Matjaž Guštin, BSc

CAN Bus Security Protocol: lightweight message confidentiality, authentication, and freshness on an automotive bus

MASTER'S THESIS

to achieve the university degree of

Master of Science (Diplom-Ingenieur)

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Marcel Carsten Baunach, Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat

Institute of Technical Informatics (ITI)
8010 Graz, Inffeldgasse 16/I

Graz, 23 May 2022

AFFIDAVIT

Abstract

In this work the *CAN Bus Security* (CBS) protocol is presented. This protocol cryptographically secures the communication between microcontrollers connected via a CAN FD bus, providing protection against sniffing, spoofing and replay attacks. This work focuses mostly on the automotive sector, as it is the origin of the CAN bus. CBS offers a simple and centralised client-server architecture based solely on symmetric cryptographic primitives, inspired by the Kerberos protocol. This allows a fast communication start-up and a simple reconfiguration or replacement of clients, e.g., in case of hardware failures.

The proposed solution includes a complete formal specification of the new protocol, the reference software implementation *Hazelnet*, its dependency the *LibAscon* cryptographic library, and the *Hazelnet Demo Platform*. The latter uses several microcontrollers to demonstrate the secure exchange of messages using CBS.

The motivations and rationales behind the protocol design and software implementation choices are explained within this document; overviews of the architecture and structure of the protocol and software are also provided.

Kurzfassung

In dieser Arbeit wird das *CAN Bus Security* (CBS) Protokoll vorgestellt. Dieses Protokoll schützt die Kommunikation zwischen Mikrokontrollern welche über einen CAN FD Bus verbunden sind kryptografisch, und bietet somit Sicherheit vor Sniffing, Spoofing oder Replay Attacken. Diese Arbeit fokussiert sich auf den Automotive Sektor, da dort der Ursprung des CAN Buses liegt. CBS bietet eine einfache und zentralisierte Client-Server Architektur, welche ausschließlich auf symmetrischer Kryptografie beruht und vom Kerberos Protokoll inspiriert wurde. Das erlaubt eine schnelle Kommunikationsaufnahme sowie eine einfache Rekonfigurierung oder einen einfachen Austausch von Clients, z.B. im Falle eines Hardware Defekts.

Die vorgeschlagene Lösung beinhaltet eine vollständige formale Spezifizierung des neuen Protokolls, die Referenzimplementierung *Hazelnet* sowie die *LibAscon* Kryptografie Bibliothek. Die gemeinsame Nutzung dieser Komponenten wird in einem Anwendungsbeispiel, der *Hazelnet Demo Platform* demonstriert. Diese Demo zeigt wie mehrere Mikrokontrollern mit dem CBS Protokoll sicher Nachrichten austauschen können.

Weiter beschreibt diese Arbeit auch die Gründe und Motivationen für die verschiedenen Protokoll sowie Software Designentscheidungen; auch ein Überblick über die Architektur und Struktur des Protokolls und der Software ist enthalten.

Acknowledgements

This thesis was done in collaboration with NXP Semiconductors Austria GmbH & Co KG, which kindly allowed me to work part-time while completing my master studies.

My sincere gratitude goes to my significant other Sara, for her unbelievable patience, support, and continuous encouragement through my years of study. *Hvala ti.*

I would like to thank my family, my manager, and my System Team colleagues for their help, support, flexibility, and simply for listening.

I am grateful to my thesis advisor, Univ.-Prof. Marcel Carsten Baunach, for assisting me in this work and for his patience.

Finally, an important thank you goes to my therapist without whom I would not have reached this point.

Contents

List of Figures

List of Acronyms and Symbols

1. **ACK**: Acknowledgement; in communication protocols it's a confirmation of the successful reception of a message.
2. **AEAD**: Authenticated Encryption with Associated Data; cryptographic operation encrypting the data while proving the authenticity and integrity of both the generated ciphertext and of arbitrary related plaintext, usually message metadata.
3. **AES**: Advanced Encryption Standard; a standard block cipher used in many security applications.
4. **API**: Application Programming Interface; set of public functions, data structures, and constants a software library, module, or service exposes to another software component for the latter to use the former.
5. **BCM**: Body Control Module; a microcontroller monitoring and operating various accessories of the vehicle's body, such as doors, windows, and ventilation.
6. **CA**: Certificate Authority; entity in PKI issuing digital certificates.
7. **CAN**: Controller Area Network; a wired automotive communication bus.
8. **CAN FD**: Controller Area Network Flexible Data-rate, extension of CAN with larger and faster frames.
9. **CBC**: Cipher Block Chaining; a block cipher mode of operation for encryption-only of long messages.
10. **CBS**: CAN Bus Security; the security protocol introduced by this thesis.
11. **CCC**: Car Connectivity Consortium; a standard-setting body for vehicle communication protocols.
12. **CRC**: Cyclic Redundancy Check; an error-detecting code, used to verify the integrity of a message.
13. **CSEc**: Cryptographic Service Engine compressed; hardware security module on NXP S32K1xx series of devices.
14. **DC**: Direct Current; constant, unidirectional electric current.

15. **DH**: Diffie-Hellman; a method to get two parties agree on a shared cryptographic secret securely over an untrusted channel.
16. **DoS**: Denial of Service; malicious attack aiming to disrupt the availability of a system.
17. **GCM**: Galois/Counter Mode; a block cipher mode of operation for authenticated encryption of long messages.
18. **HMAC**: Hash-based Message Authentication Code; a keyed-hash wrapper function used to authenticate messages with better immunity against length-extension attacks compared to a plain keyed-hash.
19. **HZL**: short for *Hazelnet*, the reference implementation of the CBS protocol.
20. **ID**: Identifier; a value representing something or someone.
21. **IP**: Internet Protocol; a communication protocol for routing data between local networks, sitting between a data-link layer and the transport layer.
22. **ISO**: International Organization for Standardization; a global standard-setting body for virtually every sector.
23. **KEM-DEM**: Key Encapsulation Mechanism – Data Encapsulation Mechanism; a hybrid encryption paradigm, using a randomly-generated symmetric key to encrypt the data (DEM) and using asymmetric cryptography to encrypt/wrap the symmetric key itself (KEM).
24. **LTK**: Long Term Key; persistent CBS cryptographic key, shared between Client and Server.
25. **MAC**: Message Authentication Code, often called *tag*; short binary fingerprint proving the authenticity and integrity of a message.
26. **NFC**: Near Field Communication; a set of low-power wireless communication protocols that let devices exchange data over distances of few centimetres.
27. **OBD**: On-Board Diagnostics; capability of a vehicle to diagnose its own abilities and report them. The OBD interface is a standard way to access this data.
28. **OS**: Operating System; software that controls the hardware resources, inputs, and outputs of a computer system, coordinating many processes using it concurrently.
29. **OSI**: Open Systems Interconnection; ISO group specifying the telecommunications layering model.
30. **PDU**: Protocol Data Unit; entire message of the current telecommunication layer,

including metadata.

31. **PFS**: Perfect Forward Secrecy; property of cryptographic protocols that ensure the confidentiality of the encrypted data protected by short term/session keys even if the long term keys are leaked or compromised in the future.
32. **PKI**: Public Key Infrastructure; roles, policies, and procedures to manage the lifetime of public key encryption and digital certificates.
33. **RGB LED**: Red-Green-Blue Light Emitting Diode; semiconductor light source, programmable to emit any colour by combining the 3 primary ones.
34. **RNG**: Random Number Generator.
35. **RTOS**: Real-Time Operating System; an operating system for applications that need to react to events within very short periods of time (usually milliseconds or less).
36. **RX**: Reception.
37. **SAD**: Secure Application Data; CBS message type containing encrypted and authenticated data from the application.
38. **SADFD**: Secure Application Data over CAN FD; a SAD message in compact form to fit within a CAN FD frame.
39. **SADTP**: Secure Application Data over a Transport Protocol; a SAD message in expended form to be fragmented into many CAN FD frames.
40. **SDU**: Service Data Unit; data (payload) carried by the current telecommunication layer, containing the PDU of the upper layer.
41. **STK**: Short Term Key; temporary CBS session key, generated by the Server and distributed to the Clients.
42. **TCP**: Transmission Control Protocol; a transport protocol meant to run over IP, which provides connections, reliability, retransmission and congestion control.
43. **TLS**: Transport Layer Security; a cryptographic protocol to secure data exchange over TCP/IP. Probably the most used one worldwide, given that it's the backbone of most secured web and internet traffic.
44. **TP**: Transport Protocol; communication protocol abstracting away some properties of the data-link layers, typically providing additional reliability, fragmentation, and flow control.
45. **TRNG**: True-Random Number Generator; hardware-based source of randomness,

usable for cryptographic purposes.

46. **TX**: Transmission.
47. **UAD**: Unsecured Application Data; a CBS message type containing plaintext and not-authenticated data from the application.
48. **UWB**: Ultra-wideband; a low-energy pulse-radio technology with ability to measure the time-of-flight, thus measure the distance between devices accurately.

1 Introduction to the CAN bus

A modern vehicle incorporates a variety of microcontrollers, operating on parts of the vehicle engine, transmission, breaks, chassis, ventilation, entertainment, door locking and more. Generally, all these microcontrollers are communicating with each other and collaborating on a task, often with one device being a controlling leader. Such communication happens via wired protocols, commonly using the CAN bus.

For example, a modern, *CCC Digital Key*-compliant vehicle access system is composed of multiple radio technologies such as Bluetooth, Ultrawide-band (UWB) and Near-Field Communication (NFC) [1]. An assortment of transceivers for each of these is commonly operated by a leader device, triggering their radio activity and controlling which messages the vehicle exchanges over the air with a smart mobile device (key fob, smartphone, smart card or other wearable). Upon successful authentication of the mobile device's identity and verification of its physical proximity to the vehicle, the leader device triggers the unlocking of the doors, finally granting physical access to the vehicle's cabin.

This work aims to secure the communication happening over the CAN bus. To explain the security problem at hand and its proposed solution, first we need to understand how the CAN bus works. This section provides a high-level overview of the most important characteristics of the CAN protocol.

1.1 The CAN bus

The Controller Area Network (CAN) bus is a vehicle bus designed to allow robust, half-duplex, multi-master, message-based communication between a set of devices in a vehicle, but it's also used in industrial applications. The devices connected to and communicating with a CAN bus are usually referred to as *nodes*. Being a bus, each message is broadcast, meaning that all other nodes on the bus are able to receive it and have to actively decide whether to process the message or discard it because it's not relevant for them.

The protocol is specified in the ISO 11898 series of standards [2][3][4][5]. In the ISO/OSI model it takes the place of the physical and data-link layers, i.e., layers 1 and 2.

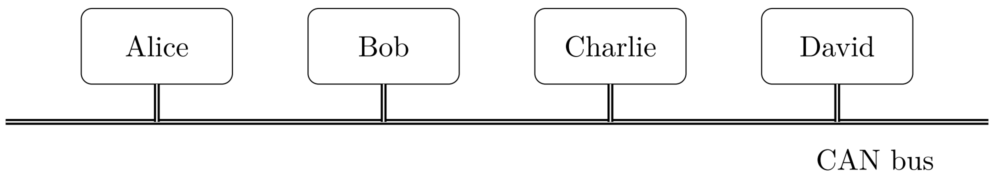

Figure 1.1: Abstract representation of a CAN FD bus shared between the nodes Alice, Bob, Charlie, and David.

We hereby avoid listing all internal workings and properties of the CAN bus as they are easy to find with a quick online search [6]. Instead, let us show only the parts which are relevant for our considerations.

1.1.1 The physical layer: differential signalling

The CAN communication physically happens on two wires with differential signalling. A diagram of the wire voltages is available in Figure 1.1.1. A brief synopsis of the physical architecture follows:

1. The differential signalling wires are denominated *CAN High* and *CAN Low*.
2. The setup of wires, resistors, and used voltages differs depending on the system requirement: *High-speed CAN* (ISO 11898-2, [3]) is the mode focusing on performance while *Low-speed CAN* (ISO 11898-3, [4]) is optimised for robustness and longer lines.
3. Resistors are connected to the High and Low wires to pull the voltage between them to a specific "at rest" level when no device is transmitting, called *recessive voltage*. Such voltage (Low-to-High) is close to 0 V for High-speed CAN and close to -5 V for Low-speed CAN.
4. *Dominant voltage* is the one actively induced on the High and Low wires by a node, opposing the recessive state. As the name implies, dominant voltage overrides the recessive one. As long as one node keeps the voltage on dominant levels, no other node is able to set it to recessive. To achieve a recessive state, all nodes on the bus must stop actively setting the voltage to dominant. Such voltage (Low-to-High) is close to $+2$ V for both High-speed and Low-speed CAN.
5. The nodes must have CAN transceivers that are able to read the bus voltage while they themselves are transmitting, enabling them to detect whether their own recessive voltage is being overwritten with a dominant one by another node.
6. The logical value 1 is represented by a recessive and 0 by the dominant state. As

a consequence, logical zero bits have a higher priority and they overwrite logical ones.

It's easy to see that performing a Denial of Service (DoS) attack on the CAN bus is trivial on a physical level: other than the obvious cutting of the wires, placing a malicious active CAN transceiver on the bus programmed to constantly transmit dominant values will prevent any other node from communicating.

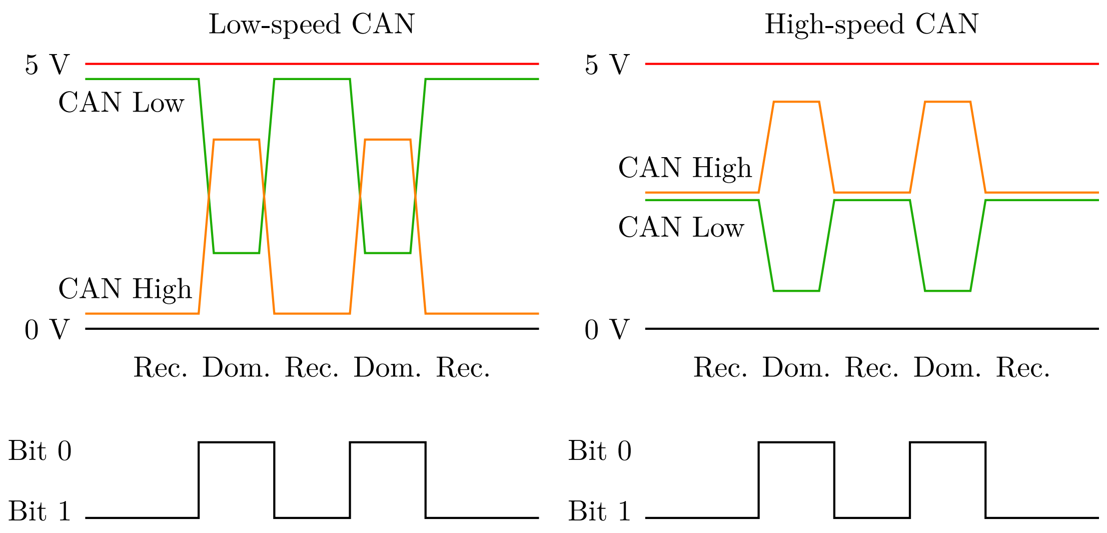

Figure 1.2: Simplified plots of voltages of CAN wires and resulting logical bits of a Low-speed (left) and High-speed (right) bus setup.

1.1.2 The data-link layer: the CAN frame

With the physical layer described above, we are now able to transmit and receive binary digits. As with most communication protocols, they are organised in a message, called *frame*. Here follows a simplified overview over the most important features of CAN frames:

1. The header of a CAN frame includes a message identifier, often referred to as *CAN ID*. This field can be either 11 or 29 bits long in the so called *Base frame format* and *Extended frame format* respectively. The CAN ID is used as a unique identifier of the message source, and may include additional metadata encoded in it, such as the type of the message content and/or its destination, depending on the application.

2. When two nodes happen to transmit simultaneously, a message arbitration proce-

dure is used to determine which message has higher priority by comparing the two CAN IDs bit-by-bit. The logic value 0 (*dominant*) takes precedence over the value 1 (*recessive*), meaning that the message with a smaller CAN ID value[1] wins the arbitration. The CAN transceiver that loses the arbitration stops transmitting, lets the winning message be written on the bus, and afterwards usually retries the transmission. For this reason, there must not be two nodes using the same CAN ID as their messages would collide after the ID transmission, causing bus errors.

3. The Service Data Unit (SDU) of a CAN frame, often called payload or data section, may carry at most 8 bytes (octets); the messages are indeed short.
4. CAN frames come with a Cyclic Redundancy Check (CRC) in the trailer, guaranteeing the integrity of the data within reasonable probability. The CRC does not offer any assurance of the authenticity of the message.
5. An acknowledgement (ACK) slot is a recessive bit at the end of each frame being transmitted and overwritten with a dominant value by any other node on the bus receiving said frame to acknowledge it. By inspecting this bit's alteration, the transmitting node can see if either no one received the frame or if at least one node received it, but without knowing which one(s). The transmitting node may automatically resend the message in case of missing acknowledgment, if configured to do so.
6. The CAN protocol defines no way of fragmenting and reassembling longer payloads; if required, it has to be handled separately by higher layers in the communication stack (such as the transport protocol). This topic is further explored in Section 1.3.
7. The CAN bus with the High-speed wiring configuration and proper transceivers can reach a gross data-rate of 1 Mbps (in some cases slightly more). The protocol overhead to transfer an SDU of 8 bytes varies between 40 and 60%, depending on the size and content of the CAN ID and CAN SDU.

CAN ID	SDU length	SDU	CRC
11 or 29 bits	4 bits	0-**8** bytes	15 bits

Figure 1.3: Abstract and simplified diagram of a CAN frame. Start of frame, end of frame and various control bits are not shown for simplicity.

[1]When looking at the ID as an 11- or 29-bit unsigned integer, big-endian, where the most significant bit is the first transmitted.

1.2 Faster and longer SDUs with CAN FD

The CAN FD bus (Controller Area Network – Flexible Data-rate) [7][8] was originally an extension of the CAN protocol by Robert Bosch GmbH, later included in the ISO standard specification [2], with the goal of transporting more data per frame at higher speeds.

1.2.1 The CAN FD frame

Let us briefly list the differences compared to classic (non-FD) CAN frames:

1. A reserved bit from the frame header (not in the CAN ID) is used to distinguish between a classic CAN and a CAN FD frame.
2. The SDU is transmitted at a higher data-rate compared to the header and trailer (up to 15 Mbps) and supports sizes of 0-64 bytes, allowing up to 8x more data per frame compared to classic CAN.
3. The SDU length is indicated with only 4 bits so the possible lengths are: 0, 1, . . . , 8 bytes as in the classic CAN or 12, 16, 20, 24, 32, 48, 64 bytes for longer payloads. A trailing padding is applied where required to reach one of the allowed length. The actual length of the useful data before the padding must be provided by other means; encoded in the CAN ID, in the SDU itself or known in advance, for instance from a preconfigured map from CAN ID to SDU format[2].

CAN ID	SDU length	SDU	CRC
11 or 29 bits	4 bits	0-**64** bytes	17 or 21 bits

Figure 1.4: Abstract and simplified diagram CAN FD frame. Start of frame, end of frame and various control bits are not shown for simplicity.

1.3 Fragmentation and reassembly

As one might expect, even when using CAN FD, 64 bytes of SDU may again not be enough for some applications, for example transferring a 4096 bit (512 bytes) RSA public key. For this reason, a transport protocol may be employed to:

[2]For example: transmitting 13 bytes will add a 3-byte padding to get the SDU size to 16 bytes and the receiving side must know somehow to cut away the last 3 bytes of the SDU before processing it any further in the application.

1. fragment a too-long SDU into smaller slices,
2. fit each slice into a frame of the underlying CAN or CAN FD protocol,
3. keep track of what has been transmitted,
4. potentially re-transmit lost frames,
5. and reassemble the full message on the receiving side.

1.3.1 The standard solution with ISO-TP

The ISO-TP [9] is a Transport Protocol defined over CAN or CAN FD frames that allows up to 4 GiB of SDU to be transmitted through fragmentation. ISO-TP is mostly used for diagnostic messages (like OBD) but can also be used for other purposes.

The protocol is specified in the ISO 15765-2 series of standards [10]. In the ISO/OSI model it takes the place of the transport layer, i.e., layer 4.

An ISO TP drawback may be the slowness of transmission of large frames due to the various timeouts, forced delays between successive fragments and confirmation frames sent periodically by the receiving party. These factors are parametrisable and can be configured to match the performance of the nodes in the system and the latency requirements. The fastest configuration supports no confirmation frames and no enforced delays between fragments.

A second important disadvantage of ISO-TP is the missing possibility to multicast fragmented SDUs: only single-frame messages can have multiple destinations, multi-frame ones have a single destination address encoded in the CAN ID.

1.3.2 Custom-tailored transport protocols

One alternative is of course to implement a custom transport protocol based on one's needs, for instance the need to support multicasting of fragmented messages. Clearly, each application may require different features from such a protocol, may it be low latency, robustness, or low bus load. For this reason, it would be unproductive to list numerous possibilities for how such a custom protocol could work.

Just to fix our minds onto something, let us give a very simple example: a basic, unreliable transport protocol, similarly to what IP does in the TCP/IP communication stack, may just fragment the too-long data into CAN or CAN FD frames and send them unreliably one after the other without acknowledgments or retries within the transport layer. The receiving party verifies whether all fragments have been received and concatenates them; if any kind of error occurs during the reassembly process, simply all received data is discarded silently.

2 Bus security and threat model

The CAN bus was designed to be robust and reliable for safety purposes. Security against attacks was not considered during its development; all messages travelling on the bus are plaintext and without authentication. For this reason, sniffing, or spoofing any data exchanged on the bus is trivial and inexpensive to attack using any CAN transceiver on the market.

Confidential data, such as cryptographic keys, the CAN nodes may exchange with each other is completely unprotected and an attacker reading it may use it to compromise other bus-connected subsystems. The messages are also not authenticated, thus carrying no guarantee that the source is a node that is originally part of the bus. For an attacker it's also easy to inject false, spoofed frames onto the bus, forcing other nodes into a desired behaviour. This could include, for example in an automotive scenario, messages that unlock the vehicle doors and start the engine, leading to vehicle theft.

Nowadays it should be obvious to see that any system transmitting plaintext messages is unsecure and unsafe. This work proposes a lightweight security protocol protecting this communication.

2.1 Threat model

As a first step, we must define what are we trying to protect.

2.1.1 Primary threat: sniffing, spoofing, replaying messages

The main vulnerability we are focusing on in this work is the protection against a *Man in the middle* (MitM) attacker tapping the wires of the CAN bus and, with the help of an *external* device, not originally part of the vehicle architecture, being able to perform some or all of the following:

1. Sniffing sensitive data, i.e., reading the content of plaintext messages in transit on the bus.

The cryptographic property we desire against such an attack is the *confidentiality* of the messages. In practice this means encrypting the message payloads, so no sniffer can understand their content.

2. Spoofing of messages, i.e., transmitting messages that claim they originate from a separate but real node on the bus.

   The cryptographic property we desire against such an attack is the *authenticity* of the messages. In practice this means adding a *Message Authentication Code* (MAC), sometimes called *tag*, to the message to guarantee the message was generated by a known party.

3. Tampering of messages, i.e., active changes in the messages in transit on the bus without the other parties realising it.

   The cryptographic property we desire in our system that opposes such an attack is the *integrity* of the messages. While such an attack is not exactly easy to achieve due to how the CAN bus works on a physical level, the message integrity typically comes for free with the authenticity property when using a secure MAC.

4. Replay attacks, i.e., reuse of old messages to induce a behaviour in the bus-connected devices.

   The cryptographic property we desire against such an attack is the *freshness* of the messages, sometimes called *timeliness*. In practice this means adding a *nonce*, a unique, never-reused, public counter or timestamp to the message to guarantee the message was generated recently.

2.1.2 Secondary threat: untrusted co-hosted applications

Another, more architectural, threat this work was considering is untrusted software applications co-hosted on the same CAN node. An application may be untrusted because it's developed by a different company and then installed on the same physical device by the entity responsible for the overall system development.

Two or more co-hosted applications could then read each other's decrypted messages from a shared message queue. The solution here is conceptually simple: moving the implementation of the protocol from a shared security layer into the application space, making the secure channel and end-to-end one, from transmitting application to receiving application. This requires the implementation of the security countermeasures to be able to co-exists with other concurrent instances of itself on the same device (e.g., by handing mutual exclusion on any shared resources).

2.1.3 Rationale for the selected threats

These threats were chosen as they are most relevant for the company (NXP Semiconductors) the thesis author was working at while the present work was being developed: in automotive access systems, the ultrawide-band (UWB) radio technology is getting traction as a secure distance-bounding system. This is used to check whether the holder of a key (physical or digital), having the right to access the vehicle, is in fact physically close-enough to the vehicle itself and the communication key-vehicle is not being relayed by thieves.

The secure part of the distance-bounding system requires each UWB node on the vehicle to obtain a cryptographic key from somewhere off-band, usually from a central leader device, such as the Body Control Module (BCM). Naturally, if the transport of this key is unprotected, the security of the entire access system falls short: sniffing the plaintext key on the bus lets the attacher inject spoofed, but accepted, UWB frames in the air, which may make the system susceptible to relays again.

2.2 Not included in the threat model

It's worth underlining what security threats this work is *not* focusing on for clarity, mostly because they were deemed out of scope for the final products of the company (NXP Semiconductors) the thesis author was working at while the present work was being developed. They are:

1. Protection against compromised, rogue nodes: the endpoints of the communication are assumed secure and well-behaving, just like in the TLS protocol. This is a requirement in order to have a symmetric cryptographic key shared between the two or more communicating parties.

   An attacker could find firmware or side-channel vulnerabilities on existing, original nodes on the bus and manipulate such nodes to sniff and decrypt messages or transmit arbitrary data that any other node would accept. Such an attack was deemed harder-enough compared to simple sniffing and spoofing of plaintext bus messages by a third party; the expertise, tools, and time required to achieve it compared to sniffing and spoofing plaintext messages on a shared bus is orders of magnitude higher.

2. Privacy of the messages and hiding the communication metadata. The rule of thumb applied here is that any user-data is protected because it's part of the secured payload of the messages and the privacy of the nodes themselves is not

really relevant, given that they are just machines with fully-automated behaviour. Details *not* being handled are thus:

- Obfuscating CAN IDs: this was deemed infeasible, as the CAN IDs are the differentiating element between messages. If this field contains confidential data, then such data should be moved to the (secured) payload and replaced with a simple enumeration of entire IDs instead of having specific bits of the ID have a decodable meaning.
- Hiding which nodes are communicating and when: the microcontrollers attached to the bus do not need to hide their behaviour, generally. If there is a real need to do so, then simply a random transmission of dummy messages to be discarded by the receiver is easy to achieve at application-level and would provide sufficient noise for an observer.
- Length of encrypted data: the messages contain a public, plaintext value indicating the amount of bytes of the plaintext. This was chosen to allow the usage of stream ciphers that generate ciphertexts of the same length as the plaintext – additionally hiding the meaning of the messages was not a requirement, only the content of the message was to be confidential.

  Hiding such information would require encrypting this field, thus making the plaintext length part of the ciphertext; additionally the plaintext would have to be padded to a minimum secure length (e.g., 128 bits) to achieve some level of obfuscation of the message content. This could be done in future changes to the proposed solution, as discussed in detail in Section 6.3.

3. Leaks of the long term keys. This work was mostly focusing on getting the nodes to communicate in a secure manner using lightweight cryptography and a simple security architecture. Perfect Forward Secrecy (PFS), the ability to unbind the short term (session) keys from the long term keys, in order to protect the former if the latter are leaked, was not a requirement.

   The choice was done in order to speed up the development process and being able to use only symmetric cryptography for a computationally lighter protocol. It has to be noted that very simple extensions to the designed solutions can be added in order to achieve PFS, as discussed in Section 6.1.

4. Secure storage of cryptographic keys used to protect the both the data in transit and at rest. To be done properly, specific hardened hardware is required such as a secure element, thus it was not in the focus of this work, as it could have the exact same implementation, only running with a protected device.

3 The proposed solution: CBS protocol

Having explained the basics of the CAN bus and what our threat model is (and what it excludes), we can now present the core part of this work, being the proposed security solution.

The *CAN Bus Security* or in short *CBS* is a centralised Client-Server protocol to enable **cryptographically secure, multicast communication within a set of trusted devices connected to the same CAN FD bus**. CBS is designed to be simple, fast, resistant to replay attacks, and allow a simple reconfiguration in case a Client requires a reset of its settings or a replacement in case of hardware failure. CBS is based on the revised nonce-based Needham–Schroeder Symmetric Key protocol [11][12], which is the basis for the Kerberos protocol, for the key distribution and on vatiCAN [13] for the freshness nonces.

CBS protects against passive and active Man-in-the-Middle attacks on CAN FD bus traffic, much like TLS does for TCP connections. The focus is solely on the security during *transport* of messages between parties who are *assumed secure* and uncompromised.

3.1 CBS requirements, briefly

CBS assumes the data-link layer is CAN FD because the 8-byte SDU in classic CAN frames are not enough[1] to carry the cryptographic overhead (nonces, MACs etc.).

The platform using CBS must provide a true (hardware) random number generator (TRNG) and a simple timestamping function, used to prove the freshness of the CBS messages. Synchronised clocks are not required.

One device on the bus must act as a Server and is expected not to lose any messages. The other devices may have arbitrary power cycles. Further ideas on improving the availability of the Server can be found in Section 6.6.

[1]In theory it should still be possible (although unpractical) to use CBS on top of a transport layer handling the fragmentation and reassembly of 64 or more bytes of SDU length into classic CAN frames of 8 bytes.

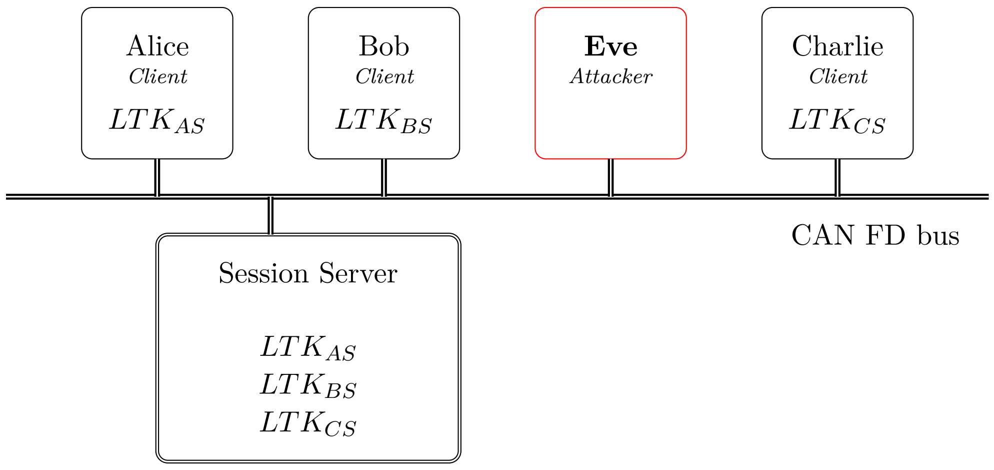

Figure 3.1: Abstract representation of a CAN FD bus shared between the Clients Alice, Bob, Charlie, and a Session Server with an attacker Eve having physical access to the same bus. Alice, Bob, and Charlie are each preconfigured to securely communicate initially only with the Session Server, without having the means to communicate with each other; it's the Session Server's role to generate and distribute additional short term keys to enable secure inter-Client communication.

3.2 The protocol specification

In the attachment ATT-1 of this document, the entire technical specification of the CBS protocol is available.

The specification includes message formats, meaning of each bit, the parties' behaviour, the checks to perform, the cryptographic primitives to use, states and configuration structures – in short everything required in order for someone not familiar with this work to write a software implementation of the whole protocol from scratch. It is in a separate document in order to distinguish between the technical protocol specification and this thesis document, which instead focuses on the academic part and explanations of the design process.

It's recommended for the reader to evaluate the specification first before continuing reading this document.

Further details on the protocol design process and rationales are discussed in Section 4.

3.3 The protocol reference software implementation

In the attachment ATT-2 of this document, the source code and documentation of the reference implementation of the CBS protocol is available, called *Hazelnet*. It is a middleware library, written in ISO C11, cross-platform, implementing both the Client and Server roles of the protocol. The library is hardware-independent and can be used both on desktops and embedded machines. It was heavily documented and unit-tested, with a particular focus on clean code, to the extent the C programming language inherently allows.

Hazelnet internally uses the the *LibAscon* [14] implementation of the Ascon cipher, which is described in Section 5.4. LibAscon was initially developed as a personal experiment for academic purposes and published as a stand-alone, open-source project. The development of the Hazelnet library was an excellent opportunity to use LibAscon in practice and see how well it works as a dependency, so it was greatly expanded to meet Hazelnet's needs. LibAscon is available as attachment ATT-3 of this document.

In the attachment ATT-4, the source code of a simple demonstrator called *Hazelnet Demo Platform* is available, which shows 2-4 CAN-connected microcontroller boards communicating securely over the bus using the CBS protocol and the Hazelnet implementation.

Further details on the software implementation are discussed in Section 5.

4 Protocol specification design choices

In this section we try to explain the requirements and restrictions posed to the development of the CBS protocol specification and for each one we expand on the way CBS tackles and why.

Adding a security layer on top of the existing CAN bus communication is not trivial due to the properties of the bus itself and some constraints of the automotive environment, which differ from a classic and well-studied point-to-point channel security; a problem handled by many protocols already, TLS being the most notorious example.

4.1 Multicast communication

The bus messages are multicast, meaning that each message has multiple receivers – and each one of them must be able to decrypt and authenticate their secured message. Regardless of whether we use a symmetric schema or an asymmetric one with KEM-DEM, in the end the message is always protected by a symmetric cipher and we need to distribute the same symmetric cryptographic key to all involved parties in a secure manner.

CBS uses a Client-Server schema, where the Server maintains the currently-in-use short term keys and distributes them to the Clients using long term key, unique per Client-Server pair. With a simple two-message handshake, each Client can obtain from the Server the symmetric short term key which is shared with other parties. The centralisation of the point-of-truth simplifies the overall architecture in terms of key distribution, in contrast to a partially or fully distributed system. The latter was explored initially but abandoned because of the increased complexity[1] in architecture and performance.

To further restrict access to each other's messages, the concept of *Groups* of parties was introduced: that is simply a virtual set of Clients and the Server (always included) that use a specific symmetric short term key for their message exchange, differing from the keys of other Groups. Multiple Groups can co-exists on the same physical bus or bus

[1]One may think of the complexity of a fully-distributed n-party key-agreement handshake ($n \geq 2$) to quickly realise a central leader makes the process much more straight-forward.

branch in order to enforce the principle of least-privilege: Clients that don't require talking to each other should be in separate Groups in order not to be able to decrypt each other's messages. At it's most extreme level, a Group can consist of only one Client that can only talk to the Server.

4.2 Arbitrary power cycles

In a real-world system in order to spare energy, any node connected to the bus may switch off or enter a low-power state at any point in time.

Because of the unpredictable availability of the interlocutors, CBS does not handle connections and re-transmissions, to stay closer to the low-level approach the CAN bus already uses; there is no tracking of the active or sleep states of other Clients. The protocol's design allows the Clients to power-down at any time, because of the Server keeping track of the messaging of the bus, of the currently used counter nonces, of the short term keys and their expiration. Any Client powering up/existing a low-power state can simply request the current state to the Server without bothering about staying in touch with all other Clients.

The only requirement CBS has in terms of power cycles is the assumption that the Server is always able to receive all messages, thus never powers-down. While this may seem as a very strict requirement, the usual leader-follower design of automotive systems where some central devices (like the Body Control Module) operates, configures, and controls the other peripheral devices ties very well with the CBS Client-Server architecture: simply the system leader also acts as a CBS Server.

4.3 Fast handshake ramp-up

Similarly, for usability and functional safety reasons, the nodes should not take too much time to perform any initial handshakes after they boot up or wake from a low-power/sleep state. However, this does not affect only a single node, but also an entire CAN bus branch using the CBS protocol: when a vehicle is initially activated (e.g., internal combustion engine start, electric vehicle boot up), the branch is be powered on, which would simultaneously trigger some synchronisation handshakes between *all* nodes on the branch, not just one or two. If each handshake requires a lot of time, the vehicle driver may have to wait a lot just for all cryptographic states to reach a ready point.

In CBS the handshake is a simple two-message exchange Client-to-Server and vice-versa for each Client and for each Group the Client is in. On the bus branch power-up,

n Clients imply $2n$ messages *per Client's Group* being exchanged on the bus for each Client to get up-to-speed and enable its secure communication. In simple initial timing measurements, a complete handshake between a single Client using 1 Group and the Server requires 2-3 ms on a CAN FD bus with 500 kbit/s arbitration-rate and equal data-rate. A few arguments for this design decision:

- The handshake process is reused for both initial Client boot-up and resume from low-power states.
- It may not be the case that all Clients are needed simultaneously and immediately after vehicle initialisation. E.g., any entertainment system may wait for 2 more seconds compared to engine health checks, reducing the initial bus load spike.
- It may not be the case that each Client has to enable each of its Groups simultaneously and immediately: some may be done lazily, on-demand, again to reduce the initial bus load spike. E.g., a Group used for the cabin ventilation system may not be initialised until the person in the vehicle activates the ventilation system in the first place.
- Short term keys are distributed wrapped by long term keys, which are unique per-Client. Having long term keys used by more than one Client could speed up the short term key distribution by means of multicasting, but per-Client long term keys are needed to simplify the replacement/reconfiguration of a Client, because they avoid reconfiguring more devices than just the Server.

4.4 Low-performance devices

Generally CAN bus nodes are small, low-power and low-performance microcontrollers. We may not have the luxury of using asymmetric cryptography because it would require too much computation time for the devices at handshake time.

The design of the CBS protocol is heavily inspired by the Kerberos protocol exactly for this reason: it uses only symmetric cryptography to keep all operations lightweight. While the used ciphers and hash functions can be swapped with different ones, the default proposal and the reference implementation uses the Ascon [15] v1.2 cipher, which was specifically designed to be a lightweight symmetric AEAD cipher and hash function. The rationale behind this choice is available in Section 4.11.

4.5 Device replacement

Physically replacing a Client should be a simple procedure, because it's a common operation when considering that these devices are placed on moving vehicles. Some CBS Clients may physically break during car collisions and it should not be too complicated to swap them with new instances.

Initial drafts of the CBS protocol were focused on Public Key Infrastructure (PKI) schemas, where asymmetric cryptography is used to represent the identities of each single device and to distribute short term keys. This simplifies the replacement or reconfiguration of a device: the only thing the new device requires is a certificate signed by a trusted Certificate Authority (CA) each other party already knows and trusts – at least that's the desired end effect. In reality, revocations of these certificates in case something is leaked, updating the list of Certificate Authorities on all devices if something is changed at the PKI root of trust, handling certificate expirations offline are all very real problems that are easy to overlook.

Newer iterations of the CBS design then focused on a centralised schema based only on symmetric cryptographic primitives. This keeps similar easiness of replacement of Clients without the problematic "side effects" of PKI. Let's imagine the Client *Alice* is physically replaced with *Alice2*: the only thing the replacement technician needs to do is generate a new 128-bit random key and install it on both Alice2 and the Server, as the new long term key used between them, i.e., LTK_{AS}. A straight-forward operation.

Quite different is the case when the Server needs to be physically replaced. The main disadvantage of a centralised architecture is of course a single point of failure. Replacing the Server with a new one requires the fresh generation of all long term keys and their installation on all of the Clients. For large buses this may be a lot of work. This of course assumes that the old Server is completely unusable and its long term keys cannot be extracted out of it (as it should be in a properly secured system).

Two notes are two be made about the replacement of the Server:

- Because the Server is usually running on a central leader device, such as the body control module, having a complete failure of such central, critical device is overall an outstanding issue. Installing a new body control module may require a lot of configuration in the first place and the cryptography of bus communication is just one of the many problems.
- The CBS protocol as of now assumes pre-installed long term keys, but the protocol could be extended, even in a backwards compatible manner, to allow a long term key agreement pairing procedure between a Client and a Server, so they generate

a new long term key instead of a human operator pre-configuring it. Details are available in Section 6.2.

4.6 Bus arbitration

The CAN bus allows transmission of only one node at the time (half-duplex) and thus collisions may happen when 2 more more nodes try to publish a message on the bus simultaneously. The CAN protocol handles it by prioritising the messages that have a more leading zeros in the CAN ID (thus a lower CAN ID numerical value), because zeros are dominant values. One transmitting node wins this arbitration and gets to finish the message transmission; the remaining nodes that lost usually try to transmit again immediately after the winning message has passed and the arbitration process repeats. On a practical side, such a bus design may introduce unpredictable delays in the transmission of the messages, because there is no control on when other bus nodes with higher priorities are transmitting. Delays mean that the content of the message may not be fresh any more.

The Secured Application Data messages in CBS employ counter nonces to prove their freshness. These nonces are integer values incremented for each message that appears on the bus. The rationale is that any node tracking which is the current nonce to use for the next message to appear on the bus can discard old messages simply be checking whether the nonce matches the expected one. The goal of such an approach is to prevent replay attacks of old messages.

The issue happens when the parties Alice and Bob (also more than two) try to transmit a message using the same counter nonce simultaneously. Let's imagine that Alice wins the arbitration process and publishes a message on the bus with a counter nonce N_{ctr}. Now all other parties would expect the next message to have a nonce of $N_{ctr} + 1$, but then Bob transmits his own pending message also carrying the nonce N_{ctr}. Clearly this is not a replay attack, but simply a consequence of the CAN bus protocol. Bob's message should be accepted.

CBS handles this by introducing tolerances: assuming N_{ctr} is the next counter nonce to appear on the bus, instead of rejecting any message containing nonces $< N_{ctr}$, the parties reject the messages with $< N_{ctr} - \texttt{ctrdelay}(m, t, S, D)$, with $\texttt{ctrdelay}(m, t, S, D) \in \mathbb{Z}$ and ≥ 0 being a dynamic tolerance level that gets smaller in case of no traffic on the bus, to reduce the attack surface for replay attacks. The formal specification of the `ctrdelay` function, its arguments m, t, S, D, and example plots of the function are available in the CBS specification ATT-1. With $\texttt{ctrdelay}(m, t, S, D) = 1$, the message from Bob in the example in Figure 4.1 would already be accepted.

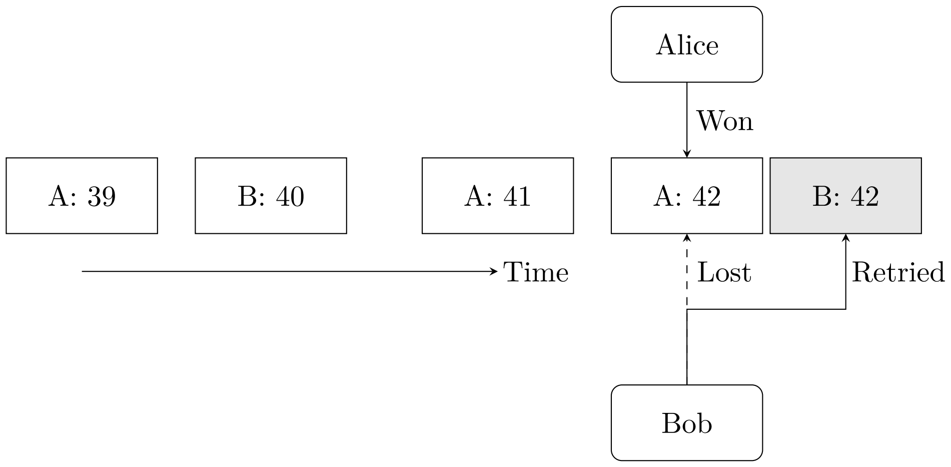

Figure 4.1: Diagram of counter nonce reuse due to bus arbitration. Alice and Bob both want to transmit a message in the same timeslot and both use the next counter nonce that should be used (42). Alice wins the arbitration and Bob automatically retries the transmission immediately afterwards. Bob's message now has a counter nonce that has already appeared in the past.

The S, D tolerance parameters of the `ctrdelay` function configure the maximum Silence Interval and the maximum Counter Nonce Delay respectively. The first decides how many messages from the past are accepted immediately after receiving a valid one, the second decides how quickly the tolerance drops to zero after receiving no messages. Configuring these two values depends on the system requirements, bus activity, amount of devices, and must be evaluated on a case-by-case basis. Some observations are:

- The tolerances may be configured per receiving Party and per Group, so some may be more tolerant than others.
- If simultaneous transmissions within the *same* Group are to be expected often, a higher maximum Counter Nonce Delay D for that Group is recommended. This depends of course on the amount of Parties in each Group.
- If long periods of silence between two transmissions within the *same* Group are to be expected, then a low maximum Counter Nonce Delay D and a low maximum Silence Interval S are recommended.
- If some Parties only transmit for extended periods without them checking any incoming messages, which would update their local counter nonce value, a higher maximum Counter Nonce Delay D is recommended. That is the case of a device without multitasking capabilities currently busy on a heavy blocking operation, which immediately transmits the operation's result, without checking the reception

queue first. A protocol extension to handle transmitting-only devices is discussed in Section 6.5.

4.7 Message logging and debugging encrypted content

CAN buses host a plethora of nodes with different functionalities and they must undergo intensive testing in the automotive industry. Having these devices communicate in encrypted manner is counter-productive for the debugging and testing process. Additionally, one may be interested in logging the data all parties exchange with each other for the same reason, but also for safety logs and fault history of a deployed system. Key escrow is a necessity. Although this topic is often discussed in context of privacy when a human is involved in the communication chain, it should pose no political issue when machines are the only communicating party in a fully automated manner.

The centralised nature of CBS is perfect for these requirement: the Server is the party generating all short term keys and distributing them to all Clients. It is thus by definition able to decrypt and validate all message coming from all Clients, regardless to which Group they are addressed to. The Server implementation may decide to decrypt all messages before logging them in plaintext or (better) logging them encrypted, but with a different key that the humans inspecting the system also have.

4.8 Optional message fragmentation

The CAN FD bus allows frames with payloads up to 64 bytes. While this may be plenty for many applications, there are definitely some that need more, as explained also in Section 1.3. A big question when designing the protocol was: where should it lay in the stack of communication layers? On top of the CAN layer or on top of a fragmentation/reassembly layer (if available)?

The CBS specification initially focused solely on securing data fitting into individual CAN FD frames with so called SADFD messages. To support transport protocols, a the SADTP message was added. These are a slight variation of the SADFD messages with longer payload and tag, as there is more space assumed available. The fragmentation of long SADTP messages is left to any underlying transport layer.

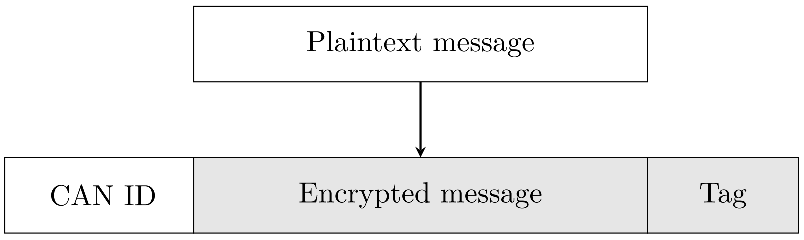

Figure 4.2: Security over CAN FD directly: the frame is secured, ready to be transmitted. The plaintext must be short-enough to fit into a secured CAN FD frame (SADFD).

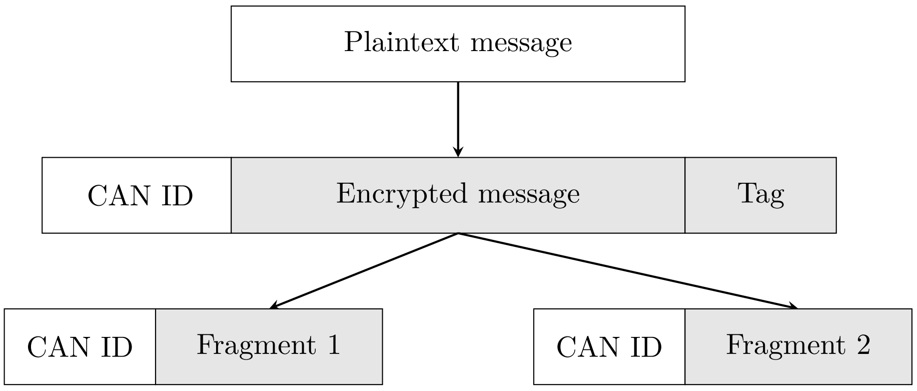

Figure 4.3: Security over transport protocol: the security is applied to the long message (SADTP), the fragments of which are sent over CAN FD. In this example transport protocol the CAN ID is reused in each single frame.

4.9 Authenticated-only messages

In some feedback about the protocol it was noted that there are no message types transporting plaintext but authenticated application data, thus having only the tag (MAC). It's worth mentioning that these messages were included in initial versions of the protocol specifications, but were later dropped in favour of AEAD-secured messages. The message structure was virtually the same: metadata, counter nonce, application data and tag; the only difference was whether the application data is encrypted or not.

The authenticated-only message were removed from the specification for the following reasons:

- Avoid human errors: the protocol user would have to decide whether to keep the application error confidential or not, which could lead to critical data being leaked if the wrong message type is selected. Thus it's better to have only one secure message type that protects also the content's confidentiality.

- Ignoring the tag: when the data is in plaintext, it may be tempting to skip the validation of the MAC and just use the data. If such data is encrypted, on the other hand, the only way to obtain it is to process it through the AEAD cipher which will automatically validate it's integrity and authenticity.
- Additional complexity: both the protocol specification and any implementation would have to deal with one more message type in a yet separate manner. More code means higher probability for bugs, thus better to keep the feature set and APIs at minimum. The same secured data transfer could be achieved by fully encrypting the payload.
- Performance: Ascon, the default cipher for the CBS protocol, has an interesting property, where its hash function has a higher computational cost than its AEAD cipher, because the hash uses more permutations[2]. Thus the cost of encrypting and authenticating the data is lower than just authenticating it.

4.10 Message length and MAC length

64 bytes of payload are not much in the modern world, where an Ethernet frame can carry 1500 bytes or even more with the *jumbo frames* extensions. The CBS SADFD message was designed in a way to reduce the overhead around the ciphertext as much as possible, exactly because of the available space being at premium.

For starters, the counter nonce was defined as a 24-bit integer rather than a 32-bit one to spare one byte – the most significant one, which would rarely be used in the first place. It was deemed that limiting the amount of messages to (up to) `0xFFFFFF` = 16 777 215 before changing short term key was enough for most real-world applications. For context, assuming all frames contain 64 bytes of pure data, that is approximately 1 gigabyte of data within one session, which is more than 2 hours of runtime assuming constant communication without overheads at 1 Mbit/s.

Secondly, the tag has been reduced to 64 bits instead of the commonly accepted security level of 128 bits. This clearly reduces the overall security level of the system by a lot, it's very worth noting. The idea of using only 8 bytes for the message authentication code comes from the compatibility of CBS with block ciphers, namely AES. The AES

[2]Specifically, the Ascon-128 AEAD cipher, v1.2, uses a 12-round p_a permutation once at initialisation and once at finalisation, the rest for absorbing input data is done with 6-round p_b permutations every 64 bits of input data. The Ascon-XOF hashing function uses instead only 12-round permutations [15].

cipher processes blocks of 128 bits (16 bytes) of data at the time. 3 blocks are 48 bytes, which leaves 16 bytes of space out of the 64 in the CAN payload for additional security metadata. Out of these 16, we have 3 for the counter nonce, 1 for the application data length and up to 3 for the SADFD-Header (assuming it's not part of the CAN ID), which leaves us with 9 bytes, which was rounded down to 8 in order to maintain 1 byte of free space available for future usage at the end of the message. Further notes on how to increase the tag length can be found in Section 6.4.

In case of stricter security requirements, the SADTP message format may be used instead, which has a 128-bit MAC. The drawback is the usage of a 32-bit field to indicate the length, thus consuming more space for the length value itself.

4.11 Motivation for choosing the Ascon cipher

- Ascon is an Authenticated Encryption with Associated Data (AEAD) cipher designed to be lightweight, which is exactly our use case. We don't need to worry about cipher modes like CBC, GCM, etc. as the cipher does already everything for us, compared to block ciphers like AES.
- Ascon has been selected as the primary choice for lightweight authenticated encryption in the final portfolio of the CAESAR competition (2014–2019) [16] and is (at the time of this writing) competing as a finalist in the National Institute of Standards and Technology (NIST) Lightweight Cryptography competition (2019–) [17].
- Ascon has a sponge-based design, making its hashes resistant to length-extension attacks, so we don't need to employ HMAC constructs, simplifying the message authentication process.
- The encryption, decryption, and hashing process reuses the same permutation function, thus we spare on code size.
- Ascon produces ciphertexts of the same size as plaintexts. This is an information leak about the plaintext, but it is not necessarily an issue for a system using CBS, for example because privacy of machine-to-machine communication may not be an issue, but in the case it is, it's easily fixed by padding the plaintext to a fixed length, much like block ciphers do. Further discussion on plaintext length obfuscation in Section 6.3.

  On the other hand, if no padding is applied, the ciphertext being no longer than necessary allows the usage of shorter CAN FD payloads, reducing thus the communication overhead.

- Ascon has some intrinsic properties due to its design, which are of of general interest for the overall system. These include some resistance to timing attacks and side channel attacks; Ascon is also very easy to implement and quite performant in software.
- Finally, Ascon was developed by TU Graz, thus the simplest motivation was to use an "in-house" developed algorithm to further see how well it can be applied.

5 Software implementation design choices

In this section we briefly tell the history of the development process, explain how the Hazelnet implementation of the protocol looks like, and motivate some software architectural decisions.

5.1 Design and development cycles

From a personal point of view, the writing of the protocol specification and the coding of the reference implementation required wearing many hats: security architect, protocol designer, software architect, and software developer. The most important self-imposed rule of the process was wearing strictly one hat at the time and, only once a task was completed, switch to another role, look at the result from a different perspective to find improvements, go back to the original hat to apply the fixes, and repeat the feedback cycle, like a one-person agile process.

For instance, one early version of the CBS protocol specification was focused on a large flowchart containing all possible choices and checks each party is supposed to perform. The mere size of the flowchart quickly proved to be ineffective to follow and understand when trying to write the software implementation of the protocol. This led to a complete overhaul of the protocol specification, using a more chronological format, explaining what each party does from booting up until powering down.

5.2 Why making a reference implementation?

A core part of the design of any kind of communication protocol, may it be cryptographic or not, is understanding how it could be implemented, whether it's easy to do it right and how usable would the API of an implementation be. Creating a reference implementation is useful for all that, but also provides a testing playground to see if the protocol even works in the first place, allows to quickly explore new functionalities, and finally, represents a secondary reference for the protocol specification. This is particularly useful when the theoretical specification is ambiguous or incomplete, so the reference software implementation could be inspected to resolve the ambiguity.

5.3 Hazelnet library

The library implements the CBS protocol. Instead of acting as a layer in the communication stack, wrapping security operations in a transmission and reception API, Hazelnet acts closer to a cryptographic library: it processes data passed to it by the application and returns the result to the same application, rather than exchanging it with neighbouring communication layers.

The user of the library must handle the physical transmission and reception manually as this library only builds the messages to transmit and unpacks the received ones. This is done to guarantee better portability across systems. The internal library state keeps track of ongoing handshakes, timeouts and other events for each Group. At the same time, such design allows multiple co-hosted applications to run concurrent instances of the library state, securing the messages independently of each other in case they don't trust each other

The library uses standard C11 code and is hardware-independent. The compile targets for a desktop OS add some optional features like heap memory allocation and use the time and TRNG functionality the OS provides. The any-platform version expects the user to provide the memory to operate on and function pointers to custom timestamping and random-number-generators of the used platform, ready for embedded systems.

5.3.1 Hazelnet library requirements

The protocol has been deliberately designed in a way to minimise the requirements for software implementations. Specifically, the need for synchronised clocks across all devices has been removed from the very start, because it's hard to achieve on a multi-master shared bus, the arbitration process being one of the reasons why. Timestamp nonces were replaced by counter nonces. Time functions are still required to check for timeouts and for wait/sleep periods, but no synchronisation to external reference clocks or between devices is needed.

The requirements thus are:

- A minimal C99 or C11 standard library, where heap-memory allocation (`malloc()` and `free()`) is optional.
- A true-random number generator (TRNG) – which requires dedicated hardware.

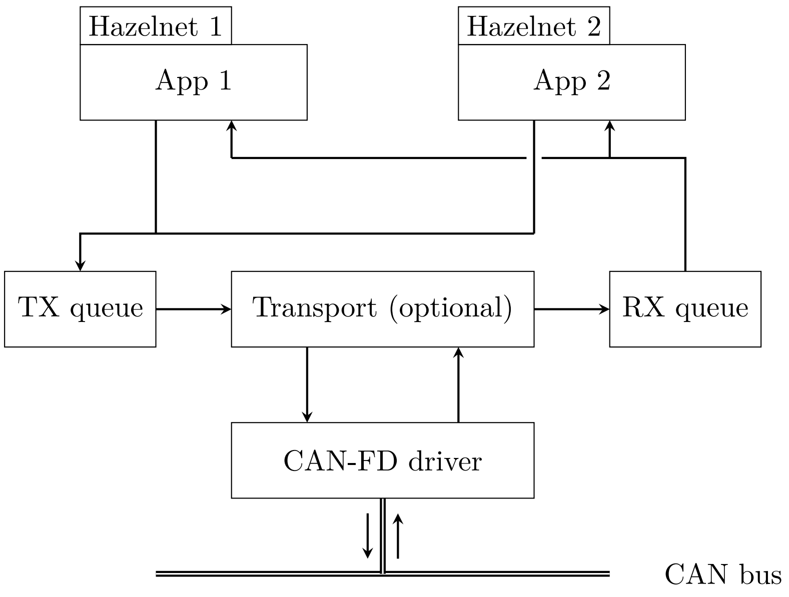

Figure 5.1: Two applications co-hosted on the same device can use different instances of the library state to independently secure the messages, even from each other, so they can share message queues. An incoming secured message popped from the queue is dispatched to the proper application, which passes it to its Hazelnet instance and obtains it back decrypted and validated. Vice-versa, the application calls the library on a plaintext message to get the message in a secure format, ready to be transmitted, and places it into a shared transmission queue.

- A local, relative, timestamping function with millisecond accuracy, like a tick-number function. It does not require to be an absolute wall-clock, as only local time differences are computed. Something as simple as the processor clock cycle counter suffices.

The last two points (TRNG and time function) are supplied as function pointers to the library, allowing the user to wrap any function that is available on the current platform into the format Hazelnet needs. The library has the two functions already prepared and are selects them automatically when running on a desktop operating system, while for embedded systems an adapter must be prepared manually. In the Hazelnet Demo Platform the RTOS current-tick function is used as time routine, while the hardware TRNG is the one the CSEc hardware security module of the S32K144 microcontroller provides.

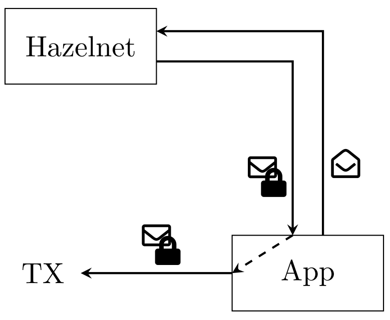

Figure 5.2: Diagram of application-library interaction on transmission: the application provides the plaintext data to the Hazelnet library, which returns it to the application encrypted, authenticated, and packed into a CBS message format, ready to be transmitted. The application has then only to forward it as-is to the transmission function or queue.

5.3.2 Hazelnet library architecture

Hazelnet has comprehensive software documentation with Doxygen[1], it's modular and easy to compile using CMake, abstracting away from a specific Make-tool, C compiler, and linker.

The library's internal code is split into 3 sections: code specifically used only by the Client role, code for the Server role and code shared by both. The overall software architecture tries to map each API function to one source code file with that function implementation and optionally some static functions.

The API is composed in a similar manner: the `hzl.h` header file is required for all roles and then the user can include `hzl_Client.h` or `hzl_Server.h` depending on the role they are choosing. The headers contain a very small amount of functions, as most of the lines of code in them are used for data structures definitions and documentation. Each API function returns an error code from an extensive enumeration that can indicate every possible error that may occur even very deep in the library's internal code.

We believe that the best way to explain what a library does, is taking a peek at its API, which in this case is very compact. A Client has the following functions available in their API (arguments and return types are omitted for brevity):

- `hzl_ClientInit()` to initialise an instance of the library state, which should happen on boot or awakening from a low-power state.
- `hzl_ClientDeInit()` being the complementary security-cleaning of the state be-

[1] https://thematjaz.github.io/Hazelnet/

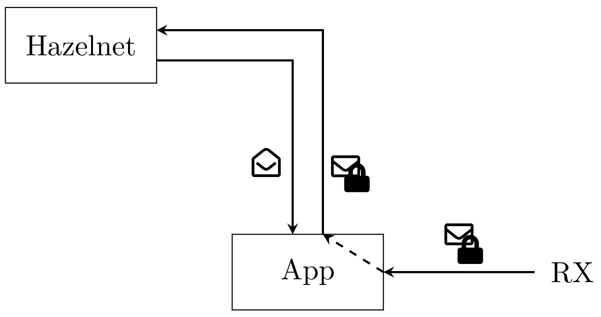

Figure 5.3: Diagram of application-library interaction on reception: the application obtains a secured, encrypted, authenticated message from a reception function/queue and forwards it as-is to the Hazelnet library. Hazelnet validates the message's format, authenticity, freshness and decrypts it, providing the plaintext data back to the application – or the proper error code if something went wrong.

fore a power-down or low-power state.

- `hzl_ClientBuildRequest()` to start a handshake: builds a Request message ready to transmit to the Server, in order to obtain the short term key of a specified Group. The transmission must be performed by the Hazelnet user (the application).
- `hzl_ClientProcessReceived()` is a function which should be called for every received message. If it's an internal one, like a Server Response or Session Renewal Notification, it will be automatically processed with no application data output for the user, optionally providing an automatic internal message for the user to transmit. If it's application data, it will be decrypted, validated, and passed to the user in plaintext.
- `hzl_ClientBuildSecuredFd()` can be used to prepare secured application data messages from plaintext data, ready to be transmitted by the user.
- `hzl_ClientBuildUnsecured()` analogously packs the application data for transmission without any cryptographic security.

The Server has a very similar API:

- These four functions behave exactly like their Client counterpart:
  - `hzl_ServerInit()`
  - `hzl_ServerDeInit()`
  - `hzl_ServerBuildSecuredFd()`

    - `hzl_ServerBuildUnsecured()`

- `hzl_ServerProcessReceived()` also processes any received message, while also automatically preparing Responses for the incoming Requests.
- `hzl_ServerForceSessionRenewal()` can be used to manually trigger a new session start right now instead of waiting for its expiration.

5.3.3 Hazelnet limitations

Hazelnet does not support SADTP message formats, thus large secured application data messages that travel over a transport protocol. This feature was skipped as not critical to test the protocol, being extremely similar to unfragmented SADFD messages.

Hazelnet is not a thread-safe library, thus a single library state-structure should not be used concurrently from multiple threads or RTOS tasks, but two distinct state-structure can coexist without issues (e.g., for two independent applications on the same host device). Support for thread-safety would require either a much more refined design or usage of locks, which makes the library harder to work cross-platform, given that every OS has different locking APIs. Support for this feature was dropped, as not critical in the initial versions to get the system running. The main drawback is that processing of received messages and building of messages to transmit cannot happen safely in separate tasks or threads of the same application.

The clear goal was to get a working, unit-tested, clear, well-documented implementation of the CBS protocol first, that is easy to understand and use rather than optimised for all features, performance, and use-cases.

5.4 LibAscon library

Before this thesis was started, the *LibAscon* implementation was created as an independent academic exploration of the Ascon cipher in order to provide a more flexible API than the reference implementation[2] developed by the Ascon designers themselves. LibAscon was then released as a stand-alone open source library, and listed among other implementations on the official Ascon website [15].

During the development of this thesis, Ascon was evaluated as an excellent cipher choice (see Section 4.11 for details) and thus the LibAscon implementation was picked, further tested in a practical project, and improved to be used specifically within the Hazelnet

[2] `https://github.com/ascon/ascon-c`

library, while still staying a stand-alone generic implementation that could be used in any project.

5.4.1 Why not using the reference implementation?

The reference C implementation implements only the offline encryption or hashing paradigm, i.e., its functions can operate only on a contiguous array of bytes already and completely available in memory. While this simplifies the reference Ascon implementation by a lot, the user of it must prepare the entire message to process in advance, which may be unfeasible or not elegant.

In the initial drafts of the Hazelnet library, one of the ideas was to encrypt/decrypt the CAN FD frames containing fragments of a large SADTP message in a streaming manner. The sender secures a large message that must be split into many CAN FD frames, but does so iteratively: as the first plaintext fragment is created, that fragment is encrypted and transmitted, then the next fragment is extracted from the large message, encrypted, and transmitted, and so on, until the final fragment is packed into a CAN FD frame alongside the tag (MAC) of the *entire* long message. The receiver would decrypt the fragments as they would come to avoid long blocking decryption times at the end and authenticate the tag once the final fragment is received. An Init-Update-Final API for the Ascon cipher was a requirement for such an implementation, allowing to operate on the data online, one chunk at the time, as it became available. However, this streaming approach was later deemed a nice-to-have optimisation step, which is not critical to achieve a working library. It remains a potential feature for a future version of Hazelnet.

Nevertheless, it's the opinion of the Hazelnet developer that the Init-Update-Final API for Ascon operations makes the code more legible and elegant, compared to organising all data in a buffer of specific format in advance to process it in one go. The code clarity issue, combined with the streaming implementation mentioned in the previous paragraph (although later abandoned), were the reasons for picking LibAscon over the reference Ascon implementation.

5.4.2 LibAscon properties

LibAscon tries to be a comprehensive, modern-C package-deal, where all variants of the Ascon cipher and hashing functions are neatly collected and available both in offline and online modes. The idea was to expand on the reference implementation in terms of API, thus getting closer to interfaces similar to other cryptographic libraries (e.g., OpenSSL, LibSodium).

We wanted the Ascon cipher to become a developer-friendly tool rather than an academic algorithm, bringing it closer to the professional software engineering world. The LibAscon implementation has comprehensive software documentation with Doxygen[3], it's modular and easy to compile using CMake, abstracting away from a specific Make-tool, C compiler, and linker.

LibAscon was never meant to be a deeply-optimised implementation or a security-hardened one. The reference repository contains specialisations of the code for different architectures and processor bitness, so there was no need to do it again. Security-hardening operations, such as masking, was not applied due to lack of knowledge in the sector. LibAscon instead focuses on being simple, understandable, general-purpose, and fully-portable.

LibAscon achieves the same performance as the official Ascon reference implementation on 64-bit machines, when compiling both in Release mode.

5.4.3 Features introduced by LibAscon

LibAscon builds on top of the reference implementation, extending it with some minor, but useful, additional functionalities:

- The already-mentioned online processing, which allows to encrypt, decrypt or hash the data one chunk at the time, even if it's not contiguous in memory.
- Arbitrary tag length for authenticated encryption: LibAscon's encryption routines can generate Message Authentication Codes of user-specified size. Ascon's reference AEAD design provides 128-bit tags; shorter versions are equivalent to truncations of this tag, while longer ones are a LibAscon extension of the algorithm using the same sponge-squeezing technique as for the extraction of an arbitrary-length digest for the Ascon-XOF hashing function.
- The AEAD tag may be provided to a separate location (separate pointer), not concatenated to the ciphertext.
- Encryption and decryption can also be performed in-place, without the need of a second output buffer. The plaintext is thus replaced with the ciphertext or vice-versa. While this functionality should work also on the reference implementation, by the looks of the source code, LibAscon contains explicit testcases for it.

[3]`https://thematjaz.github.io/LibAscon/`

5.4.4 Testing

LibAscon was tested using the official Ascon test vectors[4] to ensure compatibility with the reference implementation. A variety of fixes were applied to the code so it compiles without compiler warnings when using the most common compilers (GCC, LLVM-Clang, MSVC-CL) with most warning flags activated. The continuous integration pipeline in the repository hosting web-service allowed for quick compilation error checks and ensured the tests were passing on multiple platforms.

5.5 Hazelnet Demo Platform

The final deliverable of this thesis is a practical demonstration platform of the CBS protocol being used by microcontrollers actually communicating over the CAN bus.

5.5.1 Used hardware

The platform is made of 2, 3, or 4 microcontroller boards, each acting as a CBS party: the Clients (up to 3) are named *Alice*, *Bob* and *Charlie*, the last device is the *Server*. The boards are NXP S32K144EVB (evaluation boards of the S32K144 microcontroller) and they are connected via a simple star-shaped, high-speed CAN FD bus. The Server board provides the 120 Ω resistor line-termination on the board itself. The boards are powered using 12 V DC.

The choice for the microcontroller boards model was straight-forward: other than being obviously an in-house product, the NXP company team the thesis author was part of at the time of writing was already using them for a plethora of experiments, so access to senior engineers for support on using them was plenty. Simultaneously, they are automotive-certified products, support CAN FD and have a hardware TRNG module.

To sniff the bus traffic from a laptop and display the secured data being exchanged, a PEAK System PCAN-USB FD adapter was employed, for the same reasons as for the S32K144 microcontrollers.

[4]`https://github.com/ascon/ascon-c/blob/master/crypto_aead/ascon128v12/LWC_AEAD_KAT_128_128.txt` and analogous KAT (Known Answer Test) text files available in different locations in the reference implementation's repository for the various cipher and hashing types.

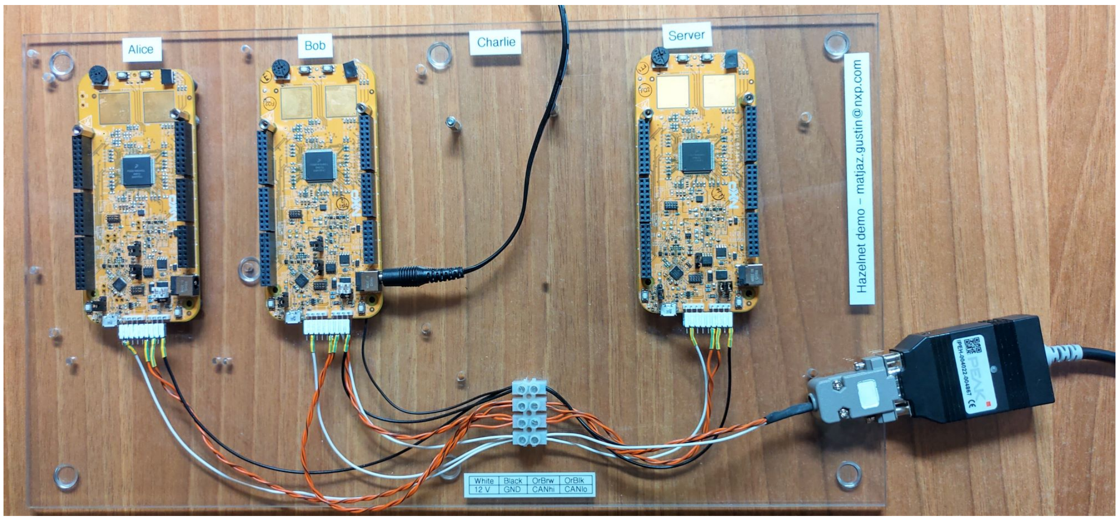

Figure 5.4: Photo of the Hazelnet Demo Platform's hardware: three NXP S32K144EVB microcontroller boards represent the parties (from left to right) *Alice*, *Bob* and *Server*. An out-of-picture power supply provides 12 V DC and the ground to the Bob board's right side with a black coaxial plug. In the bottom half of the picture, the orange twisted-pair wires are the CAN bus, the white wire is the 12 V DC power distribution, and the black wire is the ground. On the right of the picture, a PEAK System PCAN-USB FD adapter (the black box) is sniffing the traffic on the bus and providing it for a laptop/desktop computer to inspect, as an attacker would.

5.5.2 What the demo does

The devices are programmed to securely exchange dummy 8-bit integer value they increment every time they transmit it. This value is distinct from the message counter nonce, as each device has a different starting value of this 8-bit integer. Simultaneously, the devices log their activity with unsecured (UAD) messages on the bus, interleaved to the secured ones (SADFD), so it's easier to see what they are doing just by looking at the log of CAN FD messages. There is no debug logging via other peripherals.

On boot, each device logs their firmware and library version; the Server initialises a Session, while the Clients request the Session data to the Server. They then start broadcasting their integer value to everyone else on the bus, padded to a 128-bit block, using secured messages. Each device, upon receiving the message and decrypting it, it publishes its content yet again on the bus, now using an unsecured, plaintext message, which is ignored by everyone else. This last step is for demo purposes, so that it can be clearly seen in the logs who has received the message and successfully processed it – clearly no real-world application should echo security-critical data in plaintext back through the channel used to transfer it securely mere instants before.

Each Session has a preconfigured duration of 30 seconds, so a session renewal can be seen even with a relatively-short interaction with the demo. Again, clearly a real-world application would normally use much longer Sessions to reduce overheads in the distribution of the short term keys. The Server renews the Session after this timeout, notifying the Clients, who request the new short term key immediately.

In case the devices get out-of-sync for any reason, their communication with the other parties will start to raise security warnings (invalid counter nonces and message tags). Once the device's security error counter reaches a certain threshold (5 by default), their application layer decides to synchronise them again: Clients would transmit a Request for the Session Information, the Server a Session Renewal Notification instead to trigger Client Requests.

5.5.3 Firmware structure

The firmware differs only slightly between the Server and Client microcontrollers on the demo; they are otherwise structured and initialised identically. The firmware is based on the FreeRTOS[5] real-time operating system to provide some starting abstraction of the hardware and some handy functionality such as queues and clock ticks, although only a single task is employed.

The CAN driver is tasked to enqueue the received CAN FD frames upon receptions with a callback directly within the interrupt service routine that signals a reception. While not the best approach for high-intensity messaging applications, it's simple enough to implement and works for a basic demonstrator.

The only RTOS task is the initialised application, interacting with a Hazelnet instance. The devices have hardcoded configurations, again being the simplest solution for a demonstrator.

5.5.4 Firmware size

The Table 5.1 shows the size of the compiled firmware for both Client and Server roles. Both are optimised for size. The NXP S32K144 microcontrollers used have a 32-bit ARM Cortex M4F CPU using the Thumb instruction set.

[5]`https://freertos.org/`

	Client			Server		
	.text	.data	.bss	.text	.data	.bss
LibAscon	2868	0	0	2868	0	0
Hazelnet (incl. LibAscon)	**6608**	0	0	**6698**	0	0
Hazelnet example config.	0	20	192	0	24	260
Example application (incl. config.)	2046	20	302	1930	24	376
Entire firmware	33008	792	2784	32728	796	2860

Table 5.1: Sizes of components of demo platform firmwares optimised for size. All values are in bytes (decimal) and only include the used source code, i.e., the sizes are those appearing after the linker discarded unused sections.

The by-far largest part of the firmware are platform-enabling base libraries: hardware drivers (CAN, RGB LEDs, buttons, TRNG, memory regions, etc.), C standard library, RTOS (task, interrupts, queues, timers, etc.), startup code, and so on.

Clearly, the example application which exchanges only dummy messages and its simplistic configuration are not a realistic representation of what an industrial use-case would do, so it's size should not matter that much. The important detail is how much additional space the Hazelnet library code consumes.

Neither LibAscon nor Hazelnet libraries consume static memory: their contexts and buffers need to be declared or allocated by the library user. This may be on a static memory location, a declaration on the stack, a heap allocation, or even a flash memory page. This choice gives the user complete flexibility.

It has to be noted that the LibAscon library is not mandatory: the CBS protocol and Hazelnet could also work with a different cipher such as AES, which is often implemented in hardware. That could reduce the code size even more, assuming the drivers for said hardware implementation are not too heavy.

5.5.5 Laptop software

The PEAK System PCAN-USB FD adapter allows a laptop to inspect the traffic on the CAN bus. A small Python script is used to receive the CAN messages, unpack their content according to the format of CBS messages, and log them to the standard output in a human readable format. It's worth underlining that the laptop only reads the data from the bus for logging and visualisation purposes, never transmits anything, and is unable to decrypt the data because it has no cryptographic keys at all.

This logging script is not published along the thesis, as it based on an extensive companyproprietary Python wrapper of the PCAN-USB FD API, called *PyPCAN*. This wrapper is, incidentally, also a work of the same author of this thesis, predating it by a few years.

An example on how the log looks like is available in Section 5.5.7.

5.5.6 Interacting with the demo

Each board can be interacted with using its hardware buttons:

- Pressing the button 1 (marked *SW3* on the board itself, the one closest to the RGB LED) on Client boards deactivates the Client. This is to simulate either a power-down or entering a low-power state where the Client does not communicate over the bus any more. This button does nothing on the Server, which should never power-down as per CBS requirements.
- Pressing the button 2 (marked *SW2* on the board itself, the one closest to the potentiometer wheel) forces a resynchronisation of the Session Information: on a Client it triggers the transmission of a new Request message to obtain (again) the short term key; on the Server it triggers the renewal of the Session and subsequent transmission of a Session Renewal Notification message.
- Pressing the reset button (marked *SW5* and *RESET* on the board itself, the one closest to the micro-USB connector) clearly reboots the device abruptly. For Clients this triggers a new Request message, just like when pressing button 2. For the Server this causes an unexpected loss of the Session Information, because the Server is never supposed to power-down.

The RGB LED is used to show different states of the devices to the people interacting with the Hazelnet Demo Platform. The most important colours are:

- White for the Server that is waiting for the first Client to transmit a Request message after the Server boots up.
- Cyan for Clients that are in the middle of a handshake, waiting for a Response message.
- Magenta for the Server that is waiting for Clients to transmit a Request message during a Session renewal.
- Green for successful processing of received Secured Application Data messages.
- Red for security errors, such as invalid message tags.
- Cycling through the colours red-green-blue-off indicates the Client was virtually

powered-down by pressing the button 1.

- Any two colours alternating periodically indicate a fatal, unrecoverable error, such as the CAN bus wires being disconnected.

5.5.7 Example CAN log

Here we present a version of the CAN bus traffic log that we would observe when running the demo. The CBS messages are unpacked and their inner fields are shown in a more human-readable format.

The first column of each log record is a timestamp in seconds, here indicated as relative to the start of the demo for simplicity. The *GID*, *SID* and *PTY* fields are the CBS Header fields of each message. The UAD message denote logging message with plaintext data used to explain what the demo is doing. The *PL* field is the payload of the CAN FD frame, containing either a text message or binary fields, which are also unpacked and listed in hexadecimal formats.

On startup we see the devices notifying they booted up alongside with their firmware version.

```
0.870 | Starting HZL sniffer

0.878 | GID=0 | SID=BOB    | PTY=UAD
  PL=INFO: initialised Hazelnet Demo Platform v1.1.0

0.879 | GID=0 | SID=SERVER | PTY=UAD
  PL=INFO: initialised Hazelnet Demo Platform v1.1.0

0.881 | GID=0 | SID=ALICE  | PTY=UAD
  PL=INFO: initialised Hazelnet Demo Platform v1.1.0
```

Immediately afterwards, the Clients transmit Request messages to the Server, which responds within a few milliseconds:

```
0.881 | GID=0 | SID=BOB    | PTY=REQ
  PL=reqnonce=0x31302f2e2d2c2b2a,
     tag=3dc2138ef2553fdb

0.883 | GID=0 | SID=ALICE  | PTY=REQ
  PL=reqnonce=0x31302f2e2d2c2b2a,
     tag=b999ea6e79615b28

0.885 | GID=0 | SID=SERVER | PTY=RES
  PL=client_sid=2,
     ctr=0x000000,
     resnonce=0x81807f7e7d7c7b7a,
     ctext=4232c2e190903b483c65cb5a9355636d,
     tag=38b77bc962d56eb913dc5c01ea82d6c8

0.890 | GID=0 | SID=SERVER | PTY=RES
  PL=client_sid=1,
     ctr=0x000000,
     resnonce=0x8988878685848382,
     ctext=e45adeafec1788c34d7e44ecbf362f69,
     tag=0e23cc462832a8e3843b9da46e08bda5
```

Then all devices start to periodically transmit dummy encrypted application data, which is in this case just a counter, to all other parties. The counters start with `0xA0` for Alice, `0xB0` for Bob, `0xC0` for Charlie and `0xF0` for the Server. The receiving party decrypts the message and transmits it in plaintext back on the bus. `Secret counter` is the first byte of the decrypted message, i.e., the dummy value being exchanged by the Parties.

The Server transmits its first value `0xF0` encrypted, which Alice and Bob receive and publish again decrypted so we can see they processed it properly:

```
1.291 | GID=0 | SID=SERVER | PTY=SADFD
  PL=ctr=0x000000,
     ptlen=16,
     ctext=10fbe84975d0faee590d2b023d31cfb1,
     tag=20998faedfbb63ce

1.293 | GID=0 | SID=ALICE  | PTY=UAD
  PL=RX GID=00,SID=00,Secret counter=F0

1.294 | GID=0 | SID=BOB    | PTY=UAD
  PL=RX GID=00,SID=00,Secret counter=F0
```

Similarly, after about a second, Alice decides to transmit its own counter:

```
2.496 | GID=0 | SID=ALICE  | PTY=SADFD
  PL=ctr=0x000001,
     ptlen=16,
     ctext=1098f96da8667fd3dafc0dc116c3f0c7,
     tag=a19b80aa6b1078e8

2.498 | GID=0 | SID=SERVER | PTY=UAD
  PL=RX GID=00,SID=01,Secret counter=A0

2.499 | GID=0 | SID=BOB    | PTY=UAD
  PL=RX GID=00,SID=01,Secret counter=A0
```

After about 30 seconds since the start, the Server is programmed to renew the Session. The Server transmits a Session Renewal Notification message, triggering the Clients to transmit Requests once again and obtain new Responses.

```
32.887 | GID=0 | SID=SERVER | PTY=REN
  PL=ctr=0x000020,
     tag=a017d60ffb1c6430c85e3c625ae7f6d6

32.892 | GID=0 | SID=ALICE  | PTY=REQ
  PL=reqnonce=0x3938373635343332,
     tag=f1dd5d6940a99670

32.893 | GID=0 | SID=BOB    | PTY=REQ
  PL=reqnonce=0x3938373635343332,
     tag=98bf21464798e55a

32.896 | GID=0 | SID=SERVER | PTY=RES
  PL=client_sid=1,
     ctr=0x000000,
     resnonce=0xa1a09f9e9d9c9b9a,
     ctext=edce5ba74cc87164d486853fb7b9ae12,
     tag=49e8bc98015f72a40e8497fa01420171

32.900 | GID=0 | SID=SERVER | PTY=RES
  PL=client_sid=2,
     ctr=0x000000,
     resnonce=0xa9a8a7a6a5a4a3a2,
    ctext=faccaa4c37c445fec2cb975fd7f05bdd,
    tag=e8295f82a84e85d3b1db5516b6f76c78
```

6 Potential protocol extensions

No product is ever perfect and the same is true also for the CBS protocol and any of its implementations. Additional extensions and new features could be added to CBS, may it be in new minor versions by using some reserved fields and values, or by breaking compatibility and creating CBS v2.0. Regardless of the way, here we list some of the ideas that could take place, if the project will continue existing or is ever expanded by further researchers and developers. Some of these points relate to what mentioned in Section 2.2.

6.1 Perfect Forward Secrecy (PFS)

PFS is the property of cryptographic protocols that ensure the confidentiality of the encrypted data protected by short term keys even if the long term keys are leaked or compromised in the future. It is a powerful property for the privacy-robustness of the protocol.

For CBS, because the short term keys need to be distributed to multiple Clients and each of them must see the same key, a way to make it independent from the long term key is wrapping it using an ephemeral key in the Response message rather than using the long term key itself. The ephemeral key is used just for this wrapping and it's agreed on the fly as the Request comes using the Diffie–Hellman (DH) protocol. Both parties still use their long term keys as a means of authentication of the messages to avoid Man-in-the-Middle attacks. Naturally the Request and Response messages format would change.

For instance, the Request message would now provide, alongside the random nonce (still useful to prove the Response freshness), also the Client's DH public key. Other public DH parameters such as base and modulus could be included in the message or be pre-configured in both parties. The Server would then compute the DH-agreed ephemeral key on the fly, wrap the short term key with said ephemeral key, still use Client's random nonce to prove the Response freshness, and finally authenticate (but not encrypt) the entire Response message using the shared long term key.

The computational performance would clearly take a hit compared to using only symmetric cryptography. Using an elliptic curve DH implementation would certainly help, but the handshake is expected to be slower than with the pure-symmetric version. The two types of handshake might also coexist on the same bus, with some devices using the symmetric and others the PFS version, especially in different Groups with varying security requirements.

6.2 Automatic pairing of new Clients or a new Server

CBS long term keys are assumed pre-installed in the Clients and the Server, but a protocol extension could be added to allow a new party to perform a pairing sequence over the CAN bus, generate, and agree on a long term key to use from there onwards. Such pairing procedure would speed up the physical replacement of faulty devices as there would be no need for a human operator to generate and install the long term keys manually on each party. As with the previous Section 6.1, Diffie–Hellman could also be used to achieve this goal.

While such a pairing handshake based on asymmetric cryptography may be slow, it is a configuration-time operation in an assembly or repair workshop, so we can assume a waiting time for a human operator of few seconds does not impact their work. At run-time, the devices would still use only symmetric cryptography – excluding the extension mentioned in the previous Section 6.1.

The problem with DH key agreements is the verification of the identity of the interlocutor: how can the Client be sure they have just agreed on a key with the Server and not some other Client or a malicious device on the bus impersonating the other party? The two parties performing the DH handshake would need some kind of shared knowledge, maybe a pre-installed certificate from their manufacturers to sign the DH exchange with. This is a chicken-and-egg chain of trust scenario that must be carefully planned; if it falls back to a human operator verifying the exchange after installing a new device, there is no advantage.

6.3 Hiding the plaintext length

While probably not of high interest for machine-to-machine communication, there might be a need to hide the exact length of the plaintext from any passive sniffer of the bus traffic for privacy reasons. A very simple way to do so is padding the plaintext so the ciphertext length is rounded up to the next predefined block size, even when not using a block cipher. For instance, plaintext could be only multiples of 16 bytes in size.

CBS could be altered in order to encrypt the concatenation *length* || *plaintext* || *padding* rather than just the plaintext; the *ptlen* field could be thus removed from the SADFD message altogether, as the amount of blocks (0–3) could be inferred from the CAN frame's Data Length Code field. For SADTP messages the *ptlen* could also be removed as the length in blocks of the first encrypted fragment contained in the first CAN frame could also be inferred from the Data Length Code; from there on, decrypting this fragment would reveal the true length of the plaintext and thus the amount of fragments (CAN frames) that follow could be deduced from that.

An alternative solution that could already take place now without changes to the existing CBS protocol, requiring the application layer to prepare the data to be exchanged in the *length* || *plaintext* || *padding* format instead, forcing such a padding to achieve a multiple of some selected block size, e.g., 16 bytes.

6.4 Increase SAD tag length

The length of the tag (MAC) field of SADFD messages is defined to be 64 bits (8 bytes) long, so the message can contain up to 3 ciphertext blocks of 128 bits (16 bytes) each. A possible extension, fully backwards compatible, may be to extend the length of the tag field, if the CAN FD frame has some free space available, even if it's just a single byte, up to 256 bits (32 bytes) of tag length, if resistance against Grover's algorithm is a requirement.

The SADTP tag length may also be extended in an analogous manner while maintaining backwards compatibility.

6.5 Transmitting-only parties and per-party counter nonces

Some CAN bus nodes may be mostly focused on transmitting their data and may not listen for messages exchanged between every other node to spare on energy and make them cost-effective, as there is no need for heavy processing power for just a few transmissions. For example, a bus-connected temperature sensor that periodically transmits the reading should not need to listen to any incoming message.

CBS assumes all parties listen to the messages transmitted within the Groups they are part of, so they can keep track of what was the latest used counter nonce. Naturally, a transmitting-only party would quickly fall behind the rest and its messages would get discarded as too-old.

A potential solution could be using per-party rather than per-Group counter nonces. By doing so, a transmitting-only party Alice can keep on transmitting and incrementing its own counter nonce locally without the need to receive anything from the bus, with the exception of any messages from the Server. The drawback is that all other parties, which are interested into hearing from Alice, must keep track of Alice's counter nonce.

A solution of this kind is harder to scale, as most parties may have to receive and track counter nonces from most other parties. While initially explored as a CBS feature, it was abandoned because of heavier memory requirements (a data structure holding a 3-byte counter nonce per party *and* per Group rather than just per Group), but that does not mean it may not be useful for some applications.

6.6 Server replication

One may wonder how could the availability of the Server be ensured, because that's an obvious a weak spot of a centralised system. Luckily, this is not a new problem; any database, DNS, email, web server is facing it daily. The solution is usually redundancy: we need to set up two (or even more) Servers on the same bus. One is picked as *primary*, while the second is a *hot-standby replica*, to borrow the terminology from the database management system world.

The primary Server needs to share each newly generated short term key with the replica, immediately after generating it, while the replica needs to constantly monitor if the primary is still available, maybe with a heartbeat message. If the primary goes offline, the replica self-promotes and starts to act as the Server, replying to Clients and renewing Sessions, until the primary comes back online.

While the CBS protocol does not cover this functionality at the time of writing, it could be a potential extension to define CBS messages for synchronising wrapped short term keys between primary and replica (and vice-versa after a crashed primary comes back online) and heartbeat messages to verify availability.

7 Conclusion

7.1 Summary

The CAN Bus Security protocol offers a lightweight solution to secure the bus traffic against third parties reading the message contents, transmitting spoofed messages, or replaying past ones. CBS is based on a simple and well-known system architecture, supports fragmented and unfragmented messages with a small message and handshake overhead. Its protocol specification has been carefully crafted to be flexible to fit the needs of the target system while containing all required details to avoid ambiguities.

The Hazelnet library offers a portable, detailed, and highly documented reference implementation, that should be quick to set up in existing projects to upgrade their security thanks to the library's minimal API. Hazelnet has a small footprint in terms of code size and overall system performance. Multiple applications or tasks on the same device may use it to protect their messages even from each other. Finally, a demo of the library usage is also available.

7.2 Final thoughts

This project was composed by many steps: designing a security protocol from scratch, writing a formal specification for it, creating a reference software implementation, the cryptographic library for it, a demonstrator platform to showcase the result. Writing the documents in an understandable manner, covering all details, software documentation, testcases, build systems all took a lot of time to organise, but they are what make the project complete.

By releasing the specification and implementation software code to the public with permissive licenses for both, the hope is to push more industries that are currently using unprotected CAN bus communications to upgrade the security level in their products with an off-the-shelf solution without paywalls.

This thesis was done in collaboration with NXP Semiconductors Austria GmbH & Co KG.

Attachments

ATT-1 CAN Bus Security protocol specification. Protocol version 1.3, document revision 4.

- Available as embedded attachment within this PDF document.
- Available on: https://matjaz.it/cbs/

ATT-2 Hazelnet library, open-source reference software implementation of the CBS protocol.

- Repository: https://github.com/TheMatjaz/Hazelnet
- Generated documentation: https://thematjaz.github.io/Hazelnet/

ATT-3 LibAscon library source code, open-source software implementation of the Ascon cipher used within Hazelnet.

- Repository: https://github.com/TheMatjaz/LibAscon
- Generated documentation: https://thematjaz.github.io/LibAscon/

ATT-4 Hazelnet Demo Platform, open-source software example application of the Hazelnet library.

- Repository: https://github.com/TheMatjaz/HazelnetDemoPlatform

Bibliography

[1] *Car Connectivity Consortium, Digital Key, release 3.0.* URL: https://global-carconnectivity.org/wp-content/uploads/2021/11/CCC_Digital_Key_Whitepaper_Approved.pdf (visited on 2021-11-21).

[2] *ISO 11898-1:2015, Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling.* URL: https://www.iso.org/standard/63648.html (visited on 2019-03-09).

[3] *ISO 11898-2:2016, Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit.* URL: https://www.iso.org/standard/67244.html (visited on 2019-03-09).

[4] *ISO 11898-3:2006, Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface.* URL: https://www.iso.org/standard/36055.html (visited on 2019-03-09).

[5] *ISO 11898-4:2016, Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication.* URL: https://www.iso.org/standard/36306.html (visited on 2019-03-09).

[6] *CAN bus, Wikipedia entry.* URL: https://en.wikipedia.org/wiki/CAN_bus (visited on 2019-03-09).

[7] *CAN FD bus, Wikipedia entry.* URL: https://en.wikipedia.org/wiki/CAN_FD (visited on 2019-03-09).

[8] *CAN FD - The basic idea.* URL: https://www.can-cia.org/can-knowledge/can/can-fd/ (visited on 2019-03-09).

[9] *ISO-TP, Wikipedia entry.* URL: https://en.wikipedia.org/wiki/ISO_15765-2 (visited on 2021-11-21).

[10] *ISO 15765-2:2016, Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services.* Standard. Geneva, CH: International Organization for Standardization, 2016-04. URL: https://www.iso.org/standard/66574.html (visited on 2019-03-09).

[11] Roger M. Needham and Michael D. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Commun. ACM* 21.12 (1978-12), pp. 993–999. ISSN: 0001-0782. DOI: 10.1145/359657.359659. URL: https://doi.org/10.1145/359657.359659.

[12] R M Needham and M D Schroeder. “Authentication Revisited”. In: *SIGOPS Oper. Syst. Rev.* 21.1 (1987-01), p. 7. ISSN: 0163-5980. DOI: 10.1145/24592.24593. URL: https://doi.org/10.1145/24592.24593.

[13] Stefan Nürnberger and Christian Rossow. “– vatiCAN – Vetted, Authenticated CAN Bus”. In: vol. 9813. 2016-08, pp. 106–124. ISBN: 978-3-662-53139-6. DOI: 10.1007/978-3-662-53140-2_6. URL: https://doi.org/10.1007/978-3-662-53140-2_6.

[14] *LibAscon*. URL: https://github.com/TheMatjaz/LibAscon (visited on 2022-01-31).

[15] *Ascon - Lightweight Authenticated Encryption & Hashing*. URL: https://ascon.iaik.tugraz.at/ (visited on 2019-07-20).

[16] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. URL: https://competitions.cr.yp.to/caesar-submissions.html (visited on 2022-01-31).

[17] *Lightweight Cryptography competition*. URL: https://csrc.nist.gov/projects/lightweight-cryptography/finalists (visited on 2022-01-31).