CHALMERS

Securing In-Boat CAN Communication over the Internet

Using a Cell Phone as Wi-Fi Router with Limited Hardware

Master's Thesis in Computer Science and Engineering

RICKARD PERSSON

ALESANDRO SANCHEZ

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden 2015

Securing In-Boat CAN Communication over the Internet
Securing In-Boat CAN Communication over the Internet Using a Cell Phone as Wi-FI Router with Limited Hardware

Rickard, Persson
Alesandro, Sanchez

Examiner: Tomas Olovsson

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden June 2014

Abstract

Opening a communication channel to the internal vehicle bus of pleasure boats through the Internet, offers benefits such as remote diagnostics and software update of boat components. An undesirable consequence of this communication is a broad range of security threats which could can damage to the boat and company. Therefore, it is important that the communication channel is secured and these threats are mitigated. In this report, we show that it is possible to create a secure communication channel to the internal bus of a pleasure boat with an inexpensive embedded device. The solution is based on knowledge from the automotive and embedded device sector, and possible ways for protecting against these threats have been analyzed. A solution has been tested with an embedded device having access to the internal bus and connection to the Internet using a smartphone, while maintaining *confidentiality*, *integrity* and *authenticity*. The final results show that it is possible to create the communication channel with this inexpensive embedded device. The communication between the device and the server is secured with *Transport Layer Security* with sufficient throughput for various tasks. In addition to this, the embedded device need no manual input except for configuration of the smartphone. In essence, this thesis offers a proof-of-concept for an inexpensive way to improve customer support.

Acknowledgements

During the time of this project we have received valuable help from several people. We would like to thank Marco Monzani and Anders Lingfors at CPAC Systems AB, Marco for the help with the project setup and design and Anders for help with Perl programming. Pierre Kleberger and Tomas Olovsson at Chalmers University of Technology for reviewing the report and helping us with considerations on securing in-vehicle communication.

The Authors, Göteborg 2014-06-01

Contents

Terminology

Connection device The embedded device used for connecting the boat to the Internet.

CAN Controller Area Network, communication bus that connects ECUs.

TLS Transport Layer Security, Cryptographic protocol at the application layer.

SSL Secure Socket Layer, Predecessor to TLS.

IPsec Internet Protocol Security, Cryptographic protocol at the network layer.

MIPS Million of Instructions Per Second, performance measure for CPU.

DNS Domain Name System, translation between hostnames and IP addresses.

1

Introduction

The world today is and has been moving towards *the Internet of things*. More and more different types of devices and machines are being connected to the Internet and the boat industry today realizes they also need to move in this direction if they want to stay relevant in the market. By connecting the boat system to the Internet it could be possible to assist the boat owner/customer in different ways; such as performing system diagnostics and software updates for boat components among other things. This added benefit is made even more desirable given the fact that boat systems are very complex today and their complexity keeps growing, making the risk for bugs and the need to fix them bigger. Given today's, and tomorrow's, huge smartphone user base and the progresses being made in mobile phone connectivity worldwide, a new door opens to use mobile phones as a pathway for Internet access for pleasure boat systems.

1.1 Motivation

Today, many *electronic control units* (ECU) on boats can be accessed through a tool for diagnosing, updating software and even to change configuration parameters, such as enabling leisure features. The ability to do all this remotely would improve productivity by reducing the need to send out technicians, something that takes much time. To create a connection from the boat to the Internet is therefore desirable. Enabling communication with the boats' internal bus also arises security threats which can affect the system maliciously.

1.2 Gap

Opening up the internal communication bus to the outside has previously been done within the automotive industry, even some embedded devices have been possible to be accessed to from the Internet previously. Could these fields be applied to the boat sector, without a monumental price increase, in a secure way to avoid access by a third party?

1.3 Problem description

Opening a boat's network for Internet access comes with certain risks. An unauthorized individual might gain entry to the system and cause components to malfunction. A different possibility is to not harm the system but to enable software properties that were initially disabled since they require payment. It is clear that all communication to and from the boat need to be properly secured. Confidentiality, integrity and authentication are prioritized, availability on the other hand should be enforced to best effort but is not fully required.

1.4 System description

A general overview of the system's design can be seen in Figure 1.1, which shows different stakeholders and elements in the project. The *connection device* shown in the figure represents the embedded device that would be needed on the boat to bridge the boat's inner communication to the Internet. Those operating on the boat are the *customer* and *technician* who can enable the *connection device*. This device can then communicate with the internal CAN bus on the boat and connect to a wireless router to gain Internet access and thus be able to connect to the CPAC's servers. At the company the technicians or developers are then able to communicate with the internal bus of the boat.

1.5 Design requirements

The *connection device* should be able to take advantage of smartphones' access to the Internet without the need of developing apps for said phones. The wish to not develop a mobile app consists of a series of considerations, even if an application could offer better feedback to the customer. The reason for this was to avoid the need for maintaining and updating an application with the continuous upgrading of the operating system on the mobile phone. Other considerations include the need to support various *Software Development Kit* (SDK)[1][2] versions on the a specific platform, and support multiple mobile

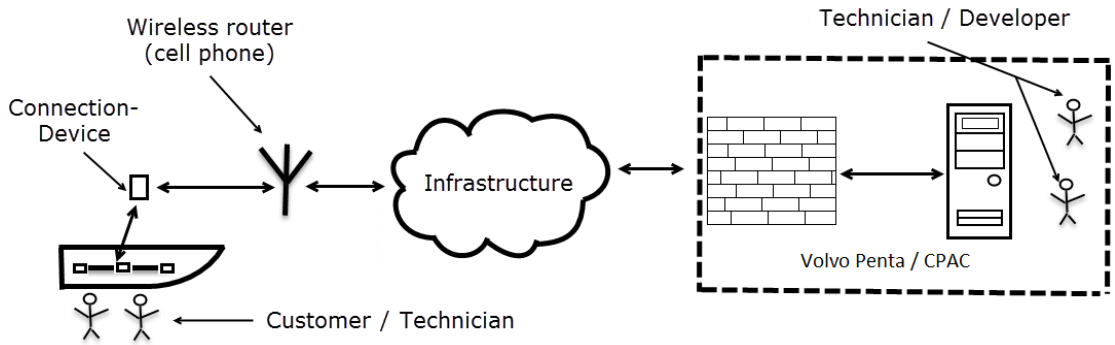

Figure 1.1: General overview of the system

platforms[3]. For this reason the *connection device* should make use of the smartphones' ability to create a WiFi hotspot in order to gain access to the Internet.

The developed system should consist of relatively inexpensive embedded hardware (*connection device*) which enables communication with the internal bus of the boat. This device should also enable communication with a router with the use of WLAN. The software on the *connection device* should primarily be a forwarding node to avoid complexity in the device which would otherwise require updating the device if other software in the boat were to change. An example of this would be if, for instance, the *connection device* was able to query the system for error codes, it would have to be updated every time the rules for querying the system changed.

Due to cost restrictions on the *connection device*, the hardware will be limited which means less processing power, memory and storage, which in turn limits the software complexity that can be supported locally on the device.

The *connection device* should be able to connect to the wireless router automatically once it has been turned on. This will keep the cost of the device low since it will not need a display and keyboard to allow the user to choose the correct router and then enter its corresponding password. Many boats today include displays and input devices that could be used for this purpose, but that would mean that old displays should be updated to offer this new functionality. Developers should then also need to include this functionality on new devices. If the *connection device* could handle this part on its own it would therefore mean reduced maintenance and upgrade costs.

The product developed should be able to achieve specific tasks when communicating with the internal bus - namely:

- Remote diagnostics
- Data logging
- Parameter configuration

- Software update

With *remote diagnostics* it should be possible to receive error codes from the system, these should be possible to identify to determine the cause of the error(s). *Parameter configuration* enables adaptation and modification which affects multiple aspects of the system, including steering and handling. *Software update* enables the possibility to update the software of the different ECUs existing on the internal bus. *Data logging* enables the viewing of the data that is being sent on the CAN-bus.

The last, but not least, important requirement of the design is that the communication between the *connection device* and the remote host should be safe to protect against the various security threats that are mentioned in chapter 2.3. A very important consideration to be taken when securing the communication is the lack of control over the router the *connection device* connects to. This is because this system should be able to connect to any wireless router- including a smartphone's Wi-Fi hot-spot - meaning in most cases the lack of ability to configure various routing settings.

To summarize the project requirements are:

- Low cost for the *Connection device*.
- Low cost for the system's maintainability.
- Simple[1] software for the *Connection device*.
- Automatically connect to WLAN.
- Secure connection to avoid compromising the channel. This includes *Confidentiality, Integrity, Availability and Authenticity*.
- Fast and reliable to be able to support the different diagnostics and configuration operations previously mentioned.

1.6 Purpose

The purpose of this project is to find different inexpensive solutions to create a secure connection between a boat and a server. Compare them and find the most suitable one. What qualifies a security solution as appropriate for this project depends on the following points:

- Able to run on inexpensive hardware.
- Be as fast as possible to reduce communication delay.
- Provide the toughest achievable security with *Confidentiality*, *Integrity*, *Availability* (CIA) and *authenticity* in mind.

[1] It should only forward CAN messages from and to the server to keep the complexity and the need to update the software low.

The project consists of implementing the software in the embedded device for establishing communication between the CAN-network to a server over the Internet, this includes identifying the WLAN router to connect to the Internet.

It also consists on determining and implementing a security solution of the system with consideration to *Confidentiality, Integrity & Availability* (*CIA*) and *Authentication*, with less focus on *Availability*. Consequently the task did not involve designing a new security protocol for this type of communication. The project did not involve writing the server software for large scale handling of multiple connections and the communication to the customer databases.

For testing the reading of data, a program capable of receiving and interpreting the CAN data sent from the *connection device* was created. EVC Flash, a program used internally at CPAC was used to test configuration of parameters and for updating software.

1.7 Research questions

This report will address the following research questions:

- Which security threats exist for in-boat networks when connected to the Internet?
- In which ways can these threats be mitigated to protect the system?
- How to ensure CIA and authenticity while connected to the Internet?

2

Background

2.1 Similar systems

Products that give access to the internal bus for tasks as diagnostics, updates, support and analysis exist both for agricultural vehicles[4] and ships[5]. The level of service that these products provide are considerable more advanced than the design scope of this project. Advanced services include the ability to communicate through satellites.

A product similar to this project is a device developed by Humphree. The major difference consists of their product having a display for the selection of WLAN. This device from Humphree also uses mobile phones to get connected to the Internet. We tried and failed to contact them to find out if they secured the connection, and if so, how.

Another product with similarities is CarIQ [6], which connects to the internal bus, transmits the data with the use of mobile antenna to connect to mobile networks which in turn enables communication with remote servers to provide the customer with information about the car. HTTPS is used to secure the connection.

Both of these solutions offer a wide variety of services which require better hardware to manage them. They are also dependent on complying with national communications standards if the product has its own transmitter and receiver which require a maintained infrastructure.

2.2 Vehicle-to-*X* communications

Vehicle-to-X consist of *Vehicle-to-Vehicle* (V2V) and *Vehicle-to-Infrastructure* (V2I) communications, both of these areas offer increased safety, traffic flow, lower emissions

and enabling larger extent of autonomy for vehicles than is today available. The systems can be used independently or in conjunction.

The distinction between the two areas can be noticed in the name. *V2V* is communication between vehicles in the same area, this can be used for deciding the flow through an intersection or highway entrance. Another area consist of platooning and automatic cruise control in which a group of cars autonomously drive in a chain to increase safety, fuel efficiency and thus lower emissions.

Vehicle-to-Infrastructure (V2I), as the name suggests, is communication between a vehicle and the infrastructure around it. Example of infrastructure include traffic signs and lights. Use case for traffic light could be notifying incoming cars that the signal will change within a certain time frame. This enables the car to make projections and adapt the speed thereafter. *V2I* communication also enables the forwarding of various road hazards to the driver or vehicle, these hazards are categorized as static or dynamic. Static hazards might be road construction, while dynamic can be weather conditions[7][8][9].

Unique for this communication in V2V and V2I is the usage of *Wireless Access in Vehicular Environments* (WAVE) also known as IEEE 802.11p when the vehicles communicate with the surroundings.

Opening up the car for these communications also brings certain security threats that are specific for this sector but also include more general threats. Known issues that are specific for *V2I* or *V2V* are the exploitation of the GPS data that is transmitted by vehicles, for example the possibility to over shadow the original *Global Positioning System* (GPS) signal with an incorrect one which could cause collisions. Another example regards privacy concerns since it could be possible to continuously track vehicles' position thus revealing the habits of individuals[10].

2.3 Security threats

This section describes various security threats which exist when establishing communication via the Internet. Also described are various methods for mitigating these threats.

2.3.1 Threats that arise by connecting to the Internet

Security threats against a system connected to an open network have various forms such as exploiting design flaws in the security protocol itself[11] or in the implementation of the security protocol[12]. Threats can also arise from not very strict specifications regarding the implementation of a protocol[13]. The threats do not exist on a single layer of the TCP/IP-stack but across them, with each of them offering different issues.

An attack against a system can take various forms. From stopping a service from offering the intended services, as is the case with *Denial-of-service* attacks (*DoS*)[14], to unauthorized access to the system, which may contain sensitive data. Manipulation of an existing communication channel for injection, removal or modification of packets. These threats are continuously evolving as flaws are detected and resolved, but also new flaws are introduced during the process of system updates[15].

Security threats can generally be divided into two categories, passive and active attacks. Passive attacks consist of eavesdropping the communication channel, and due to the nature of this attack it is difficult to detect. It can be mitigated by encrypting the communication thus making it considerably more difficult to determine the content of the communication but the amount of data and destination of the connection can still be analyzed.

Active attacks consist of acting on the communication in the stream, this can be done in various forms: from injecting, removing or modifying packets to capturing packets and then sending them again at a later time, for example *replay-attack*[16].

2.3.2 Threats on the embedded system

Kleberger et al. [17] describe the different security aspects which exist for in-vehicle networks - among them, aspects which affect the CAN bus. Most of those threats apply also for in-boat networks and other networks that use CAN. It is clear from their analysis that access to the CAN bus means high risks. *DoS* attacks on the CAN bus can easily be performed by sending high priority messages - effectively stopping all communication on the bus. By being able to simply read the CAN communication an attacker will be able to reverse engineer parts of the system, e.g. understand the protocol being used by the different ECUs to communicate. An attacker could also be able to control certain tasks of the system that could even be the cause of death, for instance by controlling the steering or speed of the boat.

Added to these attacks there exist other security threats if the attacker is able to get physical access to the system. If the system is not tampering resistant, the attacker can gain access to sensitive data that has been stored - e.g. certificates used to validate the boat which could then be used by an attacker to disguise itself as the same boat. Other attacks that involve physical access include so called side channel attacks, where the attacker measures sound, power consumption and so forth to determine the workings of the system or algorithm[18].

2.3.3 Threats related to mobile hotspot

When it comes to a cell phone's shared network not much (none that we found) research has been done on what new threats arise for this kind of Wi-Fi routing. With that

said, most, if not all, threats that apply for regular routers apply for cell phones as well. Kanawat et al.[19] list several of those threats, of which DoS attacks is the only threat for which we could not otherwise protect from in a higher layer, e.g. at the Application layer with encryption. DoS attacks could potentially make it impossible for the *connection device* to connect to the server.

2.4 Protection

The goal of adding protection to a system is to preserve the *confidentiality* of the data to avoid any third party from reading it. *Integrity* to avoid the unauthorized modification or destruction of data and lastly *availability* which establish the correct and reliable service of the information. These three aspects of security are usually mentioned as **CIA**. Last but not least **authenticity** is necessary to verify the identity of the sender or receiver in a communication channel.

A common method for achieving the protection of a communication channel is to employ a cryptographic protocol which encrypts the content in the channel. Encrypting the communication channel provides confidentiality, ensuring that the content becomes unreadable for a third party, though the amount of data sent between each host can still be observed. To provide integrity usually a *Message Authentication Code* (MAC) is used, which authenticates the message sent and provides detection of modification of the data. One common implementation of MAC is to use a cryptographic hash function such as SHA1. To provide authentication, the host needs to be verified - this can be done with the use of RSA keys which have been shared through a secure channel to avoid man-in-the-middle attacks, since the keys could be intercepted and changed[20].

3

Technical Background

3.1 Transport Protocols

Transport protocols exist on layer four in the TCP/IP stack and determine the properties of the communication in the application layer. There exist multiple protocols but the two main ones are *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP). TCP is a streaming protocol that provides reliable communication, ordered packets and error detection. UDP is a connectionless protocol that transmits data in the form of packets (datagrams) and does not provide order or reliability, though checksum exists for error detection[21][22].

3.2 Cryptographic protocols

Cryptographic protocols are used to secure a communication channel from third parties. This protection does usually involve three main aspects: The authentication of the host using the channel, confidentiality of the channel itself to avoid disclosing the content, and authentication of the messages sent to hinder manipulation of the transmitted data.

3.2.1 Transport Layer Security

Transport Layer Security (TLS), previously known as *Secure Socket Layer* (SSL), is a cryptographic protocol that exists on the application level of the TCP/IP stack. The protocol is implemented as part of an application to secure the communication. It exists

between the transport layer, more specifically TCP, and the application[23]. TLS is highly customizable in the sense that there are multiple choices to be selected for key exchange, cipher, compression and hash algorithms. Authentication is optional and when enforced TLS uses X509.v3 certificates. The first release of TLS was 1.0 which originated from SSL 3.0, the latest version of TLS is 1.2. Changes between the versions of this protocol consist of adding new options for encryption and security fixes. The security of TLS is not only dependent on the version of TLS but also on the algorithms selected for the different parts of the protocol. TLS is commonly used for securing web based communication for online-shopping and monetary transactions[24].

3.2.2 Secure Shell

SSH and TLS show strong similarities since both are implemented at the application level of the TCP/IP-stack. One major difference between the two is that SSH supports tunneling of protocols within an SSH tunnel, making it highly flexible to secure other non-encrypted protocols. SSH also supports a wide variety of authentication methods, while TLS mainly supports x509.v3 certificates. Compared to TLS, SSH requires the authentication of the client and server. The protocol supports many authentication methods[25], among them are: public key authentication, certificate authentication, password authentication and host-based authentication[26].

3.2.3 Internet Protocol Security

Internet Protocol Security (IPsec) is considerably different compared to the previous two protocols since IPsec is implemented in the network layer of the TCP/IP-stack. Because of this, IPsec can encrypt all protocols that are in the above layers, including all application and transport protocols. The protocol also encrypts protocols on the network layer, depending on operating mode. The difference depends on which mode is selected for the communication: tunneling or transport mode. Transport mode encrypts the data of the IP packet, but not the IP header, this mode is used in direct communication between client and server. Tunnel mode is used for site to site communication and in this mode, all communications are encrypted. Due to this the entire IP packet is encapsulated inside IPsec and another IP header is added with the destination of the other site. When a packet arrives at the destination site the receiver of the IPsec packet decrypts it and forwards the decrypted IP packet to the correct host on the network.

Like previous protocols, IPsec also supports two-way authentication. Since IPsec operates on a layer below TLS and SSH this protocol becomes more complex to configure, from firewall configuration to *Network Address Translation* (NAT) traversal[27].

3.3 Ciphers

There exist various cryptographic ciphers which are used to ensure that the messages sent remain confidential when transmitted over an insecure channel. Ciphers are divided into two main categories: stream and block ciphers. In the case of stream ciphers, they work by using a pseudo-random key stream which is used to continuously encrypt a data stream. With block ciphers, the algorithm works on blocks of data instead of a stream, the data to be transmitted is divided into blocks, the algorithm then operates on these blocks to generate the cipher text.

Stream ciphers are generally perceived to have a higher throughput than block ciphers. The benefit with block ciphers is that it is better understood what makes an implementation secure[28]. There exist numerous different block and stream ciphers but the more common are *RC4* which is a stream cipher, *Data Encryption Standard* (DES) and *Advanced Encryption Standard* (AES), these last two are block ciphers. DES was the previous standard for encrypting data, but flaws have been discovered which caused the development of 3DES which runs DES in three iterations. The issue with this solution is that throughput becomes low, and to resolve this issue AES was developed which is the new standard for block cipher[29].

3.4 Block cipher mode of operation

Block cipher modes in cryptography determine the *Input Vector* (IV) to be used in conjunction with the block to be encrypted and how this will be used next to encrypt the next block of data. The simplest mode of operation is to not use an IV and simply encrypt each block of data separately, this mode of operation is known as *Electronic codebook* (ECB). Three other examples of commonly known block cipher modes are *Cipher-block chaining* (CBC), *Counter mode* (CTR) and *Galois/Counter Mode* (GCM). ECB is generally not recommended since it causes identical plain texts to get identical ciphertexts. With CBC the ciphertext is XORed with the next plain text, this causes each cipher text to become unique. CTR mode uses an increasing integer value together with a nonce, these get encrypted and the result is then XORed with the plain text to obtain a ciphertext, again this means different ciphertexts for the same plain texts. GCM combines the CTR mode with the Galois mode of authentication. The strong benefit with using GCM and CTR over CBC is that they can run in parallel which is not possible with CBC due to the dependency between blocks[29].

3.5 Cryptographic hash algorithms

Cryptographic hash algorithms are used for authentication of messages. They create a unique fingerprint of the message. If the message is changed during transmission or by

a third party the hash will no longer match the message, invalidating it. There exist many different algorithms for hashing with various benefits and drawbacks, for example high throughput but lower security. The *Secure Hash Algorithm* series are commonly used and exist with various output size, *SHA-1* has 160 bits, while other SHA has the number of bits included in the name such as *SHA-256* and *SHA-384*.

4

Related Work

Alshamsi et al. [30] give a comparison between *Secure Socket Layer* (SSL) and *Internet Protocol Security* (IPsec). The comparison consists of the setup of two systems which include key exchange, *Message Authentication Code* (MAC), supported encryption algorithms etc. The report highlights different benefits and drawbacks for each protocol, such as IPSec having more overhead but multiple users can use a single connection thus reducing the overhead for establishing connections. Possible problems with a single tunnel is that if it becomes compromised it will affect all connections. SSL on the other hand has one connection per session, meaning that a compromised session will not affect other sessions. Another concern is that SSL does not support compression to the same extent as IPSec, which could make IPSec better for low bandwidth connections. IPSec is also more complex to configure due to specific requirements on the firewall, while SSL requires minimal of these configurations. There are also incompatibility issues between using IPSec from different vendors which SSL does not have.

Kleberger et al. [17] summarize security threats that exist for the internal communication within a vehicle. The report gives a brief description on which domains can introduce security threats to the connected car. These domains consist of the internal communication bus, portal of the automotive company and the communication link connecting these two domains. The paper describes weaknesses that exist in the current vehicle communication buses such as CAN. These threats consist of factors such as lack of bus protection, weak authentication, misuse of protocols and information leakage. One issue with implementing security features in the communication bus is the cost constraints for new security solutions.

Ravi et al. [31] describe the various aspects of designing a secure embedded system. From the areas that are involved in creating security, which include identification, au-

thentication, confidentiality, availability as well as tampering protection, to the threats that can compromise this security, which include logical, physical and side-channel attacks. The paper also describes the various solutions of creating this security by the usage of symmetric ciphers for data encryption, asymmetric ciphers for authentication and hashing algorithms for message authentication. Various issues specific for embedded systems are mentioned such as the problem with processing complex security protocols with increasing amount of data with limited processing power. This issue is described as the *Processing Gap*. Performance tests are also done with the SSL protocol to get a profile of how the various phases of the protocol affect performance. The results show which phases of the protocol use the most resources depending on the amount of data that is transmitted. At low data rates the authentication phase contributes the most while the symmetric encryption does at large data rates. The report also includes a graph for estimating the amount of encrypted data that can be transmitted per second given MIPS of the processor, excluding various cryptographic hardware accelerators.

Mathias Johanson et al. [32] describe the creation of a system that tunnels CAN messages from an internal bus over the Internet to a remote host. The report mentioned GPRS, 3G and WLAN as possible communication channels. The report also describes a created protocol for sending CAN messages, this protocol consists of four value types: channel, length and payload, it also mentioned that this protocol can be used over UDP, TCP and UDP/RTP. The prototype that was built consisted of an embedded Linux system, with support for using GPRS/EDGE and WLAN communication, with TCP as the transport protocol. To secure the communication the prototype used various methods. To avoid eavesdropping they used public keys for authentication together with blowfish algorithm for encrypting the communication data.

To evaluate cryptographic protocols, [30] was used, while [17] helped us get an overview of security threats to internal communication. In [31] we got an understanding of possible security threats as well as impact of adding security for embedded devices. Finally [32] has strong similarities to our product which enables us to draw experience from their solution.

5

Analysis

Figure 5.1 gives the description of the system. Those elements that are involved in this project are colored red and consist of three elements: *Remote service station* which is the server to which the client will connect to, *Telematic interface* is the *connection device* which is connected to the CAN network and this module will establish the connection to

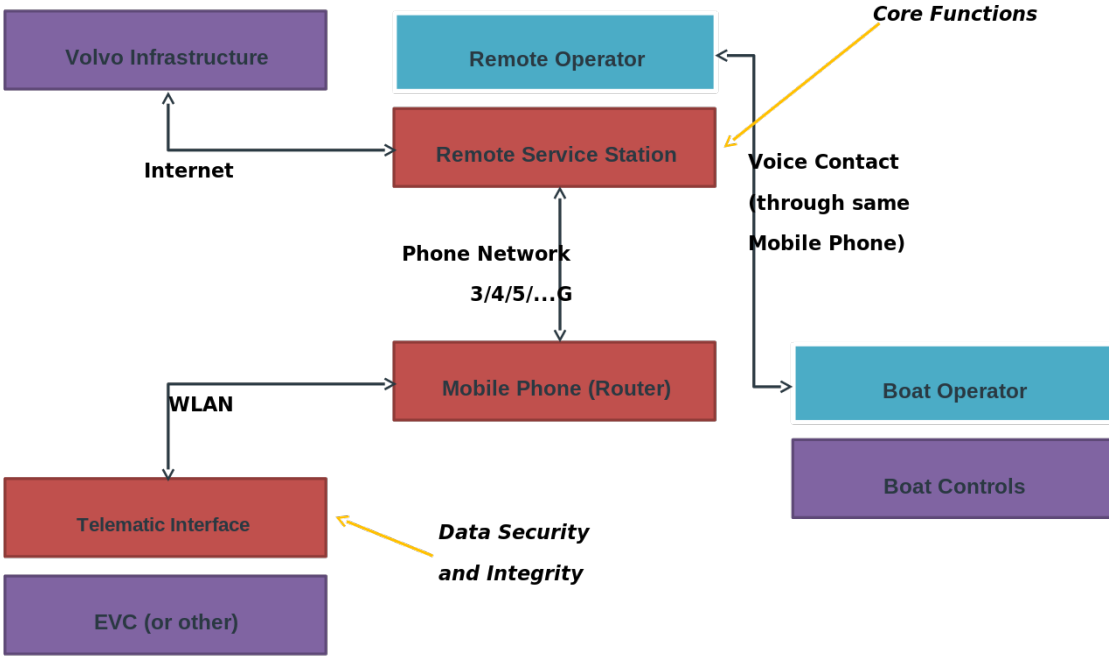

Figure 5.1: Design of the system

the server. Between these two components is the *Mobile Phone (router)* which enables them to communicate with each other. The light blue elements show individuals which are involved in using the finished product. The *Boat operator* can either be the owner of the boat or a technician, who will help the *remote operator* to analyse the system. The purple elements are hardware that exists in the sphere of the solution but that are not developed, namely the *EVC (or other)* which is the CAN system on the boat, the *Boat Controller* and the *Volvo infrastructure* which can be a customer database.

5.1 Available resources

Various resources were made available during the project. These consisted of hardware for creating the system and software for verifying its correctness. The available resources were:

- PCAN dongle
- Test rigs
- IAR KickStart kit for STM32F107VC
- Wireless LAN serial port adapter (OWS451)
- EVCFlash

CPAC's WiFi network as well as our own mobile phones were used to establish the communication between the client and server.

5.1.1 PCAN dongle

A PCAN dongle is a CAN-to-USB adapter that can be used to read and send CAN messages on a PC.

5.1.2 Test rigs

The test rig (Figure 5.6) was provided by CPAC. It mimics the system of an actual boat. This rig was used to test the various operations named under section 1.5. The rig contained two CAN buses, one for port and the other for starboard. Boats with more motors contain more buses but due to the STM32F107VC evaluation board (see 5.1.3) only having two CAN connections it was only possible to support at most two CAN connections. We used two PCAN dongles to examine the traffic on these rigs. We found out that over the two CAN buses the total amount of CAN messages per second was

~950. This was confirmed by technicians at CPAC to be a realistic value for actual boats of this kind[1].

5.1.3 IAR KickStart kit for STM32F107VC

The KickStart kit consisted of a STM32F107VC evaluation board and development software[33]. The relevant parts of the hardware were:

- Cortex-M3 CPU, 72 MHz maximum frequency[34]
- 64 to 256 Kbytes of Flash memory
- 64 Kbytes of general-purpose SRAM
- 2 USART ports
- 2 CAN ports

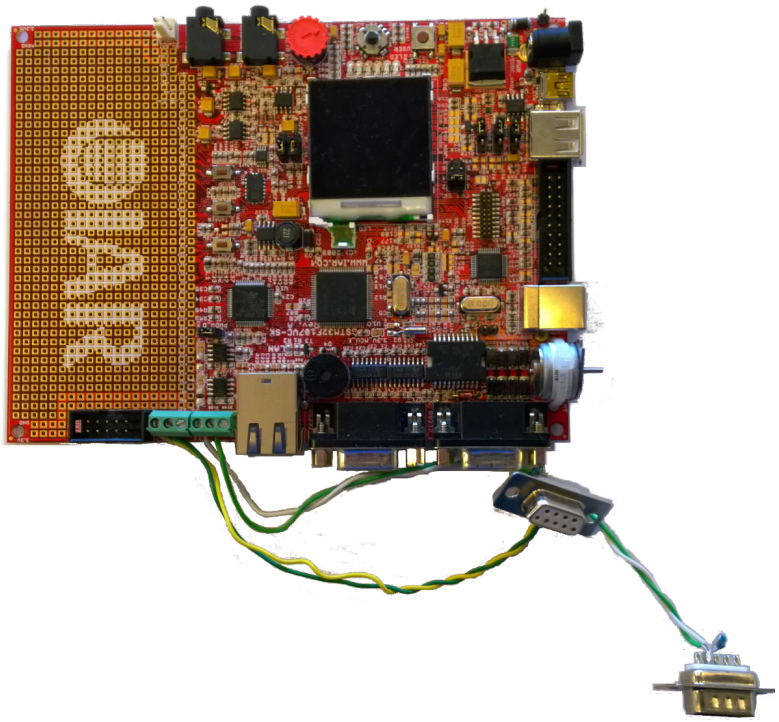

Figure 5.2: IAR development card

[1]Pleasure boats with two and four engines. More engines barely add to this number of CAN messages/second.

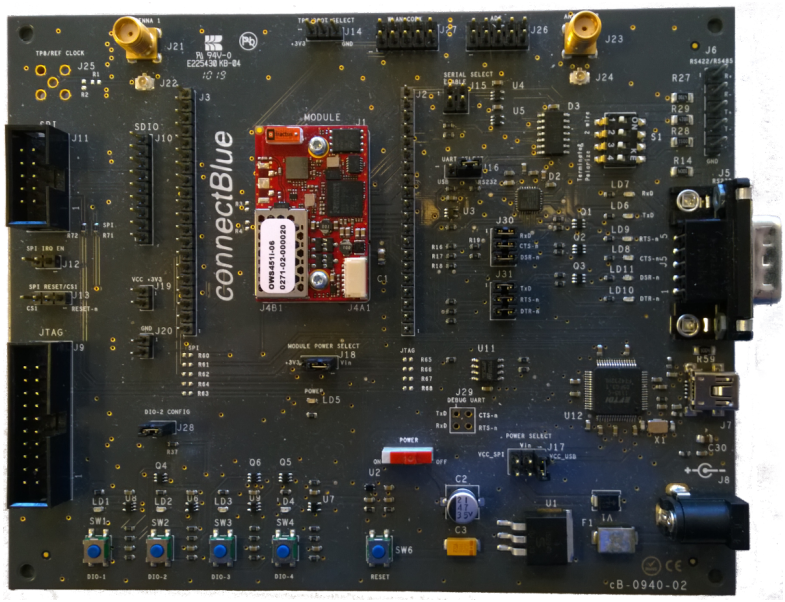

Figure 5.3: connectBlue development card

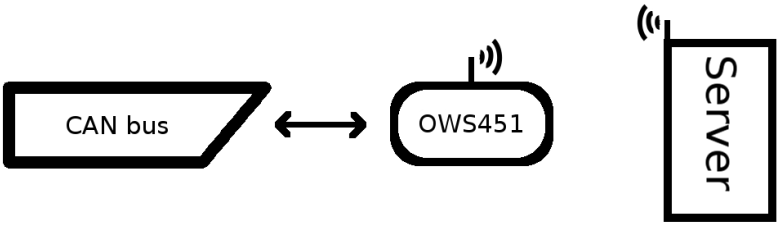

Figure 5.4: OWS451 communication

5.1.4 Wireless LAN Serial port adapter - OWS451

The WLAN Serial port adapter (Figure 5.5) was provided by connectBlue and it is able to connect to a wireless router (with WEP, WPA and WPA2 encryption[35]) to obtain Internet access (or access to a private network). This way the device is then able to communicate with a server through either UDP or TCP without giving access to lower layers of the TCP/IP stack. The module was installed on a development card (Figure 5.3) provided by the same company. Since this device does not give access to the lower layers of the TCP and UDP stacks, one cannot for example decide the size of the transmitted TCP or UDP packets, which could have been advantageous when trying to obtain better data throughput.

Figure 5.4 shows how this device establishes a TCP/UDP connection to a server making it possible to transmit and receive data to and from the server via serial connection to this device.

Figure 5.5: connectBlue OWS451 WLAN module

It supports bit rates between 300 and 2 764 800 bits/s. The default bit rate is 57 600 bits/s (or 7 200 bytes/s). Since we expect to receive 950 CAN messages per second and a CAN frame is at most 16 bytes, we get 15 200 bytes/s as the least required bit rate for unencrypted communication. This is a rough calculation but it is already obvious that the default bit rate is not enough. In addition, the protocols and encryption will also generate some overhead. We tested different bit rates on this device and even though it is supposed to support a wide range of bit rates we found that it got unstable when running over 460 800 bits/s (57 600 bytes/s), although the fault may lie on our STM32 card. This bit rate (~56 kB/s) should be more than enough for our purposes. This theory was later on confirmed to be correct, see 7.1.1.

The device contains a buffer of 256 bytes, which is where it caches data until it successfully connects to a server. If it gets full, new data sent to it will overwrite the oldest data. According to the documentation of the device (and we never seemed to experience the opposite) this seems to only be the case when the device hasn't connected to a server, not afterwards (the documentation is not 100% clear on this).

The device has two connection schemes: *connect on data* and *always connected.* The fist one ensures the device will attempt to connect only after we try to send data to the server. The latter one will connect as soon as it is powered on, it will not wait until we try to send data.

5.1.5 EVCFlash

EVCFlash is a program developed and mostly used internally at CPAC. The program is written in Perl and is capable of performing several operations once connected to the boat's system via CAN. The operations that are of interest for this project were: parameter configuration and software update. This program was thus used to test if these operations could be performed over the Internet, see Chapter 7.

Figure 5.6: Simple test rig

5.2 Selection of transport protocol

As previously named (5.1.4) we only had access to TCP and UDP for accessing a server. The requirements taken to decide which transport protocol to use consisted of the importance of detecting errors in the transmitted data to avoid sending erroneous data on the internal CAN bus, and that all packets are received and be easily ordered. This is very important when updating software.

Error detection is supported by both protocols. On the other hand, UDP does not support packet loss protection nor does it guarantee that the packets received are in the transmitted order, TCP on the other hand supports both. Another important factor in the decision was that two of the three cryptographic protocols mentioned in section 5.3 required TCP to function[2]. Due to these requirements, the choice was to use TCP as transport protocol for the communication.

5.3 Selecting cryptographic protocol

Due to the risks of opening the internal communication of the boat to the Internet, it is important to ensure that the communication is secured from access from an unauthorized party. To enforce security, the communication between client and server must be encrypted and the protocol should support two-way authentication in order to guarantee that the client connects to the correct server and vice versa.

The communication between the client and server should not be possible to be read or modified by any third party, thus, the protocol is required to support both encryption of the communication data and the verification of this data with the use of *Message Authentication Code* (MAC). With these requirements defined, there were three different cryptographic protocols identified for the project:

- **SSH** *Secure Shell*
- **TLS** *Transport Layer Security*
- **IPsec** *Internet Protocol Security*

5.3.1 Transport Layer Security

TLS has multiple benefits for the project, certificates offer the possibility to use a single root certificate for all clients. This would enable the server to know only a single certificate and each client to have unique certificates. Another benefit of TLS is the possibility

[2]There exist an implementation of TLS over UDP called DTLS but it simply tries to recreate TCP. It is only used in cases where TCP simply isn't an option. A big drawback of DTLS is that it is not updated against new threats as frequently as TLS. There is no standard implementation of SSH over UDP but Ullholm et al. [36] show how it could be possible and list potential advantages and drawbacks.

to choose from a wide variety of different cipher suits, thus it is possible to adapt it to the system requirements of the embedded device.

5.3.2 Secure Shell

There exists no need for tunneling any other protocol in the secured communication channel, thus removing a strong feature for SSH. Due to the clients not having a static hostname it will not be possible to use host based authentication, also due to security issues related to the length of the password, password authentication should be avoided. Another option would be public key authentication, but this lacks scalability with large number of clients due to key management. Certificate authentication is therefore the only valid authentication method that could have been used on this project.

5.3.3 Internet Protocol Security

Due to hardware limitations it was not possible to use IPsec for securing the communication between the client and server. This was because the embedded system does not provide access to the network layer of the TCP/IP-stack, instead only transport layer and above is available. Other concerns regarding the project is that most routers will be using NAT which could cause issues with IPsec [30].

5.3.4 Summary

Due to these considerations it was decided to use TLS for securing the communication between client and server. This decision was made since IPsec was not possible to use with the hardware and because we lacked the freedom to configure the network in which the client connects from. SSH was also a good option but due not needing features such as tunneling and due to the lack of certificate support, we decided to use TLS instead.

5.4 TLS specific considerations

After TLS had been selected as cryptographic protocol for the communication it was necessary to take various decisions regarding which TLS library and cipher suit to use.

5.4.1 Selection of TLS library

There exist various libraries that implement TLS [37]. To pick the right one for this project some requirements were taken into account. The library had to be open-source

Table 5.1: TLS libraries that matched criteria

Implementations	Open Source	Software licence	Code Size (kSLOC)[3]
Botan	Yes	Simplified BSD License	32
cryptlib	Yes	Sleepycat License and commercial license	?
CyaSSL	Yes	GPLv2 and commercial license	67
GnuTLS	Yes	LGPL	138
MatrixSSL	Yes	GPLv2 and commercial license	18
Network Security Services	Yes	Mozilla Public License	400
OpenSSL	Yes	OpenSSL / SSLeay dual-license	159
PolarSSL	Yes	GPLv2 and commercial license	14
Secure Transport	Yes	APSL 2.0	?
JSSE	Yes	GPLv2 and commercial license	37
Bouncy Castle	Yes	MIT License	56
LibreSSL	Yes	OpenSSL / SSLeay dual-license	56

in order to support adaptation of the library to the embedded device. Budget constrains also required the library to be free of charge, at least before the project goes into production. Due to security concerns with older standards of TLS it should support TLS 1.2. A list of suitable libraries are shown in Table 5.1. PolarSSL[38] was chosen as the TLS library for both the server and client because it met previously mentioned criteria and because of the good documentation, usage of standard libraries and predefined projects for Visual Studio, making it the easiest of them to start coding on.

[3]kSLOC = thousand source lines of code

5.4.2 Selection of cipher suit

PolarSSL provides a wide number of cipher suites to secure the communication. They are listed here: [39]. Our main focus when deciding which one to use was:

- Low memory requirements due to small cache on OWS451
- Small performance hit
- Information protection for as long as possible

Of the different available key exchange algorithms provided by PolarSSL, ECDHE-ECDSA was the best of them all in all of the previously mentioned criteria. This is mostly due to *elliptic curve cryptography* (ECC) being very efficient when it comes to memory concerns without affecting performance badly. ECC is also most desirable for our project since the keys can be smaller than the small cache found in the OWS451 (see 5.1.4).

When it comes to securing the information for a long time *Advanced Encryption Standard* (AES) with 128 bits was considered to be good enough as symmetric encryption since the same number of bits are considered to be secure for 25 years. For *Hash-MAC* (HMAC) at least 224 bit should be used [40], therefore it was decided to use SHA256 for HMAC. For these reasons it was decided to use *TLS-ECDHE-ECDSA-WITH-AES-128-GCM-SHA256* as cipher suit for the communication between the client and the server.

5.5 Wireless Router Connection

The *connection device* needs to connect to a wireless router in order to reach the Internet. Because of the project's requirements (see section 1.5) this should be done automatically - no input should be given to the device for it to find the correct router. Since the cell phone owner should be the one enabling Internet sharing, they could be asked to set the SSID to something of our choosing. It could then also be possible to ask him to set a specific password for it. There are two options then: Have a specific SSID and password for the *connection device* to find or have the SSID and password to be randomly generated. The second option is the most secure one. A simple rule could be done so that the beginning of the SSID starts with some standard name, like *CPAC* and followed by a random part, let's call that X. The password Y could then be generated and calculated using X. For example with X *op* Z = Y, where Z is unique value for each device, only known to it and the server and *op* an appropriate operation. We did not focus on this aspect but it is important to make sure this issue is resolved.

5.6 Server - Client communication

For the server and client to be able to communicate they should speak the same *language*. A common protocol for sending CAN messages and other kinds of messages, e.g. handshake messages, is therefore necessary. Johanson et al. [32] proposed a protocol for relaying CAN frames over the Internet. This protocol describes a message frame to be used to send messages that can be sent both over TCP and UDP. These frames can be used to send many different types of messages and allows, besides sending CAN messages, to set filtering rules, periodic frames and more. Although we do recommend the use of this protocol, with some minor changes, we found that we didn't need to fully implement all parts of the protocol for our tests. For instance, since in the system we will be using for our tests only *extended frames (29 bit addressing)* will be sent, we did not have to add support for *standard frames (11 bit addressing)*. We also needed to send timestamps for the CAN messages, something that is not supported by the previously named protocol, since this is necessary when logging data.

6

System Implementation

This chapter consists of two parts: 1) Establishing a connection between the *connection device* and the server and 2) testing the communication capabilities. Then the connection is secured and testing is done again. Testing was done with *EVCFlash* (see 5.1.5) and a program developed by ourselves for testing which operations (error code reading, diagnosis, software update etc) work. This testing was done with and without TLS implementation. The last part of the testing consisted of trying different ciphers to find the best performance. Both sides of the communication (server side 6.2 and client side 6.3) need to understand each other (see 5.6) so a communication protocol (6.1) was designed.

6.1 Communication protocol

As noted in the previous chapter 5.6, there was a need for a communication protocol between the server and the client. For that reason a protocol was designed for sending messages from and to the server (Figure 6.1). A frame of this protocol contains a header and a payload. The header consists of three parts: length, info and type. Where *length* is the number of bytes in the *payload*, *info* specifies information which is dependent on the type, and *type* defines what kind of message this is.

This protocol was simply named *CAN communication protocol*.

Figure 6.1: CAN communication protocol

6.2 Server side

The development of the server side application was done in two steps: the first one was to develop the server for the specified tasks and after this was tested the second step consisted of adding TLS to the communication.

6.2.1 Server implementation

The server was implemented as a Windows based C++ dynamic-link library (DLL), with an interface for it written in C code. The reason behind not having the interface of the DLL written in C++ was due to incompatibility in the support for C++ structures such as the *String* & *Vector* types in other languages. The need to support other languages then C & C++ comes from testing tools used to verify the product not being written in any of these two languages. The design of the server was to enable multiple clients to be connected at once, with an interface for other programs to communicate with the connected clients. This server program understands the *CAN communication protocol* only at a basic level. The information that the server interprets is type and size of the message, it is up to the program that uses this DLL to interpret the remaining parts of the messages.

6.2.2 TLS implementation

To simplify debugging of the communication it was decided to implement the TLS functionality as the last stage of the server implementation. The addition of the TLS support to the code consisted of changing the calls to write and receive functions to that of the PolarSSL library (ssl_write, ssl_read). It was also decided that each client should have its own entropy source to increase the security of the connection. The only elements that are shared between connections are the certificates, all other elements are generated when a connection is established.

6.3 Client side

The implementation of the client program was done in two steps: Establishing an unencrypted communication (6.3.1 Client implementation) to the server and then securing said communication (6.3.2 TLS implementation).

6.3.1 Client implementation

As mentioned previously in 5.1, we had access to two components that could be used to create the *connection device*. A Wireless LAN Serial port adapter (OWS451) with its corresponding development board and an IAR KickStart kit for STM32F107VC. Those two components were connected through serial communication (RS232).

The STM32F107VC was programmed to set up the OWS451 so that it connects to a particular router of our choosing and then establishes a TCP connection to our server. The router it connects to was hard coded in the beginning, and remained so during our thesis work (there is an issue with this explained in 5.5). The same thing applied for the server, the IP address was hard coded since we did not have access to a domain name.

The STM32F107VC was then configured to receive and send CAN messages. A maximum of two CAN connections could be made on the card. The CAN messages are received and buffered in a circular buffer until they are ready to be sent. They need to be buffered since during the process of encrypting the messages new ones arrive, so the new ones have to wait for their turn. How big this buffer should be was later tested and is explained in 7.1.1.

6.3.2 TLS implementation

The PolarSSL library was used on this project - the reason it was chosen is explained in section 5.4.1. The library includes a client example code that helped to make TLS work in the *connection device*. Only some minor changes were needed, e.g. specify our own certificates. The important changes that were needed were specifying which functions PolarSSL should call for *sending* and *receiving* data - which in our case were simply the functions used to send and read data via the serial port to the OWS451. Other concerns was the lack of a source of entropy on the STM32F107VC card, therefore it was specified that clock info should be used to help generate random values, while not the most optimal solution it was simply the only one available.

One of the main problems when implementing PolarSSL was the need for memory, specially RAM memory especially during the initial TLS handshake where certificates are received from the server. Besides the handshake, increased memory demand also depended on the fact that TLS added some extra time to send the messages to the

server. This extra time meant that the buffers used to save CAN messages needed to be as big as possible to avoid overwriting old CAN messages.

7

Testing and test results

The ciphers used for the testing had similarities to the cipher selected in 5.4.2, except for RC4 and DES which were used as reference.

7.1 Communication quality

Several tests were made to assert the quality of the communication. To do this we focused on:

- Data throughput
- Data loss
- Communication delay

The tests were performed with and without encryption for comparison.

7.1.1 Data throughput and data loss

Encryption impact

The purpose of this test was to see how much the data throughput was affected by the TLS encryption. To test the data throughput we programmed the STM32 chip to send as many unencrypted faked CAN messages as possible. This means that the result will depend on how fast this chip works and gives us an upper bound on how many messages this particular device is able to send. This upper bound can be compared with the number of CAN messages that are normally sent by the internal bus, were this device

should be installed. We then run the test with the different ciphers we had to choose (described in 5.4.2).

The upper bounds that we found by faking CAN messages were:

- Without timestamps: ~3300 msg/s (CAN messages per second)
- With timestamps: ~2550 msg/s
- Real value[1]: ~950 msg/s

The test was then run using the cipher suite that we had initially chosen and also with other ciphers that were found to be suitable but not preferred, for comparison. As can be seen in Figure 7.1, the values never go over the necessary 950 pkt/s for the connection to be able to send all the messages that are expected to be received in a real life setting. The reason for this is that each CAN message was encrypted separately before being sent. Better results can be obtained by encrypting a certain number of CAN messages together and then sending them at the same time. Figure 7.2 shows the impact of bundling multiple CAN messages together in relation to throughput; larger bundles increase throughput. By sending no less than seven CAN messages simultaneously any cipher can be used to ensure that no CAN message is lost, at least when the CAN messages aren't timestamped. If encrypting with RC4 only a minimum of two CAN messages are required to be sent together. For the cipher we picked (see 5.4.2) we need to send at least three messages together. Figure 7.3 shows performance when the CAN messages are timestamped. Adding timestamps require at least 10 messages to be bundle together, for all ciphers to meet the minimum requirement of 950 msg/s. But just like without timestamps, two messages are enough for RC4 to work and three for the cipher suite we initially picked.

OWS451 bit rate

As explained in 5.1.4 the bit rate at which the OWS451 device runs is important since if it is not fast enough it will not be able to properly send all the data, leading to data loss.

The maximum bit rate that the OWS451 could properly handle[2] was 460 800 bits/s, or 57 600 bytes/s. During tests in the first section of 7.1.1 we found that said bit rate was never exceeded. Even when sending twice the number of necessary CAN messages (~2000) the bit rate was around 330 kbits/s. Therefore, said bit rate was more than enough.

[1] This is the maximum number of CAN messages sent over two CAN buses in the boats where our device is expected to be installed. See 5.1.2

[2] This was explained in 5.1.4

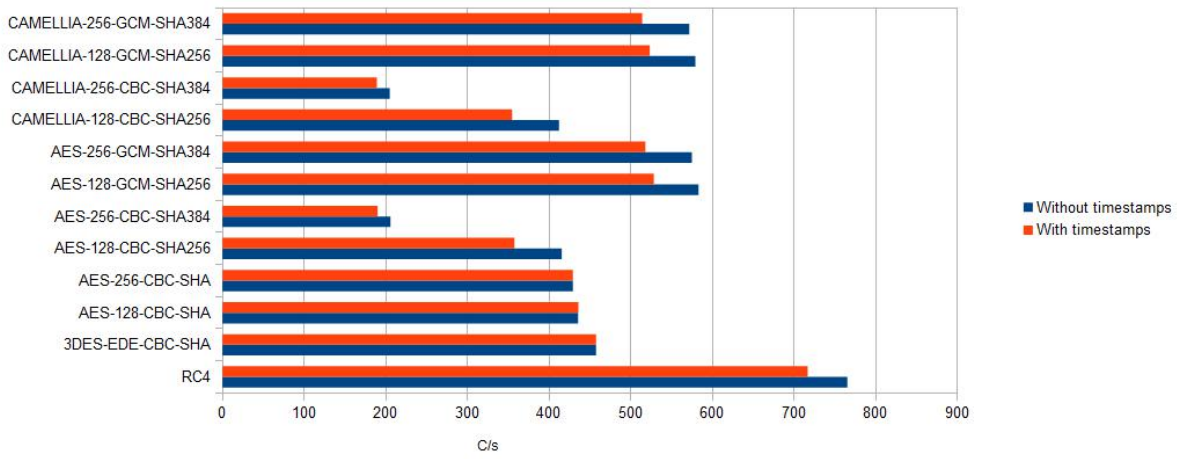

Figure 7.1: Maximum throughput (packets per second)

Buffer size

As noted in 6.3.1, the arriving CAN messages are stored in a circular buffer until they can be sent to the server. Since it takes time to encrypt messages and then send them, they need to be buffered. If we buffer too few, they get overwritten when the buffer gets full, and if we buffering too many is a waste of memory. In our case it is better to buffer too many than too few, but we can not waste too much memory, therefore it was important to find what the minimum of messages to be buffered was.

To test this we sent CAN messages to our device, both with a PCAN and also from a test rig (Figure 5.6). From the PCAN we tested sending between 1000-2000 pkt/s. The test rig sends as usual what is normally to be expected from a real boat connection: 950 pkt/s. It was then indeed noted that when ensuring that enough[3] CAN messages are encrypted and sent together the buffer did not need to be that big, a size of four was usually more than enough.

7.1.2 Communication delay

We could observe that the delay was on average four ms in round-trip. We obtained this value by continuously timestamping several messages, sending them to the device and waiting for the echo reply and calculating the time difference from transmission. This test was done by having the device connected to the same network as the server.

[3]The number of CAN messages that should be encrypted and sent at the same time depends on what cipher suite is being used at that particular time. The results from previous tests on throughput shown in Figure 7.2 and Figure 7.3 can be used for guidance.

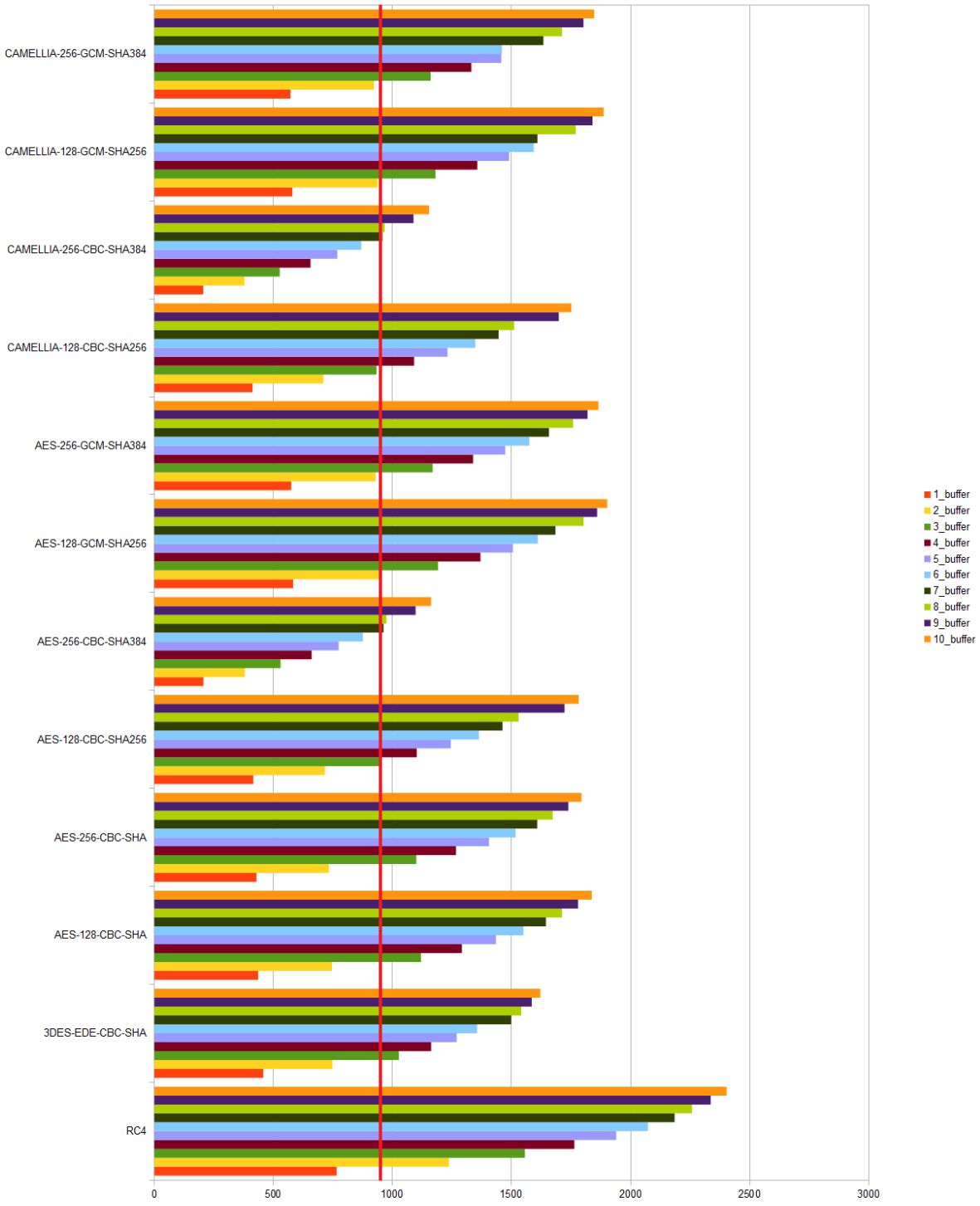

Figure 7.2: Throughput depending on packet buffer, without timestamps. *Red line* realistic throughput value when deployed.

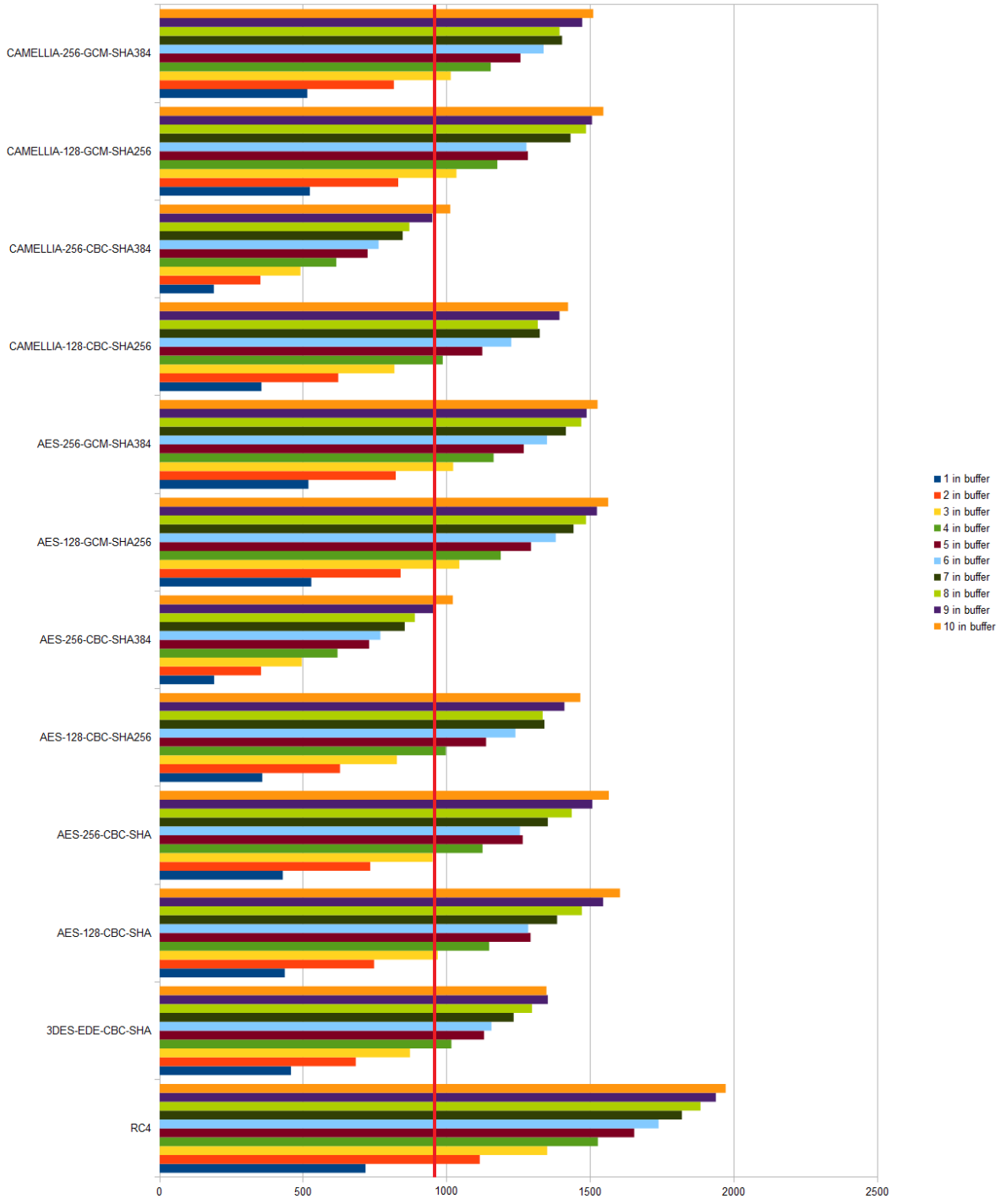

Figure 7.3: Throughput depending on packet buffer, with timestamps.
Red line realistic throughput value when deployed.

7.2 Supported operations

As noted in 1.5, the system should be able to perform certain operations, there is no point in a secure connection that can not be used for practical purposes. The operations that we tested were:

- Remote diagnostics
- Data logging
- Parameter configuration
- Software update

The goal of these tests was to verify the validity of the communication and the performance. Two different tools were used during the test, one which was developed as part of the project and EVCFlash an internal testing tool at CPAC.

7.2.1 Remote diagnostics

For this task a program called *Remote diagnostics* was created using Visual Studio and written in C#. The program was able to use the server library (DLL file) described in 6.2.1. Also, as noted in that same chapter, the server library only understands the CAN communication protocol at a basic level. This means that this program needs to understand the protocol in order to send and receive data.

The program had two different ways to use the communication:

- Show error codes
- Display and interpret all data being sent

These tasks were used to see if all data was being sent. The data received was compared to the data on the CAN bus to which we had access by being directly connected with a PCAN dongle.

7.2.2 Data logging

For this test the same program was used. All the CAN messages received are simply stored in one of two formats .asc or .trc. This task is trivial, the only requirement is to not lose CAN packets, or at least lose as few as possible. Reasons for data loss could be loss of Internet access or the buffer on the client side getting full leading to old CAN messages being overwritten by new ones.

7.2.3 Parameter configuration

The server library was added not only to the program but also to EVCFlash, in this case to utilize EVCFlash's parameter configuration capabilities. EVCFlash normally uses a CAN-to-USB adapter to connect to the bus, either a Kvaser or a PCAN dongle. The server library was disguised as a dongle in the newly implemented module for EVCFlash. This way not many changes were needed and we could test our system as a CAN-to-Internet dongle.

Once implemented, EVCFlash was used for parameter configuration as if it was connected to a regular dongle.

7.2.4 Software update

EVCFlash was used to test if software could be installed on the boat components through our system. The time it takes to update software through a regular PCAN dongle was compared to the time it took (when it worked) with our system. The EVCFlash program has two ways of updating the software on an ECU, an older that does not put any requirements on the connection's speed and a newer that does. The one that does put requirements on the speed demands the delay on the messages to be much shorter than the one we can achieve. This is also the way to be used in the future, the other less restrictive one is outdated. Our connection can only support software download with the older process, not the new one.

8

Results

8.1 System

This section describes the finished system including the client and server, and the protocol they use to communicate. Also described are the capabilities and security of the finished system.

8.1.1 Communication protocol

The communication between client and server is done with the protocol seen in Figure 8.1, this protocol consists of four different sections. The first, *Length*, consists of an 8 bit unsigned char which specifies the length of the *Payload*. The second section, *Info*, contains information directly related to the Type part. *Type* defines how the *Payload* should be interpreted, since the content of the payload differs between types. Both *Type* and *Info* consist of unsigned values. Lastly the *Payload* is the content of the packet. The different types are shown in Table 8.1.

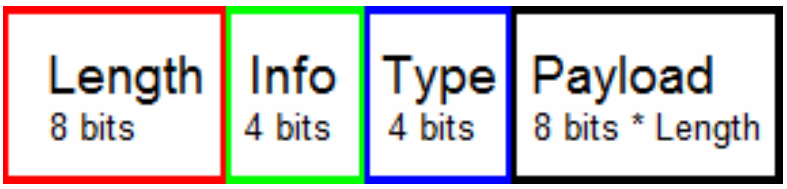

Figure 8.1: Communication protocol

Table 8.1: CAN transfer protocol types

Type name	Type value	Info specifies	Payload contains
Handshake	0	Number of buses connected	Boat name
Single CAN	1	Source/destination channel	CAN ID, data
Timed CAN	2	Same as Single CAN	Timestamp, CAN ID, data
Single RTR CAN	5	Same as Single CAN	Empty
Times RTR CAN	6	Same as Single CAN	Timestamp

8.1.2 Client system

The client system, or *connection device*, consists of *simple* hardware that can be connected to a CAN bus and to a wireless router to gain Internet access and then connect to a server. This way the server and the client can exchange CAN communication over the Internet.

Hardware - Connection Device

The *connection device* we developed consists of two main components: An STM32F107VC card and a OWS451 Wireless LAN Serial port adapter from *connectBlue*. These devices are better described in 5.1.3 and 5.1.4. The two are serially (RS232) connected.

Software

The software running on the *connection device* has three main tasks:

- CAN transfter protocol
- CAN communication
- OWS451 communication

The software is as simple as possible in the sense that it only handles the sending and receiving of CAN messages and not how to perform certain tasks (like the ones described in 1.5). This in order to keep one of the requirements named at the start of 1.5. The software only uses 227 kB of flash memory and 40 kB of RAM.

8.1.3 Server system

The server software is a DLL library to facilitate its incorporation to other software without the need for recompiling. The code of the library could also be easily included directly into C++ projects if necessary. The software waits for a *connection device* to try to establish a TLS communication and then accepts it if the certificate provided by the client is valid. After that both can communicate using the CAN transfer protocol described in 8.1.1.

This server library can be used by a program[1] to send and receive messages to and from a *connection device*.

8.1.4 System quality

As noted in 7.1.1 it was possible to ensure that no data is lost regardless of which of the selected cipher suites (see 5.4.2) is used. It is only necessary to ensure that enough CAN messaged are encrypted and sent together. Chapter 7.1.2 also shows that the delay (over a private network) is on an average of four ms in round-trip which is completely acceptable for a communication expected to take place over the Internet.

8.1.5 System capabilities

This system supports the use of most of the operations that were part of a requirement for this project (see section 1.5). As was found under 7.2, the only operation that cannot be fully implemented is *software update*. This is due to the requirements for low delays to be shorter than what can be realistically achieved over the Internet. Given that the other operations lack this requirement they are totally feasible for as long as it can be guaranteed that no data will be lost.

8.2 Security

To ensure the *Confidentiality*, *Integrity* and *Authenticity* for the communication *Transport Layer Security* (TLS) 1.2 was used. Authenticity between client and server is guaranteed by *Elliptic Curve Digital Signature Algorithm* (ECDSA) with *Elliptic Curve Diffie–Hellman Exchange* (ECDHE) for key exchange. The size of the generated keys used by ECDHE must be 224 bits or longer. Confidentiality is achieved with the usage of *Advanced Encryption Standard* (AES) of 128 bits, with the usage of *Galois/Counter Mode* (GCM) as mode of operation. Lastly the authentication of the entire message

[1]This server library does not fully understand the CAN transfer protocol - it only knows the part shown in Figure 8.1 without knowing what the different type values mean. This means that any program that uses this library needs to understand the rest of the protocol.

was done with the usage of HMAC of 256 bytes, namely SHA256. These security fetures are combined in the TLS cipher suit *TLS-ECDHE-ECDSA-WITH-AES-128-GCM-SHA256*.

With the selected solution it would be possible to protect against a subset of the security threats described in section 2.3. The attacks which the system does not protect against are *Denial of Service* (DoS) attacks, against the CAN communication or implementation issues in the TLS library itself. The system does protect against attacks which involve eavesdropping or alternating the communication, the solution also protects against impersonation due to the mutual authentication.

9

Discussion

9.1 Addressing the requirements

This section goes through the different requirements which were mentioned in 1.5 and discusses how well the finished product responds to these.

9.1.1 Low cost for the *Connection device*

Given the simplicity of the client side software (see 9.1.3) the hardware needed is then also very simple, meaning that the cost for the hardware can be kept low. The prototype for the *connection device* consists of very simple hardware: A STM32F107VC development board and a OWS451 WLAN to serial port adapter. A final version of the product could simply consist of the necessary components from the STM32 evaluation board, listed in 5.1.3. Those specifications cover the requirements for the client side software. As noted in 8.1.2 the software's footprint is 227 kB of flash memory and 40 kB of RAM. Most of that comes from the PolarSSL library. Seeing how the memory footprint of the software is very close to the limits of the STM32 it might be wise to try a different TLS library to make that footprint smaller, like CyaSSL or MatrixSSL. That way it is not necessary to buy more memory to ensure that the hardware will be able to be reprogrammed with newer versions of the software down the road.

Since the solution uses a router to connect to the Internet it was not required for the connection device to support mobile broadband. This reduces the complexity of the hardware and consequently the price.

9.1.2 Low cost for the system's maintainability

Once the client is installed will it be self-sufficient, since only bug-fix updates are needed to maintain a working product. This is due to the simplicity of the device in which very little processing of the data is taking place. Also the decision to not support an application for smart-phones reduces the cost of maintainability, as we do not have to keep track of changes in the smart-phone operating system or SDK. The only cost for maintainability comes from having the servers running.

9.1.3 Simple software for the *Connection device*

One of the core requirements which were specified in section 1.5 was that the node should only consist of being a forwarding node to reduce the complexity of the code. This part was accomplished with the usage of a simple protocol which enabled good flexibility. One concern would be that the current implementation does not provide the possibility to extend the protocol with more types without updating the software in the embedded device, but on the other side, backwards compatibility can easily be achieved which reduces the need for updating old systems.

9.1.4 Automatically connect to WLAN

The automatic WLAN connection was not completed due to time constrains, the working demo that was produced was able to connect to a WLAN by means of hard-coding the SSID for the routers. This solution would not be practical since it would require a unique compiled version on every embedded device, alternatively having identical SSID and password for all WLAN. It is important to have a pattern for finding the correct network, to distinguish between networks in order to connect to the correct one. Another important issue is security, if it is possible to determine the connection details used by the embedded device, an attacker could gain anonymous Internet access. This security risk is partly mitigated with the fact that the router will only be active for a short duration during the time when the customer is in need of support.

9.1.5 Secure connection to avoid compromising the channel

Since TLS 1.2 was used as cryptographic protocol for all the communication between the connection device and client, it was possible to support *Confidentiality*, *Integrity* and *Authenticity* but not *Availability*.

9.1.6 Supported operations

The operations which were specified in section 1.5 were *Remote diagnostics*, *Data logging*, *Parameter configuration* and *Software update*. As mentioned before it was possible to perform all of these except for the last one due to the communication delay, mainly added by communicating over the Internet, which made this impossible or impractical depending on update procedure (not all of the update procedures had a requirement on low delay by this is the direction that CPAC/Volvo is taking).

9.2 Software Design

The design of the *communication protocol* 8.1.1 went through multiple iterations as more of the project became defined. The end specifications showed remarkable similarities with the protocol designed by Mathias Johanson et al.[32], which only was discovered after our protocol was finished. The differences between the two consisted of the order of the different elements of the segments in the header, and the size of these. Their protocol also included some added features that may be worth taking into consideration, such ability to communicate through GPRS/EDGE.

9.3 Library selection

The selection of PolarSSL and as the TLS library, could possible be replaced by either of *CyaSSL* and *MatrixSSL* since there are strong similarities between all of these three. If the project was done during a longer time period then all of these would have been tested to see which library has the best performance and smallest size. The importance of these factors are mainly for the embedded device since the system resources are considerably more limited compared to those on the server. In general we believe that either of these libraries would have worked for this project.

9.4 Future Work

During the development of this project it was noticed that there exist areas where future work could be done. This to improve the solution or to show if other solutions are more suitable.

9.4.1 Hardware accelerated cryptography

If the embedded device incorporated cryptographic circuits it should be possible to increase the throughput, this would be especially useful if the circuit did support AES and

GCM.

9.4.2 Cryptographic Protocol

Due to the strong similarities between TLS and SSH it would be possible to apply a similar security solution for the communication with the use of the SSH protocol instead of the TLS protocol.

If another device was used instead of the OWS451 (see 5.1.4) which gave access to the network layer of TCP/IP-stack, then it would be possible to use IPSec for security.

9.4.3 Entropy algorithm

The entropy algorithm could be considerably improved (currently the clock is used as input for the algorithm). The importance of this part should not be underestimated due to the importance of random values in creating strong and secure communication. Once more, better hardware or entropy algorithm provided by the operating system could make this an easier task.

9.4.4 TLS libraries

Since only one TLS library was tested it could be of interest to test some other libraries (MatrixSSL and CyaSSL recommended) to compare the toll they take on the hardware's performance.

9.4.5 Automatic WLAN connection

While a proposition for this issue was given (see 5.5) it was never actually tested and some more thought should be put onto this aspect.

9.4.6 CAN security

As stated in 9.1.3, the connection device only forwards packets from and to the CAN-bus, therefore it is possible to send any kind of CAN messages. This could potentially be a security risk, even if the communication is protected with TLS it does not stop an authorized host to send incorrect messages by mistake. Possible solutions could be adding authorization levels to different operations on the server side. Another solution would be to add restrictions on which CAN packets are allowed to be transmitted over the Internet by filtering these in the connection device. Finally, having a firewall is important to mitigate attacks below the Application layer of the TCP/IP-stack.

10

Conclusion

This thesis describes the implementation a remote secure channel using TLS 1.2 to communicate with the internal CAN-bus on a boat. With this channel it is possible to perform various operations such as remote diagnostics, parameter configuration and data logging over the Internet. The implementation was done on low cost embedded device, which used WLAN to connect to a router for Internet access. Since any router can be used to access the Internet, this means the added benefit of mobility since the customers' own smart-phones with mobile hotspot can be used. To ensure simplicity and low maintenance the embedded device acts merely as a forwarding node.

The implementation included evaluation of which security risks could exist when connected to the internal CAN bus of a vehicle via the Internet. This assessment showed that the general threats of being on the Internet exist but also specific ones with strong similarities to the automotive sector. This was due to the commonly shared internal CAN bus which controls the vehicle.

During the thesis a few areas were identified where future work would be needed. This included creating an algorithm for determining which WLAN to connect to, since no user input should be needed, improved entropy source on the embedded device to improve security, evaluate multiple TLS libraries to determine the best one, etc.. We also see that improvements can be done for the hardware, specifically the introduction of a cryptographic circuit to enhance the communication speed.

Overall, Confidentiality, Integrity and Authenticity were solvable by using TLS.

Bibliography

[1] A. Inc, Sdk compatibility guide, last visited 2010-11-15.
URL https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/cross_development/cross_development.pdf

[2] Google, Device compatibility, last visited 2014-05-03.
URL http://developer.android.com/guide/practices/compatibility.html#Versions

[3] A. Hammershøj, A. Sapuppo, R. Tadayoni, Challenges for mobile application development, Intelligence in Next Generation Networks (2010) 1–8.
URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?&arnumber=5640893

[4] New john deere strategy takes intelligent farming to the next level, last visited 2014-05-03 (nov 2011).
URL http://www.deere.com/wps/dcom/en_NAF/our_company/news_and_media/press_releases/2011/nov/farm_sight.page

[5] Kongsberg, Kongsberg k-ims, last visited 2014-05-03.
URL http://www.km.kongsberg.com/ks/web/nokbg0397.nsf/AllWeb/737F465C7ECC8A7AC12579EA002EFA83/$file/K-IMS.pdf

[6] CarIQ, Cariq, last visited 2014-05-07.
URL http://mycariq.com/product

[7] S. Andrews, Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) Communications and Cooperative Driving, Springer London, pp. 1121–1144, last visited 2014-05-16.
URL http://link.springer.com/referenceworkentry/10.1007%2F978-0-85729-085-4_46

[8] H. Zhu, X. Shen, R. Lu, X. Lin, Security in service-oriented vehicular networks 16,

last visited 2014-05-16.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5281251

[9] IEEE (Ed.), Collaborative Efforts for Safety and Security in Vehicular Communication Networks, last visited 2014-05-16.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6065075

[10] C. Laurendeau, M. Barbeau, Threats to Security in DSRC/WAVE, pp. 266–279.
URL http://link.springer.com/chapter/10.1007/11814764_22

[11] A. Bittau, M. Handley, J. Lackey, The final nail in wep's coffin, last visited 2014-05-07.
URL http://tapir.cs.ucl.ac.uk/bittau-wep.pdf

[12] W. R. S. Kevin R. Fal and, Attacks involving window mangement, in: TCP/IP Illustrated, 2nd Edition, Vol. 1, Pearson, 2011, pp. 723–723.

[13] M. Baggett, Ip fragment reassembly with scapy, last visited 2014-05-08.
URL http://www.sans.org/reading-room/whitepapers/tools/ip-fragment-reassembly-scapy-33969

[14] CPNI, Security assessment of the transmission control protocol, last visited 2014-05-07.
URL http://www.bsdcan.org/2009/schedule/attachments/75_tn-03-09-security-assessment-TCP.pdf

[15] C. Vulnerabilities, Exposures, Cve-2014-0160, last visited 2014-05-07.
URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[16] W. Stallings, Security attacks, in: Cryptography and Network Security:Principles and Practice: International Edition, 5th Edition, Pearson, 2010, pp. 39–42.

[17] P. Kleberger, T. Olovsson, E. Jonsson, Security Aspects of the In-Vehicle Network in the Connected Car, Intelligent Vehicles Symposium (IV) (2011) 528–533.
URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5940525

[18] B. Yang, K. Wu, R. Karri, Scan based side channel attack on dedicated hardware implementations of data encryption standard, Test Conference, 2004. Proceedings. ITC 2004. International (2004) 339–344.
URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1386969

[19] S. D. K. P. S. Parihar, Attacks in wireless networks 1.
URL http://interscience.in/IJSSAN_Vol1Iss1/paper25.pdf

[20] H. Xia, J. C. Brustoloni, Hardening web browsers against man-in-the-middle and eavesdropping attacks, International World Wide Web Conference Commit-

tee (2005) 489–498.
URL http://dl.acm.org/citation.cfm?id=1060817

[21] Transmission control protocol (sep 1981).
URL http://tools.ietf.org/rfc/rfc793.txt

[22] J. Postel, User datagram protocol (aug 1980).
URL http://www.ietf.org/rfc/rfc768.txt

[23] W. Stallings, Secure socket layer and traport layer security, in: Cryptography and Network Security:Principles and Practice: International Edition, 5th Edition, Pearson, 2010, pp. 513–526.

[24] T. Dierks, E. Rescorla, The Transport Layer Security (TLS) Protocol (aug 2008).
URL http://tools.ietf.org/rfc/rfc5246.txt

[25] S. C. S. Corp., Ssh authentication, last visited 2014-05-03.
URL https://support.ssh.com/manuals/server-admin/32/Authentication.html

[26] E. C. Lonvick, T. Ylonen, The Secure Shell (SSH) Protocol Architecture (jan 2006).
URL http://www.ietf.org/rfc/rfc4251.txt

[27] S. Kent, K. Seo, Security Architecture for the Internet Protocol (dec 2005).
URL https://tools.ietf.org/rfc/rfc4301.txt

[28] C. D. Cannière, A stream cipher construction inspired by block cipher design principles, last visited 2014-05-28.
URL http://link.springer.com/chapter/10.1007/11836810_13

[29] K. W. R. James F. Kurose, Computer Networking:A Top-Down Approach: International Edition.

[30] A. N. Alshamsi, T. Saito, A Technical Comparison of IPSec and SSL, Advanced Information Networking and Applications, 19th International Conference on 2 (2005) 395–398.
URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1423719

[31] S. Ravi, A. Raghunathan, P. Kocher, S. Hattangady, Security in embedded systems: Design challenges, ACM Transactions on Embedded Computing Systems 3 (3) (2004) 461–491.
URL http://dl.acm.org/citation.cfm?id=1015049

[32] M. Johanson, L. Karlsson, T. Risch, Relaying controller area network frames over wireless internetworks for automotive testing applications, in: Systems and Networks Communications, 2009. ICSNC '09. Fourth International Conference on, IEEE.
URL http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?tp=&arnumber=5281251

[33] IAR, Iar kickstart kit for stm32f107vc, last visited 2014-05-03.
URL https://old.iar.com/website1/1.0.1.0/3096/1/?item=prod_prod-s1%2F489

[34] STMicroelectronics, Stm32f105xx, stm32f107xx, last visited 2014-05-03.
URL http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00220364.pdf

[35] connectBlue, cB-OWS451 and OWL253 Electrical Mechanical Data Sheet, last visited 2014-05-19.
URL http://support.connectblue.com/display/Dashboard/OWS451

[36] M. Ullholm Karlsson, A. Habib, Ssh over udp, last visited 2014-05-03.
URL http://publications.lib.chalmers.se/records/fulltext/123799.pdf

[37] Comparison of tls implementations, last visited 2014-05-06.
URL http://en.wikipedia.org/wiki/Comparison_of_TLS_implementations

[38] O. B.V, Polarssl, last visited 2014-05-06.
URL https://polarssl.org/

[39] PolarSSL, Supported ssl / tls ciphersuites, last visited 2014-10-25.
URL https://polarssl.org/supported-ssl-ciphersuites

[40] E. II, Ecrypt ii yearly report on algorithms and keysizes.
URL http://www.ecrypt.eu.org/documents/D.SPA.17.pdf