculate(_tbsHash, (const char *)_toBeSigned,
         ↪  lenTBS);

        char hash[48];
        int hashlen = 0;

// calculate JOINT HASH (toBeSigned Hash computed in
 ↪  ToBeSignedCertificate_oer_encoder + signerHash)
        memcpy(_tbsHash + _tbsHashLength, _signerHashRCA,
         ↪  _signerHashLength);
        sha256_calculate(hash, _tbsHash, _tbsHashLength +
         ↪  _signerHashLength);
        hashlen = sha256_hash_size;
```

```
24
25          // computation digest of AA
26          data[cert_length - 1] = 0x80;
27          sha256_calculate(_signerHashBuf, data, cert_length);
28          _signerHashAA = &_signerHashBuf[0];
29          _signerHashLength = sha256_hash_size;
30
31          char *pubKeySigner =
            ↪  search_public_Dilithium_key(signerAA);
32          // verification function
33          OQS_STATUS result = OQS_SIG_dilithium_2_verify(hash,
            ↪  hashlen, sig, OQS_SIG_dilithium_2_length_signature,
            ↪  pubKeySigner);
34          if (result != OQS_SUCCESS){
35                  fprintf(stderr, "\nERROR:
                    ↪  OQS_SIG_dilithium_2_verify failed!\n");
36          }
37      }
38      else
39      {
40          // TO BE SIGNED HASH
41          _tbsHashLength = 32;
42          sha256_calculate(_tbsHash, (const char *)_toBeSigned,
            ↪  lenTBS);
43
44          char hash[48];
45          int hashlen = 0;
46          // calculate joint hash (toBeSigned Hash computed in
            ↪  ToBeSignedCertificate_oer_encoder + signerHash)
47          memcpy(_tbsHash + _tbsHashLength, _signerHashAA,
            ↪  _signerHashLength);
48          sha256_calculate(hash, _tbsHash, _tbsHashLength +
            ↪  _signerHashLength);
```

```
49          hashlen = sha256_hash_size;
50
51          char *pubKeySigner =
            ↪  search_public_Dilithium_key(signerAT);
52
53          // verification function
54          OQS_STATUS result = OQS_SIG_dilithium_2_verify(hash,
            ↪  hashlen, sig, OQS_SIG_dilithium_2_length_signature,
            ↪  pubKeySigner);
55          if (result != OQS_SUCCESS){
56                  fprintf(stderr, "\nERROR:
                    ↪  OQS_SIG_dilithium_2_verify failed!\n");
57          }
58      }
59  }
```

A.10 Signing and Injection of CAM

The function responsible for the signing and injection phase of the new CAMs.

```
1  void MsgGenApp_Send(FitSec *e, MsgGenApp *a)
2  {
3      struct pcap_pkthdr ph;              FSMessageInfo m = {0};
4      gettimeofday(&ph.ts, NULL);          ph.ts.tv_sec += _tdelta;
5      m.message = (char *)&buf[SHIFT_SEC];
6      m.messageSize = sizeof(buf) - SHIFT_SEC;
7      m.sign.signerType = FS_SI_AUTO;     m.position = position;
8      m.generationTime = timeval2itstime64(&ph.ts);
9      if (_changePseudonym)
10         FitSec_ChangeId(e, FITSEC_AID_ANY);
11     size_t len = a->fill(a, e, &m);
12     if (len > 0)
```

```
13      {
14          if (!_gn_src && m.sign.cert)
15          {
16              FSHashedId8 id = FSCertificate_Digest(m.sign.cert);
17              memcpy(buf + 6, &id, 6);
18          }
19          if (m.payloadType == FS_PAYLOAD_UNSECURED)
20              buf[SHIFT_GN] = 0x11;
21          else  buf[SHIFT_GN] = 0x12;
22
23          if (flag_PQC == 1)
24          {
25              if (round_send % 10 == 0)
26                  ph.caplen = ph.len =
                    ↪  (uint32_t)msgLenWithCert_dilithium;
27              else
28                  ph.caplen = ph.len =
                    ↪  (uint32_t)msgLenWithDigest_dilithium;
29          }
30          else
31              ph.caplen = ph.len = (uint32_t)(m.messageSize +
                ↪  SHIFT_SEC);
32
33          mclog_info(MAIN, "%s Msg sent app=%s gt=" cPrefixUint64 "u
            ↪  (%u bytes)\n",
34                     strlocaltime(ph.ts.tv_sec, ph.ts.tv_usec),
35                     a->appName, timeval2itstime64(&ph.ts), ph.len);
36          if (flag_PQC == 1)
37          {
38              char unsecuredData[lenUnsecuredData];
39              int j = 0;
40              for (int i = 25; i < 106; i++)
```

```
41              unsecuredData[j++] = buf[i];
42          int hashLen = 32;
43          char hashUnsec[32];
44          sha256_calculate(hashUnsec, unsecuredData,
            ↪  lenUnsecuredData);
45          char *secretKey =
            ↪  search_private_Dilithium_key(pathMyCert);
46          char *signatureMessage =
            ↪  malloc(OQS_SIG_dilithium_2_length_signature);
47          size_t lenSignature =
            ↪  OQS_SIG_dilithium_2_length_signature;
48          OQS_STATUS check =
            ↪  OQS_SIG_dilithium_2_sign(signatureMessage,
            ↪  &(lenSignature), hashUnsec, hashLen, secretKey);
49          if(check!=OQS_SUCCESS)
50              fprintf(stderr, "Error signing message\n");
51          unsigned char h_cert[32];
52          unsigned char lsb[8];
53          if (round_send % 10 != 0)
54          {
55              // computation digest of certificate
56              sha256_calculate(h_cert, myCert.buf, myCert.size);
57              memcpy(lsb, &h_cert[24], 8);
58              int index = 117;
59              buf[index++] = 0x80;  // type of signer: digest
60              // filling digest field
61              for (int i = 0; i < 8; i++)
62                  buf[index++] = lsb[i];
63              // type of signature: Dilithium
64              buf[index++] = 0x85;
65              // filling signature field
66              for (int i = 0; i < 2420; i++)
```

```
67                      buf[index++] = signatureMessage[i];
68              }
69              else
70              {
71                  int index = 120;
72                  // filling certificate field
73                  for (int i = 0; i < myCert.size; i++)
74                      buf[index++] = myCert.buf[i];
75                  // type of signature: Dilithium
76                  buf[index++] = 0x85;
77                  // filling signature field
78                  for (int i = 0; i < 2420; i++)
79                      buf[index++] = signatureMessage[i];
80              }
81          }
82          _packet_handler(&h, &ph, buf);
83      }
84      else
85          usleep(1000000 / _rate);
86  }
```

Bibliography

[1] Raphael Couturier Lama Sleema Hassan N. Noura. «Towards A Secure ITS: Overview, Challenges and Solutions». In: *Journal of Information Security and Applications* (2020). URL: https://www.sciencedirect.com/science/article/abs/pii/S2214212620307997 (cit. on pp. 4, 5, 13).

[2] Sandro Gagliano. «Analisi, Progettazione e Sviluppo di Cellular-V2X Software Utilities». MA thesis. Politecnico di Torino, 2019. URL: https://webthesis.biblio.polito.it/18658/ (cit. on pp. 5, 7–9, 16).

[3] Qualcomm. «ITS stack». In: *Qualcomm Technologies* (2019). URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/c-v2x_its_stack.pdf (cit. on pp. 6, 7).

[4] Autocrypt. «DSRC vs. C-V2X: A Detailed Comparison of the 2 Types of V2X Technologies». In: *Autocrypt* (2021). URL: https://autocrypt.io/dsrc-vs-c-v2x-a-detailed-comparison-of-the-2-types-of-v2x-technologies/ (cit. on pp. 9, 15, 19).

[5] Car2Car. «Introduction to Cellular V2X». In: *Qualcomm Technologies* (2019). URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/c-v2x_intro.pdf (cit. on pp. 9, 15).

[6] Car2Car. «The ITS Directive brings legal certainty and security to C-ITS». In: *Car 2 Car communication consortium* (2023). URL: https://www.car-2-car.org/news-events/news/detailview/the-its-directive-brings-legal-certainty-and-security-to-c-its-135 (cit. on pp. 9, 10).

[7] Car2Car. «C-ITS: Cooperative Intelligent Transport Systems and Services». In: *Car 2 Car communication consortium* (2023). URL: https://www.car-2-car.org/about-c-its (cit. on pp. 10, 11).

[8] Michele Rondinone (HMETC) / Alejandro Correa (UMH). *Definition of V2X message sets*. Tech. rep. Horizon 2020 ART-05-2016 – GA Nr 723390. Universidad Miguel Hernández (UMH), 2018. URL: https://www.transaid.eu/wp-content/uploads/2017/Deliverables/WP5/TransAID_D5.1_V2X-message-sets.pdf (cit. on p. 14).

[9] RF Wireless World. «Advantages of C-V2X in 5G | Disadvantages of C-V2X in 5G NR». In: *RF Wireless World* (). URL: https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-C-V2X-in-5G-NR.html (cit. on p. 16).

[10] Micron Technology. «C-v2x: A sixth sense for ADAS and autonomous vehicles». In: *Micron Technology* (2021). URL: https://www.micron.com/about/blog/applications/automotive/cv2x-a-sixth-sense-for-adas-and-autonomous-vehicles (cit. on p. 17).

[11] AutoTalks. «How to Add V2X to ADAS». In: *AutoTalks* (). URL: https://auto-talks.com/wp-content/uploads/2024/01/ADAS-whitepaper.pdf (cit. on p. 18).

[12] Auto21. «Da Euro NCAP un premio alla tecnologia Car-2-X Volkswagen». In: *Auto21* (2020). URL: https://www.auto21.net/2020/03/18/da-euro-ncap-premio-a-volkswagen-per-tecnologia-car2x-wifi/ (cit. on p. 19).

[13] *Intelligent Transport Systems (ITS); Security; ITS communications security architecture and security management*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2018. URL: https://www.etsi.org/deliver/etsi_ts/102900_102999/102940/01.03.01_60/ts_102940v010301p.pdf (cit. on pp. 20, 24, 27).

[14] *Intelligent Transport Systems (ITS); Communications Architecture*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2010. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/302665/01.01.01_60/en_302665v010101p.pdf (cit. on p. 23).

[15] *Intelligent Transport Systems (ITS); Security; Security Services and Architecture*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2010. URL: https://www.etsi.org/deliver/etsi_ts/102700_102799/102731/01.01.01_60/ts_102731v010101p.pdf (cit. on p. 24).

[16] *Intelligent Transport Systems (ITS); Security; Security header and certificate formats; Release 2*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2021. URL: https://www.etsi.org/deliver/etsi_ts/103000_103099/103097/02.01.01_60/ts_103097v020101p.pdf (cit. on pp. 31, 52, 67, 68).

[17] *IEEE Standard for Wireless Access in Vehicular Environments — Security Services for Applications and Management Messages*. IEEE, 2022 (cit. on pp. 32, 33, 35–37).

[18] *ECDSA: Elliptic Curve Signatures*. SoftUni (Software University). 2018. URL: https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages (cit. on p. 33).

[19] Bart Preneel Takahito Yoshizawa. «Post-Quantum Impacts on V2X Certificates – Already at The End of The Road». In: *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*. June 2023. URL: https://ieeexplore.ieee.org/document/10199793 (cit. on pp. 37, 38).

[20] Ian Smalley Josh Schneider. «What is quantum computing?» In: (2024). URL: https://www.ibm.com/topics/quantum-computing (cit. on p. 38).

[21] a Tim enterprise brand Telsy. «Nuove proposte di firme digitali post-quantum: la seconda competizione del NIST». In: (2023). URL: https://www.telsy.com/nuove-proposte-di-firme-digitali-post-quantum-la-seconda-competizione-del-nist/ (cit. on p. 39).

[22] Manohar Raavi Simeon Wuthier Pranav Chandramouli Yaroslav Balytskyi Xiaobo Zhou and Sang-Yoon Chang. *Security Comparisons and Performance Analyses of Post-Quantum Signature Algorithms*. Tech. rep. University of Colorado, Colorado Springs, USA Department of Computer Science - University of Colorado, Colorado Springs, USA Department of Physics and Energy Science, 2021. URL: https://link.springer.com/chapter/10.1007/978-3-030-78375-4_17 (cit. on pp. 40, 43–46).

[23] wolfSSL. «Dilithium vs. Falcon». In: (2024). URL: https://www.wolfssl.com/dilithium-vs-falcon/ (cit. on p. 46).

[24] Cohda Wireless. «Leading the technology revolution in cooperative intelligent transport systems». In: (). URL: https://www.cohdawireless.com/ (cit. on p. 49).

[25] Umesh Kumar Singh Kaul Meenakshi Choukse Dharmendra. «A Modified approach for implementation of an efficient padding scheme in a digital signature system». In: (). URL: https://www.researchgate.net/publication/45825703_A_Modified_approach_for_implementation_of_an_efficient_padding_scheme_in_a_digital_signature_system (cit. on p. 73).

[26] *Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 4: Geographical addressing and forwarding for point-to-point and point-to-multipoint communications; Sub-part 1: Media-Independent Functionality*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2019. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/3026360401/01.04.00_20/en_3026360401v010400a.pdf (cit. on p. 74).

[27] Long Van Le Andreas Festag Roberto Baldessari Wenhui Zhang. «V2X Communication and Intersection Safety». In: (). URL: https://www.researchgate.net/publication/227260711_V2X_Communication_and_Intersection_Safety (cit. on p. 75).

[28] *Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2019. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/3026360501/02.02.00_20/en_3026360501v020200a.pdf (cit. on p. 75).

[29] *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service*. 650 Route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE: ETSI, 2014. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.03.01_30/en_30263702v010301v.pdf (cit. on p. 75).