POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Protecting In-Vehicle Services with a Secure SOME/IP Protocol

Supervisors
prof. Fulvio Risso
prof. Riccardo Sisto
dott. Fulvio Valenza

Candidate
Marco Iorio

Company tutor
Italdesign
dott. ing. Massimo Reineri

Academic year 2017-2018

Abstract

Vehicles are becoming every generation more smart and ICT oriented: modern cars are characterized by dozens of different Electronic Control Units (ECUs), each one hosting one or more applications devoted to monitor and manage every single aspect of the vehicle itself. Advanced Driving Assistance Systems, nowadays becoming a standard, are going further, moving the control of safety critical systems, like braking and steering, to computers, algorithms and software.

Previous research demonstrated that many security flaws do exist in commercially available vehicles due to the massive presence of software. Network protocols designed without taking into proper account security design principles and application bugs have been exploited by researchers to remotely take over the control of different vehicular systems, without the possibility for the actual drivers to react.

The thesis originates from these problems, and focuses on novel mechanisms and algorithms to provide improved security to the applications that are executed in the vehicle, which are based on the principle of defining exactly who can talk to whom, hence allowing each service to be contacted only from trusted parties.

The desire for a future-proof solution, compatible with service relocatability and not tied to static configurations, has driven the research towards an emerging communication middleware designed for automotive use-cases: SOME/IP. While being very promising as a protocol, thanks to the service-oriented abstraction and the transparent service discovery functionalities, it is characterized by no security features for protection from malicious or compromised ECUs.

After having identified the major areas requiring protection, a security framework integrated with SOME/IP has been designed. It aims at guaranteeing both the authentication of the different parties involved in the communications and the protection of the actual messages transmitted over the network, without giving up the dynamism typical of this middleware.

The designed framework has been implemented as a proof of concept inside the `vsomeip` stack, an open source implementation of the SOME/IP specifications. The functionalities and performances have been evaluated both quantitatively, by measuring the penalties introduced by the modifications, and by means of a demonstrator, which shows the different features of the proposed solution in an environment mimicking various ECUs deployed within a vehicle.

Contents

1 Introduction 5
- 1.1 Goal of the thesis . . . 6
- 1.2 Structure of the work . . . 6

2 Background 8
- 2.1 Vehicular networks . . . 8
  - 2.1.1 In-vehicle networks . . . 9
  - 2.1.2 The CAN bus . . . 9
  - 2.1.3 Automotive Ethernet . . . 11
- 2.2 Car Hacking . . . 11
  - 2.2.1 Jeep Cherokee Hacking . . . 12
- 2.3 Related works . . . 14

3 SOME/IP 16
- 3.1 Communication paradigms . . . 16
  - 3.1.1 Request/Response . . . 17
  - 3.1.2 Publish/Subscribe . . . 17
- 3.2 Transport protocol bindings . . . 18
  - 3.2.1 UDP . . . 19
  - 3.2.2 TCP . . . 19
- 3.3 On wire format . . . 20
- 3.4 Service discovery . . . 21
- 3.5 vsomeip . . . 23
  - 3.5.1 Example of applications . . . 24
  - 3.5.2 Configuration files . . . 25
  - 3.5.3 Architectural description . . . 26
  - 3.5.4 Security functionalities . . . 28

4 Securing SOME/IP 29
- 4.1 Security levels . . . 29
  - 4.1.1 Nosec . . . 30
  - 4.1.2 Authentication . . . 30

4.1.3 Confidentiality . . . 31
4.2 Cryptography overview . . . 31
4.2.1 Symmetric cryptography . . . 31
4.2.2 Asymmetric cryptography . . . 32
4.3 Security protocol . . . 32
4.3.1 Keys granularity . . . 33
4.3.2 Session establishment . . . 34
4.3.3 Message protection . . . 37
4.4 High-level architecture . . . 41
4.4.1 Security module . . . 41
4.4.2 Secure storage . . . 42
4.4.3 Execution manager . . . 42
4.5 Limitations . . . 43

5 PoC Implementation 44
5.1 Cryptography . . . 44
5.1.1 Cryptography libraries . . . 45
5.1.2 Cryptography algorithms . . . 46
5.1.3 Benchmark methodology . . . 46
5.1.4 Results evaluation . . . 48
5.2 Functional modules . . . 53
5.2.1 Cryptography abstraction . . . 53
5.2.2 Session establishment . . . 54
5.2.3 Message protection . . . 55
5.3 Security configuration . . . 56
5.3.1 Digital certificates . . . 56
5.3.2 Configuration files . . . 57

6 Experimental Evaluation 60
6.1 Automatic testing . . . 60
6.2 Performance measurements . . . 62
6.2.1 Benchmark methodology . . . 62
6.2.2 Results evaluation . . . 64
6.3 The demonstrator . . . 74
6.3.1 Dashboard . . . 74
6.3.2 Services . . . 75
6.3.3 Attacker . . . 76
6.3.4 Conclusions . . . 77

7 Conclusions and Future Work 79

Bibliography 81

Chapter 1

Introduction

During recent years, the automotive sector has known a tremendous innovation, evolving at a speed previously inconceivable for an already mature industry. While the basic mechanical components which enable a vehicle to accelerate, steer and brake still resemble the ones already in use many years ago, a completely different world, made of copper, silicon and lines of code, has made his way inside every type of car, from low-cost to luxury, and is gaining every day more and more prominence.

On one side, tens of Electronic Control Units[1] (ECUs), interconnected through a wired nervous system, monitor thousands of values and parameters coming from sensors positioned everywhere in the car, to guarantee its perfect functioning and to take countermeasures in case something weird is detected: vehicles are safety-critical systems after all. On the other one, advanced infotainment systems, equipped with functionalities that only a couple of years ago were defined futuristic, are becoming one of the most distinctive features to make every new car model unique and somehow special. Artificial intelligence, advanced driving-assistance systems and autonomous driving are marketing buzzwords which all point towards more computational power, algorithms, communication networks and protocols: bits are stealing the stage to the steel.

Computerization does not only bring advantages to the automotive world: a whole new group of challenges is thrown inside a sector whose first goal must be the safety, the protection from hazards which can easily become deadly. Nonetheless, while testing mechanical components is already an established practice, controlled by strict standards and made easier by years and years of experience, the same cannot be said for what regards software, which is often devoted to much less life-threatening activities. Furthermore, with the increase in the number of automatized functions and connections towards external systems, software and network communications are becoming tempting targets for a wide variety of dishonest and wicked individuals,

[1] Any embedded system in automotive electronics that controls one or more of the electrical systems or subsystems in a vehicle.

which can leverage the decennial experience in both local and remote attacks in the ICT world to insinuate themselves inside the "computers on wheels" and eventually gain the control. How long will it last before the first automotive malware is discovered?

1.1 Goal of the thesis

This thesis work, which has been carried on as a collaboration between the Computer Networks Group of Politecnico di Torino and Italdesign, a design and engineering company part of the Volkswagen Group based in Moncalieri (Italy), originates from the increasing sensibility about the importance of cybersecurity inside the automotive world. It focuses on novel mechanisms and algorithms to provide improved security to the applications that are executed in the vehicle, by providing a framework to express and enforce high-level policies defining which are the allowed communications, preventing services from being contacted by malicious parties.

Before starting thinking about any possible solution, an initial investigation has been carried on to understand which are the main architectures and network protocols which play an important role inside every vehicle. Two of them, in particular, emerged prominently: on one side the CAN bus, which constitutes the nervous system of the current generation of vehicles, and on the other Automotive Ethernet, which is deemed to gain more and more importance in the near future.

Aiming at the design of a future proof solution, compatible with emerging trends like on-demand services and independent of hard-coded configurations, the research turned the attention towards the analysis of an emerging communication middleware designed for typical, Ethernet oriented, automotive use-cases: SOME/IP. Most of the work has been devoted to the design of a security framework integrated into SOME/IP, and to its implementation inside the open source `vsomeip` stack. Finally, the goodness of the approach has been verified both by means of quantitative measurements of the introduced latencies and by the development of a demonstrator which, mimicking different ECUs deployed within a vehicle, shows how the unprotected system is easily compromised by an attack, while the secured one is able to survive.

1.2 Structure of the work

The remainder of the discussion is structured as follows:

- **Chapter 2** outlines some background aspects concerning vehicular networks, depicts successful episodes of car hacking and concludes with an analysis of possible solutions already available in literature;

- **Chapter 3** provides an in-depth overview of the SOME/IP framework and its open source implementation `vsomeip`, upon which the security features are designed;
- **Chapter 4** describes in great detail the design of the proposed solution, with emphasis on the security properties provided by each functionality;
- **Chapter 5** offers a discussion of the main architectural choices performed during the implementation of the conceived solution and motivates the decision about the selected cryptography library;
- **Chapter 6** explains how the implementation of the proof of concept has been validated, shows the results of the quantitative measurements and describes the realization of the demonstrator;
- **Chapter 7** closes this thesis work with final conclusions and proposals for future enhancements.

Chapter 2

Background

In this chapter, an overview about different network technologies exploited in the automotive industry is presented, with a particular emphasis on two important standards for in-vehicle communications: the well-established CAN bus and the emerging Automotive Ethernet. Subsequently, the car hacking problem is analyzed, by showing the results of previous research demonstrating that commercially available vehicles are vulnerable to cyberattacks. Finally, a discussion about related works is introduced, by considering different solutions already available to protect in-vehicle communications.

2.1 Vehicular networks

The different computer systems distributed throughout the vehicle cannot operate in complete isolation: communications between the different units must be established to exchange useful information and to provide the requested services. Moreover, the number of signals coming from outside the chassis, being them GPS data or warnings originated from preceding vehicles, is going to increase in the near future to guarantee better security and to enhance the on-board experience. Vehicular networks can be mainly divided into three broad areas:

- *In-vehicle*, which represent the nervous system in charge of interconnecting the different ECUs present in a vehicle both one to another and with the plethora of sensors deployed all over the vehicle;
- *Vehicle-to-Vehicle (V2V)*, which allow automobiles to "talk" to each other forming an ad-hoc wireless network, exchanging information to, e.g., automatically adapt the speed to the preceding vehicles or to react faster to braking, even with limited visibility;
- *Vehicle-to-Infrastructure (V2I)*, a communication model which allows vehicles to share information through short range wireless signals with the so-called

Roadside Units, smart devices such as traffic lights, lane markers, streetlights, signage and parking meters, in order to receive real-time advisories for increased safety and fuel economy.

Furthermore, even if not strictly related to the automotive sector, modern vehicles are becoming equipped with a great amount of network interfaces common in our smartphones, to provide every day a more complete infotainment experience: Bluetooth, Wi-Fi and 4G are all technologies which are gaining more and more prominence also inside the cars.

2.1.1 In-vehicle networks

Being this thesis work focused on in-vehicle services, an analysis in greater detail of the different network technologies and protocols exploited by these systems is carried on in the following, with particular emphasis on the two most relevant media: the CAN bus and Automotive Ethernet.

In modern vehicles, multiple communication buses, realized with different technologies and characterized by distinct network protocols do coexist for two main reasons: on one side, such design choice is due to the need for keeping physically separated systems which operate at different levels of criticality: nobody wants that a faulty radio makes the ABS unusable. On the other one, different systems have different requirements in terms of bandwidth, timings and positioning: while the vast majority of the interconnections is realized with wired technologies, a small role is also played by really low bandwidth wireless media, as in the case of tire pressure monitoring systems. Figure 2.1 shows the complexity of a typical automotive wiring harness design, which is already the third heaviest and third most costly component of automotive designs [1].

A central gateway plays the role of an aggregator, by collecting and analyzing the signals coming from the different interfaces, and forwarding through the other buses only the ones of interest; moreover, this device often acts also as a terminator for the information originating from both the external world and the on-board diagnostic interfaces, and routes them towards the expected destinations.

2.1.2 The CAN bus

A Controller Area Network (CAN bus) is a robust and widespread vehicle bus standard designed to allow microcontrollers and devices to communicate with each other at a maximum transfer speed up to 1 Mbps. It is a broadcast and message-based protocol, whose most noteworthy feature is the usage of a lossless bitwise arbitration method of contention resolution, which prevents the need for retransmissions when multiple nodes try to send a message at the same time. This behavior makes it suitable for strongly real-time oriented communications.

Figure 2.1: Typical wiring harness in a car.

Every CAN packet is composed by two main elements: an arbitration ID, which uniquely identifies the type of transmitted data and acts as a priority field, and the actual payload, which can be up to eight bytes long; both identifiers and payloads meanings are not standardized, being different for each car maker. Extensions have been proposed to increase the identifier size and to allow longer payloads by using packet chaining.

Messages transmitted over the CAN bus can be broadly divided into three categories:

- *Informative*, which are periodically sent to advertise the readings coming from sensors (e.g. the speed of each wheel);
- *Requesting an action*, which target a specific device, although the message is anyway broadcasted, to ask for an action to be performed (e.g. to lift a window in consequence of a stimulus);
- *Diagnostic*, which are transmitted by mechanics through particular instruments to obtain diagnostic information or to trigger specific functions.

A very simple broadcast communication protocol used in the automotive industry to complement the CAN bus is called Local Interconnect Network (LIN). It is designed to be as inexpensive as possible and can support up to 16 slave nodes which primarily just listen to the master device, often connected to the CAN bus.

The maximum speed of 20 kbps makes it useful only for very low-end peripheral systems.

2.1.3 Automotive Ethernet

While new applications are requiring every day more and more communication bandwidth to fulfill their requirements, existing high-speed proprietary standards, such as the MOST Protocol[1] and the FlexRay bus[2], are arising new challenges due to the complexity in the implementation and the expensiveness caused by licensing issues.

Although Ethernet is a mature and standardized technology with over thirty years of use in the networking market, it has not been widely adopted in the automotive industry until recent years, mainly due to the strict electromagnetic interference (EMI) requirements for the sector. In order to overcome this limitation, the OPEN (One-Pair Ethernet) alliance, made up of car makers, suppliers and semiconductor companies, started sponsoring the Broadcom's 100 Mbps BroadR-Reach solution, which enables full-duplex transmission over a single pair at a reduced base frequency to meet the automotive EMI specifications. Later on, this technology has been standardized by the IEEE 802.3 group as 100BASE-T1 [2] (100 Mbps) and 1000BASE-T1 [3] (1000 Mbps) and it is usually referred to as Automotive Ethernet.

Two main advantages can be associated to Automotive Ethernet: firstly, it is characterized by a wider bandwidth with respect to the other networking media used for in-vehicle communications, which is of particular relevance for demanding video streams (e.g. those required by autonomous driving) and multimedia applications. Secondly, being the modifications only at the physical layer, it is completely compatible with the protocols used every day together with vanilla Ethernet, thus offering the possibility to leverage all the knowledge already acquired in the ICT world. Nonetheless, new protocols are currently under development, which aim at providing functionalities required by the automotive industry: one example includes SOME/IP, a service-oriented middleware laid on top of the TCP/IP stack which, being the backbone of the work, will be analyzed in great detail in chapter 3.

2.2 Car Hacking

With the advent of more and more computer-based systems inside the vehicles, a whole new set of challenges, previously unknown for the automotive world, are

[1] Media Oriented Systems Transport Protocol: a high-speed multimedia network technology optimized by the automotive industry, which is used in almost every car brand worldwide.

[2] A high-speed bus (10 Mbps) that is geared for time-sensitive communications, such as drive-by-wire, steer-by-wire, brake-by-wire and so on.

threatening the life of millions of unaware drivers. During recent years, different researchers concentrated their efforts in studying the feasibility of carrying on attacks to vehicles by exploiting both bugs in ECUs' software and specifically crafted messages injected into vehicular networks, both locally and remotely: the discoveries are certainly not encouraging for the industry.

In 2008, Tobias Hoppe et al. [4], depicted four different attack scenarios which, exploiting messages injected into the CAN network and the broadcast nature of the transmissive medium, allowed the researchers to perform simple actions. In particular, two offenses, aiming at opening the window lift when a specific condition was met and keeping the anti-theft system off, were based on flooding the network with malicious messages to cause message confliction, which arises when the target ECU receives opposing information. Different ECUs may react differently to this condition but, usually, simpler ones merely consider valid the last message received: the attacks succeed even if legit messages, which are periodically broadcast, are not removed from the medium. The researchers also demonstrated that it was possible to replace an entire system, such as the airbag, with a bogus chip which, sending messages mimicking the original device, made the substitution unnoticeable; furthermore, they showed that, due to implementation flaws, it was possible to exploit diagnostic messages to extract privacy sensitive information from the vehicle.

In 2010, researchers from the University of Washington and the University of California San Diego [5] went further and showed that if they were able to inject messages into the CAN bus of a vehicle, they could make physical changes to the car, such as controlling the display on the speedometer, killing the engine, as well as affecting braking. The research received widespread criticism because people claimed there was not a way for an attacker to inject these types of messages without being close to the vehicle and, with that type of access, they could just cut a cable or perform some other physical attack. The next year, the same research group [6] showed that they were able to remotely perform the attacks, by exploiting interfaces such as the MP3 parser of the radio, the Bluetooth stack and the telematics unit to get the code executed.

2.2.1 Jeep Cherokee Hacking

In 2015, Charlie Miller and Chris Valasek ended the debate by clearly demonstrating that remote car hacking of an unaltered vehicle was indeed possible, as stated by the journalist who took part to the experiments [7], and detailed by the authors themselves [8].

The vehicle chosen by the researchers to perform their tests was a 2014 Jeep Cherokee, depicted in figure 2.2, which was deemed to provide the best opportunities of success after having evaluated the attack surface, the network architecture and other parameters. The internal architecture was interesting from the researchers' point of view, thanks to the presence of a central unit, which was the source for

Figure 2.2: 2014 Jeep Cherokee.

infotainment, Wi-Fi and cellular connectivity, navigation and apps, while being connected at the same time to both the CAN buses present in the vehicle. Most of the functionalities were executed on a single 32-bit ARM processor running the QNX[3] operating system, while a low-power microcontroller managed the communications through the CAN buses.

Different possible entry points for an attacker have been identified by the researchers, with the most promising being the Bluetooth stack, which would anyway require the attacker to be near to the target, the pay-per-use Wi-Fi hotspot service available in the Jeep, again suffering from a limited range, and the cellular radio used to retrieve information from the Internet. During their analysis, the hackers discovered serious vulnerabilities, comprising the faulty generation of the WPA2 password which could be easily obtained by an attacker and the presence of different TCP and UDP open ports, including one dedicated to D-Bus over IP, a protocol allowing inter-process communication and remote procedure call mechanisms. Exploiting the newly discovered backdoor, accessible both from Wi-Fi and the cellular network and lacking any authentication system, the researchers succeeded in controlling almost every parameter related to the infotainment system, including changing the radio station or the volume, setting the fans to arbitrary speed, obtaining the GPS coordinates of the vehicle and so on.

[3]A commercial Unix-like, real-time and micro-kernel based operating system, aimed primarily at the embedded systems market.

Being the compromised system not able to communicate directly through the CAN bus, gaining access to vital systems of the vehicle required a great effort from the hackers, which needed to reverse engineer the firmware of the microcontroller to forward messages from and to the main unit. They exploited one more time the available D-Bus connection to flash the modified software and eventually succeeded in injecting whichever message they wanted into both CAN buses, obtaining the possibility to shut down the engine or disable the brakes issuing commands through a laptop wherever in the US.

The research clearly demonstrates that remote car hacking is possible and real, although accessing life-threatening features required very strong hacking skills and motivation: the attackers spent months in reverse engineering and disassembling firmwares, understanding messages and deeply analyzing core system files in order to achieve their goals. Nonetheless, it highlights how the trend of adding more and more functionalities inside the vehicles to make them appear like smartphones is exposing the automotive industry to new dangers, which are mainly caused by the incorrect integration of systems developed and manufactured by different companies, whose specifications and implementations are almost always kept secret.

2.3 Related works

Being a vital system in every vehicle, CAN bus has been the subject of a high number of studies to increase the security and prevent malicious attacks. Both active protections, exploiting cryptographic functions to guarantee message authentication [9], and intrusion detection systems, continuously monitoring the network to detect traffic pattern changes [4], have been proposed. Nonetheless, they all suffer from strong limitations due to the reduced computational power typical of microcontrollers and the challenging real-time needs characterizing this protocol.

Moving to more powerful systems and Ethernet-based communications, the security of messages exchanged between remote devices has already been tackled by many researchers, especially in the broader field of ICT networking. Various protocol suites are in use today, which approach the problem at different levels to guarantee the features required by the specific use-cases. Firstly, security can be provided at the network layer of the ISO/OSI stack by the IPSec protocol [10], which establishes secured tunnels between interconnected hosts; secondly, it is possible to climb up to the transport layer, where the ubiquitous TLS [11] is in charge of protecting connections between applications residing on remote systems. Although being both characterized by proved effectiveness and maturity, they appear not to fit well the peculiarities of in-vehicle networks and especially of the SOME/IP framework: the former is restricted by a limited granularity due to application unawareness, while the latter does not play well with multicast communications and requires a rather complex authentication handshake.

Mohammad Hamad et al. [12] recently proposed a framework that targets modern vehicular networks and aims at providing secure communications between ECUs by exploiting security policies to define who should talk to whom. The described solution is made up of two main building blocks: on one side, a framework used to build secure communication policies gradually by integrating them throughout the design and life cycle of the software component, which enforces trust relationships and allows delegation through the usage of a Public Key Infrastructure (PKI). On the other one, a security module which acts as a connection policy checker vetting the incoming and outgoing communications and enforcing the security policies in a distributed manner. While being the proposed solution undoubtedly noteworthy, and in a certain way adopting a solution similar to the one presented in this thesis work, different elements push towards the adoption of a completely divergent approach which, leveraging the integration with SOME/IP, aims at exploiting provided functionalities, such as service discovery, to simplify the infrastructure and relax the need for low-level policies.

Chapter 3

SOME/IP

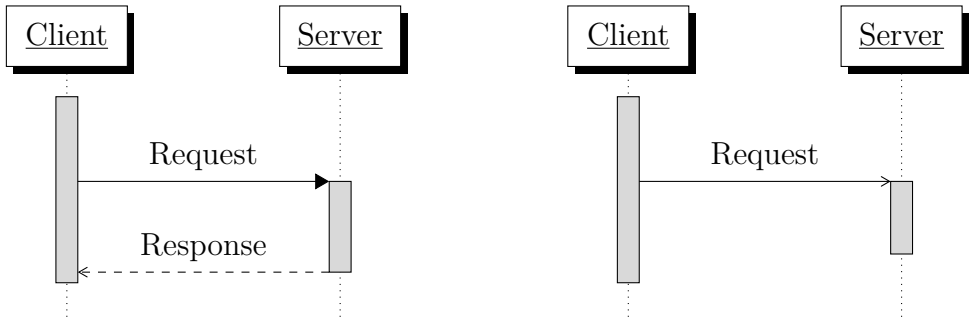
Scalable service-Oriented MiddlewarE over IP (SOME/IP) is a communication middleware designed for typical, Ethernet oriented, automotive use-cases. It has been standardized by AUTOSAR,[1] as part of the effort in the development of a solution which is oriented towards emerging scenarios like highly automated driving, requiring high-performance computing hardware and intense network communications. According to its specifications [14], the main reasons behind the definition of a new Remote Procedure Call (RPC) mechanism include the desire for fulfilling the hard requirements regarding resource consumption in vehicles, the need for scalability from tiny to large platforms running different operating systems and the inclusion of all the features required by automotive use-cases.

3.1 Communication paradigms

SOME/IP is designed to provide a service-oriented abstraction on the top of one or more different transport protocols, mainly UDP and TCP. In the terminology introduced by the middleware, a *service* represents an atomic entity that groups together zero or multiple *methods*, *events* and *fields*, the building blocks which make it possible for distinct applications to transmit information from one to another; different *instances* of the same service may coexist at the same time, and reside on different ECUs as well as on the same device.

Two different communication patterns, request/response and publish/subscribe, are offered by SOME/IP which, due to their importance, are analyzed in the following. Additionally, by their combination, the concept of *field* originates: it represents a value associated to a getter, a setter and a notification event.

[1] AUTomotive Open System ARchitecture, a worldwide development partnership of automotive interested parties founded in 2003 which pursues the objective of creating and establishing an open and standardized software architecture for automotive ECUs [13].



(a) Classical request/response. (b) Fire & forget.

Figure 3.1: Request/response communication diagrams.

3.1.1 Request/Response

One of the most common communication paradigms is the one referred to as request/response, which corresponds to standard Remote Procedure Call. This pattern aims at providing the possibility to invoke functions which do not belong to the application currently under execution but, instead, are made available by a remote service and identified as *methods*, according to the SOME/IP terminology.

The process, depicted in figure 3.1a, starts whenever a communication partner, playing the role of a client, wants to trigger a remote execution: a designated SOME/IP message is constructed, identifying the target destination and containing the possible parameters which the caller may want to include in. When the message is correctly delivered to the destination, the server can decode the request and perform any involved operation: once the processing terminates, a response containing either the result of the computation or an identifier of the occurred error is finally generated and dispatched, and the invocation terminates.

One special case of the pattern under examination, represented by figure 3.1b, is defined fire & forget, and entails a one-way communication by not requiring the callee to send back a response: its only purpose consists in requesting a remote node to perform an action, without caring about the result.

3.1.2 Publish/Subscribe

The alternative messaging pattern available in SOME/IP is represented by the notification concept, depicted in figure 3.2, which is rather typical in automotive networks. It decouples the sender from the recipients of the messages: whenever a value changes, the service in charge of it publishes a new notification targeting the corresponding *event*; applications willing to receive updates, on the other hand, express their interest by subscribing to the event. This way, publishers do not program the messages to be sent directly to specific receivers but, instead, leave the middleware the task of delivering them only to the intended destinations.

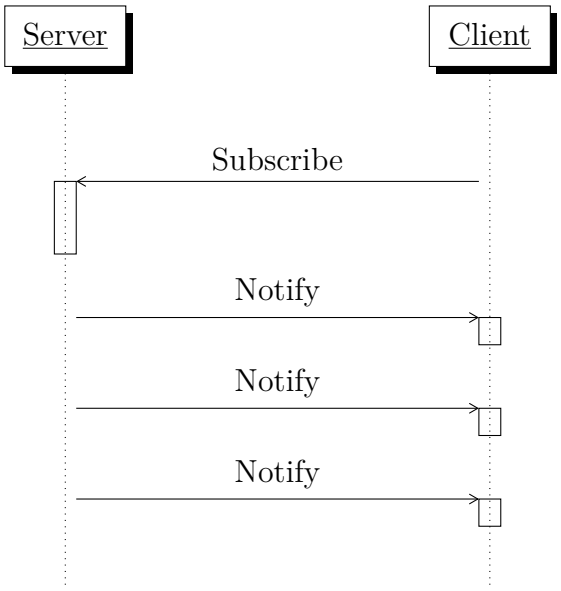


Figure 3.2: Publish/subscribe communication diagram.

Depending on the specific use case, different strategies for sending notifications are possible: common examples include *cyclic updates*, where an updated value is sent in a fixed interval, *updates on change*, dispatched as soon as a modification of the tracked value is detected and *epsilon change*, transmitted only when the difference with respect to the last value is greater than a certain threshold.

One of the main advantages offered by the adoption of the publish/subscribe communication paradigm is related to traffic optimization: by knowing the number and the location of the subscribers, the middleware is able to transparently exploit features offered by lower network layers, such as multicast messages, to save transmissions on the communication medium; moreover, it can also completely avoid sending notifications when no applications are interested in them.

SOME/IP, actually, takes care only of the notification phase, while the initial subscription procedure is managed by SOME/IP–SD, a companion protocol which is entailed of managing service discovery functionalities, and that is described in §3.4.

3.2 Transport protocol bindings

As seen before, SOME/IP operates on the top of a transport protocol, which is in charge of providing the functionalities needed to deliver messages from the sender to the recipient(s). Two main bindings are currently supported, corresponding to the protocols that make the communication across the Internet possible: UDP and TCP; nonetheless, external transport mechanisms, such as Network File System (NFS) or Automotive Pixel Link, could be used if more suited for the specific use-case.

3.2.1 UDP

Being a connectionless protocol, the delivery is achieved by enveloping SOME/IP messages in UDP packets and simply transmitting them across the network, without prior connection establishments. In order to save transmissions, it is possible for multiple messages, clearly all targeting the same destination, to be grouped together and to become part of the same datagram: the recipient is able to separate the various slices by exploiting the length field contained in every SOME/IP header, and comparing it with the remaining amount of data still to be read.

The UDP binding is able to transport only SOME/IP messages that fit directly into an unfragmented IP packet: the maximum allowed payload size has been arbitrarily fixed to 1400 bytes, considering the Ethernet Maximum Transmission Unit (MTU), equal to 1500 bytes, and leaving some extra space to allow for future changes to the protocol stack (e.g. changing to IPv6 or adding security means).

To permit the delivery of bigger messages over UDP, a protocol extension, named SOME/IP Transport Protocol [15], has been standardized by the same consortium: it implements the message fragmentation concept directly within the framework, by providing the possibility of splitting extra-sized payloads in multiple parts, each one with a copy of the header and additional information necessary for the recreation, transmitted in different UDP datagrams.

According to the guidelines provided by the specifications, UDP should be the preferred binding, either vanilla or using the SOME/IP–TP functionalities in case of large messages, being a very lean protocol which introduces few overhead and is suitable also in case of hard latency requirements. Indeed, due to the cyclic nature of many communications in automotive applications, the best approach to errors often consists in just waiting for the next data transmission, instead of trying to repair the last one. Finally, it supports multicast messages, thus offering a way to optimize the network utilization.

3.2.2 TCP

The alternative choice to convey SOME/IP messages entails TCP, a much more complex transportation protocol which integrates different robustness features directly out-of-the-box, by transparently managing packet losses, reordering and duplications, and by automatically issuing retransmission requests whenever necessary.

A new TCP connection towards the service provider is opened by the client when the first message has to be transmitted, and it is reused for all methods, events and fields belonging to the same service instance; in a corresponding manner, the connection is closed by the client when of no more use.

Being a heavyweight protocol, which introduces features regarding flow and congestion control, TCP is suggested to be used only in case of very large chunks of data need to be transported and no hard latency requirements in case of error exist.

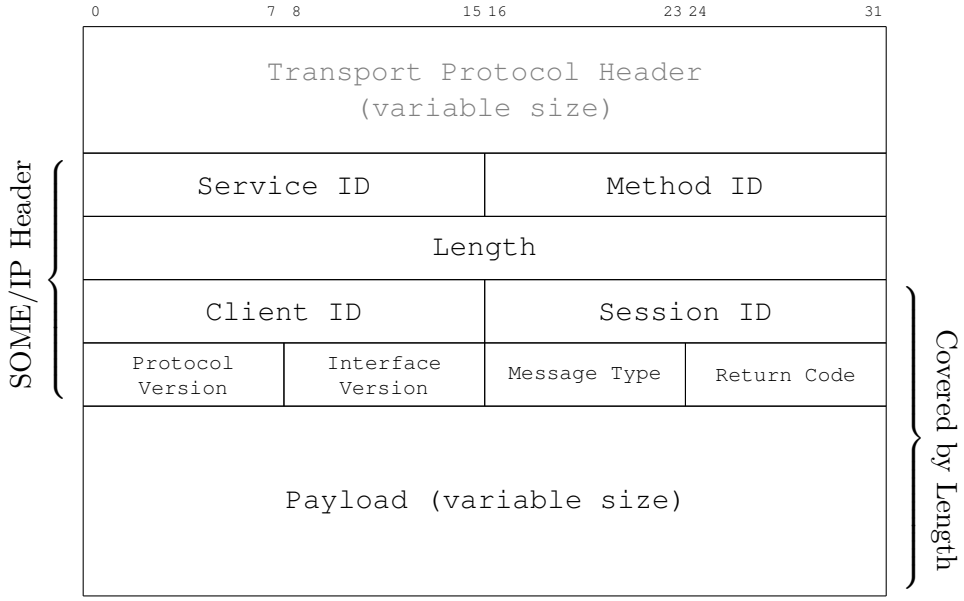


Figure 3.3: SOME/IP message.

3.3 On wire format

A typical SOME/IP message, depicted in figure 3.3, is composed by a *header*, which specifies the communication parties and the type of the message, and a *payload*, which contains the actual data to be transferred.

The header, 16 bytes long and encoded in network byte order, is made up of the following fields:

- `Service ID`, which uniquely identifies the targeted service and, together with the Method ID, forms the Message ID; it is worth noting that there is no way to distinguish a specific instance of a service, which is instead specified by the used transport layer port;
- `Method ID`, that specifies, within the chosen service, the RPC method to be executed or the event to which the notification belongs;
- `Length`, containing the length in bytes starting from the Client ID until the end of the SOME/IP message;
- `Client ID`, which identifies the application triggering the specific request; together with the Session ID, it forms the Request ID, to differentiate multiple parallel uses of the same method or event;
- `Session ID`, an identifier incremented whenever a new message is sent;

- `Protocol Version`, specifying the SOME/IP version;
- `Interface Version`, indicating the major version of the service interface;
- `Message Type`, used to differentiate the type of the message: possible values include `REQUEST`, `REQUEST_NO_RETURN`, `RESPONSE`, `ERROR` and `NOTIFICATION`, plus the corresponding counterparts used when the payload is fragmented according to the SOME/IP–TP specifications;
- `Return Code`, specifying, in case of a response, whether the requested operation succeeded or identifying the occurred error.

In case consecutive data exists, it forms the payload which, depending on the message type, may convey different pieces of information: parameters in case of a request, results of computations for responses or updated data in notifications. The specifications of SOME/IP do also provide the serialization rules to be adopted for the payload, regarding both basic numerical types and more complex data structures, including strings and arrays.

3.4 Service discovery

One of the main features introduced by SOME/IP is the one named *service discovery*, which is provided by its companion protocol SOME/IP–SD. According to the specifications [16], defining formats, message sequences and semantics of the protocol, its main tasks include communicating the availability of the different services as well as advertising and managing the subscription phase regarding events.

It enables the decoupling between the application providing a given service, which merely announces its capabilities to the framework, and the ones requiring it to operate, that, similarly, just have to signal their need. Furthermore, being able to detect where the various applications are located, SOME/IP–SD relaxes the requirement for static configurations specifying the network parameters of all the services willing to communicate with: dynamism is guaranteed by the possibility of starting and stopping applications at any moment, as well as of transparently relocating them.

SOME/IP–SD messages are sent by exploiting the features offered by SOME/IP, although the transport protocol is constraint to be UDP; they can be delivered either in unicast, in case directed to a specific client, or exploiting multicast functionalities, to reach all the listening parties.

As outlined by figure 3.4, messages are composed by one or more *entries*, each one used to synchronize the state of a service instance or for publish/subscribe handling; moreover, different *options* are often associated to each entry, to specify additional pieces of information like, e.g., the network parameters, in terms of

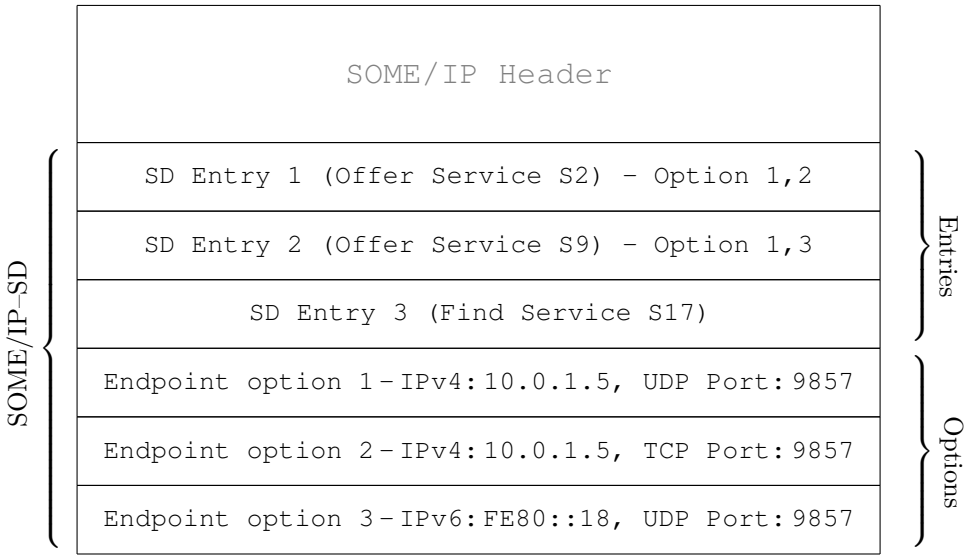


Figure 3.4: Structure of a SOME/IP–SD message.

addresses and ports, at which the advertised service is accessible. The main types of entries include:

- **Offer Service**, advertising the availability of a given service instance, characterized by a certain version, together with the network parameters of the endpoint where the instance is accessible; it is periodically sent by the protocol, and can also be used to declare that a service is no more available, by specifying a TTL value equal to zero;

- **Find Service,** used to speed-up the detection process, by requesting a specific service if its current state is still unknown;

- **Subscribe Eventgroup**, that, as the name suggests, it used by a client to subscribe to an eventgroup[2] offered by a remote service, by specifying also the endpoint where the notifications shall be delivered;

- **Subscribe Eventgroup Acknowledgement**, communicating whether the subscription has been accepted or not by the provider and, in the positive case, announcing the network parameters used for multicast notifications.

[2]A logical grouping which includes one or more events offered by the same service; every event may be part many eventgroups.

3.5 vsomeip

The `vsomeip` stack[3] is an open-source `C++` implementation of the SOME/IP specifications, designed as part of the GENIVI[4] project. The stack, currently at version 2.10.21 dated May 2018, consists of three main modules, described in the following.

- `libvsomeip.so`, the main library, which is in charge of providing all the basic functionalities according to the protocol specifications. It encompasses creation and serialization of message headers, establishment of connections, management of the communication endpoints and packets transmission; moreover, it provides the event loop abstraction which simplifies the development of applications based on the `vsomeip` stack. `libvsomeip.so`, however, does not provide functionalities for the serialization of data structures composing the payload: the task is left to the application developer, possibly by exploiting the SOME/IP binding of `CommonAPI`, a higher-level framework which abstracts the inter-process communication protocol chosen through the logical definition of application interfaces and the automatic generation of stub code [18].

- `libvsomeip-sd.so`, a companion library implementing the service discovery functionalities as described by SOME/IP–SD. It is automatically loaded only in case it is required and manages both the detection of running service instances and the subscription phase concerning event notifications: while not being mandatory, therefore, it must be activated to enable the publish/subscribe communication paradigm.

- `libvsomeip-cfg.so`, whose main task consists in the interpretation of the `json` configuration files which specify all the necessary parameters for application execution, as described in §3.5.2, and in providing easy access to them from the other modules.

While mainly sticking to the SOME/IP specification, some described features are currently not implemented by `vsomeip`: these include mainly the already mentioned payload serialization, the SOME/IP–TP protocol for message fragmentation, explained in §3.2.1, and some special options defined by the SOME/IP–SD protocol, like the one for load balancing between different instances of the same service, which seem to be simply ignored by the framework.

[3] `https://github.com/GENIVI/vsomeip`

[4] GENIVI is a nonprofit industry alliance committed to driving the broad adoption of open source, In-Vehicle Infotainment (IVI) software and providing open technology for the connected car. [17]

Listing 3.1: Server example code.

```
1  #include <...>
2  #include <vsomeip/vsomeip.hpp>
3
4  std::shared_ptr<vsomeip::application> app;
5
6  int main() {
7      app = vsomeip::runtime::get()->create_application("Server");
8      app->init();
9      app->register_message_handler(SERVICE_ID, INSTANCE_ID, METHOD_ID, on_message);
10     app->offer_service(SERVICE_ID, INSTANCE_ID);
11     app->start();
12 }
13
14 void on_message(const std::shared_ptr<vsomeip::message> &request) {
15     auto its_response = process_request(request);
16     app->send(its_response);
17 }
```

3.5.1 Example of applications

Before continuing in the analysis of the `vsomeip` internals, an example of two programs, respectively a server and a client which communicate between them through the request/response paradigm is presented, to provide the reader an idea about how applications based on this stack are implemented.

Listing 3.1 depicts the very basic instructions – no error checks are performed – necessary to implement a simple `vsomeip` based server, which continuously listens for new messages and responds to them. Firstly, in the `main` function, an object of type `application`, which provides access to most of the middleware functionalities, is created; secondly its `init` method is executed, to load the configuration and set-up the whole framework. A *message handler* is then registered and the framework is informed about which are the services offered by the application. Finally, the `start` method is executed, which begins the event loop: it never terminates until the corresponding `stop` function is called.

The interaction between `vsomeip` and the application is highly based on *handlers*, callback functions which are specified during the initialization and linked to specific circumstances, like reception of messages, changes in the availability of services or subscriptions to events: whenever the corresponding action is triggered, the stored function is invoked, by passing the correct parameters to characterize the occurred event. Back to the example, `on_message` is executed every time a message targeting the method of interest is received: it processes the request and finally sends back the prepared response, by exploiting the specific method of the `application` class.

Listing 3.2, on the other hand, illustrates its counterpart, again stripped down to the essentials; the basic structure is very similar to the previous case, with an initialization phase, where the handlers are registered and the needed service is

Listing 3.2: Client example code.

```
#include <...>
#include <vsomeip/vsomeip.hpp>

std::shared_ptr<vsomeip::application> app;

int main() {
    app = vsomeip::create_application("Client");
    app->init();
    app->register_availability_handler(SERVICE_ID, INSTANCE_ID, on_availability);
    app->register_message_handler(SERVICE_ID, INSTANCE_ID, METHOD_ID, on_message);
    app->request_service(SERVICE_ID, INSTANCE_ID);
    app->start();
}

void on_availability(...) {
    if (/* the requested service is available */) {
        vsomeip::message request = prepare_request();
        app->send(request);
    }
}

void on_message(vsomeip::message response) {
    process_response(response);
}
```

requested, followed by the `start` invocation. When the service discovery module detects[5] a running instance of the requested service, the `on_availability` function is executed, which, in this case, simply sends a request to the server; as before, if a response is received, it is delivered directly to the `on_message` method.

It is worth noting that, except for very simple examples like the ones presented here, the exploitation of background threads is mandatory: being the `start` method blocking, it is not possible to perform further operation once it is executed; moreover, long-lasting computations cannot occur within message handlers, to avoid blocking the threads directly managed by `vsomeip`.

3.5.2 Configuration files

The configuration of a wide number of parameters, ranging from network addresses associated to each offered service to logging features, is managed by means of `json` files, which are loaded during application initialization. While not deepening into the details, exhaustively described by the user guide available together with `vsomeip`, simple configuration files, which may be used for the examples presented in §3.5.1, are depicted in listings 3.3 and 3.4. Four main properties are specified:

[5] In case this functionality is disabled, whether or not a service is available depends on the parameters specified in the configuration file.

Listing 3.3: Server configuration.

```
"unicast" : "192.168.12.1",
"applications" : [{
    "name" : "Server",
    "id" : "0x1343"
}],
"services" : [{
    "service" : "0x1234",
    "instance" : "0x5678",
    "reliable" : { "port" : "31000"},
    "unreliable" : "31000"
}],
"service-discovery" : {
    "enable" : "true",
    "multicast" : "224.244.224.245",
    "port" : "30490",
    "protocol" : "udp"
}
```

Listing 3.4: Client configuration.

```
"unicast" : "192.168.12.2",
"applications" : [{
    "name" : "Client",
    "id" : "0x1344"
}],
"service-discovery" : {
    "enable" : "true",
    "multicast" : "224.244.224.245",
    "port" : "30490",
    "protocol" : "udp"
}
```

- `unicast`, indicating the local interface which is used for transmitting messages towards remote ECUs;
- `applications`, listing the names of the applications associated to each configuration, along with the corresponding identifiers used as Client IDs inside the SOME/IP header;
- `services`, stating which are the service instances offered by the applications, together with the transport protocols, and the corresponding ports, to be used; in case the publish/subscribe paradigm is exploited, this section describes also available eventgroups and relative multicast addresses;
- `service-discovery`, configuring whether this optional feature is enabled, together with the network parameters necessary for its operation.

It is worth noting that the `services` section does not need to enumerate requested ones, since they are automatically detected by the service discovery module: in case it is disabled, on the other hand, it is mandatory to state all of them, together with the network parameters at which they are reachable.

3.5.3 Architectural description

A representation of the architecture provided by `vsomeip` is depicted in figure 3.5 [19], which shows two ECUs interconnected through an Ethernet link. Different applications, based on the `vsomeip` communication framework, are being executed at the same time on the top of a Linux kernel, each one characterized by its own instance of the `vsomeip` library, which can be further subdivided into two main building blocks.

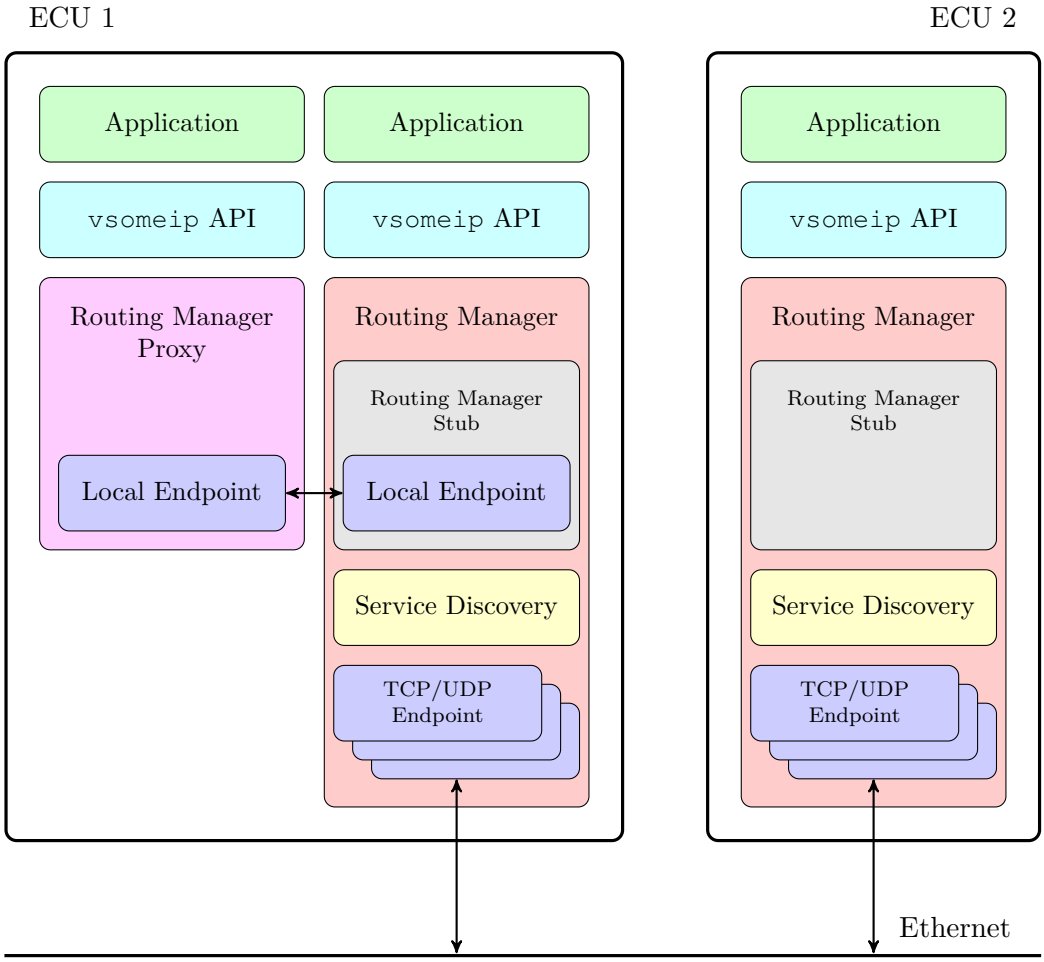


Figure 3.5: `vsomeip` architecture.

In the upper part, it is possible to glimpse the module which provides the public API exploited by all applications for the interaction with the library itself. It is mainly constituted by the `application` class already seen previously in the examples, which implements the event loop abstraction and acts as an interface between the applications and the underlying core module of the library. Moreover, it comprises the classes which allow the representation and automatic creation of SOME/IP messages and that, by means of `serializers` and `deserializers`, can be converted to and from the on-wire format mandated by the standard.

The core part of the library, on the other hand, is the one whose cornerstone is constituted by the `routing_manager` class, responsible for the actual delivery of the messages towards either applications residing on the same ECU, by means of Inter-Process Communication (IPC) mechanisms, or remote devices, using one of the available transport protocols. As exemplified by the picture, it is possible to have two different fashions of the routing manager that, anyway, are completely

transparent from the applications' point of view:

- **routing manager**, the full-fledged version of the module, which is loaded by only one application per each device, either the first which is executed or according to the configuration files. This instance is the only one responsible for sending messages to and receiving them from applications residing on remote devices, by actually establishing the network connections and managing the transport endpoints (i.e. the TCP and UDP sockets). Furthermore, it is in charge of loading the `libvsomeip-sd.so` library in case the service discovery functionalities are enabled, advertising to the other ECUs all the services offered by the applications executed inside the same device and internally propagating the received information.
- **routing manager proxy**, that is executed by all the other instances and performs only local communications, towards one of the other applications residing on the same ECU, by means of IPC mechanisms. Messages which need to be dispatched to remote recipients are, therefore, firstly sent to the central routing manager and then forwarded to the intended destination, where the inverse process may need to be carried on. Service information are constantly exchanged between the proxies and the master instance, to share the knowledge about offered and required services and to manage the problematics related to events.

3.5.4 Security functionalities

According to the `vsomeip` documentation, the framework includes some security features which are based on UNIX credentials, and therefore available only for local communications. In case this mechanism is activated by means of the configuration file, every local connection is authenticated during the establishment by exploiting the standard UNIX credential passing mechanism. It entails the usage of user and group IDs, which are matched against the policies specified in the configuration to decide whether the communication is allowed or not. Furthermore, information about other parties in the system must be received from the authenticated routing manager, to avoid malicious applications faking it and being able to wrongly inform other clients about services running on the system.

While being certainly an improvement with respect to the completely unsecured version standardized by the specifications, these features appear to be somehow limited and weak, due to the total lack of protection for what regards network-based communication between remote devices and the usage of unauthenticated configuration files.

Chapter 4

Securing SOME/IP

This chapter presents in great detail the security framework designed during the thesis work, which aims at protecting the communications occurring between in-vehicle services. It has been conceived tightly integrated within the SOME/IP middleware, due to its emerging importance in the automotive industry and with the desire of exploiting as much as possible its peculiar characteristics, especially the service discovery functionalities, to avoid the need for static configurations.

Dynamism and backward compatibility of applications have been two main cornerstones of the research, with the goal of providing a future-proof solution which is easily configurable without delving into low-level parameters. At the same time, the challenging vehicular environment, made up of a variety of different devices with unique characteristics has been kept in mind: while the usage of cryptography limits the target of the presented solution to only microprocessor-based ECUs, trade-offs have been considered to avoid the design of a very powerful but completely unusable framework.

4.1 Security levels

The security framework has been designed to operate at a *service instance granularity*, thus considering each instance of a SOME/IP service as a unique entity to which a specific application can be either allowed or denied access. Such decision, strongly influencing the whole design, has been taken as a compromise between two conflicting needs: on the one hand the requirements for strong isolation, which push towards a very fine granularity discriminating methods and events, and on the other one the constrained resources, which limit the number of session establishments and authentications that can be performed without increasing the latency to an unsustainable level. Nonetheless, being services logical abstractions grouping together methods and events, but unrelated from the concept of application, the final granularity depends on the architectural decisions made by application developers.

The conceived solution, beyond offering the possibility to specify for each application which are the service instances that can be offered or required, allows developers to associate to each one a specific *security level*, which selects the cryptographic function used to protect the actual messages transmitted across the network. Depending on the criticality of each service, different levels of security may be suitable, considering again a balance between protection and computational complexity, which may fruitlessly increase the latency and saturate the physical device. Three different security levels are provided by the designed framework, respectively denominated *nosec*, *authentication* and *confidentiality*, which are analyzed in the following.

4.1.1 Nosec

The *nosec* level is the simplest one, which merely corresponds to vanilla SOME/IP. While not providing any security guarantee, it has been made available for compatibility reasons, since it adds no complexity to the transmission; moreover, it may be useful for services which do not impose security requirements, to avoid wasting ECUs' power in useless computations.

In case of services operating at *nosec* level, potential wicked individuals, gained access to the transmissive medium, are expected to be able to both sniff messages, thus reading the exchanged information, and to inject spoofed packets, faking the legit sender without the possibility for the receivers to detect the attack.

4.1.2 Authentication

The second available security level in the proposed framework is named *authentication*: it guarantees that only allowed applications are able to send messages associated to a specific service, providing message authentication.

Before a SOME/IP packet is sent out by the middleware, a cryptographic signature, technically denominated Message Authentication Code (MAC), is attached to it. When the destination receives a message, the signature can be verified: in case it matches, the receiver knows that the packet originates from a trusted party and has not been modified while passing through the network; in other words, it can be sure about its *authenticity* and *integrity*. Furthermore, by adding a sequence number whose trustworthiness is guaranteed by the signature, it is possible to prevent *replay attacks*, which are characterized by the capture of valid packets for a subsequent retransmission to, in case no protection is in place, trigger again the same action.

Wrapping up, in case of *authentication*-level services, an attacker continues to be able to sniff information transmitted across the network, but loses the capability of injecting fake messages, which can be easily detected and dropped by the receivers. It can be considered a good trade-off since, while exchanged data may not need to remain secret, it is usually of the most importance to avoid that wicked individuals are able to trigger physical actions by means of spoofed commands.

4.1.3 Confidentiality

Finally, the most complete level is denominated *confidentiality*. It includes all the security properties offered by the previous one, thus guaranteeing authenticity and integrity of the exchanged messages, and preventing replay attacks. In addition, before the transmission of every message takes place, the payload[1] is encrypted with a cryptographic function, to prevent unauthorized parties from accessing it: data *confidentiality* is assured.

An intruder, in case of services operating at *confidentiality* level, is therefore neither able to inject messages into the network, which are discovered through signature verification, nor to sniff the transmitted data, missing the key necessary to extract the semantic meaning from random strings of bytes. Concluding, this is clearly the security level which provides the greatest number of guarantees but, at the same time, also the most expensive from the computational point of view: a careful analysis should be carried on by application designers, to decide whether the additional functionalities are worth the introduced penalties in terms of latency.

4.2 Cryptography overview

Before deepening into the analysis of the security framework, a brief overview about cryptography is presented: while certainly not being an exhaustive dissertation, it provides a description of the main concepts used in the following discussion.

Talking about cybersecurity, cryptography is clearly one of the most important building blocks, the Swiss Army knife necessary to protect private communications between applications across a computer network. It can be broadly divided into two areas, depending on the characteristics of the keys required by the algorithms.

4.2.1 Symmetric cryptography

Symmetric cryptography uses the same secret key for encryption and decryption operations, which must be known and shared between all the different parties participating in the communication. It is commonly used for protecting great amounts of data from access and modifications with relatively limited requirements in terms of computational power. Its main limitation resides in the need for a key sharing phase, which is clearly critical from the security point of view.

In the proposed solution, symmetric cryptography algorithms are extensively used for message protection, in case of services operating both at *authentication* and *confidentiality* level. The two main classes include:

[1] It is clearly not possible to encrypt the header, since its information must be accessed to complete the delivery; nonetheless, it is authenticated by the signature, and therefore it is not possible for an attacker to modify the specified parameters.

- **Message Authentication Code (MAC)**: a cryptographic checksum produced by a function which takes in input the message to be protected and the secret key; the output can be used by the receivers, knowing the same secret key, to detect modifications of the transmitted data, both accidental and intentional, and to verify that the originator owned the same key.
- **Authenticated Encryption with Associated Data (AEAD)**: a class of symmetric algorithms which combine together message encryption and MAC computation, with the usage of a unique key, to provide faster and less error-prone programming interfaces; they are particularly suitable for network packets, being able to encrypt the payload and at the same time authenticate the whole message, comprising the headers.

4.2.2 Asymmetric cryptography

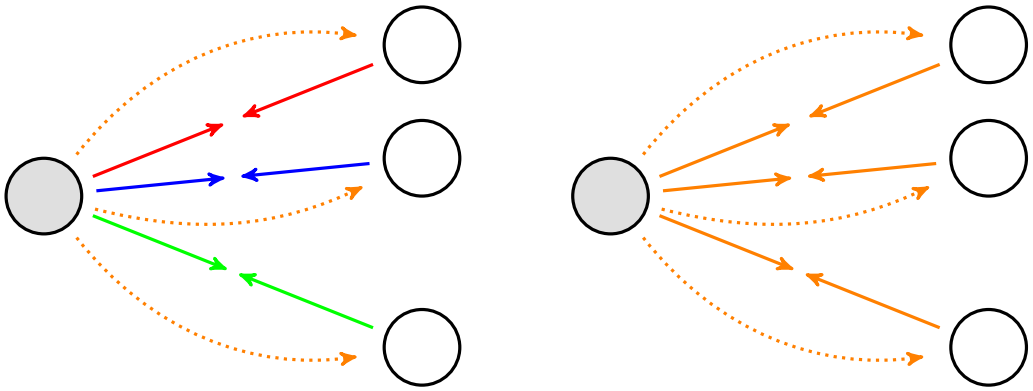
Asymmetric cryptography, also known as public cryptography, differently from the previous case, needs a pair of related keys for its operation. While being interchangeable from the mathematical point of view, the two keys are distinguished according to how they are conserved: the former, referred to as *private*, is kept secret by its owner, while the latter is made available to everybody, usually by means of a *public certificate*; it is emitted by a trusted authority and associates the *public* key with the identity of its owner.

Public cryptography is characterized by being highly computational demanding and usable only with very limited amounts of data. Within network protocols, it is mainly used during an initial handshake phase, to exchange the secret key later used for actual message protection and to authenticate the participants to the communication by computing digital signatures.

4.3 Security protocol

The core part of the conceived solution resides in the design of the security protocol which, given in input the level associated to each service, provides the actual protection to the messages across the transmissive medium.

An approach similar to the one adopted by the widespread Transport Layer Security (TLS) protocol, made up of an initial handshake phase for session establishment followed by subsequent message transmissions, has been chosen during the design. This decision, which provides a clear separation of roles between the two phases, originates from the need for the usage of different cryptographic techniques: asymmetric algorithms to authenticate the applications and verify whether they are allowed to offer or request specific services and symmetric cryptography to limit the computational time required when a packet has to be sent out. While it were possible, from the theoretical point of view, to avoid the session establishment phase



(a) Alternative 1. (b) Alternative 2.

Figure 4.1: Comparison of strategies for protection of messages belonging to a service instance — different colors correspond to different symmetric keys.

and directly authenticate network messages by means of asymmetric cryptography, large latencies would have been introduced due to the computational complexity brought in by public cryptography, exacerbated in case of embedded systems with processors of limited power.

4.3.1 Keys granularity

The desire for a solution which does not impose limitations on the SOME/IP protocol makes it necessary to consider not only unicast communications but also to allow for multicast messages, exploited for efficient delivery of notifications to all the interested subscribers. Two possible strategies can be conceived to secure messages belonging to a given service instance: on the one hand, as shown in figure 4.1a, it is possible to use a different symmetric key between the offerer and each requester, plus an additional one for multicast packets (depicted as dotted lines in figure), known by all communication parties; on the other one, represented by figure 4.1b, only a single key can be shared between all applications granted access to a specific service instance.

As a trade-off between security and complexity, the second alternative, associating a single symmetric key to each service instance, has been chosen for the designed solution: while the former provides a better isolation within a service instance, the additional complexity introduced is considered not being worth for two reasons. Firstly, symmetric keys, being automatically regenerated every time a service is started, are deemed to last only for a very limited time, reducing to a great extent the possibilities for a successful attack; secondly, in case an intruder gets the control of an application, all the services to which it has access would be anyway easily compromised, also in case different keys were used.

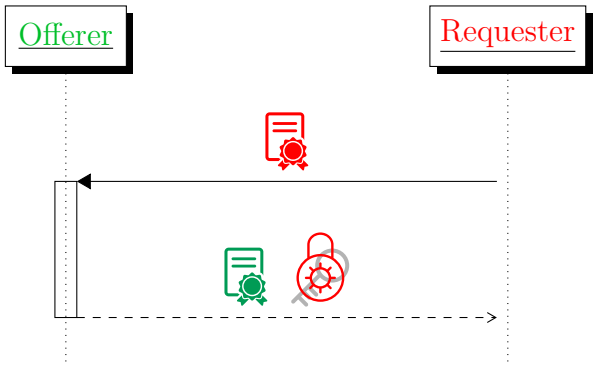


Figure 4.2: Session establishment handshake.

4.3.2 Session establishment

According to the chosen key granularity, whenever an application starts offering a service instance, a new symmetric key is randomly generated by the framework and stored within a local data structure. At this point, the session establishment phase consisting of an exchange of messages carried on separately between each application willing to require the service and its offerer, can start. It aims at mutually authenticating both parties in order to verify whether they should be allowed access or not and, in case of success, to exchange the parameters, including the secret key, necessary for subsequent communication.

Before delving into the details, it is fundamental to notice that each application needs to be accompanied by a private key and the corresponding signed digital certificate, which enumerates in a trustworthy manner all the instances of services that it is allowed to either offer or request, along with the minimum security level which must be guaranteed for each of them. For the time being, it is assumed that private keys are kept secret and accessible only to the corresponding application, while digital certificates are stored in a public area replicated on each ECU: later on, in §4.4, a more in-depth analysis is carried on about which are the security properties that must be guaranteed for the cryptographic material, and how they can be provided.

Protocol description

Figure 4.2 depicts the simple protocol that constitutes the session establishment phase, made up of two message exchanges. The process is started by the client, sending an initial message to begin the communication and pointing out its digital certificate, the manifest stating its allowed capabilities. Once the request is received by the service provider, the certificate is retrieved and, after its validation by means of the trusted root certificate, it is verified whether the handshake can continue or the request shall be denied. In case of successful outcome, the response is built up

by the server, to share its own certificate and the parameters necessary for later communications, including the symmetric key which is encrypted with the public key contained in certificate of the requester. Finally, the server affixes its own digital signature to the message and dispatches the response: the client, after having validated the certificate and verified the capabilities of the offerer, can check the digital signature and, in case of match, decrypt and store the symmetric key, which will be used for later protection of actual messages.

It is worth noting that the security level at which a service instance operates is decided preventively by the offerer and must be greater than the one stated within its own certificate: otherwise, every subsequent session authentication would fail, rejected by the clients discovering that the offerer is violating its constraints. At the same time, during the handshake, the offerer verifies the service level marked in the peer's certificate and, in case it is greater than the provided one, refuses the connection, to prevent the access to a service which is less secure than its needs.

As outlined before, two main tasks are carried on during the session establishment phase:

- **Mutual authentication**, to verify that both the server and the client have respectively the right to offer and request the considered service. It is based on the usage of asymmetric cryptography, to show the association between a digital certificate and an application by demonstrating the possession of the corresponding private key. The verification is carried on differently for what regards the server and the client: the former performs an *explicit* authentication, by signing the response with its own private key, while the latter is *implicitly* authenticated, since it needs to use its private key to decipher the transferred symmetric key.

- **Symmetric key transfer**, to let the client, in case the authentication succeeds, to acquire the secret key necessary for the subsequent communication: it is a clearly critical task since, in case an attacker obtained the cryptographic material, the protection for the whole service would be lost. To avoid being accessed by malicious parties, the symmetric key is transferred across the network in encrypted form, by using the public key contained in the digital certificate of the client: only the intended destination, in fact, owns the corresponding private key necessary for the decryption. Moreover, as explained above, the successful deciphering constitutes a proof which guarantees the authenticity of the peer.

Messages description

As seen in the previous section, the session establishment handshake follows the request/response communication pattern: being the security framework designed to be part of SOME/IP, the middleware is clearly exploited for the delivery of

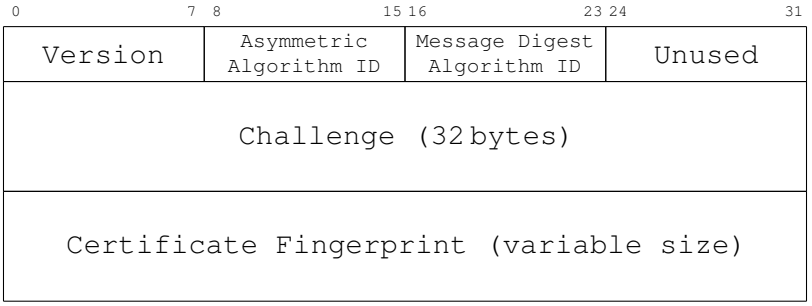


Figure 4.3: Authentication request message.

the messages. Specifically, all pieces of information are transported by SOME/IP packets, targeting the service for which the authentication is carried on and, in particular, a special method which has been devoted to the task.

Figure 4.3 depicts the fields constituting an authentication request, which are now analyzed in detail:

- `Version`: the version of the handshake protocol;
- `Asymmetric Algorithm ID`: the identifier of the asymmetric cryptography algorithm that is used during the handshake, corresponding to the type of the public key stored in the certificate;
- `Message Digest Algorithm ID`: the identifier of the cryptographic hash function that is used to compute the certificate fingerprint;
- `Unused`: reserved for future usage;
- `Challenge`: a randomly generated sequence of bytes, which is used to associate a request to the corresponding response, preventing that a valid message previously sniffed by an attacker can be replayed with success;
- `Certificate Fingerprint`: the identifier of the digital certificate of the requester; its length depends on the digest function used for the computation.

It is worth noting that, while it were possible to transfer every time the whole digital certificate, it would be certainly a waste of network bandwidth due to its considerable size (few kilobytes). Moreover, being the vehicle a closed system, it is possible to assume the placement of the necessary cryptographic material inside every ECU at applications deploy time. For these reasons, according to a well-established practice, certificates are identified by a fingerprint, a unique identifier computed by means of a cryptographic hash function.[2]

[2]A mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash) and is designed to be a one-way function, that is, a function which is infeasible to invert [20].

Figure 4.4, on the other hand, shows the structure of an authentication response message which, in addition to the initial identifiers with the same meaning as before, the challenge which is copied from the request and the fingerprint of the digital certificate of the offerer, consists of:

- `Security Level ID`: the identifier of the security level (*nosec*, *authentication* or *confidentiality*) at which the service instance is offered;
- `Symmetric Algorithm ID`: the identifier of the symmetric cryptography algorithm used to guarantee the specified security level;
- `Instance ID`: an identifier associated by the offerer to each communication party, which may be used, in case it is required by the chosen symmetric algorithm, to form the initialization vectors;
- `Encrypted Key Length`: the length of the encrypted symmetric key transmitted as the consecutive field;
- `Encrypted Symmetric Key`: the symmetric key necessary for the run-time protection of the messages, encrypted with the public key contained in the digital certificate pointed out by the request;
- `Signature Length`: the length of the digital signature transmitted as the consecutive field;
- `Digital Signature`: the digital signature computed among all the previous fields by the offerer, to guarantee the integrity of the message and demonstrate the possession of the private key.

4.3.3 Message protection

After having successfully established a secure session, it is possible to start transmitting the messages containing application data. The technique adopted for the run-time protection varies, clearly depending on the security level at which the service operates. While in case of *nosec* services, with no guarantees in place, vanilla SOME/IP messages are simply serialized, *authentication* and *confidentiality*-level packets are respectively processed by the selected Message Authentication Code and Authenticated Encryption algorithm, and the additional information necessary for later verification are appended to the message before the dispatch. Similarly, when a message is received, its level is firstly compared against the expected one; secondly, *authentication* and *confidentiality*-level packets are processed by the corresponding cryptographic function to verify their authenticity and, in the latter situation, decrypt the payload: if a mismatch is detected, the message is immediately discarded.

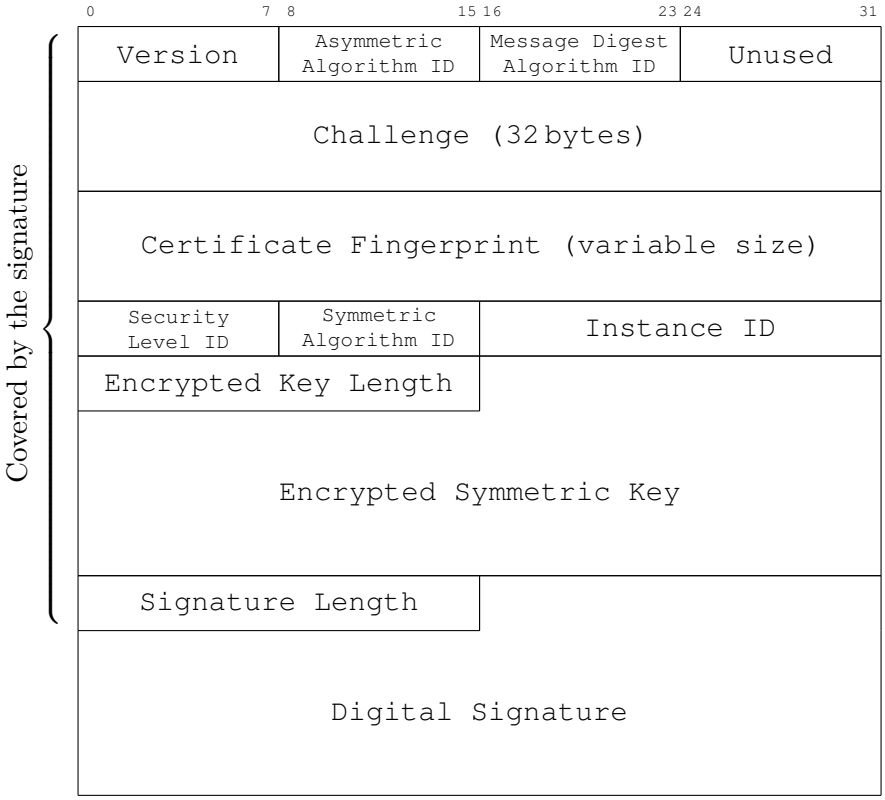


Figure 4.4: Authentication response message.

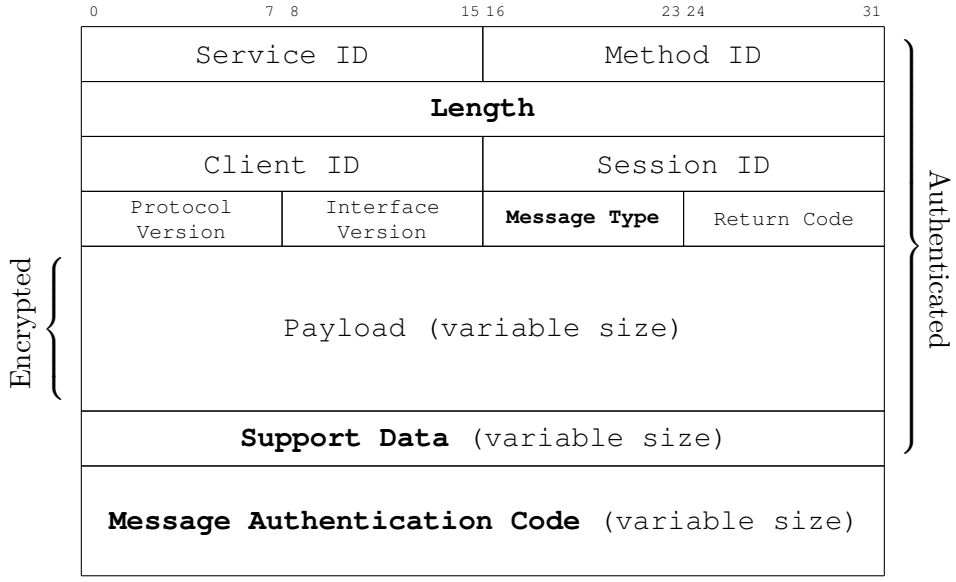


Figure 4.5: Secured SOME/IP message.

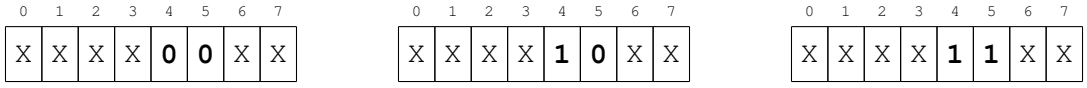
Figure 4.5 shows the format of a secured packet, highlighting in bold the differences with respect to a vanilla SOME/IP packet. While both in case of *authentication* and *confidentiality* levels the entire message, including the SOME/IP header, is authenticated, the latter provides also the encryption of the payload, which carries application data. The modifications are analyzed in the following:

- `Length`: since the secured packet comprises more information with respect to vanilla SOME/IP, the content of the length field needs to be updated to reflect the changes, in order to allow for a correct deserialization at reception side;

- `Message Type`: to specify which is the security level of the current packet, two previously unused bits of the message type field are exploited as flags, as depicted in figure 4.6;

- `Support Data`: includes all the pieces of information required to be transmitted along with the Message Authentication Code, to perform validation and decryption when the message is received; while its size and content varies depending on the adopted algorithm, it always consists of a sequence number, which can be used for replay protection;

- `Message Authentication Code`: the output of the cryptographic function, which allows the receiver to verify the authenticity and integrity properties of the message; its size depends on the specific symmetric algorithm decided by the offerer of the service instance.



(a) Nosec. (b) Authentication. (c) Confidentiality.

Figure 4.6: SOME/IP message type – Security level flags.

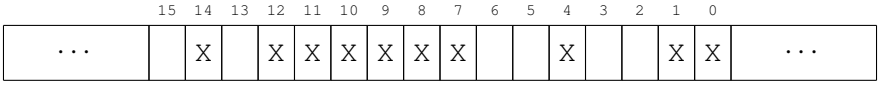


Figure 4.7: Sliding window for replay protection.

The protection from message replay is guaranteed through the usage of an authenticated sequence number, which is added to every message of service instances operating at *authentication* and *confidentiality* level. While SOME/IP already includes a Session ID which is incremented whenever a new message is sent out, its usage is mandated only when the request/response communication pattern is exploited: it is, therefore, not suitable for replay protection of notifications.

Being SOME/IP usable both on top of reliable and unreliable transport protocols, it is possible to assist to message losses and reordering: for these reasons a sliding window technique, similar to the one adopted by Datagram Transport Layer Security (DTLS) [21], is adopted. For each communication party of every service instance with an established secure session, a data structure like the one depicted in figure 4.7 is maintained: it remembers, starting from the biggest sequence number received up to now, the history of whether the messages characterized by the preceding identifiers have already been received or not. A sequence number may fall into three different positions, with respect to the sliding window:

- **to the left**, meaning it is older than the last recorded packet and shall be automatically discarded;

- **inside**: the data structure can be checked to verify whether the message has already been received or not; in the first case, the packet is rejected, otherwise it is processed and the stored information updated;

- **to the right**, signifying it is newer than all the previously received messages: in this case the sliding window moves to the right, until it comprises the newly detected sequence number.

The exploitation of a sliding window mechanism, instead of storing all the received sequence numbers, is dictated by efficiency reasons; while the choice of the length of the window itself is left implementation dependent, the mechanism shall be able to record at least the status of the last 32 packets.

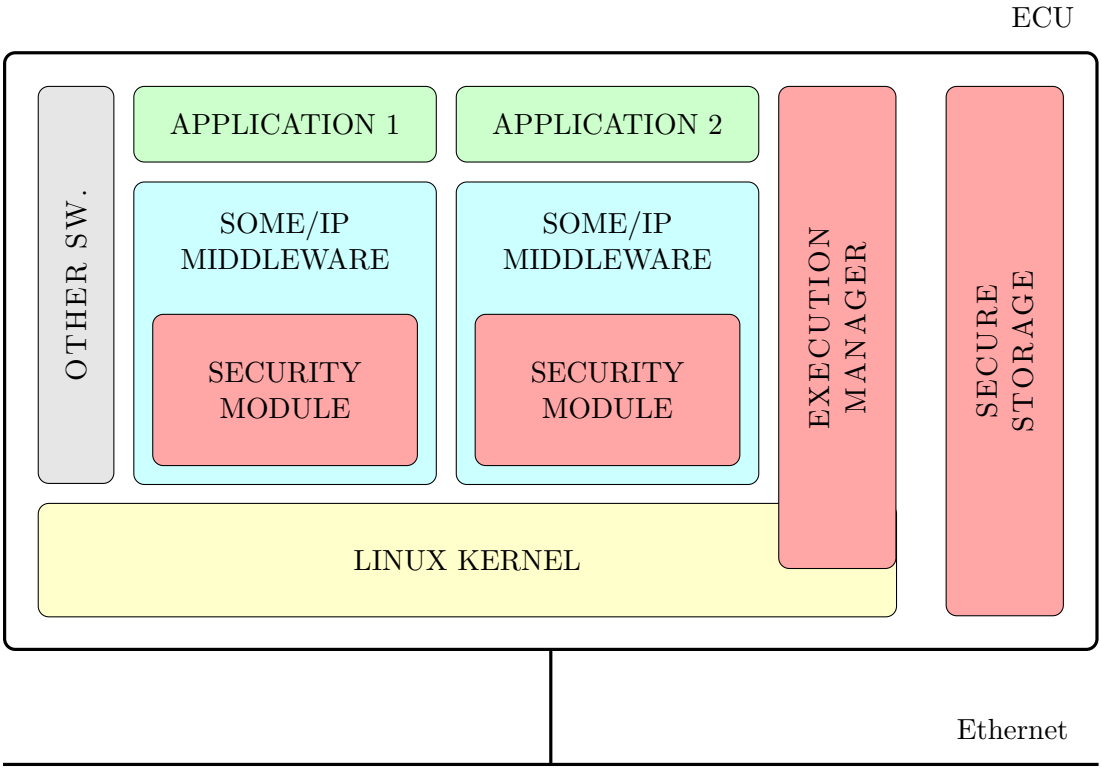


Figure 4.8: High level architecture.

4.4 High-level architecture

The conceived solution, designed to be integrated within the SOME/IP middleware, is composed by multiple modules necessary for its operation. Figure 4.8, with reference to the architecture provided by the `vsomeip` implementation, depicts a typical microprocessor-based ECU interconnected with other devices by means of an Ethernet link; internally, a Linux based operating system runs user applications exploiting the SOME/IP middleware to transparently manage the communications, as well as other software, which handles different tasks in background. Finally, the blocks highlighted in red constitute the solution which is being presented and are analyzed in the following.

4.4.1 Security module

The security module represents the core part of the solution, the one strictly bound to the SOME/IP middleware, which provides the actual protection to the messages transmitted across the network. It is designed to implement the two phases of the protocol analyzed in §4.3, thus including both the session establishment handshake and the run-time protection of the messages. Moreover, it manages all the related

aspects, such as the coordination with the other parts of the middleware and the processing of signed configuration files specifying security options.

4.4.2 Secure storage

One of the main assumptions made during the design of the handshake protocol relates to the security properties guaranteed to each piece of the cryptographic material: in case some of those fall through, the whole solution becomes useless. Firstly, private keys shall be actually kept secret by the framework, and made accessible uniquely to the corresponding application: in case an attacker took possession of one of them, he would become able to successfully carry on authentications on behalf of the legit owner and either access or provide the same services. Secondly, while applications certificates do not need special protection, being their authenticity and integrity guaranteed by means of a digital signature, the same is not true for what regards the root certificate, the one, unique for each vehicle, that contains the public key necessary to verify all the other signatures, that is self-signed. Once a wicked individual succeeded in replacing the root certificate stored within the various ECUs, counterfeit certificates would be considered valid by the framework and forbidden communications would be enabled.

While a simple solution to guarantee the required protections could involve the separation mechanisms provided by the operating system, such as the creation of a different user for each application and the careful setting of file permissions, a software protection, alone, would be inherently weak, being easily bypassable by an attacker with physical access. Stronger protection requires hardware support, providing tamper-resistant storage and burned-in keys that can be exploited to build up, together with software functionalities, the necessary guarantees in terms of security and trustiness. Being cryptographic hardware modules already quite widespread in the general computing world, standardized as Trusted Platform Modules and offered in different flavors by all the main CPU vendors, the topic is not deepened further in this discussion and it is assumed that available technologies are able to secure the required cryptographic material.

4.4.3 Execution manager

The final element considered in the presented architecture is the one named execution manager, a software module acting as an interface between the security module and the hardware storage, to abstract the low-level and platform-dependent aspects inherent the access of the protected cryptographic material. It is conceived to cooperate with and exploit the functionalities provided by the hardware support to authenticate the applications and the whole framework at start-up time, granting access to the corresponding private keys only in case all the elements correspond to the expected ones. Also in this case, while the thematic is so wide to require

an extensive dissertation, for the current analysis it is not deepened anymore, concentrating the efforts on the security module.

4.5 Limitations

While being carefully designed to try to reduce as much as possible the vulnerabilities, the solution presented in this discussion is characterized by a few weak points, both due to the trade-offs necessary to avoid a too heavyweight solution and the implied limitations of every security protocol.

Primarily, in addition to the considerations already performed in previous sections and especially about the service instance granularity (§4.1), it is worth noting that the conceived solution does not provide any type of protection against Denial of Service (DoS) attacks, which try to make applications and services unavailable by overloading them with tons of invalid requests. Moreover, while the introduction of cryptographic functions, constituting the backbone of the security framework, is mandatory to protect the data transmitted across the network, it accentuates the risk of DoS attacks: being relatively expensive from the computational point of view, in fact, the additional mechanisms may become targets of attacks which aim at overloading the ECUs to prevent the execution of the actual applications. Possible malicious behaviors may target both the session establishment phase, by flooding a huge amount of authentication requests to force the offerer performing useless but expensive computations to provide a response, and the run-time protection, by overwhelming the framework with tons of unauthenticated SOME/IP packets. While DoS attacks can be the cause of many problems, it is actually hard to find possible countermeasures able to prevent them without, at the same time, exposing new vulnerabilities which can be exploited for the same purpose.

One of the SOME/IP functionalities for which no protection is provided by the conceived solution is the service discovery module: while, due to its importance, it could certainly be the target of a possible attack to redirect legit requests towards malicious applications, the authentication of SOME/IP–SD messages would increase the computational load of the devices and introduce new possible vulnerabilities for a DoS attack without, at the same time, increasing the overall security. Hijackings, in fact, are already prevented by the mutual authentication carried on during the handshake phase, which prevents the connection if one of the two peers is not authorized to either offer or request the specific service, depending on its role.

Finally, being the considered solution conceived to operate within the SOME/IP middleware to secure communications between user applications, it does not provide protection against attacks targeting lower levels of the network stack, like IP, TCP and UDP, either aiming at preventing the correct transmission and reception of messages or trying to exploit vulnerabilities in smaller companion protocols to make the victim ECU execute malicious code.

Chapter 5

PoC Implementation

This chapter, which presents the main insights about the implementation of the solution presented in chapter 4, is divided into three broad parts. Firstly, being cryptography the backbone of the conceived solution, a comparison about different cryptographic libraries and algorithms, corroborated by the results of performance measurements, is proposed to justify the preliminary decisions took at the beginning of the implementation process. Secondly, a high-level description of the modules developed within the `vsomeip` framework is presented, to show the designed features that have been implemented as a proof of concept, together with the main developed abstractions. Finally, an overview of the configuration options is presented, to show how it is possible to both specify the level at which each service instance operates and configure the broad range of security parameters influencing the behavior of the implementation under discussion.

5.1 Cryptography

Before starting the actual implementation of the conceived solution within the `vsomeip` framework, an analysis about the different available possibilities for the computation of the necessary cryptographic functions has been carried on for two main reasons. On one side, being the designed security protocol deeply based on the usage of cryptographic algorithms and the target of the conceived solution embedded systems with possibly limited computational capabilities, it becomes fundamental to select an efficient and highly optimized implementation to avoid the introduction of unnecessary performance penalties. Secondly, while the subsequent PoC development has been driven by the usage of abstractions hiding as much as possible low-level details, the selected cryptography library is a so important building block that, guiding some architectural choices, becomes with no doubt difficult to be replaced at a later stage. The initial decision about a specific algorithm, on the other hand, is certainly less fundamental, being possible to provide multiple

implementations and let the possibility to select the most suitable one by means of the configuration: nonetheless, for the sake of completeness, a comparison between different alternatives is also depicted.

5.1.1 Cryptography libraries

From an initial scouting phase, narrowed down both by the desire of avoiding any bound with proprietary software for the proof of concept and by the requirement for `C` or `C++` APIs mandated by the `vsomeip` implementation, four different open-source candidates primarily emerged for a more in-depth comparison, which are briefly described in the following.

OpenSSL,[1] a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is made up of two modules: on one side `libcrypto`, a general-purpose cryptography library developed in `C` and implementing a wide range of algorithms accessible through both low-level and more consistent high-level interfaces; on the other one `libssl` that, combining the functionalities of the core module, provides an open-source implementation of the SSL and TLS protocols. It is licensed under a permissive Apache-style license.

LibreSSL,[2] a cryptography library and TLS implementation forked from OpenSSL in 2014 by the OpenBSD project, with goals of modernizing the codebase, improving security, applying best practice development processes and removing obsolete or broken features. It is characterized by a similar structure and API with respect to OpenSSL and is licensed under BSD-style Open Source licenses.

CryptoPP,[3] a free `C++` library for cryptographic schemes including ciphers, message authentication codes, one-way hash functions and public-key cryptosystems. The library is licensed as a compilation under the Boost Software License 1.0, while the individual files are all public domain.

Botan,[4] a cryptography library written in `C++11`, which implements a wide range of symmetric and asymmetric cryptographic algorithms as well as high-level, ready to use, protocols such as the well-known TLS; it is licensed under the permissive Simplified BSD license.

[1] `https://www.openssl.org/`

[2] `https://www.libressl.org/`

[3] `https://www.cryptopp.com/`

[4] `https://botan.randombit.net/`

5.1.2 Cryptography algorithms

Before presenting the methodology used during the benchmarking phase and analyzing the obtained results, a brief discussion about the cryptography algorithms chosen for the comparison is provided. In order to see the overall picture, a common subset of candidates provided by all the libraries and belonging to the three categories exploited by the security framework has been identified, by considering well-established and widespread competitors. For a fair comparison, moreover, the same security strength, corresponding to 128 bits, has been considered in every case,[5] by adapting the key length according to the characteristics of the algorithms.

The first investigated grouping embraces authenticated encryption, the class of algorithms belonging to the symmetric cryptography branch that are used to guarantee data authentication, integrity and confidentiality. Two different candidates have been considered for this initial evaluation, AES-CCM and AES-GCM: they are both modes of operation combined with the ubiquitous Advanced Encryption Standard (AES) block cipher, that can operate with data of any size.

Secondly, algorithms computing Message Authentication Codes have been taken into account, due to their usage in protecting communications of services operating at *authentication* level. In this category, the two selected candidates are represented by the authentication-only variants of the ones chosen for the previous category, which are used without feeding data to be encrypted.

Finally, the last considered category includes asymmetric cryptography algorithms fundamental during the session establishment phase described in §4.3.2; only the well-known RSA has been investigated as a candidate for this grouping, due to the lack of possible competitors with comparable functionalities and speed.

5.1.3 Benchmark methodology

After having identified the candidates for the comparison, four different `C++` applications, corresponding to the considered libraries, have been developed and, for each different algorithm, a function following the logic depicted by the pseudocode in listing 5.1, which refers to authenticated encryption, has been implemented. Trying to remove as much as possible potential factors of interference, the core instructions are executed repeatedly within a loop and the total elapsed time is considered in the throughput computation; moreover, all plaintexts are generated in advance and initialization vectors, if necessary, are kept fixed during all the different iterations. To represent various message sizes, functions are executed with four different values for the plaintext length, which are deemed to be quite representative: 16 bytes,

[5] Actually, this statement is not true for what regards RSA that, if a 2048 bits key is used as in this case, provides only 112 bits of equivalent security; nonetheless, the obtained results are not invalidated since no comparison is performed between symmetric and asymmetric algorithms.

Listing 5.1: Authenticated encryption benchmark function.

```
1  function bench_algorithm(algorithm, iterations, size)
2  begin body
3      plaintexts <- random_plaintexts(iterations, size)
4
5      start_time <- now()
6      for i from 1 to iterations
7          (ciphertext, MAC) <- encrypt(algorithm, plaintexts[i])
8          (plaintext, auth) <- decrypt(algorithm, ciphertext, MAC)
9          assert(auth is true)
10     end for
11     end_time <- now()
12
13     print_throughput(start_time, end_time, iterations, size)
14 end body
```

64 bytes, 256 bytes and 1024 bytes; in case of asymmetric algorithms, only the latter is considered since, for small sizes, the difference between one and the other is negligible. Finally, it is worth mentioning that, for every cryptography library, the latest stable version available at the time has been downloaded and compiled with out-of-the-box settings; moreover, applications have been built all in release mode with always the same set of compilation flags.

Since the PoC under implementation is thought to be run mainly on embedded systems, benchmarks have been executed on two different environments:

- a laptop constituted by an Intel Core i7-4600U CPU operating at 2.10 GHz, 8 GB of RAM and running Debian Stretch as operating system;
- a development board, characterized by two ARM Cortex-A7 cores operating at 1.20 GHz, 1 GB of RAM and executing an embedded Linux distribution built through Yocto Project.[6]

As an effort in trying to further reduce possible interferences, CPU scaling functionalities have been temporarily disabled on both systems and other applications or unnecessary background tasks terminated. Each benchmark has been executed ten times specifying as a parameter the number of iterations for the main loop of each algorithm, selected as a tread-off to achieve good precision while limiting the execution time: 100 000 and 1000 repetitions have been chosen respectively for symmetric and asymmetric algorithms in case of the x86 device, while they have been reduced by ten and five times each for the less powerful ARM-based board. The results have been post-processed by means of a custom script to compute, for each library, algorithm and size, the average execution time $\bar{t}$ and the corresponding standard deviation σ_t, calculated according to equation (5.1); finally, the throughput

[6] https://www.yoctoproject.org/

value has been derived, together with the associated standard deviation, and plotted for easier comparison.

$$\sigma_t = \sqrt{\frac{\sum_{i=1}^{N}(t_i - \bar{t})^2}{N}} \tag{5.1}$$

5.1.4 Results evaluation

Considering the results depicted in figures 5.1 to 5.6, it emerges at a first glance the huge difference in terms of performance between the laptop with a x86 processor and the ARM based embedded system, which justifies the considerations made during the design phase about the necessary trade-offs to avoid the introduction of excessive overheads.

Looking more in detail at the different libraries, and limiting the analysis to x86 benchmarks, it emerges that, both for what regards authenticated encryption and MAC algorithms, OpenSSL and LibreSSL achieve similar throughputs, except in few cases where the former tends to be more optimized; Botan and CryptoPP, instead, are characterized by performance limited, in most situations, to less than one half the ones provided by the competitors. Finally, considering asymmetric algorithms, OpenSSL is again the leader, with CryptoPP chasing and the other libraries more distant. Switching to ARM, although the ranking is confirmed in most situations, OpenSSL emerges as being the most optimized library providing, in many cases, twice as much throughput with respect to the others. Concluding the analysis, it is clear that OpenSSL, being particularly performing both with x86 and ARM processors, is the chosen library for the PoC implementation.

Before going on, a further comparison has been performed, to select the algorithms offering the best throughput, both in case of authenticated encryption and message authentication codes. For what regards the first category, CHACHA20-POLY1305, a combination of two algorithms designed to be efficient when implemented in software, and AES-OCB have been added to the previously considered competitors; the second group, instead, saw the inclusion of both the authentication-only variant of CHACHA20-POLY1305 and the HMAC algorithm combined with the widespread SHA256 cryptographic hash function. Figure 5.7 shows the results of the benchmark in case of the x86 processor, where AES-GCM, and its GMAC variant, emerge as clear winners in both categories. Instead, considering the results obtained with the embedded system, shown in figure 5.8, it is evident the efficient software implementation of CHACHA20-POLY1305 which, especially in case of bigger message sizes, is two or three times more performing with respect to the other competitors. Wrapping up, two different algorithms, AES-GCM and CHACHA20-POLY1305, have been selected for the implementation due to the provided throughputs: they are both able to protect service instances operating at either *authentication* or *confidentiality* level.

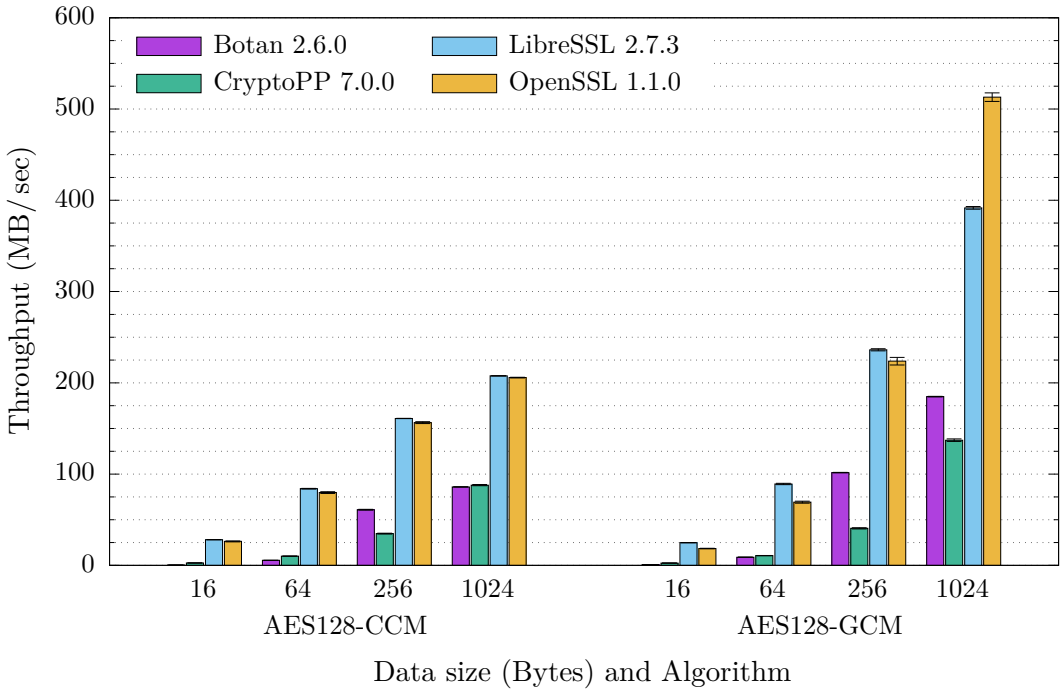


Figure 5.1: Libraries benchmark — Authenticated Encryption — x86

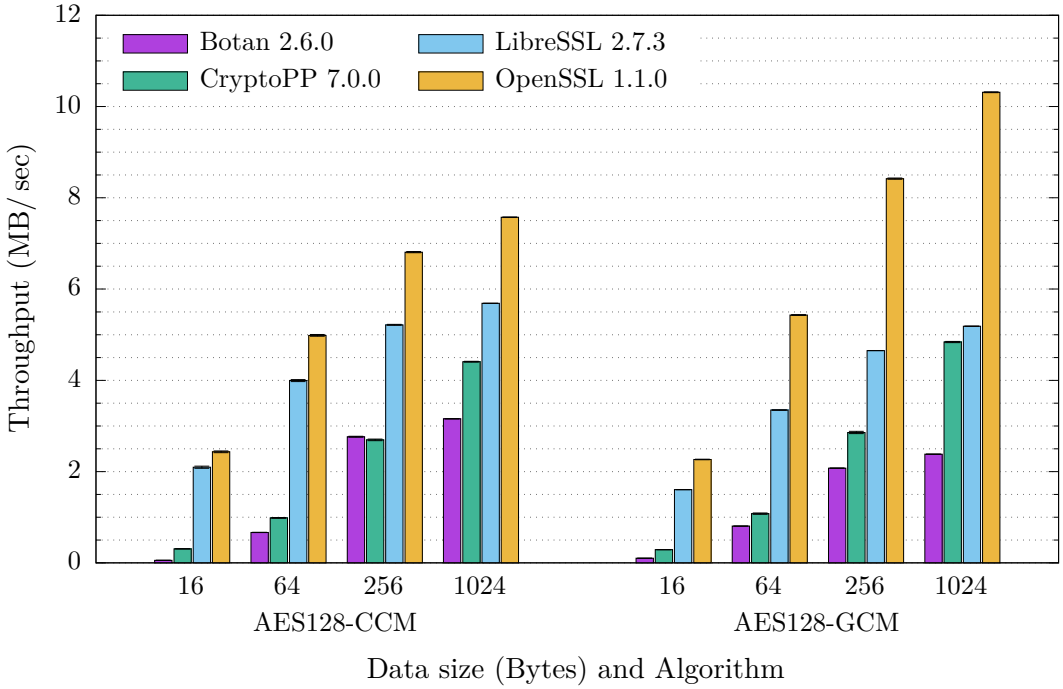


Figure 5.2: Libraries benchmark — Authenticated Encryption — ARM

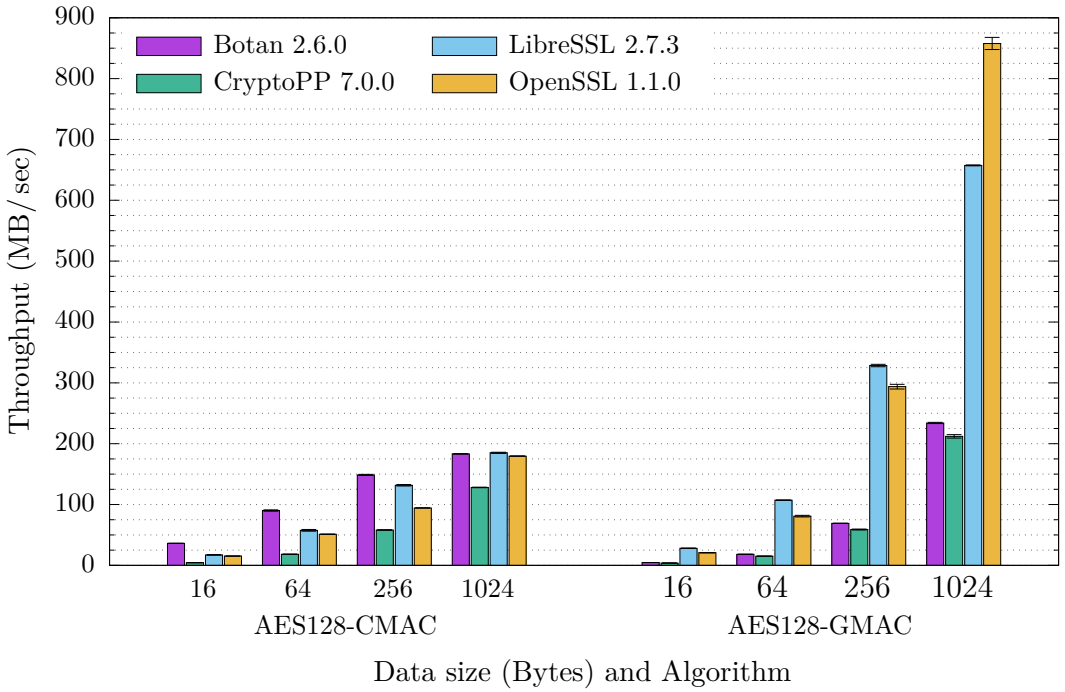


Figure 5.3: Libraries benchmark — MAC algorithms — x86

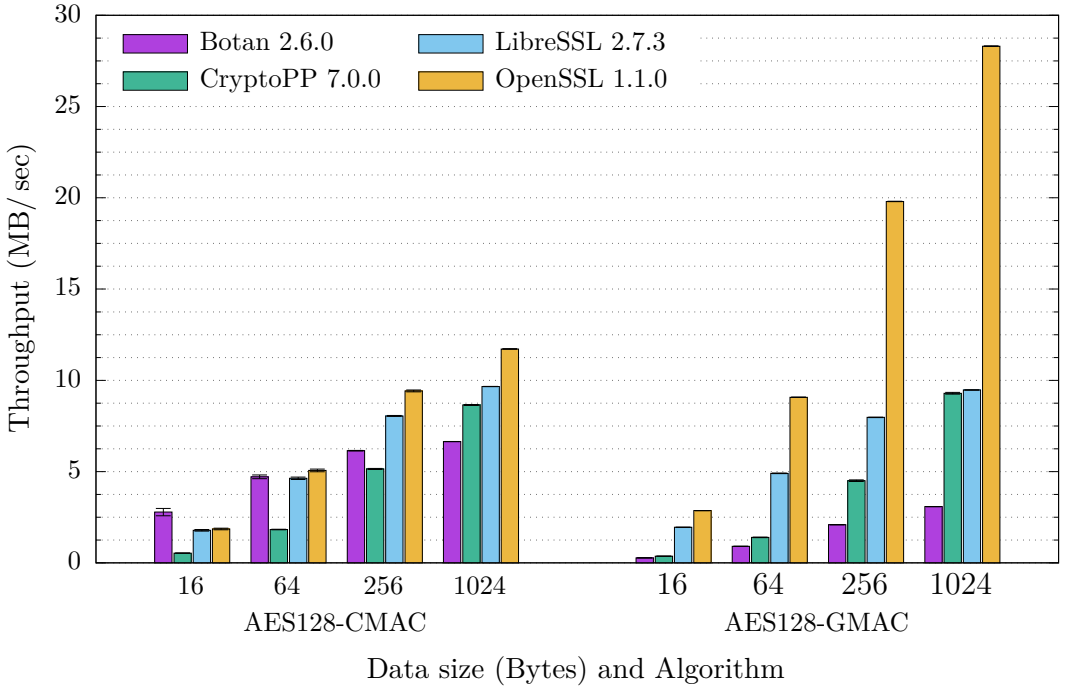


Figure 5.4: Libraries benchmark — MAC algorithms — ARM

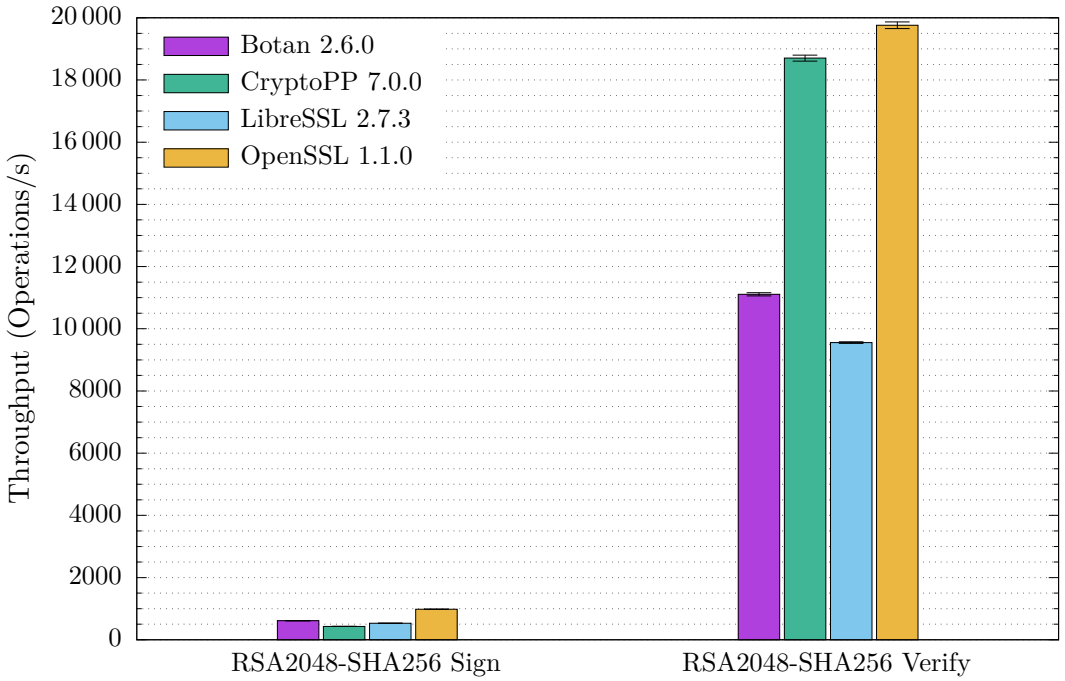


Figure 5.5: Libraries benchmark — Asymmetric algorithms — x86

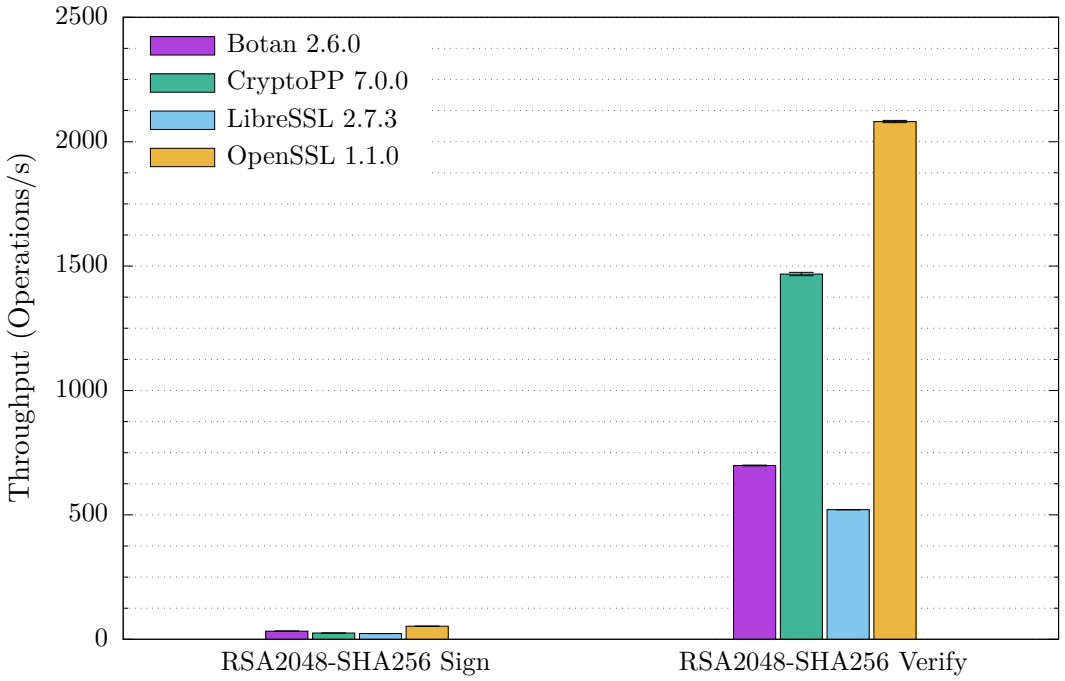


Figure 5.6: Libraries Benchmark — Asymmetric algorithms — ARM

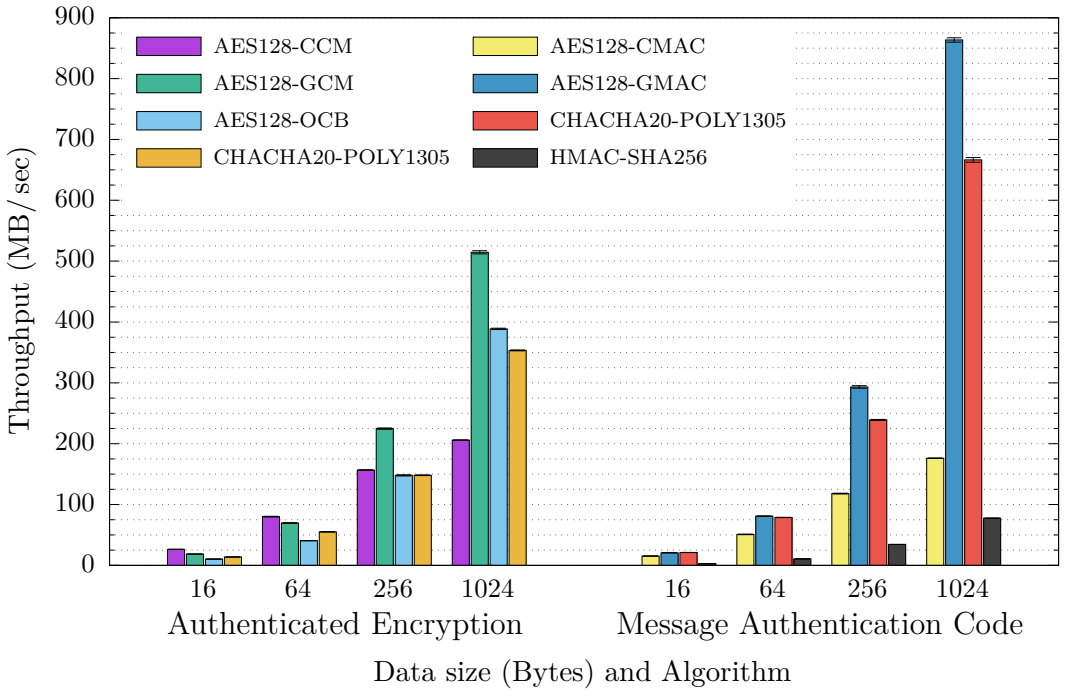


Figure 5.7: Symmetric cryptography algorithms benchmark — x86

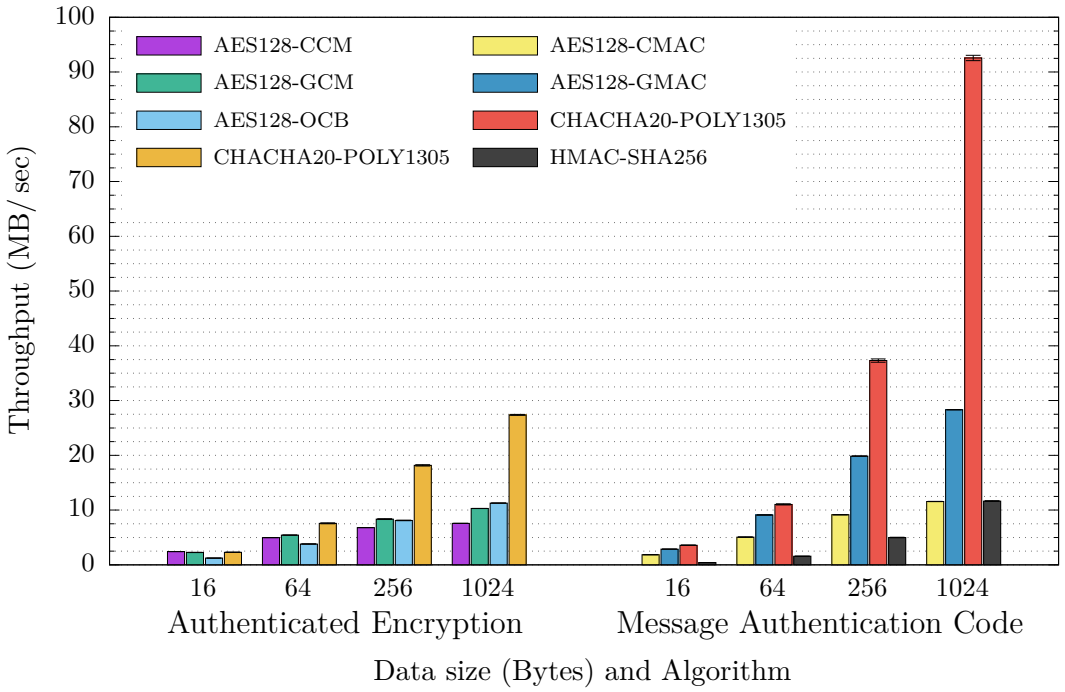


Figure 5.8: Symmetric cryptography algorithms benchmark — ARM

5.2 Functional modules

After having decided the cryptography library and algorithms to be used, the actual implementation as a proof of concept of the conceived security framework took place. In particular, most of the efforts have been concentrated on the integration of the functionalities designed as part of the security module, including the authentication phase and the run-time protection, within `vsomeip`; moreover, an entire new group of configuration options, described more in detail in §5.3.2, has been introduced, to let the possibility to customize various aspects of the provided solution.

As already stated before, for what regards the PoC implementation, cryptographic material, comprising the root certificate and private keys, has been assumed to be protected by means of operating system facilities. While, as described in §4.4.2, only limited guarantees are provided this way, the decision strongly simplifies the initial development, leaving more time for working on important parts of the solution instead of concentrating on hardware-dependent technological aspects.

Back to the description, the implementation can be divided at a high-level into different functional modules, made up of tightly bound classes cooperating to provide a specific set of functionalities, which are briefly analyzed in the following.

5.2.1 Cryptography abstraction

Being the depicted solution highly based on cryptography, a quite significant number of classes has been designed to abstract the various functionalities, decoupling as much as possible the interface from the actual implementation, which is clearly based on the APIs provided by OpenSSL.

Considering symmetric cryptography, for example, two different interfaces have been designed, abstracting respectively the concepts of authenticated encryption and authentication-only. They allow the transparent coexistence of multiple implementations based on different algorithms, hiding specific and low-level aspects. Moreover, every derived class may choose to tackle the replay protection differently to exploit, if possible, part of the initialization vector as a sequence number.

Similarly, in case of asymmetric cryptography, different interfaces are used to represent private and public keys, along with the functionalities enabled by each one. The same occurs for what regards digital certificates, defining the methods available to both access the contained information, according to the format described in §5.3.1, and verify their authenticity, by traversing the chain of truth. Furthermore, the concept of certificate store has been introduced, to keep in cache the digital certificates already loaded during the current execution and prevent the need for repeated validations, which are expensive from the computational point of view.

Finally, a set of enumerations has been introduced, to associate numeric identifiers to the various algorithms, and utility methods provided for easy conversion from and to the human-readable format used in configuration files and logs.

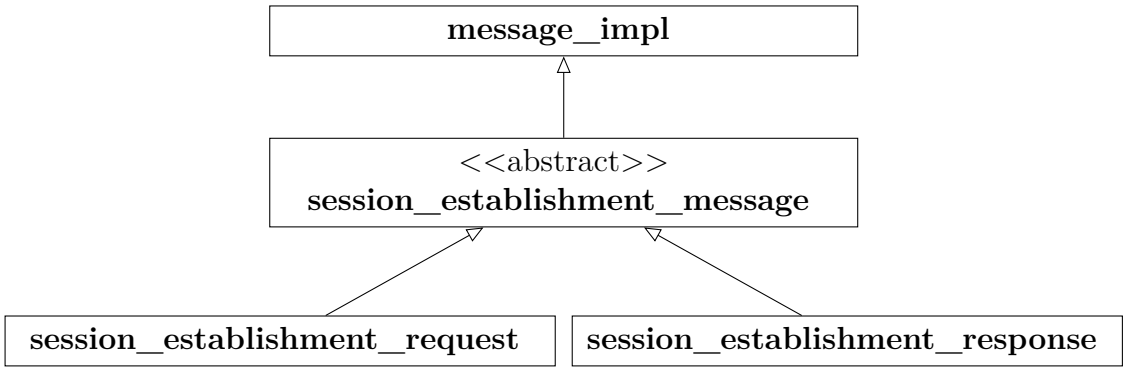


Figure 5.9: Session establishment messages inheritance diagram.

5.2.2 Session establishment

The first core functionality provided by the PoC is the one implementing the session establishment phase according to the protocol designed in §4.3.2. It is integrated directly within the `routing_manager_base` class, to detect the actions requested by the applications and react consequently. For instance, when the `offer_service` method is executed, the framework performs an initial check to verify, according to the digital certificate associated to the application,[7] whether the operation is allowed and which is the minimum security level to be guaranteed: depending on the outcome, either the session parameters are initialized or the request is rejected. Similarly, for the client side, the execution of the `request_service` function triggers a verification, which can culminate into the block of the request itself: nonetheless, also in case of positive outcome, no authentication messages are sent until an available instance of the requested service is detected. When this event occurs, the message handshake mandated by the protocol can start: it is managed entirely within the routing manager, in a transparent way from the applications point of view, by intercepting the packets targeting the special method devoted to the specific purpose. If the authentication phase succeeds, the client obtains the session parameters necessary for the communication and the service availability is advertised to the application. Otherwise, different behaviors may occur depending on the situation: in case either an authentication request or a response is fallacious, or the peer is not authorized to perform the specific operation, the process is immediately aborted, with no further exchanges. If no authentication response is received from the offerer, on the other hand, the client may retry establishing the session multiple times, according to the configured parameters, before giving up.

[7] It is worth noting that, in this initial phase, the framework does not verify whether the application has access to the private key corresponding to the certificate, which is instead checked during the mutual authentication phase.

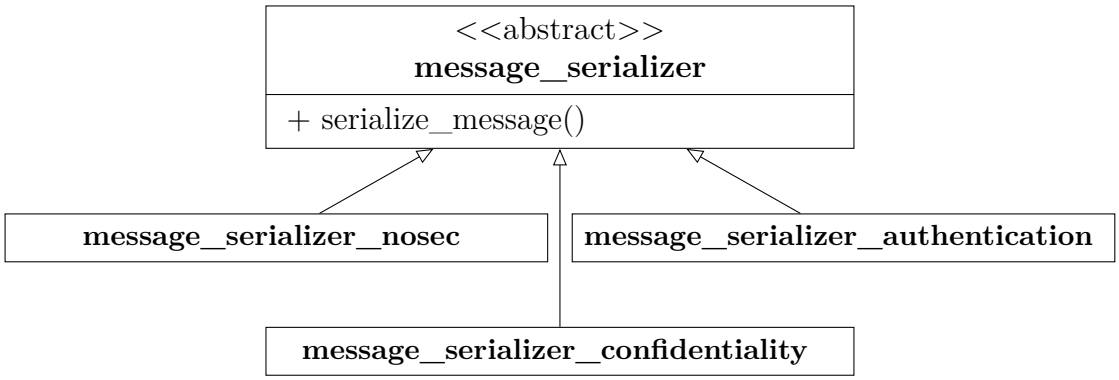


Figure 5.10: Message serializer inheritance diagram.

Figure 5.9 shows the inheritance diagram of the main classes used to represent the messages exchanged during this phase: the root is formed by `message_impl`, already present in `vsomeip`, which abstracts a standard SOME/IP packet by providing accessors for the various header fields, together with `serialize` and `deserialize` methods to convert to and from the on-wire format. Immediately below, it is introduced the `session_establishment_message` abstract class, which implements the common part of authentication messages, present both in requests and responses, and its serialization, by extending the above-mentioned methods. Finally, the two leaves of the diagram represent the actual messages: the response, in particular, provides the functionalities to transmit and retrieve the session parameters, including the encrypted symmetric key, necessary for the subsequent run-time protection.

5.2.3 Message protection

The second fundamental feature included inside the PoC consists in the run-time protection of all exchanged messages, according to the specifications provided in §4.3.3. It is implemented through two main class hierarchies: the one depicted in figure 5.10, and the symmetrical one devoted to message deserialization.

With reference to the mentioned diagram, the root of the inheritance is constituted by `message_serializer`, an abstract class which encapsulates a vanilla `serializer` object, as provided by `vsomeip`, and provides a single method, receiving in input the message to be processed and returning in output the stream of bytes ready to be transferred. According to the instantiated specialization, which depends on the parameters exchanged during the session establishment phase, the behavior of the function changes, to guarantee the corresponding security level. Considering `message_serializer_authentication`, for instance, it performs two main operations during serialization: firstly, the encapsulated `serializer` object is exploited to convert the message into raw bytes and the header is modified

according to the specifications, by setting the corresponding flags and adapting the length value. Secondly, serialized data is processed by the object implementing the symmetric cryptographic algorithm associated to the current session and the additional pieces of information, including both the initialization vector and the actual Message Authentication Code, are appended to the original stream of bytes, ready to be dispatched.

5.3 Security configuration

One of the main driving aspects emerged during the initial design of the solution under analysis is related to the usage of high-level rules to define the security guarantees associated to each service. In the following, a brief analysis of the possibilities for configuring security-related aspects of the PoC implementation is presented, to show how the defined goal has been reached thanks to the integration of the functionalities within SOME/IP, which abstracts the low-level aspects of the network communication.

Two different techniques have been adopted to fulfill the task: on one side, information related to the capabilities of each application are directly embedded within its associated digital certificate. On the other one, `json` files, already used for `vsomeip` configuration, are extended with additional security options: moreover, to guarantee the authenticity and integrity of the contained information, a digital signature is stored and verified every time the file is read by the framework.

5.3.1 Digital certificates

Being the security framework based on the concept of defining for each application the associated capabilities, in terms of services allowed to be offered or requested, it is clearly necessary to provide the possibility to specify the needed information in a simple and trusted way. It is, in fact, an essential requisite to guarantee that the configured values cannot be altered by a possible attacker, otherwise the whole provided security would fall down like a house of cards. Since the parameters under examination are exploited during the session establishment phase, which is highly based on the usage of asymmetric cryptography to perform the authentication, it is deemed to be quite straightforward the storage of security information directly within the digital certificate associated to each application, whose trustworthiness is certified by an authority and verified before extracting the contained data.

More in detail, with reference to the X.509 specifications defining the format of public certificates, a specific extension, denominated Subject Alternative Name (SAN), has been exploited for the purpose. It allows listing, within the certificate, a set of names, according to different possible standards, that are bound to the stored public key: a URI-like format, depicted in figure 5.11, has been chosen to list

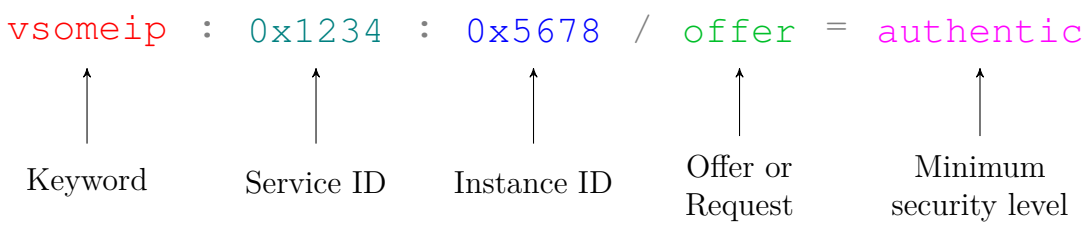


Figure 5.11: Format of a `vsomeip` SAN entry inside the digital certificate.

the capabilities of each application. It is composed by five different elements: the initial `vsomeip` keyword, which characterizes the type of the entry, followed by the identifiers of the considered service instance, according to the SOME/IP format, and the capabilities associated to the application. In particular, it is specified whether the application is allowed to either `offer` or `request` the service and the minimum security level that must be guaranteed during the communication. No bounds are dictated about the number of entries that can be present within a certificate, which are only influenced by how many different services can be accessed by the considered application.

Furthermore, two additional digital certificates are requested for the correct operation of the framework: the former, already mentioned different times, is the self-signed root certificate which is used to validate all the others. The latter, on the other hand, contains the public key necessary to verify the signatures affixed to configuration files and is identified by the `vsomeip:configuration-signature` Subject Alternative Name.

While configuration parameters considered up to now clearly adhere to the demand for high-level rules, criticism may arise about the complexity of embedding actual entries, according to the presented format, within digital certificates. Nonetheless, it is deemed to be trivial the development of a GUI-based application, which abstracts the whole process by hiding low-level details.

5.3.2 Configuration files

While the core security rules are specified through digital certificates, as explained in the previous section, a wide range of additional parameters may be tuned by means of extended `vsomeip` configuration files. In the following, the various security options available are briefly described, referring to the example shown in listing 5.2; before continuing, it is worth noting that fingerprints and signatures, shown in a shortened format to avoid lengthy lines, must be written in full inside the actual `json` files for being accepted. Configuration options can be broadly divided into three groups, corresponding to the main `json` objects shown in the example.

Listing 5.2: Security configuration options example.

```
...
"applications" : [{
    "name" : "example-application",
    "id" : "0x0001",
    "private-key-path" : "./example-application.key",
    "certificate-fingerprint" : "910F4D028E63CD8B..31A8A2B53A900163",
    "application-fingerprint" : "A0EEF580CAE3B616..A6D1D0AC87542FFA"
}],

"service-security" : {
    "certificates-path" : "/var/certificates",
    "root-certificate-fingerprint" : "63ED9DB6A52A4B5C..1414A5E72EB5F3C5",

    "services" : [{
        "service" : "0x1234",
        "instance" : "0x5678",
        "security-level" : "confidential",
        "security-algorithm" : "aes-gcm-128"
    }],

    "default-algorithms" : [{
        "security-level" : "authentic",
        "security-algorithm" : "chacha20-poly1305-256"
    },{
        "security-level" : "confidential",
        "security-algorithm" : "aes-gcm-256"
    }],

    "repetitions-max" : "3",
    "repetitions-delay" : "1000" ,
    "repetitions-delay-ratio" : "2.0",

    "check-application-fingerprints" : "true"
},

"configuration-security" : {
    "signature-algorithm" : "rsa2048-sha256",
    "certificate-fingerprint" : "08F7338656BEDEBE..36A623EC3720E33A",
    "signature" : "5507C9115B288C167C830875AD328081..9B84494A671AD986C45D2733DA4C633E"
}
...
```

- `applications`, enumerating additional parameters strictly related to the actual applications. Three main pieces of information may be specified: on one side, it is necessary to tell the framework where it is possible to retrieve the required private key, along with the identifier of the corresponding digital certificate. On the other one, it is possible to specify the expected fingerprint of the application: if the optional verification is enabled, the framework compares the indicated value against the actual one, and the execution is aborted in case a mismatch is detected. While no strong guarantees are provided, since libraries are not checked for genuineness, it implements an initial form of authentication of the applications.

- `service-security`, providing the possibility to redefine most of the parameters influencing the behavior of the framework: nonetheless, two pieces of information are mandatory, to specify both the folder containing all the necessary certificates and the fingerprint identifying the root one. Additionally, for what regards each offered service instance, it is possible to configure both the security level, which cannot be lower than the one mandated by the certificate, and the cryptographic algorithm chosen for the protection. If no explicit settings are present, the framework defaults in one case to the value advertised by certificate, and in the other one to either the algorithm associated to the given security level or, if not provided, to a predetermined choice. For what concerns requested services, on the other hand, it is possible to modify the behavior in case an authentication response is not received, by specifying the maximum number of requests that can be performed before giving up and the time which should elapse between one and the subsequent. Finally, it is possible to decide whether the verification of the application fingerprint is enabled or not, by setting the corresponding property.

- `configuration-security`, specifying the information required to verify the authenticity and integrity of the whole `json` file. In particular, it points out the asymmetric cryptography algorithm used for the computation and the digital certificate containing the public key necessary for verification. Furthermore, the actual signature is stored: it is computed among the entire file, with the signature value replaced by a number of zeros corresponding to its expected length. After having loaded a configuration file, the security framework verifies the genuineness of the contained signature and, in case of mismatch, the execution is immediately terminated.

Chapter 6

Experimental Evaluation

This chapter, which can be broadly divided into three main parts, examines the methodologies adopted to evaluate the functionalities of the developed proof of concept and presents the results of performance measurements. Firstly, a brief description of the automatic tests part of the `vsomeip` bundle is provided, together with an analysis about the incompatibilities with the implemented security framework. Secondly, the benchmark methodology is depicted, showing the results of quantitative measurements carried on to evaluate the penalties introduced by the additional features. Finally, the design of the demonstrator, realized to exhibit the implemented security functionalities in a simple and effective way by mimicking the variety of ECUs present in modern vehicles, is presented to the reader.

6.1 Automatic testing

While, during the initial phases of the PoC implementation, simple application examples, based on both request/response and publish/subscribe communication patterns, have been exploited to check the correct functioning of the additional modules, a more systematic testing phase has soon become necessary. Luckily, a set of test cases, made up of more than one hundred units implemented by means of the Google Test framework,[1] is already shipped together with the `vsomeip` source code. They are mostly composed by two or more applications, communicating between one another either locally or across the network according to the possibilities provided by the `vsomeip` framework; moreover, various combinations of configuration files are exploited, to cover a wide range of available settings.

Since a great amount of test cases needs to be run on two different devices interconnected by a network, the necessary testing environment has been deployed before starting the actual execution of the applications. For the sake of simplicity,

[1] `https://github.com/google/googletest`

virtualization has been exploited to prevent the need for a complex staging: two lightweight Docker containers have been created within the development machine and interconnected through a virtual switch, to mimic the expected configuration. Subsequently, all the `json` files used for the set-up of the `vsomeip` framework have been modified, by including the pieces of information necessary to enable the security functionalities. To prevent unneeded complexities, a single certificate allowing access to all the service instances part of the tests has been created: while being clearly a bad practice from the security point of view, it is deemed to be a fully acceptable strategy to be adopted during this testing phase, which aims at verifying the global functioning of the framework.

Completed the initial set-up of all the necessary elements, the actual test and fix phase could start. Multiple iterations of the process have been performed for each failing test case: unnecessary limitations have been removed and correct operation enabled also for specific corner cases overlooked during the initial analysis of the `vsomeip` source code. Finally, after having completed the procedure, it is possible to state that the developed proof of concept is almost completely compatible with vanilla `vsomeip`: only a handful of unit tests is still failing, due to reasons explained in the following, which mainly depend on the introduced security guarantees.

Starting the analysis about problematic tests, it is necessary to mention that, in few situations, they have been implemented by sending hand-crafted SOME/IP packets through raw endpoints, without exploiting the abstractions provided by the framework. It is obvious that, being the security functionalities designed exactly to prevent the communication with unauthenticated parties, these kinds of messages are immediately blocked by the receivers, forcing the failure of the involved tests. Similarly, the behavior in very specific corner cases may slightly vary: considering the subscription to two eventgroups comprising both the same event, for example, vanilla `vsomeip` delivers twice every notification, while the secured version only once; the second message, in fact, would be anyway detected as replayed and discarded by the framework. Furthermore, it is worth mentioning that slight adaptations of a couple of tests has been made necessary to account for the modifications introduced by the run-time protection. They mainly concern the reduction of the maximum payload size, necessary to make room for authentication data in case of limited UDP packets: although it would have been possible to extend the maximum SOME/IP message length and preserve the payload, a different strategy has been adopted during the development of the PoC, to account for authentication data of different lengths.

Concluding this discussion, the relationship between the security framework and the service discovery functionalities presented in §3.4 is briefly analyzed. Albeit not being strictly mandatory from the operational point of view, the usage of the automatic detection features provided by SOME/IP is strongly suggested in conjunction with the conceived solution, to prevent the occurrence of critical situations. In case the service discovery module was disabled, for example, client applications would have absolutely no way to detect the reboot of the offerer,

preventing the correct reestablishment of the session. Nonetheless, this requirement is not deemed to be a limitation, being the additional module already necessary to enable the publish/subscribe communication pattern.

6.2 Performance measurements

After the competition of the proof of concept implementation and the successful verification of the correct functioning through automatic tests, a quantitative analysis of the performance has been carried on. It aimed at measuring the extra latencies introduced by the additional security functionalities that, being highly based on cryptographic functions, are with no doubt computationally expensive. In particular, the attention turned to the run-time protection, evaluating the penalties associated to each security level, to understand whether they are bearable or not in critical and constrained environments like the automotive one. In the following, the benchmark methodology is firstly introduced, to continue with the analysis of the results.

6.2.1 Benchmark methodology

In order to perform the necessary measurements, two applications, clearly based on the framework under analysis, have been developed to communicate between each other. In particular, a couple of test cases already provided for benchmarking purposes together with `vsomeip` have been adapted to the specific needs. The former implements a server, in charge of offering a service instance and answering to the received messages, while the latter is a client, which performs repeated requests to the peer and measures the round trip time necessary to obtain the acknowledgement.

Three different strategies have been adopted for performance evaluations, corresponding to alternative implementations of the core function executed by the client. Firstly, it is considered the request/response communication pattern through the logic depicted in listing 6.1, which adopts a synchronous approach: after having sent a request to the server, in fact, the corresponding response is awaited before proceeding with the next iteration of the loop. This way, the measured round trip time comprises the elapsed time from the very beginning of the request to the reception of the response, including both the latencies introduced by the framework and the ones due to the transmission across the network. As always, to reduce the variabilities and obtain more consistent results, the process is repeated within a loop, and the total elapsed time is considered for the final computation: the number of iterations varies according to the specific situation, to obtain a good compromise between precision and execution time. Secondly, an asynchronous approach is adopted, by waiting for the reception of the responses only at the end of all iterations. Although the measured quantity does not strictly correspond to a round

Listing 6.1: Synchronous request/response benchmark function.

```
1  function rtt_benchmark(payload_size, iterations)
2  begin body
3      request <- create_request(payload_size)
4
5      start_time <- now()
6      for i from 1 to iterations
7          send_request(request)
8          wait_response()
9      end for
10     end_time <- now()
11
12     print_rtt(start_time, end_time, iterations)
13 end body
```

trip time, it aims at highlighting as much as possible the penalties introduced by the framework through the continuous delivery of messages at the highest possible pace. Finally, the alternative communication pattern, publish/subscribe, is considered, to verify whether the usage of notifications alters the results obtained with the previous techniques.

In addition to round trip time measurements, CPU usage has been evaluated during each execution run, to check whether the introduced security functionalities influence the processor load. The estimation is performed exploiting the statistics provided by Linux through the `procfs` pseudo-filesystem, by computing the percentage of time spent for actually executing the core loop of the benchmark with respect to the total CPU time elapsed in the meanwhile. While being characterized by a limited precision due to the coarse granularity of the information reported by the kernel, obtained results allow for a high-level analysis of possible variations caused by the modifications.

Each benchmark has been executed both exploiting vanilla `vsomeip`, took as a reference, and the proof of concept implementation: in the second case, all the three possible security levels have been used, to compare the overheads introduced by each one; for what regards *authentication* and *confidentiality*-level services, CHACHA20-POLY1305 has been picked up as selected cryptographic algorithm. Applications based on the request/response pattern have been executed considering all the three types of network bindings offered by `vsomeip`. On one side, performance related to local communications, occurring between applications residing on the same ECU by means of Unix sockets, is considered; on the other one, remote services have been taken into account, by evaluating the message round trip time both in case UDP and TCP are chosen as transport protocols. Finally, for what regards notifications, only UDP is exploited, to allow for multicast messages. Additionally, message size variability has been kept in mind, by repeating the measurements with different payload lengths, ranging from 1 byte to 1024 bytes, which are deemed to be quite representative of actually used values: only the requests are modified between one

execution and the subsequent, while acknowledges are always characterized by the absence of the payload.

Similarly to what already performed for the evaluation of cryptography libraries, as explained in §5.1.3, measurements have been executed considering both a x86 based device and embedded systems equipped with ARM processors. It is worth noting that, for what regards the first case, only the laptop already presented in the above-mentioned discussion has been used: for the sake of simplicity, in fact, virtualization has again been exploited to emulate remote communications. In the second one, on the other hand, two identical NXP's SABRE Boards for Smart Devices, already used for cryptography benchmarking purposes, and interconnected by means of a Fast Ethernet link[2] composed the actual benchmark environment. They are based on the i.MX 7Dual Applications Processor, characterized by two ARM Cortex-A7 cores operating at 1.20 GHz, and equipped with 1 GB of RAM.

As already explained previously, moreover, many precautions have been adopted to reduce as much as possible any possible factor of interference, stopping all unnecessary background tasks and executing ten times every benchmark. It is worth pointing out that, especially for measurements adopting an asynchronous pattern, causing the transmission of high quantities of data in limited time, it has been necessary to tweak some operating system parameters related to the size of UDP buffers, to prevent the loss of packets due to saturation.

At the end of the benchmarking phase, the obtained results have been post-processed to compute the average values among the repeated executions, together with the associated uncertainties. Moreover, a comparison with respect to vanilla `vsomeip` has been extracted, to point out the penalties introduced by the security modifications; finally, the results have been plotted for easier analysis.

6.2.2 Results evaluation

In the following, the outcome of the different benchmarks is presented, both for what regards the x86 system and the embedded devices. All depicted graphs are organized in three parts. Firstly, the measured round trip time is presented, followed by a direct comparison of each security level against vanilla `vsomeip`, to highlight the differences. Finally, CPU usage measurements are plotted, to verify whether correlations with the other values do exist.

x86 benchmarks

Starting with the analysis related to the x86 system, figures 6.1 and 6.2 depict a subset of the obtained results, which are deemed to be enough for a high-level

[2] Although the development boards are equipped with Gigabit Ethernet interfaces, the speed has been limited to 100 Mbps to emulate the connection types available within actual vehicles.

evaluation. Being all the benchmarks performed on a single device, in fact, only limited variabilities are introduced by the usage of different transport bindings, thanks to the absence of the latencies due to the transmission over physical networks. Additionally, before going further, it is worth pointing out that, for what regards CPU loads, the maximum possible value is limited to 50 %, due to the contemporary execution of two applications on the same machine.

Analyzing more in detail the first chart, which shows the results of the synchronous request/response benchmark across UDP, it is possible to notice a slight difference between vanilla `vsomeip` and the secured version operating at *nosec* level. While no additional operations are carried on by the latter, variations can be probably ascribed to modifications regarding the usage of some thread synchronization primitives, required for the correct operation of the proof of concept. Considering greater security levels, on the other hand, latencies introduced by cryptographic functions become a bit more evident, although still remaining confined in the 15 % to 20 % band. Moving the attention towards CPU usage, it emerges an opposite trend with respect to the previous case, due to the limitations imposed by the security measures: replay protection, in fact, prevents the concurrent authentication of multiple messages belonging to the same service instance, slightly reducing the maximum exploitable parallelism.

Obtained results are mostly confirmed also by the second plot, shown in figure 6.2: while the average round trip time is much lower than the situation analyzed before, due to the asynchronous approach adopted, the same global trend can be inferred. Although the statistical error is more prominent in this specific case, due to the smaller time intervals took into consideration, penalties introduced by authentication and encryption are still quite contained, never accounting for more than 15 %; for what regards CPU usage, on the other hand, a homogeneous situation is presented, with no noticeable differences between the various cases.

Wrapping up, analyzed measurements demonstrate that, at least in case of powerful x86 systems, implemented security functionalities do not introduce unsustainable latencies, given the quite limited increases in the round trip time. Moreover, no evident differences are present between *authentication* and *confidentiality* levels, with the latter just slightly slower, and across the different payload sizes considered. Finally, although not having presented the graphic, definitely similar conclusions can be drawn also in case the publish/subscribe pattern is exploited, characterized by an absolute correspondence with the values depicted in figure 6.1.

ARM benchmarks

Being the intended targets of the developed solution, a more complete overview of the results related to embedded systems is presented. Moreover, having used two physical devices for benchmarking purposes, the difference between local and remote communications is clearly more evident, highlighting the significant latencies

introduced by the actual transmission across the medium. Nonetheless, talking about transport protocols, definitely similar results have been obtained both with UDP and TCP: for the sake of brevity, only the plots about the former are presented and analyzed in the following.

Beginning with synchronous request/response benchmarks, shown in figures 6.3 and 6.4, it is evident at a first glance the huge difference between the involved round trip times, almost ten times bigger in case of actual network communications. Being the variation so pronounced, it greatly influences all the considerations about the PoC implementation. While in case of local communications, in fact, security functionalities do introduce additional latencies accounting for about one third of the total round trip time, the same differences are almost negligible if packets flow across Ethernet. Similarly, although in the first situation the CPU load is certainly noteworthy, considering the 50 % limit due to the execution of two applications at the same time, the second one shows an almost completely idle processor. Finally, considering the results concerning notification-based communications, depicted in figure 6.5, no evident differences can be extrapolated with respect to the simpler request/response pattern.

Moving the attention towards the second exploited benchmarking technique, characterized by the continuous dispatch of messages at the highest possible pace, similar results, shown respectively in figures 6.6 and 6.7, are obtained both in case of local and remote communications. Comparing vanilla `vsomeip` with *authentication* and *confidentiality* levels, additional penalties accounting for at most one half the reference time are introduced by the implemented protections. At the same time, the processor appears to be quite overloaded, becoming a possible bottleneck: nonetheless, comparing the values associated to the different security levels with the reference implementation, they are all contained within the same band.

Wrapping up, analyzed data demonstrates that, also for what concerns embedded systems, the security functionalities integrated within `vsomeip` do not dictate unsustainable penalties. Considering the likely situation of UDP or TCP based communications, in fact, additional latencies are definitely negligible, also in a constrained environment like the automotive one. Anyhow, even stressing the use-case, by overloading the devices with an excessive amount of traffic, slowdowns are deemed to be still sustainable. Moreover, similarly to the x86 situation, *authentication* and *confidentiality* levels appears to be characterized by very similar performance, with a higher cost associated to the latter which becomes noticeable only for bigger payloads.

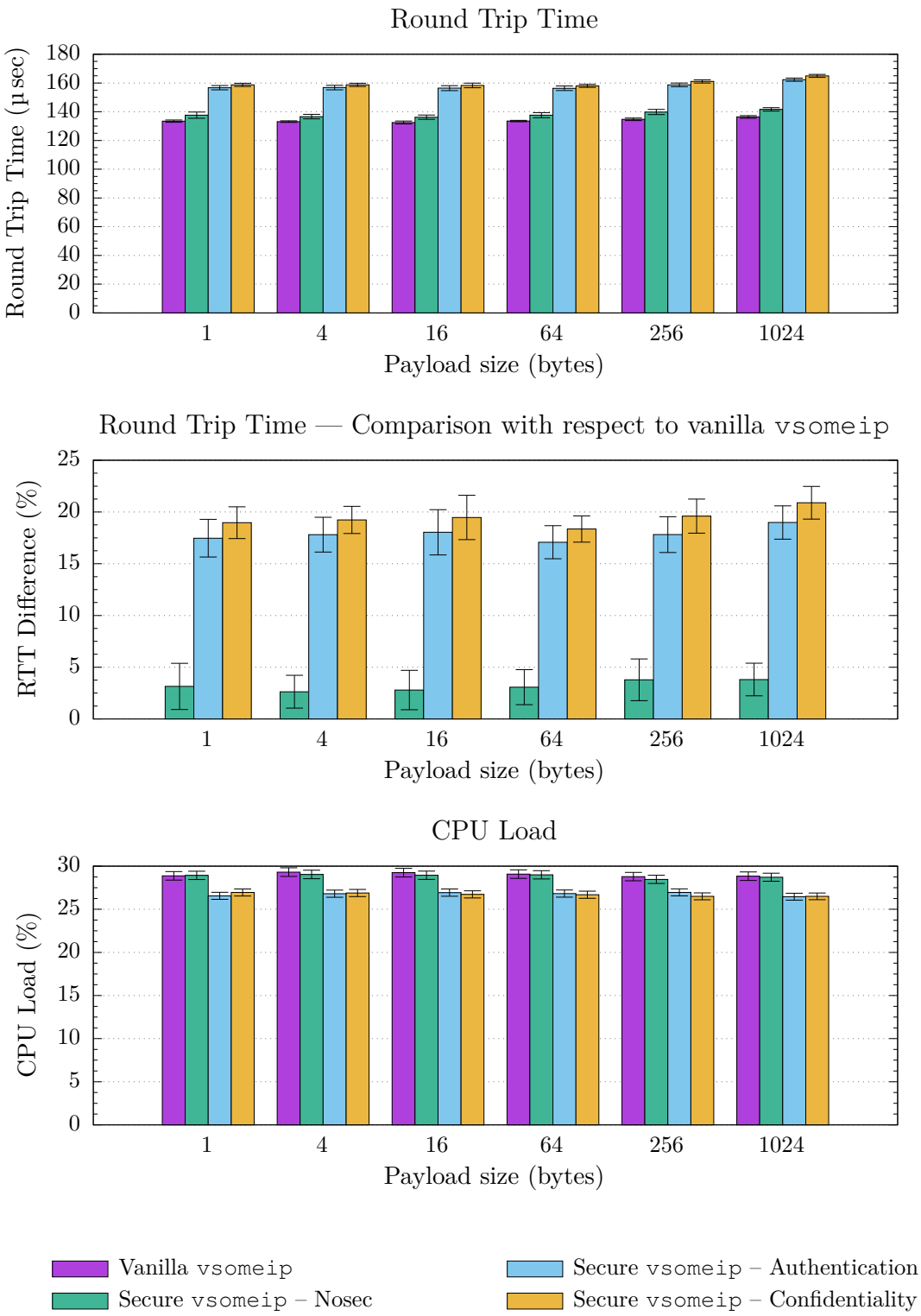


Figure 6.1: Synchronous request/response benchmark — UDP — x86

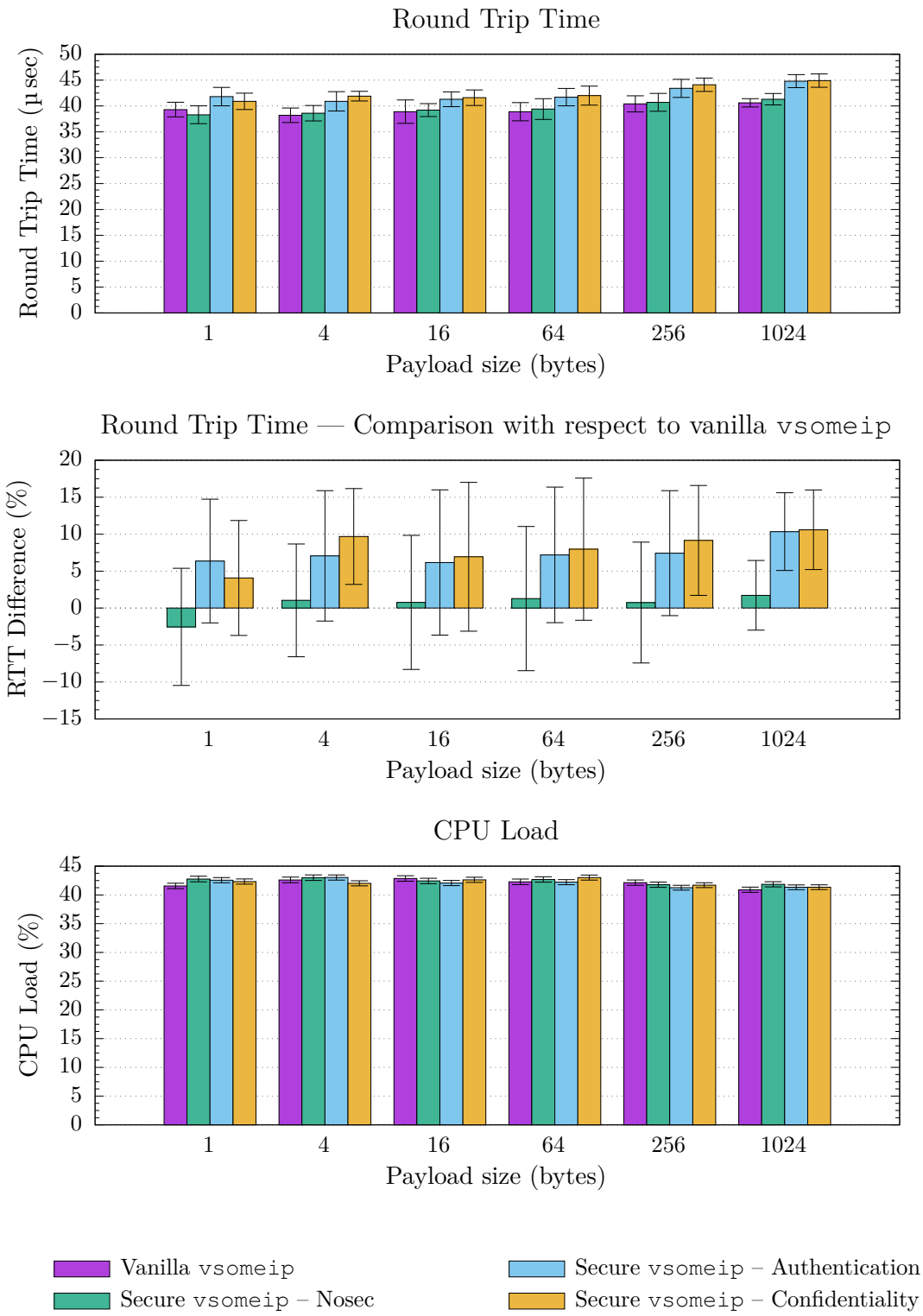


Figure 6.2: Asynchronous request/response benchmark — UDP — x86

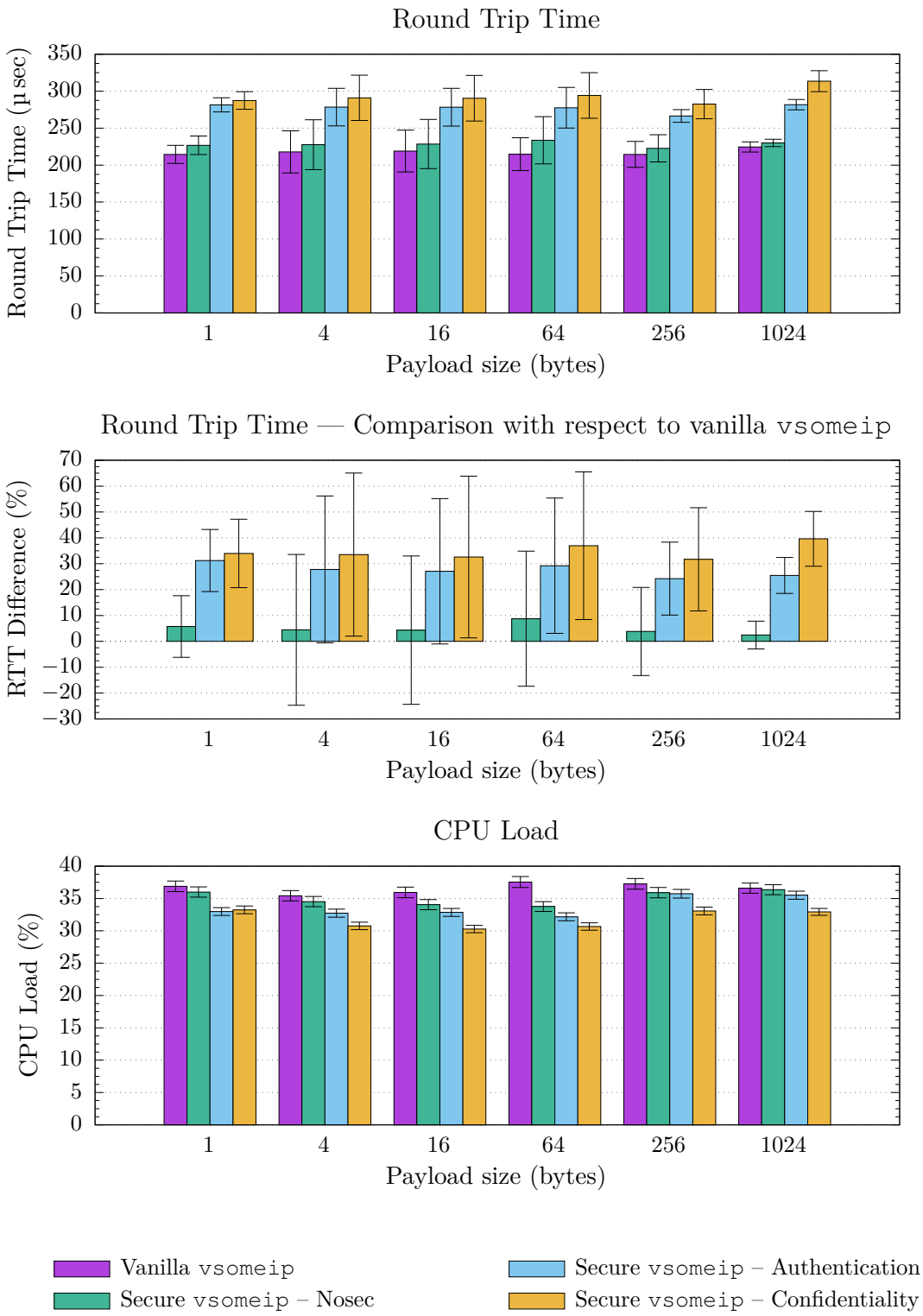


Figure 6.3: Synchronous request/response benchmark — Same device — ARM

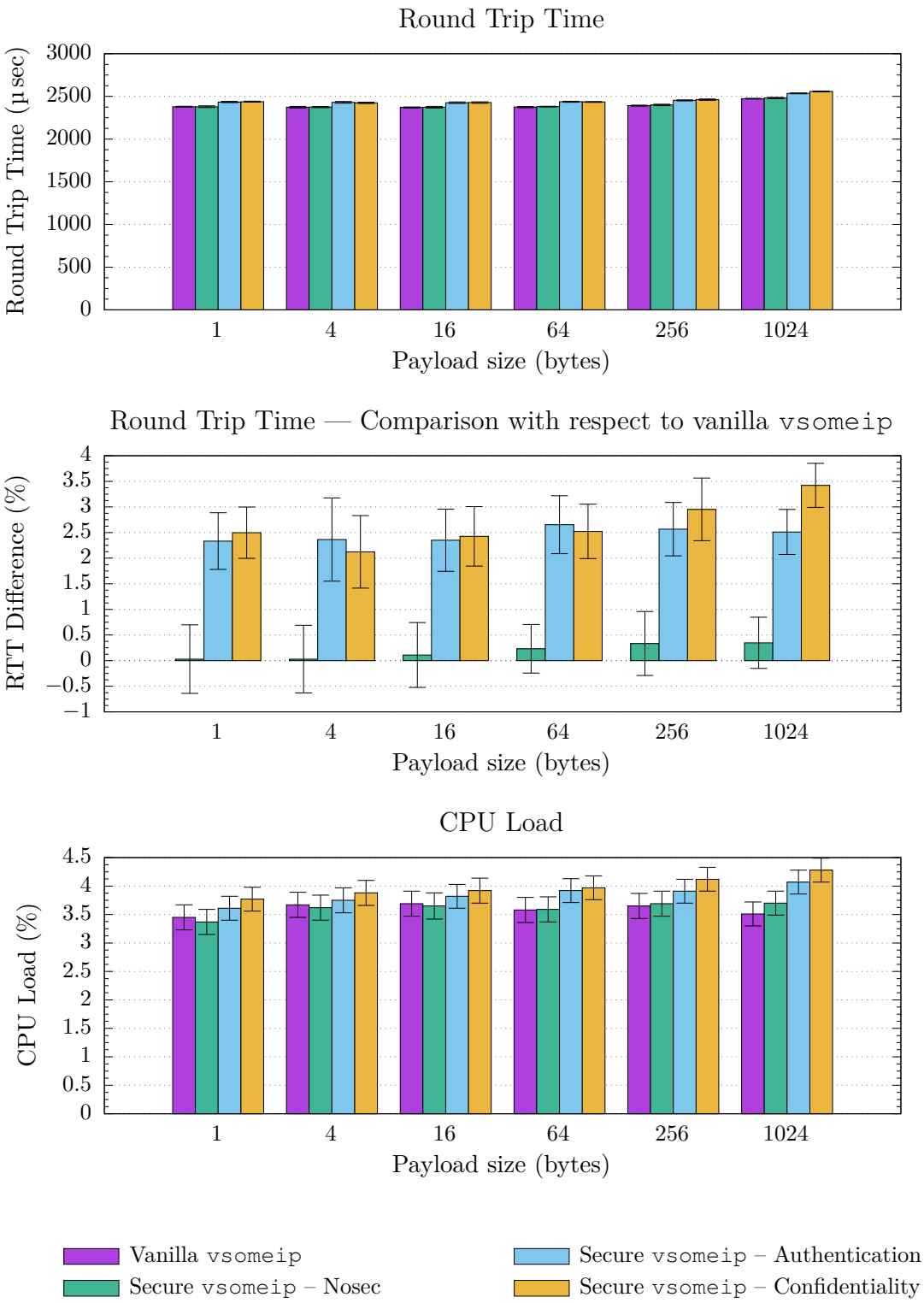


Figure 6.4: Synchronous request/response benchmark — UDP — ARM

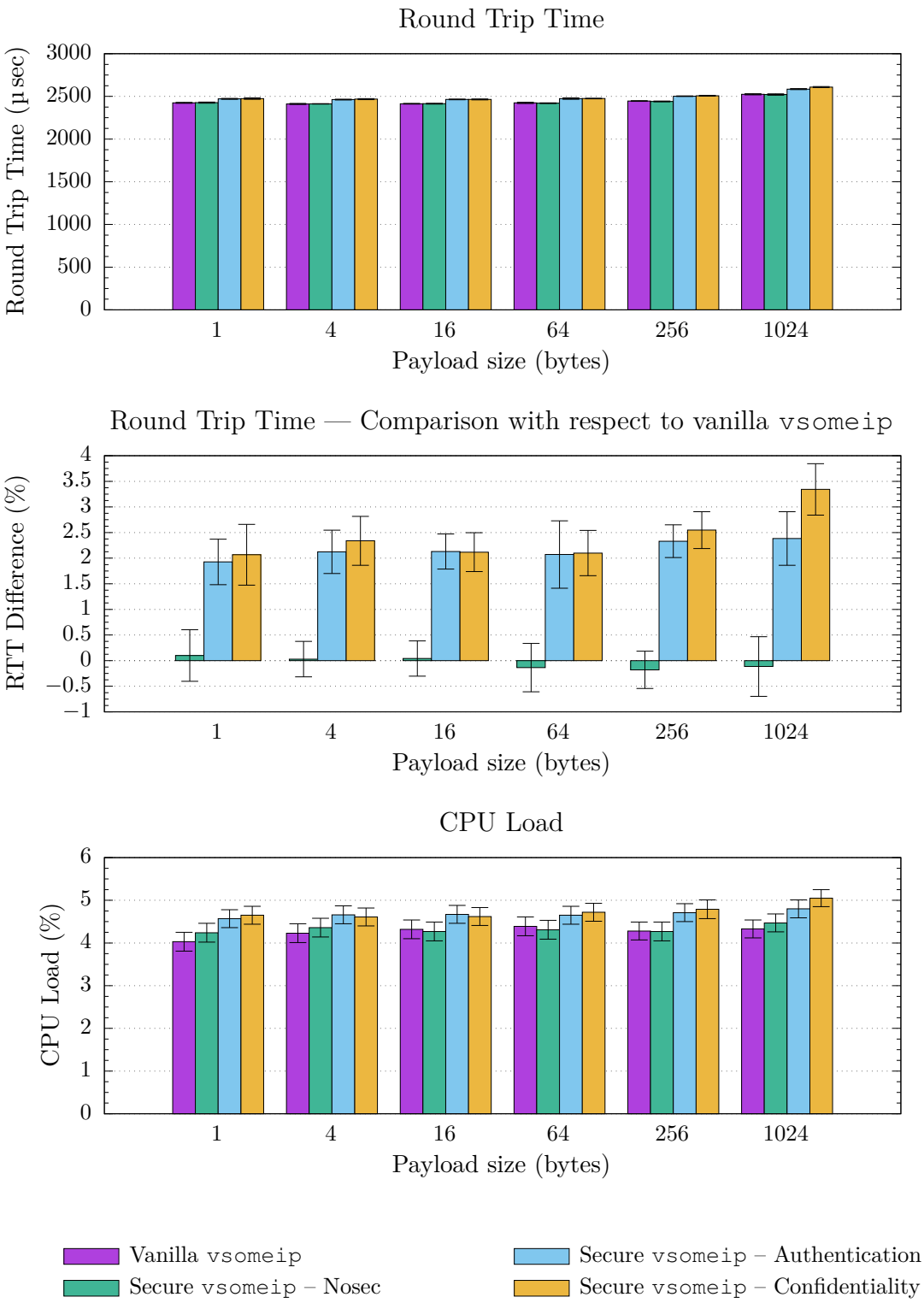


Figure 6.5: Publish/subscribe benchmark — UDP — ARM

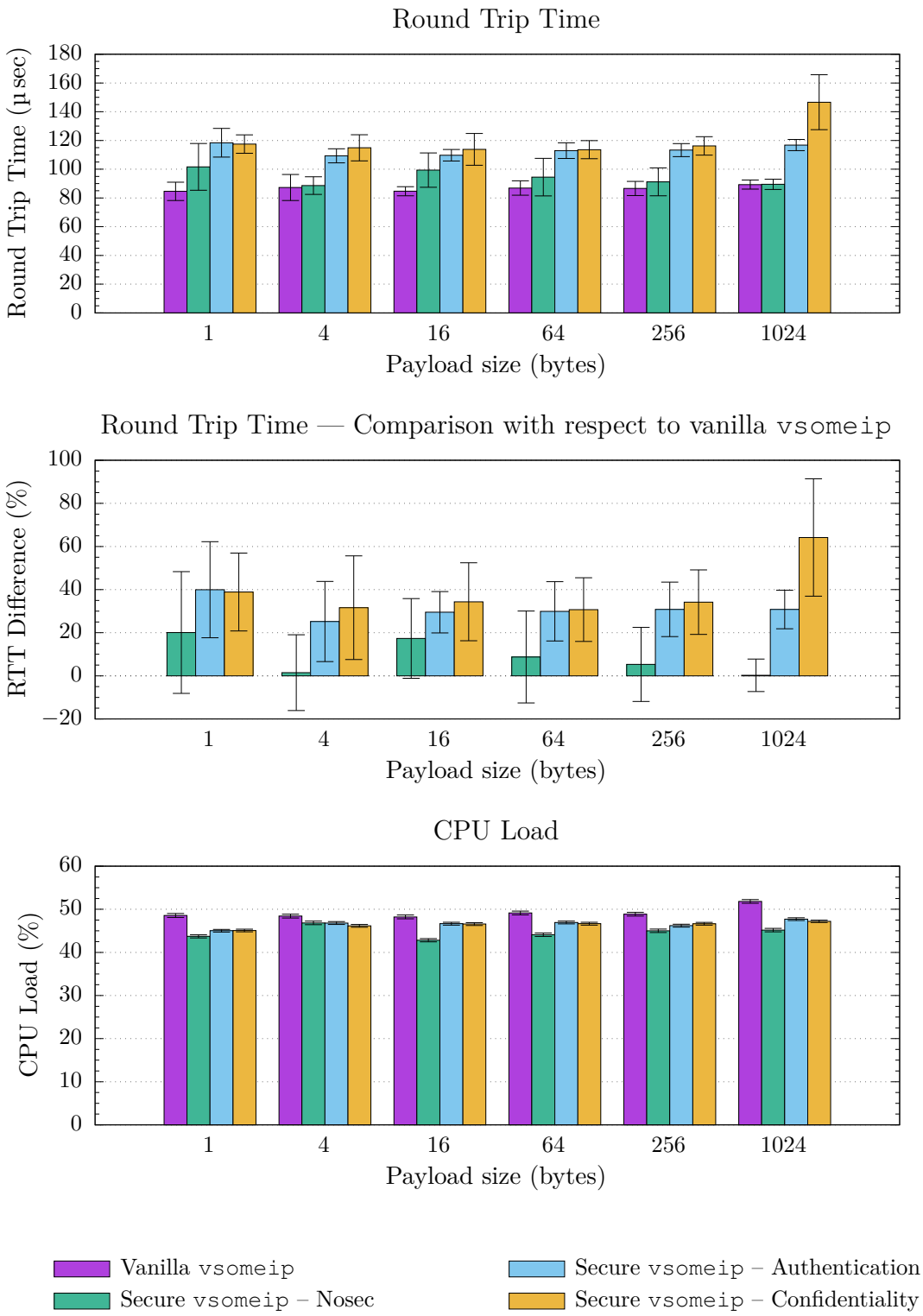


Figure 6.6: Asynchronous request/response benchmark— Same device — ARM

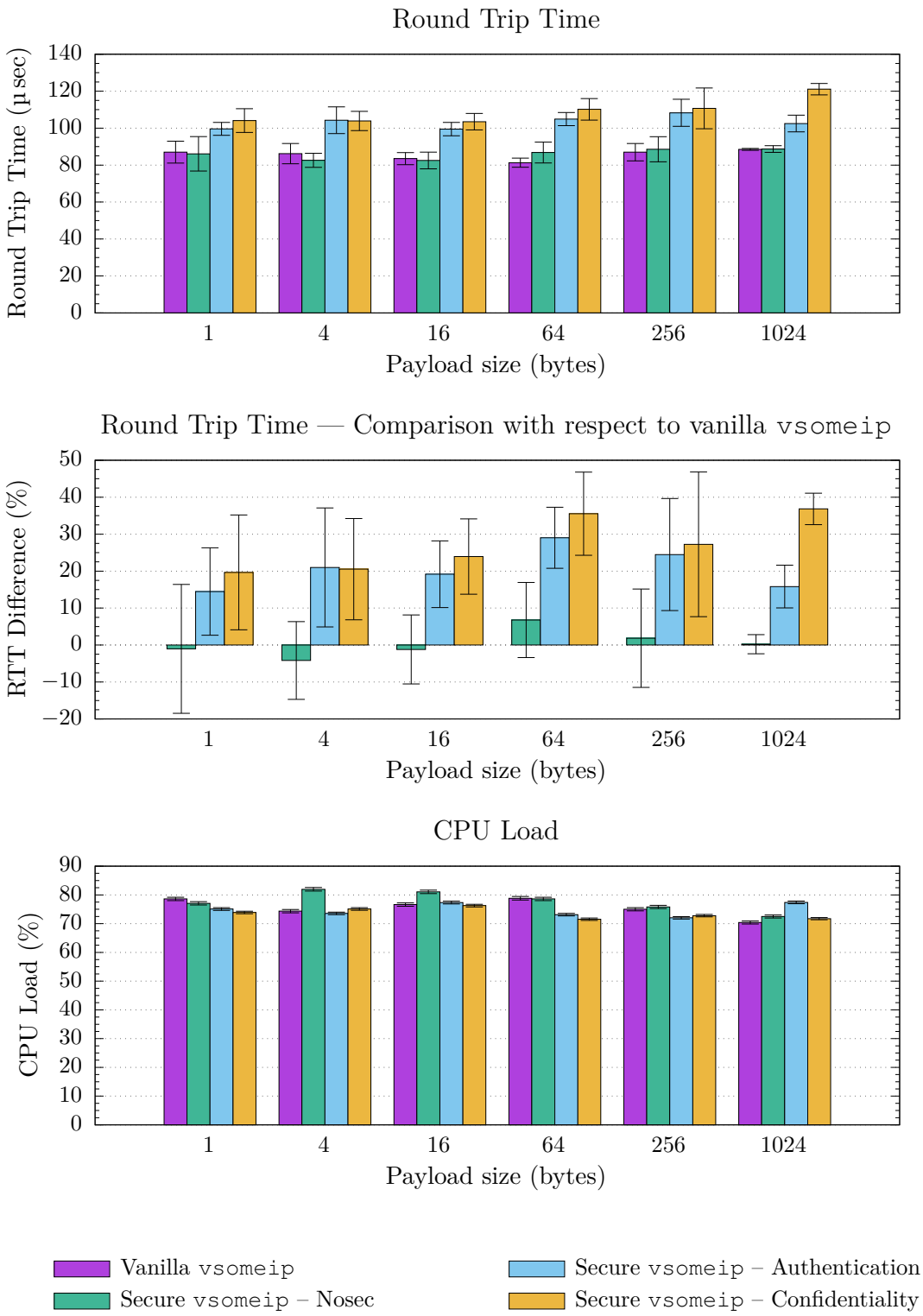


Figure 6.7: Asynchronous request/response benchmark — UDP — ARM

6.3 The demonstrator

As a final task of this thesis work, a demonstrator has been designed and developed, to illustrate why security is important in network communications and which are the functionalities introduced by the implemented proof of concept. Being the research under analysis strictly anchored to the automotive world, it has been realized to mimic different ECUs deployed within a vehicle: three different development boards, interconnected by an Ethernet switch, have been used to execute `vsomeip` applications which communicate between one another by means of the conceived framework. Additionally, a standard laptop has been used to represent an attacker that, by means of a specific application, tries both to extract information from the exchanged messages and to inject malicious packets into the network to trigger unwanted actions.

Before going on with a more in-depth analysis of the various elements composing the demonstrator, a very high-level overview of the adopted topic for the showcase is presented. The desire for an immediate and easily understandable staging has driven the decision of realizing the simulation of a typical car's instrument panel: while one application displays the dashboard, actual values to be shown are transmitted across the network exploiting `vsomeip` functionalities. Additionally, in order to make the whole fitting much more realistic, a hardware printed circuit board (PCB), made up of knobs, switches and buttons, has been prototyped to emulate the presence of actual sensors. Finally, physical switches are also exploited to configure the security level at which the services operate, providing an easy way to move from one to another and compare the provided guarantees.

6.3.1 Dashboard

The first element of the demonstrator analyzed during this discussion is the dashboard, implemented by a `C++` application. It is made up of two different building blocks: on one side, the graphical user interface depicting a typical car's instrument panel and, on the other one, the communication module, which receives the data to be displayed from the network.

Talking about the GUI, whose screenshot is shown in figure 6.8, it has been realized exploiting the Qt framework and, in particular, the `QML` declarative language, which allowed the creation of a nice interface in a relatively easy way. In this regard, it is necessary to admit that a quite gorgeous `QML` dashboard was already provided as an example shipped together with the framework,[3] which has been exploited for the realization by tweaking some parameters and implementing the interfaces necessary to read the data received from the communication module. For the sake of

[3] `https://doc-snapshots.qt.io/qt5-5.12/qtquickextras-dashboard-example.html`

Figure 6.8: Dashboard GUI screenshot.

completeness, it is worth mentioning that the actually used GUI is characterized by a more appealing dark theme, which has been adapted for the current presentation due to graphical reasons.

Moving to the communication module, on the other hand, it is composed by a set of classes that, exploiting the `vsomeip` framework, act as a client of the different available services, by subscribing to the offered events and receiving the associated notifications. Whenever a new value is received, the dashboard is immediately updated through the peculiar functionalities offered by Qt, namely signals and properties, which allow the communication across the `C++` to `QML` boundary.

Concluding the discussion about the dashboard, the considered application is executed by an embedded system composed of an Embedded Artists' iMX6 Quad COM Board, equipped with a high-performance quad-core Cortex-A9 ARM processor operating at 1 GHz, 2 GB of RAM and supporting both 2D and 3D graphical acceleration. It is bundled with the COM Carrier Board developed by the same vendor, providing a wide range of input/output interfaces towards the external world. An Embedded Linux operating system, compiled through the Yocto Project build system, is run on the top of bare hardware, to execute the application of interest which, together with the developed security framework, has been cross-compiled by exploiting the available SDK.

6.3.2 Services

Going on with the discussion, the second building block of the demonstrator is composed by two applications, running on distinct devices. Each one is associated to a different set of instruments displayed by the cluster and is in charge of obtaining and transmitting the information to be shown. As in the previous situation, two modules can be identified as the base of each program, corresponding to the distinct tasks that have to be performed.

Starting with the analysis of the communication part, it is organized in different `vsomeip` services, each one in charge of a specific gauge. A publish/subscribe approach has been adopted to notify the dashboard for the presence of a new available value: while being deemed to be the most suitable strategy for the particular use-case, thanks to the decoupling between senders and receivers, the chosen communication technique allows the usage of multicast messages which, as explained later on, greatly simplify the implementation of the attacker.

Secondly, each application is required to obtain the values to be transmitted: albeit it was easy to input the data by means of a keyboard and a textual interface, the result would have been visually poor, losing each association with the automotive world. For this reason, as already stated in the introduction, a more appealing staging has been realized, to emulate the information that would normally be provided by sensors, through the usage of physically controllable knobs and switches. After having completed the construction of the PCB comprising the various components, it has been connected to the two development boards by exploiting both general-purpose input/output interfaces (GPIO) and a ready-to-use analog to digital converter. Moving to the software side, input values are continuously accessed by the applications through the facilities provided by the Linux kernel and, particularly, by reading the content of special files located within the `sysfs` pseudo-filesystem. Finally, raw data is converted to calibrated values, ready to be dispatched within the corresponding notification.

Two identical development boards, already exploited for benchmarking purposes, have been used to run the considered applications. Custom scripts have been prepared to automatically execute the programs, by setting the necessary parameters without the need for user intervention. Moreover, they allow switching back and forth between the three different security levels provided by the proof of concept by simply acting on the buttons part of the PCB. Finally, a bunch of leds are managed by the same piece of code, to provide an immediate and visual feedback about the current operating mode.

6.3.3 Attacker

Given that the conceived security framework aims at protecting in-vehicle communications, the third fundamental element constituting the demonstrator is clearly an attacker, which tries to perform different malicious operations to subvert legit message exchanges. It is represented by a `C++` application, which is meant to be run on a standard laptop connected to the same Ethernet switch of the other devices. For the sake of simplicity, as already introduced before, multicast messages have been used for service communications: this way, the attacker is able to automatically receive all the exchanged packets, by just subscribing to the same group, without the need for putting in place additional malicious techniques. Nonetheless, it is worth noting that, while not being of any benefit for the showcase under analysis,

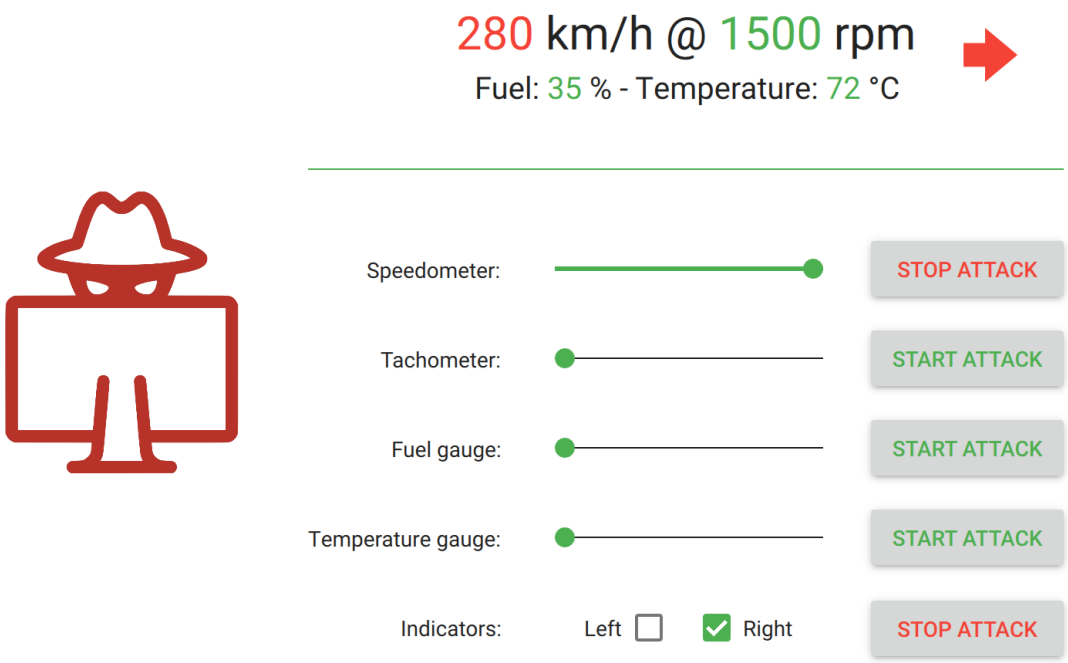


Figure 6.9: Attacker application screenshot.

the execution of man-in-the-middle attacks, forcing the packets flowing through the hijacker's device, would have been similarly trivial, by exploiting off-the-shelf tools.

Analyzing more in detail the developed application, implemented exploiting the functionalities provided by the Qt framework, it is constituted by a back-end, in charge of managing network communications, and a front-end, responsible for visually presenting both the intercepted information and a simple control panel. Starting with latter, figure 6.9 depicts the aspect of the QML based graphical user interface, which is divided into two parts corresponding to the available typologies of offenses: the topmost area shows the results of a sniffing attack, carried on by passively listening for the messages exchanged between legit applications. The second section, on the other hand, provides the possibility to perform a spoofing attack, by flooding the network with a high volume of specifically crafted messages, trying to force the dashboard showing malicious information. Looking under the hood, bare network sockets are managed through the abstractions provided by the Qt framework, allowing both to listen for multicast messages and to send fake SOME/IP packets used to try fooling the dashboard.

6.3.4 Conclusions

Switching back and forth between the three security levels offered by the implemented proof of concept, the demonstrator provides a clear picture about the differences in terms of guaranteed protections, by observing whether each attack has success or is

prevented by the framework. Starting with the *nosec* level, corresponding to vanilla SOME/IP messages, the attacker does not encounter any kind of opposition, being able both to read the exchanged information and to force the dashboard showing injected data. Increasing the security level by adding authentication functionalities, it becomes immediately evident that, while sniffing is still possible, message flooding becomes completely useless, being no more able to condition the values displayed by the different gauges. Finally, going further and enabling the *confidentiality* level, the attacker loses also the capability of intercepting exchanged messages: although every packet continues to be received, in fact, the encryption prevents the extraction of useful data from the payload and the malicious application displays only meaningless information.

Chapter 7

Conclusions and Future Work

During this thesis work, a security framework integrated within SOME/IP has been conceived to protect the communications between in-vehicle services. Although different well-established protocols do already exist to prevent attacks to network transmissions in the broader ICT world, the challenging environment dictated by the automotive sector and the desire for a future-proof solution pushed the research towards a new approach. After an initial analysis, SOME/IP emerged as a promising communication framework, driven by dynamism and oriented towards high-bandwidth and computationally demanding applications. Albeit comprising a wide range of functionalities, no reference to cybersecurity is made inside the framework specifications, leaving the transmitted information completely unprotected from malicious attacks.

Most of the work has been devoted to the design of a protocol aiming at protecting SOME/IP messages flowing across the network by defining, for each service, the set of allowed communications through high-level rules. It has been conceived to avoid introducing any limitations to the original framework, and is organized in two distinct parts: an initial session establishment phase, which mutually authenticates applications willing to communicate between one to another, and the run-time enforcement engine, applying the necessary protection to actual network packets. Moreover, a complete security-oriented architecture has been outlined, to point out the functional modules necessary to achieve a full protection.

The depicted solution has been mostly implemented as a proof of concept within the `vsomeip` stack, and a demonstrator has been developed to visually show up the goodness of the approach by mimicking different ECUs deployed within a vehicle. Moreover, benchmarks have been performed to quantitatively measure the impact of the additional features: although obviously certifying the introduction of some penalties from the latency point of view, the results are deemed to be pretty promising for a still improvable implementation. Considering a typical communication across the network, in fact, cryptography functions never impacted for more than five percent of the total round trip time. Furthermore, also in case of more extreme

benchmarks, simulating devices overloaded by excessive amounts of traffic and neglecting network overheads, additional latencies never exceeded unsustainable thresholds. Anyway, additional measurements should still be performed both to quantify the overheads introduced by the initial session establishment phase and to explore the possibility of exploiting available hardware accelerators to fatherly reduce the penalties, especially in case of more limited embedded systems.

As a future work, the designed security protocol could be reviewed by means of formal analysis, to verify the correctness and guarantee the absence of faults; moreover, known limitations shall be examined in great detail to conceive possible solutions. In particular, for what regards the security granularity, initially confined for the fear of introducing excessive overheads, in the light of the performance results it may be possible both to increase the number of available security levels and to differentiate the communications belonging to the same service, by introducing the usage of multiple symmetric keys. Furthermore, denial of service attacks should be reconsidered, to verify whether it would be possible to put in place additional protections aiming at reducing their effectiveness.

Another important future plan could consist in exploring the different technologies necessary to complete the realization of the security architecture outlined during this thesis work. Techniques available both in the ICT field and already provided by automotive grade devices may be analyzed, to study the different possibilities that could be exploited both for cryptographic material binding and platform integrity verification. Finally, it should be evaluated the capability of assembling all the different building blocks, by always keeping in mind the constrained environment dictated by the automotive world.

Bibliography

[1] Micha Risling. *In-Vehicle Connectivity: Dealing with the "Elephant in the Car"*. Apr. 2018. URL: https://innovation-destination.com/2018/04/26/in-vehicle-connectivity-dealing-with-the-elephant-in-the-car/.

[2] "IEEE Standard for Ethernet Amendment 1: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1)". In: *IEEE Std 802.3bw-2015 (Amendment to IEEE Std 802.3-2015* (Mar. 2016), pp. 1–88. DOI: 10.1109/IEEESTD.2016.7433918.

[3] "IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 1 Gb/s Operation over a Single Twisted-Pair Copper Cable". In: *IEEE Std 802.3bp-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by-2016, and IEEE Std 802.3bq-2016)* (Sept. 2016), pp. 1–211. DOI: 10.1109/IEEESTD.2016.7564011.

[4] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. "Security threats to automotive CAN networks — Practical examples and selected short-term countermeasures". In: *Reliability Engineering & System Safety* 96.1 (2011). Special Issue on Safecomp 2008, pp. 11–25. ISSN: 0951-8320. URL: http://www.sciencedirect.com/science/article/pii/S0951832010001602.

[5] K. Koscher et al. "Experimental Security Analysis of a Modern Automobile". In: *2010 IEEE Symposium on Security and Privacy*. May 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[6] Stephen Checkoway et al. "Comprehensive experimental analyses of automotive attack surfaces." In: *USENIX Security Symposium*. San Francisco. 2011, pp. 77–92.

[7] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway — With Me in It*. July 2015. URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[8] Charlie Miller and Chris Valasek. "Remote exploitation of an unaltered passenger vehicle". In: *Black Hat USA* (2015), p. 91.

[9] Q. Wang and S. Sawhney. "VeCure: A practical security framework to protect the CAN bus of vehicles". In: *2014 International Conference on the Internet of Things (IOT)*. Oct. 2014, pp. 13–18. DOI: 10.1109/IOT.2014.7030108.

[10] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. RFC Editor, Dec. 2005. URL: http://www.rfc-editor.org/rfc/rfc4301.txt.

[11] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, Aug. 2018. URL: http://www.rfc-editor.org/rfc/rfc8446.txt.

[12] M. Hamad, M. Nolte, and V. Prevelakis. "A framework for policy based secure intra vehicle communication". In: *2017 IEEE Vehicular Networking Conference (VNC)*. Nov. 2017, pp. 1–8. DOI: 10.1109/VNC.2017.8275646.

[13] Wikipedia contributors. *AUTOSAR — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=AUTOSAR&oldid=856923595.

[14] AUTOSAR. *SOME/IP Protocol Specification*. 2016. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-0/AUTOSAR_PRS_SOMEIPProtocol.pdf.

[15] AUTOSAR. *Specification on SOME/IP Transport Protocol*. 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf.

[16] AUTOSAR. *SOME/IP Service Discovery Protocol Specification*. 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-3/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf.

[17] *GENIVI Alliance*. URL: http://genivi.org/.

[18] *CommonAPI C++*. URL: https://genivi.github.io/capicxx-core-tools/.

[19] Genivi. *vsomeip in 10 minutes*. URL: https://github.com/GENIVI/vsomeip/wiki/vsomeip-in-10-minutes.

[20] Wikipedia contributors. *Cryptographic hash function — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=860912721.

[21] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. RFC Editor, Jan. 2012. URL: http://www.ietf.org/rfc/rfc6347.txt.