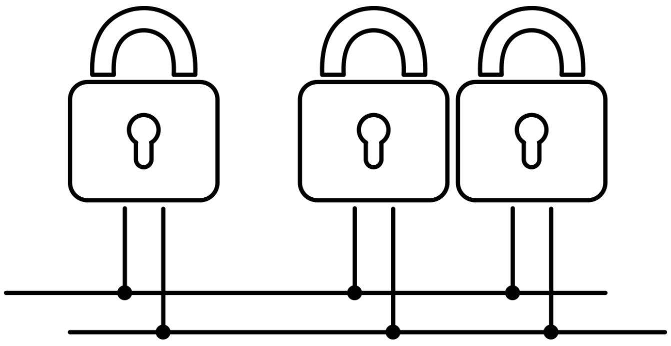
CAN Bus Security protocol specification

Protocol version: 1.3 – Document revision: 4

Matjaž Guštin
gustin@matjaz.it

2022-05-22



Contents

1 Definition and threat model **5**

1.1 Hardware requirements . . . 6

1.2 Communication stack . . . 7

2 General definitions **8**

2.1 Notation . . . 8

2.2 Parties . . . 8

2.3 Cryptographic keys . . . 9

2.4 Client and Server . . . 9

2.5 Groups . . . 9

2.5.1 Single vs. multiple Groups . . . 10

2.6 Session . . . 10

2.7 Endianness . . . 11

2.8 Warnings . . . 11

2.9 Time functions . . . 12

2.9.1 Current timestamp . . . 12

2.9.2 Time difference . . . 12

2.9.3 Current Counter Nonce delay . . . 13

2.10 Cryptographic functions . . . 14

2.10.1 Authenticated encryption with associated data . . . 14

2.10.2 Hashing . . . 15

2.10.3 Random number generation . . . 15

3 Client and Server states **16**

3.1 Client configuration . . . 16

3.2 Client state variables . . . 17

3.3 Server configuration . . . 17

3.4 Server state variables . . . 19

4 Message Header **20**

4.1 Header Types 20
4.2 Physical location of CBS Header within the CAN FD frame 21
4.3 Payload Types 23

5 Protocol timeline **25**
5.1 Server initialisation 25
5.2 Client initialisation 26
5.2.1 Request generation 26
5.2.2 Request message format 27
5.3 Server processing Request 27
5.3.1 Request message validation 27
5.3.2 Response message format 28
5.4 Client processing Response 29
5.4.1 Response message validation 29
5.4.2 Waiting for a Response 31
5.5 Application Data transmission 31
5.5.1 Incrementing Counter Nonces 32
5.5.2 Unsecured Application Data message format 32
5.5.3 Secured Application Data over CAN FD message format 33
5.5.4 Secured Application Data over Transport Protocol message format . 34
5.6 Application Data reception 35
5.6.1 Unsecured Application Data validation 35
5.6.2 Secured Application Data validation 36
5.6.3 Counter Nonce synchronisation and acceptance 37
5.7 Server renewing session 38
5.7.1 Expiration criteria 38
5.7.2 Server-side Session renewal phase 39
5.7.3 Session Renewal Notification message format 41
5.8 Client processing Session Renewal Notification 42
5.8.1 Session Renewal Notification validation 42

5.8.2 Client-side Session renewal phase 44

6 Checks against replays, spoofs and DoS attacks 46

6.1 Server processing messages only the Server may transmit 46

6.2 Client processing Requests 46

6.2.1 Request message validation 46

6.3 A note on Denial of Service (DoS) attacks 46

7 Low-power modes of the Parties 48

7.1 Client low-power mode 48

7.1.1 Client caching Request before low-power 48

7.2 Session Server low-power mode 49

Appendices 51

A Alternative AEAD ciphers and hash functions 51

B Reconfiguring Long Term Keys 52

Intended audience

This document is the formal reference description of the protocol, including detailed behaviours, message formats descriptions and validations. It is aimed at developers that want to implement the protocol or understand its behaviour in detail.

It is assumed that the reader is *already familiar* with the following notions:

1. basic cryptography including symmetric ciphers, authenticated encryption, hashing, cryptographic protocols, replay attacks, nonces, Alice & Bob notation,
2. CAN and CAN FD buses, their characteristics and behaviour,
3. (optional) the ISO-TP transport protocol over CAN and CAN FD.

1 Definition and threat model

The ***CAN Bus Security*** or in short *CBS* is a 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 of a Client requires a settings reset or a replacement in case of hardware failure.

CBS is based on the revised nonce-based Needham–Schroeder Symmetric Key protocol [1][2], which is the basis for the Kerberos protocol, for the key distribution and on vatiCAN [3] for the timeliness 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.

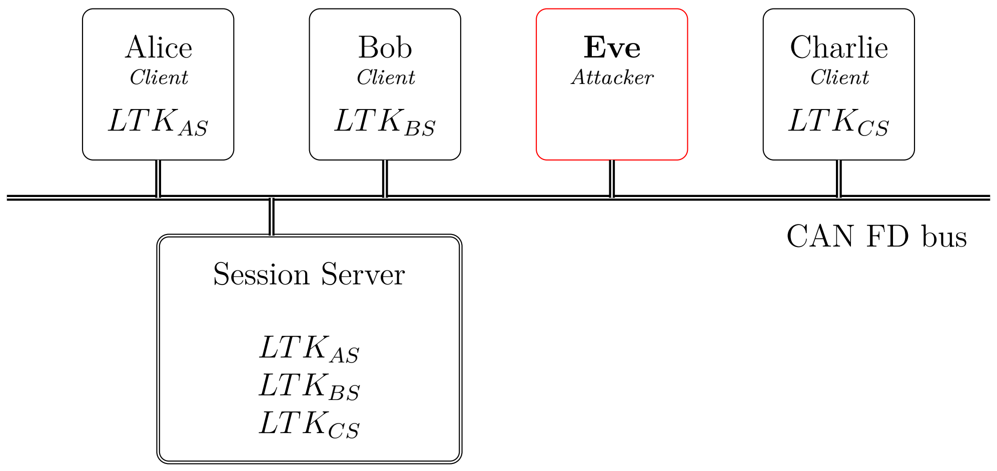


Figure 1: Abstract representation of a CAN FD bus shared between the Client nodes Alice, Bob, Charlie and a Session Server node with an attacker Eve having physical access to the same bus. Alice, Bob and Charlie are each preconfigured to securely communicate with the Session Server only at the beginning, without having the initial 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.

CBS **does protect** against:

1. tampering of messages, i.e. active changes in the messages on the bus,
2. sniffing sensitive data, i.e. reading the content of messages,
3. spoofing of messages, i.e. transmitting messages claiming to be someone else,
4. within a configurable time interval, also replay attacks, i.e. reuse of old messages to induce a behaviour in the bus-connected devices.

CBS does **not**:

1. hide communication metadata including:
   - CAN IDs,
   - which Parties are communicating,
   - when are they communicating,
   - length of encrypted data,
2. protect against software applications co-hosted on the same device reading each other's decrypted messages from a shared message queue[1],
3. protect against compromised Parties: the end nodes of the communication are assumed secure and well-behaving,
4. guarantee Perfect Forward Secrecy (PFS),
5. handle secure storage of cryptographic keys,
6. handle secure logging of messages.

Rationale: The systems connected with CAN buses are generally closed and have a small, fixed number of devices that communicate in a multicast fashion. A major security threat is a foreign device attached to the same bus that could sniff, spoof and replay the messages to force the whole system into a desired behaviour. Privacy is generally not an issue, as the communicating Parties are not humans, so the metadata is not being focused on.

1.1 Hardware requirements

CBS assumes the data-link layer is CAN FD because the 8 bytes of payload in classic CAN frames are not enough to carry the cryptographic overhead (nonces, MACs etc.). 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 payload length into classic CAN frames.

The platform using CBS must provide a hardware (true) random number generator (TRNG), used to prove the freshness of the CBS messages.

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. See Section 7 for details.

[1]In case of untrusted co-hosted applications, the implementation of the protocol can be moved from a shared security layer into the application space, as seen in the third stack example in Section 1.2.

1.2 Communication stack

The CBS implementation can potentially be placed in a variety of positions within the existing stack of communication layers, depending on the user's requirements. Figure 2 shows 3 common patterns. From left to right:

1. *CBS over CAN FD*: supporting only short messages, which are secured before being transmitted over CAN FD and decrypted before being passed to the appropriate application. Assuming a shared queue of messages, the co-hosted applications may still read each other's messages once decrypted. The whole physical device acts as a Party (see definition in Section 2.2).
2. *CBS over Transport*: similar to the previous point, but larger messages are supported, too. They are secured before being fragmented and reassembled before being decrypted. Assuming a shared queue of messages, the co-hosted applications may still read each other's messages once decrypted. The whole physical device acts as a Party.
3. *One CBS instance per application*: for cases will no trust between co-hosted applications, each one uses a separate queue and CBS implementation instance or is part of a different Group (see Section 2.5), allowing them to hide queued messages from each other, as they are provided in plaintext only to their own application. Each co-hosted application acts as a Party.

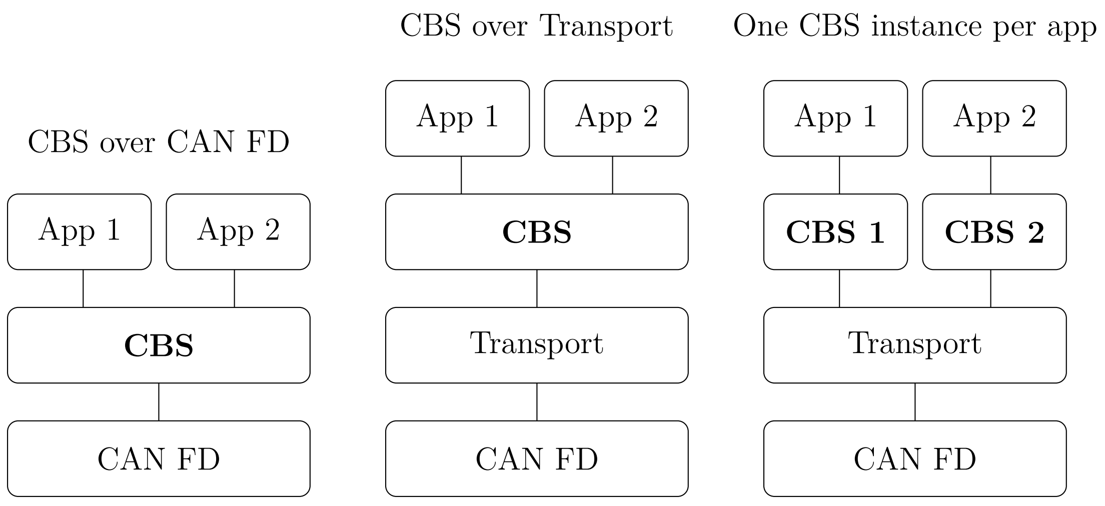


Figure 2: Possible positions of the CBS implementation in the communication stack. The Transport layer handles fragmentation and reassembly of long CBS messages that don't fit into a single CAN FD frame.

2 General definitions

2.1 Notation

- $:=$ indicates assignment of values to variables.
- $=$ indicates equality, like in a mathematical equation.
- $||$ is the concatenation operation.
- Integer data types are denoted similarly to the ISO C standard integer library `stdint.h`: `uint8` being an unsigned 8-bit integer, `uint64` being an unsigned 64 bit integer and so on. A *byte* is synonymous with `uint8` and is shortened with "B". A `uint24` type is equivalent to a `uint32` without its most significant byte.
- Arrays are indicated with the forms:
  - `uint8[16]` for fixed-length in bytes, in this case 16 (128 bits)
  - `uint8[*]` for arbitrary length
- `0x` is the prefix denoting that the following number is in hexadecimal notation. E.g. `0x012A` is 298.

2.2 Parties

Let us define a *Party* as an abstract entity willing to communicate securely with others on the bus. Party names use the Alice and Bob [4] notation. The Parties are thus listed as *Alice*, *Bob*, *Charlie*, etc. and map bijectively to their initials A, B, C etc.

The Party may be a physical device or a software module, even co-hosted on the same physical device along with other software modules, all of them using CBS.

The *Source Identifier*[2] id^A is a `uint8` integer value in $[0, 255] \cap \mathbb{Z}$ that uniquely represents the Party A. The numbering starts with 0 and is used sequentially with no gaps, e.g. using the identifiers $\{0, 2, 3\}$ is not allowed, while $\{0, 1, 2\}$ is. In other words, the largest Source Identifier used must be equal to the amount of Parties on the bus minus 1.

The **Session Server has always the Source Identifier 0 (zero)**. As a consequence there may be at most 256 Parties and of these at most 255 Clients.

Example with 4 Parties S, A, B, C:

$$id^S = 0, id^A = 1, id^B = 2, id^C = 3$$

[2]While *Party Identifier* may have seemed a more intuitive naming choice, its acronym *PID* could have caused some confusion with the *Process Identifier* of an operating system, especially if the same physical device runs multiple processes, each acting a CBS Party.

2.3 Cryptographic keys

Two types of cryptographic keys are used in CBS:

- *Long Term Keys* or *LTKs* are 128 bits or more long static, constant, preconfigured, symmetric keys that uniquely identify a Party. Each Party has only one unique LTK. Unless a security incident occurs, LTKs never change. They must be kept in persistent storage (preferably secure storage). LTKs are used to securely obtain STKs.

- *Short Term Keys* or *STKs* are 128 bit long symmetric keys randomly generated at runtime that identify a Session with a limited lifespan. STKs may be kept in volatile storage (preferably secure storage). They are used to secure application data transferred between Parties.

2.4 Client and Server

CBS is based on the client-server paradigm, where one central Party called the *Session Server* keeps track of the security Session and provides the current STK to the Clients upon request. A *Client* is a Party having only one LTK and using it to obtain the current Session information from the Session Server, which in turn keeps a copy of the LTK of each Client, enabling the Server to securely communicate with each Client independently. Clients trust the Session Server by definition.

The Session Server is a *role* one Party assumes. Said Party may only perform this role (e.g. a dedicated device on the bus) or may also be assigned other tasks, including exchanging application data with other Parties.

A generic Client is indicated with an *A* or *Alice* while the Session Server is always indicated with an *S*.

2.5 Groups

Groups are preconfigured *ordered* sets of Clients that use the same STK. Different Groups have different STKs, effectively isolating them from each other. By definition, the Session Server is always part of every Group, because it has access to all STKs. The Group cardinality must be at least 2, thus the Session Server and one or more Clients.

A generic Group is indicated with a G; any variable or constant indicated with a G subscript (e.g. x_G) is an instance of that variable for a specific Group G.

The *Group Identifier* id_G is a `uint8` integer value in $[0, 255] \cap \mathbb{Z}$ that uniquely represents the Group G. As a consequence there may be at most 256 Groups. The numbering starts with 0 and is used sequentially with no gaps, e.g. using the identifiers $\{0, 2, 3\}$ is not allowed, while $\{0, 1, 2\}$ is. In other words, the largest Group Identifier used must be equal to the amount of Groups on the bus minus 1.

The **Group Identifier 0 (zero) must always be available** and **all Parties belong to it**; in other words, it's the broadcasting group.

Example with 4 Parties S, A, B, C and 3 Groups J, K, L:

$$\begin{aligned} id_J &= 0,\ J \coloneqq \{S, A, B, C\} \\ id_K &= 1,\ K \coloneqq \{S, A, B\} \\ id_L &= 2,\ L \coloneqq \{S, B\} \end{aligned}$$

2.5.1 Single vs. multiple Groups

In the simplest setup of a bus, the broadcasting group with $id_G = 0$ is the only available Group, allowing no domain separation between the messages on the bus: everything is accessible to any other CBS Party (but not to non-CBS-enabled nodes). In case there is only a single, broadcasting Group, the Group Identifier may be omitted from the message headers (see Section 4.1).

A more sophisticated approach uses many Groups to allow only some Parties to be able to decrypt specific messages and prevent other Parties from doing so to reduce the attack surface in case a Party is compromised or simply untrusted.

Because of the multicasted nature of the communication and the use of symmetric cryptography, any Party within a Group could sniff sensitive messages other Group members send to each other and generate fake message claiming they come from someone else in the Group. All Parties within a Group are trusted and assumed to behave properly, in the sense that they don't spoof or alter each other's messages within the Group. To prevent this kind of possibility completely, the bus may be configured to have many Groups of just 2 Clients for 1:1 Client-Client or 1 Client for 1:1 Client-Server communication. The Session Server is anyhow in every group.

2.6 Session

A *Session* is the time interval where a certain STK is chosen by the Session Server to be the current one for a Group and distributed to Clients upon request. The STK is then used to enable secure application data exchange between all Parties within the same Group.

The Session Server starts a new Session when the previous expired or initially immediately after its boot. A Session has an expiration in time (amount of seconds since being established) and in number of messages sent by all involved Parties combined, whichever comes first, and it's automatically replaced with a new Session after expiration. See Section 5.7 for details.

2.7 Endianness

All multi-byte integers such as a `uint64` must be encoded with **little-endian** byte order: least significant byte first, at byte with index 0. When concatenating values, the left-most value in the concatenation occupies the least significant bytes. When concatenating binary arrays, the left-most value in the concatenation occupies the bytes with smaller indices in the array. Bit sequences and bytes are always indicated with the most significant bits on the left-most side and with the bit at index 0 being the least significant; "MS" and "LS" are used to indicates the Most Significant and Least Significant bits respectively.

These rules are valid for all messages as well as for all inputs of encryption, decryption and hashing functions.

A few examples:

- `uint16` $x = 1$ is encoded as `uint8[2]` $= (1, 0) =$ `0x0100`.
- Assuming `uint16` $x = 1$, `uint16` $y = 2$, the concatenation $x||y$ is encoded as `uint8[4]` $= (1, 0, 2, 0) =$ `0x01000200`.
- Assuming `uint8[2]` $x = (1, 2) =$ `0x0102`, `uint8[3]` $y = (3, 4, 5) =$ `0x030405`, the concatenation $x||y$ is encoded as `uint8[5]` $= (1, 2, 3, 4, 5) =$ `0x0102030405`.

Rationale: Little-endian is the dominant endianness for x86 and many ARM architectures (many ARM architectures have configurable endianness but little is default). Other than that, simply one endianness must be picked to avoid confusion.

2.8 Warnings

In case a security issue or attack is detected, a warning should be issued by the CBS implementation to the application by any means, in order for it to decide how to handle the issue (e.g. request a retransmission of a broken message or ignore it).

The Warnings, their shortenings/acronyms and encoding values are:

- Success (OK) $:= 0$. The usage of this warning is optional at the successful validation of any received message or other successful operation.
- Invalid tag (INV) $:= 1$. The message is not intact or not authentic or does not use the proper values in the hashing/authenticating operations.
- Message From Myself (MFM) $:= 2$. The message contained the receiver's Source Identifier as the transmitter's identity.
- Not Expecting a Response (NER) $:= 3$. The Client received a Response addressed to it while not expecting any.

- Server-Only Message (SOM) $:= 4$. The message is of a type which only the Session Server can transmit, but the transmitter's identity was not the Session Server.
- Response Timeout (RTO) $:= 5$. The Client did not receive a Response to its Request within the preconfigured timeout.
- Old message (OLD) $:= 6$. The message contained a too-old counter nonce.
- Denial of Service (DOS) $:= 7$. The system is receiving too many suspect messages.
- Not In Group (NIG) $:= 8$. The Party the Request originated from does not belong into the requested Group.
- Received Overflown Nonce (RON) $:= 9$. The received message contained a Counter Nonce that exceeded its maximum allowed value.
- Received Zero Key (RZK) $:= 10$. The received Response message, once decrypted, contained an all-zeros STK_G that cannot be used.
- Values $\in [11, 15]$ are reserved for future use (RFU).

The values are chosen to fit into 4 bits.

2.9 Time functions

2.9.1 Current timestamp

The `currenttime`() function is a generic timestamp-generating function. As the timestamps are never transmitted in the messages, their format is not relevant for the purposes of this protocol; for this reason each Party may have a different implementation of this function. The function is not required to provide the absolute current timestamp, in the sense of date and time; a relative timestamp suffices, in the sense of a rolling counter since an arbitrary point in time. The required accuracy is 1 millisecond.

2.9.2 Time difference

The `timedelta`(t_1, t_2) function provides the elapsed time in milliseconds between two timestamps generated by `currenttime` with a resulting accuracy of 1 millisecond or better. It assumes that t_2 represents a timestamp in the future compared to t_1, even if the actual values are $t_2 < t_1$. The implementation of this function must take care of any roll-around of the timestamps if too much time passes between them. As with the `currenttime` function, the exact binary format of the output of this function is not relevant; for this reason each Party may have a different implementation of this function.

2.9.3 Current Counter Nonce delay

The $\texttt{ctrdelay}(m, t, S, D)$ function provides the current `uint24` Counter Nonce validity delay used to validate the freshness of the received messages. This delay is used to accept also slightly older messages in case of congestion on the bus (see Section 5.6.3). It takes 4 input parameters:

1. the timestamp m of the last valid message received which is assumed to be in the past, i.e. older or equal than $\texttt{currenttime}()$,
2. the timestamp t when to compute the Counter Nonce validity delay, which is generally going to be the reception instant of the message containing said Nonce, obtained through $\texttt{currenttime}()$,
3. the maximum Counter Nonce Delay allowed $D \in \mathbb{Z}, \geq 0$, usually a constant,
4. the maximum Silence Interval allowed $S \in \mathbb{Z}, \geq 0$, usually a constant, in the same units as the output of the `timedelta` function.

The function is defined on $[0, +\infty[$ as:

$$\texttt{ctrdelay}(m, t, S, D) = \begin{cases} 0 & \text{if } \texttt{timedelta}(m, t) \geq S \\ \left\lceil D\left(1 - \frac{\texttt{timedelta}(m,t)}{S}\right)\right\rceil & \text{otherwise} \end{cases}$$

It can be broken down as follows:

- $\texttt{timedelta}(m, t)$ is the elapsed time since the last valid message was received,
- $D(1 - \frac{\texttt{timedelta}(m,t)}{S}) = D - \frac{D}{S}\texttt{timedelta}(m, t)$ is a real-numbered linear decay from D to zero over a time interval of S, indicated as l in the Figure 3. D is the offset of the line equation while $-\frac{D}{S}$ is its slope,
- the ceiling function $\lceil \cdot \rceil$ is used to convert the real-numbered decay to integers,
- the $\texttt{timedelta}(m, t) \geq S$ condition is used to provide a zero-delay after the maximum Silence Interval S has passed, effectively allowing no Counter Nonce delays after that. This condition also simplifies error handling, as it prevents a division by zero, in case $S = 0$, which would be the case of no tolerance for older Counter Nonces.

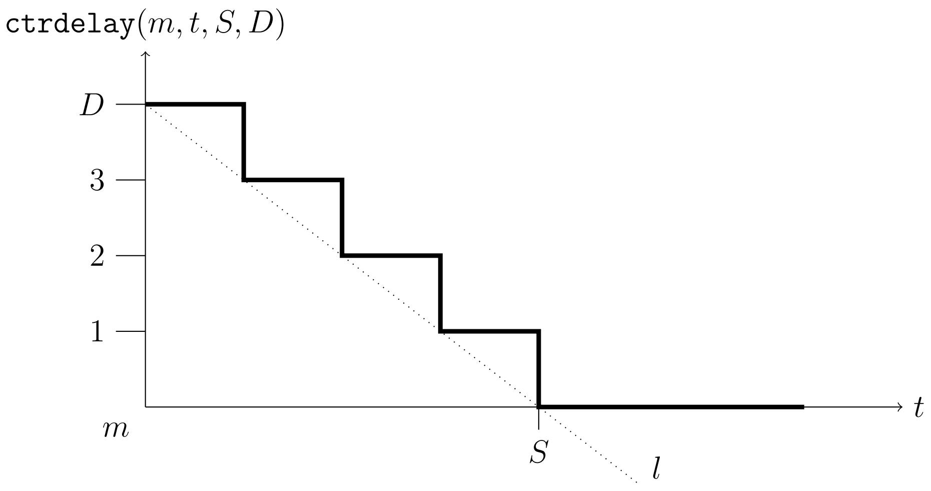


Figure 3: Plot of the `ctrdelay` function with since the fixed instant m of the last valid received message, $D = 4$ and S a fixed positive constant. The S parameter is used to stretch the decaying of the delay over time, while the D parameter is used to indicate the starting delay. The line l is the real-value decaying line, then converted to the ceiling-integer equivalent. Note that the value of `ctrdelay` stays at 0 for $t \geq S$.

2.10 Cryptographic functions

2.10.1 Authenticated encryption with associated data

`AEAD`(K, n, ad, pt, tl) is the declaration of the function for authenticated encryption with associated data with variable tag length. The Ascon128 v1.2 cipher [5] is chosen as its definition[3]. It takes 5 input parameters:

1. the secret key K, 128 bit;
2. the unique public nonce n, 128 bit;
3. the public associated data ad to be authenticated but not encrypted;
4. the plaintext pt of arbitrary length to be authenticated and encrypted;
5. the desired length of the output tag tl in bytes.

The `AEAD` function has two outputs:

1. the *ciphertext*, being the encrypted plaintext. It has the same or larger length than the plaintext;
2. the *tag* (a.k.a. Message Authentication Code or MAC) of arbitrary length, proving the authenticity and integrity of the associated data and the ciphertext.

[3]For the usage of different ciphers, see the Appendix A.

Rationale: Ascon is a sponge-based authenticated encryption algorithm designed to be lightweight and have some countermeasures against side-channel attacks. Ascon has been selected as the primary choice for lightweight authenticated encryption in the final portfolio of the CAESAR competition (2014–2019).

Ascon has a very small state, making it memory-efficient for embedded systems; it performs a single-pass authenticated encryption without the need for complex modes such as AES-CBC with EtM or AES-GCM; it also supports arbitrary input data length and generates ciphertext with the same length as the plaintext, removing the need for padding of the plaintext and in some cases of the padding that CAN FD applies to the payload, sparing on message size and transmission time.

2.10.2 Hashing

`Hash`(x, dl) is the declaration of the cryptographic hash function with variable digest length and resistance to length-extension attacks. The Ascon-XOF v1.2 function [5] is chosen as its definition. It takes 2 input parameters:

1. the input data x to be hashed, arbitrary amount of bits;
2. the desired length of the output digest dl in bytes.

Rationale: Ascon-XOF has the same state and permutations as Ascon128 (see Section 2.10.1), allowing a reduced code size due to code reuse with the same security properties.

2.10.3 Random number generation

`TRNG`(n) is the true-random number generator producing n random bits. This could be provided by a hardware module or by an operating system[4].

[4]For practical reasons it could be implemented using a CSPRNG with a secret initial seed, however such implementation is discouraged, as it must take care of never repeating the same value, not even after a reboot of the device or reinitialisation of the CSPRNG. Such a mistake could lead to nonce reuses which could leak information about the plaintext.

3 Client and Server states

Here is indicated the formal list of constant configuration fields and runtime-changing variables for Clients and Sessions Server.

3.1 Client configuration

Clients hold a few global constant settings which do not change at runtime and are thus kept in persistent memory. The default values are just recommended values and can be configured differently by the protocol user.

- The *Long Term Key* LTK_{AS} to communicate with the Session Server.

  It must be at least 128 bits (16 B) in length and not all-zeros.

  Default[5] length: 128 bits.

- The *Response timeout* t_{reqres}, being the amount of time since the transmission of a Request in which a Response is expected.

  It must be $\in [0, 65535]$ ms.

  Default: 100 ms.

- Its own *Source Identifier* id^A.

- The *Header Type* h as per Section 4.1 to use for all messages. Must be the same for all Parties.

  Default: H0, $\forall G$.

- The *Set of Groups* the Client belongs to. This is defined as a set of tuples:

  $$(id_G, D_G^{max}, S_G^{max}, t_G^{ren}), \forall G$$

  with per-Group configuration, containing:

  - The *Group Identifier*: id_G
  - The *Max Counter Nonce Delay*: D_G^{max}, used to filter out recent messages from old ones. Note that different Parties may have different values for this value. It is recommended to use small values for stricter security requirements. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.

    It must be $\in [0, 2^{22}], \forall G$.

    Default value: $20, \forall G$.

[5] Keys with more than 128 bits may require a different implementation of the `AEAD` function. See Appendix A.

- The *Max Silence Interval*: S_G^{max}, used to filter out recent messages from old ones. Note that different Parties may have different values for this value. It is recommended to use small values for stricter security requirements. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.

  It must be $\in [0, 65535]$ ms, $\forall G$.

  Default value: $5000\,\text{ms} = 5\,\text{s}, \forall G$.

- The *Client-side Session Renewal time Duration*: t_G^{ren}, used to know when the information about the expired Session must be deleted.

  It must be $\leq 6\, t_G^{ntf}, \forall G$, where t_G^{ntf} is the Delay between Session Renewal Notifications of the same Group the Session Server uses (see Section 3.3).

  It must be $\in [0, 65535]$ ms, $\forall G$.

  Default value: $5000\,\text{ms} = 5\,\text{s}, \forall G$.

3.2 Client state variables

Clients hold a few state variables for each Group G which change at runtime. They may be kept in volatile memory. They must be all initialised to zeros.

- The *Short Term Key* STK_G of the Group. Exactly 128 bit (16 B) in length.
- The *Counter Nonce* N_G^{ctr} is a `uint24`, used in messages exchanged within the Group G to prove the freshness of the messages.
- The *Request Timer* r_G is a variable of the same data type as the output of the `currenttime` function, used to check whether a Request's timeout is expired.
- The *Message Timer* m_G is a variable of the same data type of the output of `currenttime` that keeps the timestamp of the latest valid received message to scale down the acceptable Counter Nonce interval. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.
- The *Request Nonce* N_G^{req} is a `uint64` that keeps a copy of the random nonce used in the Request messages for the STK_G and used to validate the Response. It is also used to indicate that a Response is expected in the first place when $\neq 0$.
- The information of the just-expired Session: $STK_G^{old}, N_G^{ctr,old}, m_G^{old}$, which have the exact same format and role than STK_G, N_G^{ctr}, m_G, but they refer to the previous Session rather than the current one.

3.3 Server configuration

The Session Server holds a few global constant settings which do not change at runtime and are thus kept in persistent memory:

- The *Long Term Keys* of *each* Client: $LTK_{AS}, LTK_{BS}, LTK_{CS}$ etc., to secure the STK distributions.

  They must each be at least 128 bits (16 B) in length and not all-zeros.

  Default length: 128 bits.

- The *Header type h* as per Section 4.1 to use for all messages. Default value: H0. Must be the same for all Parties.

- The *Set of all Groups*, defined as a set of tuples:

$$(id_G, P_G, D_G^{max}, S_G^{max}, N_G^{exp}, s_G^{exp}, t_G^{ntf}), \forall G$$

  with per-Group configuration, containing:

  - The *Group Identifier*: id_G
  - The *Set of Parties* belonging to this Group: P_G
  - The *Max Counter Nonce Delay*: D_G^{max}, used to filter out recent messages from old ones. Note that different Parties may have different values for this value. It is recommended to use small values for stricter security requirements. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.

    It must be $\in [0, 2^{22}], \forall G$.

    Default value: $20, \forall G$.

  - The *Max Silence Interval*: S_G^{max}, used to filter out recent messages from old ones. Note that different Parties may have different values for this value. It is recommended to use small values for stricter security requirements. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.

    It must be $\in [0, 65535]$, ms$\forall G$.

    Default value: $5000\,\text{ms} = 5\,\text{s}, \forall G$.

  - *Counter Nonce Upper Limit*: N_G^{exp}, used to know when the current Session expires due to the amount of messages sent. It is recommended to use small values for stricter security requirements.

    It must be $\leq 2^{24} - 2^7 = 16777088 = \texttt{0xFFFF80}, \forall G$.

    Default value: $2^{24} - 2^{16} = 16711680 = \texttt{0xFF0000}, \forall G$.

  - *Session Time Duration*: s_G^{exp}, used to know when the current Session expires due to enough time passed since its establishment. It is recommended to use small values for stricter security requirements.

    It must be $\leq 2^{32} - 1\,\text{ms}, \forall G$, which is approximately 49 days.

    Default value: $3600000\,\text{ms} = 3600\,\text{s} = 1\,\text{hour}, \forall G$.

  - *Delay between Session Renewal Notifications*: t_G^{ntf}, used to know when to send the next Notification.

    It must be $\in \left]0, \left\lfloor \frac{s_G^{exp}}{6} \right\rfloor\right[$ ms$, \forall G$.

    Default value: $2000\,\text{ms} = 2\,\text{s}, \forall G$.

There are no Source Identifiers in the Session Server configuration, as they are always 0 for the Session Server for each Group.

3.4 Server state variables

The Session Server holds a few state variables for each Group G which change at runtime. They may be kept in volatile memory[6]. They must be all initialised to zeros.

- The *Short Term Key* STK_G of the Group. Exactly 128 bit (16 B) in length.
- The *Counter Nonce* N_G^{ctr} is a `uint24`, used in messages exchanged within the Group G to prove the freshness of the messages.
- The *Session Timer* s_G is a variable of the same data type as the output the `currenttime` function, used to check whether a Session is expired.
- The *Message Timer* m_G is a variable of the same data type of the output of `currenttime` that keeps the timestamp of the latest valid received message to scale down the acceptable Counter Nonce interval. See definition of the `ctrdelay` function in Section 2.9.3 for usage details.
- The information of the just-expired Session: $STK_G^{old}, N_G^{ctr,old}, m_G^{old}$, which have the exact same format and role than STK_G, N_G^{ctr}, m_G, but they refer to the previous Session rather than the current one.

[6] In case the state variables are kept in volatile memory and the Session Server reboots, it will not be able to communicate with other Clients, but the Clients will be able to communicate with each other, if they already obtained the Session information from the Server before its reboot. The protocol is designed assuming the Session Server should not reboot. For higher availability, it's recommended to keep the state variables in non-volatile memory.

4 Message Header

Each CBS message contains two parts: the *Header* with the metadata and the *Payload* with either CBS-related data or with wrapped application data.

The Header has always the same fields in every message:

1. *GID*: contains the Group Identifier id_G of the Group the message is destined to. It defines the Parties that should have the STK to decrypt and validate the content of the message. Optional in case only a single, broadcasting Group is used on the bus, as it's value could be assumed to be 0 (zero) instead of being transmitted.

2. *SID*: contains the Source Identifier id^A of the Party A transmitting. It is always 0 for the messages transmitted by the Session Server.

3. PTY: contains the Payload Type as per Section 4.3, which indicates the content and format of the Payload, to process it properly.

4.1 Header Types

Many Header encodings are possible in order to spare space and transmission time. The possible Header Types are numbered starting from 0 and shortened as H0, H1 etc. Header Types $\in [0, 6]$ are defined in the Figures 4, 5, 6, 7, 8, 9, 10. Header Types $\in [7, 31]$ are reserved for future use (RFU). Any custom, implementation-defined Headers can be indicated with numbers ≥ 32.

Header 0

`uint8`	`uint8`	`uint8`
8 bit	8 bit	8 bit
GID	*SID*	PTY

Figure 4: *Header 0* (H0): the most explicit and largest in amounts of Groups and Parties, but also the least efficient. Occupies 3 bytes, allowing up to 256 Groups and up to 256 Parties (Session Server included).

Header 1

`uint8`	`uint8`	
8 bit	MS 5 bit	LS 3 bit
GID	*SID*	PTY

Figure 5: *Header 1* (H1): occupies 2 bytes, allowing up to 256 Groups and up to 32 Parties (Session Server included).

Header 2

uint8	uint8	
8 bit	MS 5 bit	LS 3 bit
SID	*GID*	*PTY*

Figure 6: *Header 2* (H2): occupies 2 bytes, allowing up to 32 Groups and up to 256 Parties (Session Server included).

Header 3

uint8		
MS 3 bit	2 bit	LS 3 bit
GID	*SID*	*PTY*

Figure 7: *Header 3* (H3): occupies 1 byte, allowing up to 8 Groups and up to 4 Parties (Session Server included).

Header 4

uint8		
MS 3 bit	2 bit	LS 3 bit
SID	*GID*	*PTY*

Figure 8: *Header 4* (H4): occupies 1 byte, allowing up to 4 Groups and up to 8 Parties (Session Server included).

Header 5

uint8	uint8
8 bit	8 bit
SID	*PTY*

Figure 9: *Header 5* (H5): when all Parties belong only to a single, broadcasting Group (which is equivalent to saying that there are no Groups at all), the *GID* field becomes redundant, so it can be removed. Occupies 2 bytes, allowing only 1 Group and up to 256 Parties (Session Server included).

Header 6

uint8	
MS 5 bit	LS 3 bit
SID	*PTY*

Figure 10: *Header 6* (H6): like for Header 5, it has no *GID* field as a single, broadcasting Group is assumed. Occupies 1 byte, allowing only 1 Group and up to 32 Parties (Session Server included).

4.2 Physical location of CBS Header within the CAN FD frame

CBS does not enforce a specific physical location of the Header within a CAN FD frame to better fit the needs of a closed, static system. This means that some or all of the Header fields may be encoded within the CAN ID instead of being placed into the CAN FD payload (prepended to the CBS Payload), which is otherwise the default choice. Such default is chosen as the most portable, completely independent of the CAN ID, effectively decoupling the CAN FD data-link layer from the CBS layer. The possibilities are also depicted in Table 1.

Encoding (parts of) the CBS Header into the CAN ID can free some valuable space in the CAN FD payload to carry more application data and at the same time leverage the hardware-based filtering that some CAN transceivers implement to ignore specific CAN IDs. Such an encoding is up to the implementation.

Because the CBS Header may be encoded in the CAN ID and it may have different sizes as per the previous Section, its size is not included in the format and validity checks performed upon receiving any CBS message. It is assumed that the CBS implementation performs the en-/decoding of the CBS Header to/from the proper location.

It is required that all Parties on the bus use the same way of encoding of the Header.

Possibility 1	Full CBS Header	CBS Payload	
Possibility 2	CBS Header part 1	CBS Header part 2	CBS Payload
Possibility 3	...	Full CBS Header	CBS Payload
CAN frame	CAN ID	CAN FD payload	

Table 1: Possibilities for the physical position of the CBS Header: completely encoded in the CAN ID, partially encoded in the CAN ID and partially within the CAN FD payload or completely included in the CAN FD payload, prepended to the CBS Payload.

Some examples of how the CAN ID is used to encode the CBS Header:

- Assuming many large Groups, a Header 0 (H0) could be encoded in the middle of a 29 bit ID, leaving the upper bits for priority indication (to override the implicit priority of the rest of the ID during arbitration) and the lowest bits to indicate the type of application-level data carried within the CBS Payload. The latter could be useful to leverage the hardware filtering for messages of interest. Depicted in Figure 11.

29	24	16	8	0
	p pppp	gggg gggg	ssss ssss	ttta aaaa

Figure 11: Example encoding of a Header Type 0 (H0) into a 29 bit CAN ID: `p` are Priority bits, `g` are the GID bits, `s` are the SID bits, `t` the PTY bits and `a` contain optional application-level metadata.

- Assuming fewer small Groups, the same thing could be done in an 11 bit ID using a smaller Header, like H4. Depicted in Figure 12.

```
11 |  8 |         0 |
   | pgg| ssst ttaa |
```

Figure 12: Example encoding of a Header Type 4 (H4) into an 11 bit CAN ID: `p` is a Priority bit, `g` are the GID bits, `s` are the SID bits, `t` the PTY bits and `a` contain optional application-level metadata.

- A completely different approach would interpret the CAN ID (or part of it) as an integer enumeration of all possible message from all possible Groups, Sources and Payload Types. This could also include application message types, which indicate what the content of the application data in secured messages is about. All Parties ought to have have a bijective map (e.g. lookup table or conversion function) to translate from value to meaning and vice versa. Depicted in Table 2.

CAN ID	*GID*	*SID*	*PTY*	Appl. msg. type
0	0	0	REN	N/A
1	0	0	RES	N/A
2	0	0	REQ	N/A
3	0	0	SADFD	Status request
4	0	0	SADFD	Status response
5	0	1	REN	N/A
⋮	⋮	⋮	⋮	⋮
n	3	12	SADFD	Configuration data

Table 2: Example mapping between a numeric CAN ID and all fields of the CBS Header, along with the application message type determining the category of the content within Secured Application Data messages. Messages of the CBS layer that do not carry application data don't use this field (indicated as "N/A").

4.3 Payload Types

The *Payload Type* is an identifier of the format and content of each CBS message, used to know how to process the received messages.

The Payload Types, their shortenings/acronyms and encoding values are:

- Session Renewal Notification (REN) := 0. Format described in Section 5.7.3.
- Response (RES) := 1. Format described in Section 5.3.2.
- Request (REQ) := 2. Format described in Section 5.2.2.
- Secured Application Data over Transport Protocol (SADTP) := 3. Format described in Section 5.5.4.

- Secured Application Data over CAN FD (SADFD) $:=$ 4. Format described in Section 5.5.3.
- Unsecured Application Data (UAD) $:=$ 5. Format described in Section 5.5.2.
- Values 6 and 7 are reserved for future use (RFU).

Rationale: The Payload Type values are chosen to fit within 3 bit and are sorted to have a decent implicit priority for the CAN arbitration (low value = high priority), if they are encoded in the CAN ID (see Section 4.2), with the idea that:

- security-enabling messages are more important than application data messages, otherwise said data cannot be secured (REN, RES, REQ > others),
- transport-protocol messages take priority over single-frames to avoid interleaved messages mid-stream of fragments (TP > FD),
- Secure Application Data messages are more important than unprotected ones (SE > UN).

5 Protocol timeline

In this section the actions of the Session Server and of each Client, the messages they send and the Parties' behaviour are described in chronological order of runtime appearance, from the initialisation of the CBS implementation (e.g. at device boot) onwards.

Each described step refers to a generic Group G and has thus to be performed *for each preconfigured Group* in the Session Server and in each Client.

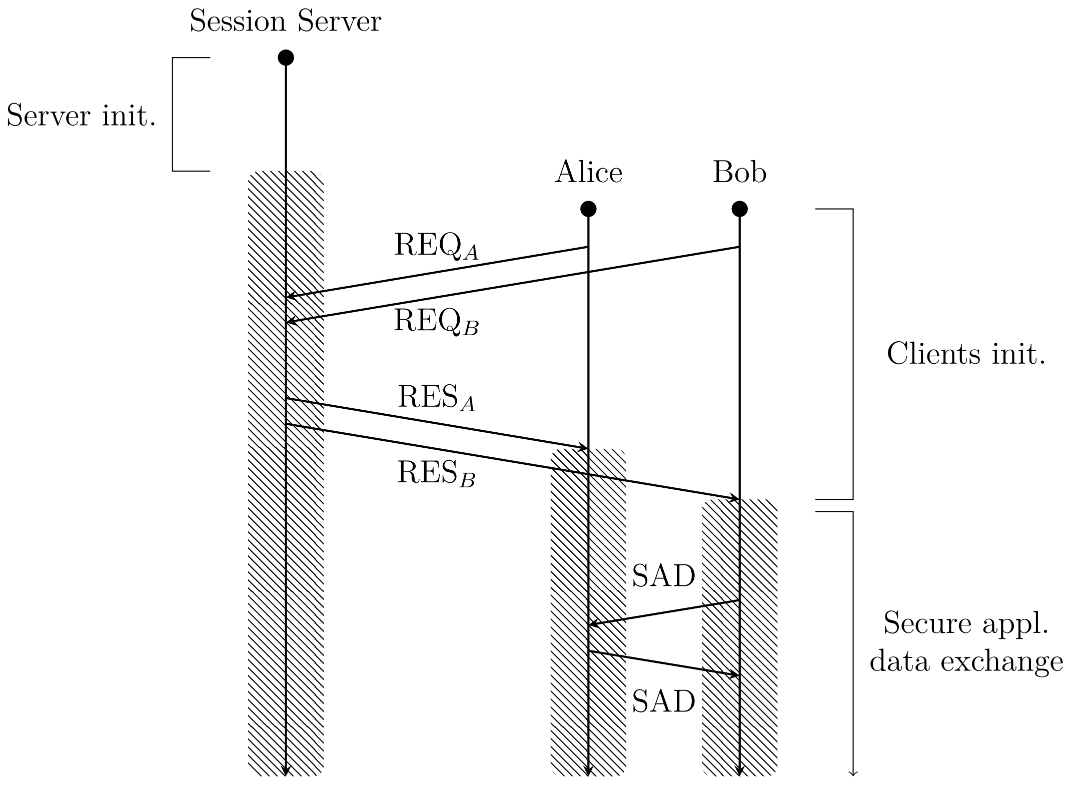


Figure 13: Timeline of the initialisation of Session Server and the Clients, including the initial Request and Response messages and the following secured communication of the Clients. The black dots indicate the start of the CBS initialisation of the Party (e.g. at device boot). The patterned areas with diagonal lines indicate the time interval when the Party possesses the current STK_G, N_G^{ctr} and is thus able to communicate securely with others. Not to scale.

5.1 Server initialisation

This is the behaviour that happens during the initialisation of the CBS implementation on the Session Server, e.g. at device boot. The Session Server generates a new, fresh, non-zero STK of 128 bit for each Group in its configuration using the true-random number generator[7]. Simultaneously it also stores the timestamp of when said STK was generated,

[7]For the generation of many STKs in a row for performance reasons, a CSPRNG could be used instead of a TRNG, as long as properly seeded with a secret, non-zero, true-random value.

to later know when it expires, and resets the Counter Nonce of the Group to 0. The same timestamp of STK generation is also used to initialise the m_G value, i.e. the timestamp of the last valid message received:

$$\begin{aligned} STK_G &\coloneqq \texttt{TRNG}(128\,\text{bit}) \neq 0, \forall G \\ s_G &\coloneqq \texttt{currenttime}(), \forall G \\ m_G &\coloneqq s_G, \forall G \\ N_G^{ctr} &\coloneqq 0, \forall G \end{aligned}$$

After that, the Session Server starts listening for incoming Request messages from Clients, to which it will reply with Response messages.

At the same time, the Session Server is already able to transmit Application Data messages to other Clients, both secured and unsecured, noting that Clients are not yet able to receive the secured ones. As m_G is going to be updated upon the successful processing of a Request, as long as $m_G = s_G$ for a Group G, no Client in that Group has yet obtained the STK_G, N_G^{ctr} values to be able to validate and decrypt a Secured Application Data message[8].

5.2 Client initialisation

This is the behaviour that happens during the initialisation of the CBS implementation on the Client, e.g. at device boot. At this point in time, the Client has no STKs and is thus unable to transmit Secured Application Data messages. Unsecured ones on the other hand are available, because they are STK-independent.

For each Group[9], the Client performs the Request generation and its transmission as follows to obtain its STK_G, N_G^{ctr} from the Session Server.

5.2.1 Request generation

For the given Group G a unique, fresh, non-zero, true-random number N_G^{req} called the *Request Nonce*, is generated and stored in local volatile memory. This number will be used to verify the freshness of each expected Response message from the Session Server. N_G^{req} is then used to build a Request message. After *building* such message but *before transmitting* it, the current timestamp is also stored in volatile memory, later used to check whether a Response has been received before the timeout:

[8]The CBS implementation may decide to either completely prevent the transmission of Secured Application Data messages until $m_G \neq s_G$, as no Client could process them, or allow them and handle any potential reply timeout in the application using CBS.

[9]To avoid a high computing and bus load spike during the initialisation, the Client may decide to postpone the Request generation for some Groups that are not immediately needed. The Request Nonces N_G^{req} of said subset would thus remain set to zeros to indicate that no Response is expected.

$$r_G := \texttt{currenttime}()$$
$$N_G^{req} := \texttt{TRNG}(64\,\text{bit}) \neq 0$$

5.2.2 Request message format

The fields of the message are defined in Figure 14, including their data types, names and values.

REQ-Header			REQ-Payload	
GID	*SID*	*PTY*	`uint64` *reqnonce*	`uint8[16]` *tag*
id_G	id^A	REQ	N_G^{req}	(see note below)

Figure 14: Diagram of a Request message.

The value of the *tag* field is the keyed, labelled hash of the Header and *reqnonce* fields:

$$tag := \texttt{Hash}(LTK_{AS}||m_{req}||GID||SID||PTY||reqnonce, 16\,\text{B})$$

The Label m_{req} is the constant `uint8[11]` array:

$$m_{req} := (99, 98, 115, 95, 114, 101, 113, 117, 101, 115, 116)$$

being the ASCII encoding of the string `cbs_request` without null-terminations. The *GID*, *SID* and *PTY* fields in the concatenated hashing input are each encoded into a `uint8`. If no *GID* field is present in the Header, it's assumed to have the value 0.

5.3 Server processing Request

Upon receiving a Request message, the Session Server must validate it. If valid, a Response message is generated and transmitted.

5.3.1 Request message validation

1. The *GID* field, if present, must indicate a known Group.
2. The *PTY* field must indicate a valid type (in this case REQ).
3. The *SID* field must not be 0 (thus not being the Session Server): $id^A \neq 0$. Otherwise, the CBS implementation must provide a MFM-warning to the application.
4. The (*GID*, *SID*) pair must be valid, i.e. the requesting Client must belong to the Group indicated by *GID*. Otherwise, the CBS implementation must provide a NIG-warning to the application.

5. The REQ-Payload must have length ≥ 24 B (sizes of *reqnonce* + *tag* fields). Longer Payloads are allowed because the additional bytes may be used by future CBS versions.

6. The *reqnonce* field must not be 0: $N_G^{req} \neq 0$

7. The *tag* must be valid. Otherwise, the CBS implementation must provide a INV-warning to the application. This check effectively makes the Session Server discard Requests that are spoofed or not intact. Replayed Requests are still accepted (see Section 6.3).

In case any of these checks fails, the message is discarded and the warning, if any, is provided to the application. Otherwise the Session Server:

1. Saves the instant of reception of the Request:

$$m_G := \texttt{currenttime()} \begin{cases} s_G + 1 & \text{if } \texttt{currenttime()} = s_G \\ \texttt{currenttime()} & \text{otherwise} \end{cases}$$

   This will ensure $m_G \neq s_G$ even for Requests being received within the same millisecond from the Session start s_G, indicating to the Session Server that at least one Client in the Group G has requested and obtained STK_G, N_G^{ctr}. In other words, in the Group G there is now at least one Client able to successfully receive and process Secured Application Data messages.

2. Builds a Response message as per the next section.

3. Transmits said Response message.

5.3.2 Response message format

A represents the Client in the Group G that sent the Request to which the Session Server is replying with this message. The fields of the message are defined in Figure 15, including their data types, names and values.

RES-Header

GID	*SID*	*PTY*
id_G	$id^S = 0$	RES

RES-Payload

`uint8` *client*	`uint24` *ctrnonce*	`uint64` *resnonce*	`uint8[16]` *ctext*	`uint8[16]` *tag*
id^A	N_G^{ctr}	N_G^{res}	(see note below)	

Figure 15: Diagram of a Response message.

The *GID* field contains the same Group Identifier as in the Request id_G, the *client* field contains the Source Identifier of the Client id^A that sent the Request, the *resnonce* field contains a freshly generated, true-random value $N_G^{res} \coloneqq \texttt{TRNG}(64\,\mathrm{bit})$, the values of *ctext* and *tag* fields are the two outputs of the same `AEAD` function call, which authenticates the Header, *client* and *ctrnonce* fields, using both the *reqnonce* and *resnonce* field as AEAD-nonce N_{aead}. The plaintext is the STK, which is encrypted into the *ctext* field:

$$\begin{aligned}
ctext, tag &\coloneqq \texttt{AEAD}(LTK_{AS}, N_{aead}, ad, pt, 16\,\mathrm{B}) \\
N_{aead} &\coloneqq N_G^{req} || N_G^{res} \\
ad &\coloneqq m_{res} || GID || SID || PTY || client || ctrnonce \\
pt &\coloneqq STK_G
\end{aligned}$$

The Label m_{res} is the constant `uint8[12]` array:

$$m_{res} \coloneqq (99, 98, 115, 95, 114, 101, 115, 112, 111, 110, 115, 101)$$

being the ASCII encoding of the string `cbs_response` without null-terminations. The *GID*, *SID*, PTY and id^A fields in the concatenated associated data are each encoded into a `uint8`. If no *GID* field is present in the Header, it's assumed to have the value 0.

The Counter Nonce N_G^{ctr} is **not incremented** after building the Response message, as it would happen for other messages transmitted by the Session Server.

5.4 Client processing Response

Upon receiving a Response message, the Client A must validate it. If valid, the obtained STK is used for Secured Application Data messages.

5.4.1 Response message validation

1. The *GID* field, if present, must indicate a known Group.
2. The PTY field must indicate a valid type (in this case RES).
3. The *SID* field must be 0 (being the Session Server), thus $id^S = 0$. Otherwise, the CBS implementation must provide a SOM-warning to the application.
4. The RES-Payload must have length ≥ 44 B (sizes of *client* + *ctrnonce* + *resnonce* + *ctext* + *tag* fields). Longer Payloads are allowed because additional bytes may be used by future CBS versions.
5. The *client* field must be equal to the identity of the Client validating the message. This check effectively makes Clients, that are currently expecting a Response, discard Responses aimed at other Clients.

6. The Client must be expecting a Response for the Group G. This can be verified by checking that the Request Nonce kept in the Client's memory for the Group G is non zero:
$$N_G^{req} \neq 0$$
Otherwise, the CBS implementation must provide a NER-warning to the application. This check effectively makes Clients discard Responses when they were not expecting any.

7. The Response must be received within a pre-configured timeout t_{reqres} from the saved timestamp r_G. This can be verified by checking that the inequality holds:
$$\texttt{timedelta}(r_G, \texttt{currenttime}()) \leq t_{reqres}$$
Otherwise, the CBS implementation must provide a RTO-warning to the application.

8. The received Counter Nonce must be smaller than its maximum value, according to its data type:
$$\textit{ctrnonce} < \max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$
where the *ctrnonce* is the field of the received message. Otherwise, the CBS implementation must provide a RON-warning to the application.

9. The *tag* must be valid. Otherwise, the CBS implementation must provide a INV-warning to the application. This check effectively makes Clients discard Responses that are spoofed, not intact or not constructed using the N_G^{req}, which indicates they are not a Response to the transmitted Request.

10. The STK_G, decrypted from *ctext*, must be non-zero. Otherwise, the CBS implementation must provide a RZK-warning to the application.

In case any of these checks fails, the message is discarded and the warning, if any, is provided to the application. Otherwise, in case of a valid Response, the Client:

1. Sets the Request Nonce variable to zero, to indicate that no Response is currently expected and saves the instant of reception of the Response. The same timestamp is also used to initialise the m_G value, i.e. the timestamp of the last valid message received, used then to compute the Counter Nonce validity delay with the `ctrdelay` function:
$$\begin{aligned} N_G^{req} &:= 0 \\ r_G &:= \texttt{currenttime}() \\ m_G &:= r_G \end{aligned}$$

2. Sets the short term key STK_G and Counter Nonce N_G^{ctr} to the ones decrypted/taken from the Response message. The Counter Nonce value is just copied, not incremented, as it represents the *next* Counter Nonce that should appear on the bus.

The Client is now fully enabled to transmit Secured Application Data messages within the Group G.

5.4.2 Waiting for a Response

While waiting for a Response or the timeout t_{reqres}, no new Request must be transmitted for the same Group; multiple concurrent Requests for different Groups are allowed. A local non-zero Request Nonce $N_G^{req} \neq 0$ variable indicates that a Response is currently expected for the group G. After said timeout and no valid Response received, a new Request may be transmitted, but the decision whether to do so is left to the application. Some example decisions are: immediately retry with a new Request up to n tries; wait first for some time and then retry with a new Request; fallback to Unsecured Application Data messages; issue a warning to a human operator; fallback to a custom solution; a combination of the previous points.

5.5 Application Data transmission

The CBS protocol offers 3 variants of messages for transferring application data:

- Unsecured, where there data is in plaintext without any protection.

  Unsecured Application Data messages may be transmitted *at any point in time* by any Party. This includes the Session Server and all Clients across every Group, even before the first Request is transmitted or while waiting for a Response. This is possible because the Unsecured version is not dependent on Counter Nonces or STKs. Unsecured Application Data messages are useful:

  – as a fallback mechanism in case the Session Server is offline,
  – if support to legacy systems is required,
  – for benchmarking or debugging purposes during the system development,
  – to transport data which is already secured by the application itself without additional cryptographic overhead,
  – to enable communication between Parties that don't have any Groups in common[10].

- Secured over CAN FD, where the data is encrypted, authenticated and fresh but also limited in size, as the message is optimised to fit within one CAN FD frame.

- Secured over Transport Protocol (TP), where the data is encrypted, authenticated, fresh and with larger Payloads, as the message is ready to be transferred over any

[10]This is only recommended to reduce the size of the configuration data structure of each Part if and only if the transferred data is already secured by the application. Otherwise a common Group should be added to the configuration of the communicating Parties and Secured Application Data messages should be used instead.

transport layer on top of CAN or CAN FD that handles fragmentation and reassembly of long messages. Because many transport protocols exists, the examples in this document will take ISO-TP [6] as a reference.

Any Party, including the Session Server, is allowed to transmit Application Data messages.

The transmission of Secured Application Data messages is allowed at any point in time, when the transmitting Party has the Session information to do so, but it is discouraged in situations where all other Parties on the bus lack said Session information, thus making any receivers unable to validate and decrypt the message properly. This is namely the case for the Session Server when it just generated a new STK (on initialisation or Session renewal) but has not yet distributed said STK to not even one Client.

5.5.1 Incrementing Counter Nonces

The Counter Nonce N_G^{ctr} is used by all Parties (all Clients and the Session Server) in all Secured Application Data messages to prove the freshness of the message.

After building but *before attempting transmission* any Secured Application Data message, the local variable N_G^{ctr} is incremented by 1, so that any following new message is built with a new nonce. It is critical that the increment happens before transmission and regardless of transmission success, to avoid any potential issues where the transmission routine reports an error and but the message is actually transmitted or vice-versa. By incrementing before transmitting, no Counter Nonce is transmitted twice. Counter Nonces are further updated upon receiving a valid Secured Application Data message, as indicated in Section 5.6.2.

Before the Counter Nonce reaches the upper limit of its variable's data type:

$$\max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$

the Session Server will renew the Session and reset the Counter Nonce. In any case, the Client must never reuse the same Counter Nonce twice with the same STK. In practice, this means that once the variable reaches its upper limit, Secured Application Data messages must not be transmitted any more, the Counter Nonce is not incremented any more and the Client should transmit a new Request message to obtain the current Session information again, just to be sure there are no synchronisation issues between the Client and the Session Server.

5.5.2 Unsecured Application Data message format

The fields of the message are defined in Figure 16, including their data types, names and values.

The max *plaintext* length varies depending on where the CBS Header fields are physically located and whether the message is carried over CAN FD or a transport layer. For CAN FD the maximum varies within:

UAD-Header			UAD-Payload
GID	*SID*	*PTY*	`uint8[*]` *plaintext*
id_G	id^A	UAD	Arbitrary bytes

Figure 16: Diagram of an Unsecured Application Data message

- 61 B: when the largest Header H0 is placed completely in the CAN FD payload;
- 64 B: when the whole Header is encoded in the CAN ID.

In case the message navigates over a transport layer, the max *plaintext* length also varies depending on the chosen transport protocol. For ISO-TP the maximum varies within:

- $(2^{32}-4)$ B: when the largest Header H0 is placed completely in the ISO-TP payload;
- $(2^{32}-1)$ B: when the whole Header is encoded in the CAN ID.

5.5.3 Secured Application Data over CAN FD message format

This message is optimised to fit within a CAN FD frame. The fields of the message are defined in Figure 17, including their data types, names and values.

SADFD-Header

GID	*SID*	*PTY*
id_G	id^A	SADFD

SADFD-Payload

`uint24` *ctrnonce*	`uint2` *rfu*	`uint6` *ptlen*	`uint8[`*ctlen*`]` *ctext*	`uint8[8]` *tag*
N_G^{ctr}	0	$[0, 52]$	(see note below)	

Figure 17: Diagram of a Secured Application Data over CAN FD message.

The values of *ctext* and *tag* fields are the two outputs of the same `AEAD` function call, which authenticates the Header and *ptlen* field, using the *ctrnonce* field to build the AEAD-nonce N_{aead}. The plaintext is arbitrary data from the application, which is encrypted into the *ctext* field:

$$\begin{aligned} ctext, tag &:= \texttt{AEAD}(STK_G, N_{aead}, ad, pt, 8\,\mathrm{B}) \\ N_{aead} &:= N_G^{ctr} || GID || SID || \overbrace{0 \ldots 0}^{88\,\mathrm{bit}} \\ ad &:= m_{sadfd} || GID || SID || PTY || ptlen \\ pt &:= \text{arbitrary } ptlen \text{ bytes} \end{aligned}$$

The Label m_{sadfd} is the constant `uint8[14]` array:

$$m_{sadfd} \coloneqq (99, 98, 115, 95, 115, 101, 99, 117, 114, 101, 100, 95, 102, 100)$$

being the ASCII encoding of the string `cbs_secured_fd` without null-terminations. The *GID*, *SID* and *PTY* fields in the concatenated associated data are each encoded into a `uint8`. If no *GID* field is present in the Header, it's assumed to have the value 0.

ctlen represents the ciphertext length in bytes, while *ptlen* contains the plaintext length in bytes. They are equal for Ascon128[11]. The max *ctlen* varies depending on where the Header fields are physically located and is between:

- 49 B: when the largest Header H0 is placed completely in the CAN FD payload[12];
- 52 B: when the whole Header is encoded in the CAN ID.

The *rfu* field (Reserved for Future Usage) is unused and its bits are set to 0.

5.5.4 Secured Application Data over Transport Protocol message format

This message is optimised to carry large secured payloads, because the whole message is meant to be fragmented and reassembled by a transport layer existing between the CBS layer and the data-link layer (CAN or CAN FD). Two differences to notice compared to the Secure Application Data over CAN FD: a larger *ptlen* field for larger Payloads and a larger *tag* field for increased security, as more space is now available. The fields of the message are defined in Figure 18, including their data types, names and values.

SADTP-Header

GID	*SID*	*PTY*
id_G	id^A	SADTP

SADTP-Payload

`uint24` *ctrnonce*	`uint32` *ptlen*	`uint8[`*ctlen*`]` *ctext*	`uint8[16]` *tag*
N_G^{ctr}	$[0, 2^{32} - 26]$	(see note below)	

Figure 18: Diagram of a Secured Application Data over Transport Protocol message.

The values of *ctext* and *tag* fields are the two outputs of the same `AEAD` function call, which authenticates the Header and *ptlen* field, using the *ctrnonce* field to build the AEAD-nonce. The plaintext is arbitrary data from the application, which is encrypted into the *ctext* field:

[11]If a different cipher is chosen as per Appendix A, then $ctlen \geq ptlen$ due to padding, e.g. for block ciphers, but it's easy for the receiving Party to reconstruct *ctlen* from *ptlen*.

[12]This is a design choice to enable a SADFD message with the worst overhead from the CBS Header to still carry 3 blocks of a 128 bit block cipher (48 B in total) such as AES-128, in case a different `AEAD` implementation is chosen as per Appendix A.

$$\begin{aligned} ctext, tag &\coloneqq \texttt{AEAD}(STK_G, N_{aead}, ad, pt, 16\,\mathrm{B}) \\ N_{aead} &\coloneqq N_G^{ctr}||GID||SID||\overbrace{0\ldots0}^{88\,\mathrm{bit}} \\ ad &\coloneqq m_{sadtp}||GID||SID||PTY||ptlen \\ pt &\coloneqq \text{arbitrary } ptlen \text{ bytes} \end{aligned}$$

The Label m_{sadtp} is the constant `uint8[14]` array:

$$m_{sadtp} \coloneqq (99, 98, 115, 95, 115, 101, 99, 117, 114, 101, 100, 95, 116, 112)$$

being the ASCII encoding of the string `cbs_secured_tp` without null-terminations. The *GID*, *SID* and PTY fields in the concatenated associated data are each encoded into a `uint8`. If no *GID* field is present in the Header, it's assumed to have the value 0.

ctlen represents the ciphertext length in bytes, while *ptlen* contains the plaintext length in bytes. They are equal for Ascon128[13]. The max message size is depending on the chosen transport layer. The max *ctlen* varies depending on where the Header fields are physically located; this maximum for ISO-TP varies between:

- $(2^{32}-26)$ B: when the largest Header H0 is placed completely in the ISO-TP payload;
- $(2^{32}-29)$ B: when the whole Header is encoded in the CAN ID.

5.6 Application Data reception

Upon receiving any Application Data message, the receiving Party B must validate it, based on its message type.

5.6.1 Unsecured Application Data validation

1. The *GID* field, if present, *should* indicate a known Group, but this is not a requirement, as UAD messages may be exchanged between Parties that don't have any Groups in common. The receiving Party may decide freely whether to enforce this rule or not.
2. The PTY field must indicate a valid type (in this case UAD).
3. The *SID* field must be different than the validating Party's Source Identifier id^B. Otherwise, the CBS implementation must provide a MFM-warning to the application.

[13]If a different cipher is chosen as per Appendix A, then *ctlen* $\geq$ *ptlen* due to padding, e.g. for block ciphers, but it's easy for the receiving Party to reconstruct *ctlen* from *ptlen*.

In case any of these checks fails, the message is discarded and the warning, if any, is provided to the application. Otherwise the plaintext is passed as-is to the application.

5.6.2 Secured Application Data validation

These checks are valid for both Secured Application Data messages over CAN FD and Transport Protocol.

1. The *GID* field, if present, must indicate a known Group.
2. The PTY field must indicate a valid type (in this case SADFD or SADTP).
3. The *SID* field must be different than the validating Party's Source Identifier id^B. Otherwise, the CBS implementation must provide a MFM-warning to the application.
4. The SADFD-Payload must have length $\geq 12 + \textit{ptlen}$ B (sizes of *ctrnonce* + *rfu* + *ptlen* + *tag* fields plus the value of the *ptlen* field), while the SADTP-Payload must have length $\geq 23 + \textit{ptlen}$ B. Longer Payloads are allowed because the additional bytes may be used by future CBS versions.
5. The *ptlen* field must have a value in the allowed ranges (respectively for SADFD- and SADTP-Payloads), depending on where the Header fields are physically located. This is mostly a standard memory safety check to avoiding reading more data than the underlying CAN FD or Transport Protocol could carry.
6. The validating Party must have a non-zero STK_G, thus be enabled to validate and decrypt the message.
7. The received Counter Nonce must be smaller than its maximum value, according to its data type:
$$\textit{ctrnonce} < \max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$
where the *ctrnonce* is the field of the received message.
8. The validating Party's local variable of the Counter Nonce must be smaller than its maximum value, according to its data type:
$$N_G^{ctr} < \max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$
In other words, once the Counter Nonce reaches the maximum, every secured message is discarded. The Session should anyway be renewed way before such limit from the Session Server.
9. The received Counter Nonce must be fresh. This means that the following expression must be true:
$$\textit{ctrnonce} \geq N_G^{ctr} - \texttt{ctrdelay}(m_G, \texttt{currenttime}(), S_G^{max}, D_G^{max})$$

where the *ctrnonce* is the field of the received message, N_G^{ctr} is the local Counter Nonce variable of the validating Party, S_G^{max}, D_G^{max} are taken from the validating Party's configuration and m_G from its current state. Otherwise, the CBS implementation must provide a OLD-warning to the application. This check effectively drops the messages that are not fresh.

10. The *tag* must be valid. Otherwise, the CBS implementation must provide a INV-warning to the application.

In case any of these checks fails[14], the message is discarded[15] and the warning, if any, is provided to the application. Otherwise the receiving Party:

1. Saves the timestamp of the received valid message. On the Session Server:

$$m_G \coloneqq \texttt{currenttime}() \begin{cases} s_G + 1 & \text{if } \texttt{currenttime}() = s_G \\ \texttt{currenttime}() & \text{otherwise} \end{cases}$$

   This ensures $m_G \neq s_G$ for the Session Server, just like in Section 5.3. On the Clients:

$$m_G \coloneqq \texttt{currenttime}()$$

2. Increments the local Counter Nonce variable considering the largest between the received and the local variable:

$$N_G^{ctr} \coloneqq max\{N_G^{ctr}, ctrnonce\} + 1$$

3. Passes the decrypted plaintext[16] without any padding to the application.

5.6.3 Counter Nonce synchronisation and acceptance

Secured Application Data messages carry the Counter Nonce value the transmitting party has. The Counter Nonce is used as proof of freshness of the message: it is incremented at every transmitted and received Secured Application Data and Session Renewal Notification message, effectively keeping all Parties within the same Group in sync about the value of such nonce. There are a few corner cases that prevent the ideal syncing of the value:

[14]When over a TP, all checks except for the *tag* validity can be performed after the reception of only a few fragments (maybe even after just the first, depending on the transport protocol). In case of invalidity, the remaining incoming fragments may be discarded immediately and, assuming the transport protocol supports it, the sender may be notified to stop the transmission of subsequent fragments to reduce the bus load.

[15]If many Secured Application Data messages from the same Group are being discarded, the application may decide to request the current STK and Counter Nonce to the Session Server again, just to avoid possible synchronisation issues.

[16]When over a TP, to avoid large memory buffers, the implementation may also opt to pass the plaintext fragment-by-fragment to the application as they are being received with the notice that the *tag* validation is deferred until the reception of the very last fragment.

- A Party may be busy performing intensive tasks and transmitting messages without inspecting the reception message queue, where messages with newer (greater) Counter Nonces can be found.
- A Party's transmission loses the CAN arbitration in case of message collision. The CAN peripheral transmits the message in the first available slot after the winning message. Assuming the winning and losing message had the same Counter Nonce value, the second (losing) carries an already old value; a duplicate of the winning when it should be greater by 1.

To avoid messages being lost, not just the latest Counter Nonce is accepted, but also a few older ones, specifically any value such that:

$$\geq N_G^{ctr} - \texttt{ctrdelay}(m_G, \texttt{currenttime}(), S_G^{max}, D_G^{max})$$

where N_G^{ctr} is the local Counter Nonce variable of the validating Party, S_G^{max}, D_G^{max} are taken from the validating Party's configuration and m_G from its current state. The Party-local variable N_G^{ctr} represents the Counter Nonce expected to appear in the next received Secured Application Data or Session Renewal Notification message, while `ctrdelay` represents the tolerance for older Counter Nonces.

A large of `ctrdelay` leads to a more messages being accepted but leaving a large replay window for an attacker. A small one may lead to some messages to be discarded as too old in case oh high congestion of the bus. This is also affected by D_G^{max}, S_G^{max}. In particular configuring[17] $D_G^{max} = 0$ reduces the acceptance interval to just the next expected Counter Nonce and would not accept a message that loses the arbitration from the example above.

The Figure 19 shows how the result of the `ctrdelay` function changes over time as new valid messages are received.

5.7 Server renewing session

A Session and thus its STK expire in time and in amount uses, i.e. total amount of messages sent with the same STK, whichever comes first. The expiration and following renewal of a Session are handled solely by the Session Server. During the Session renewal phase, a new STK is generated and distributed, but the old STK is kept available for a bit longer, to allow any pending messages secured with the old STK to be still accepted.

5.7.1 Expiration criteria

1. The Session expires when the Counter Nonce reaches the preconfigured upper limit N_G^{exp}:

$$N_G^{ctr} \geq N_G^{exp}$$

[17]The implementation may also decide to alter D_G^{max}, S_G^{max} dynamically at runtime based on the current bus load.

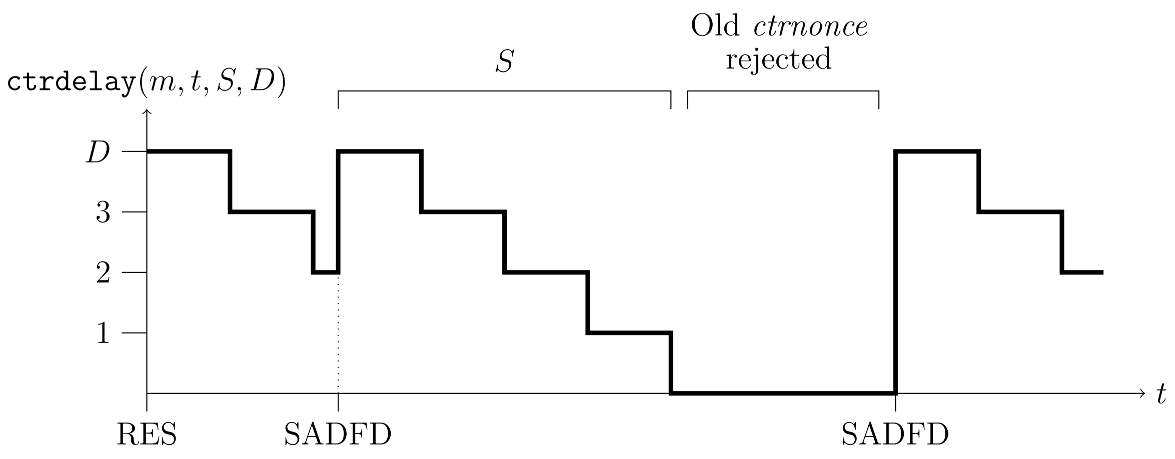


Figure 19: Plot of how the result of the `ctrdelay` function changes over time as new valid messages are received, indicated on the X-axis, which refresh the value of m to that instant. In this example $D = 4$ and S is a fixed positive constant.

2. The Session expires when more than s_G^{exp} time has passed since the Session start s_G:

$$\texttt{timedelta}(s_G, \texttt{currenttime}()) > s_G^{exp}$$

Note that different Groups may have different validity periods (different N_G^{exp} or s_G^{exp}).

5.7.2 Server-side Session renewal phase

During this phase the Session Server starts a new Session just like during the initialisation from Section 5.1, while temporarily keeping also the information of the old Session active. Let G be the generic Group that had just expired.

1. The Session Server copies the expired STK, Counter Nonce and Message Timer into temporary variables:

$$\begin{aligned} STK_G^{old} &:= STK_G \\ N_G^{ctr,old} &:= N_G^{ctr} \\ m_G^{old} &:= m_G \end{aligned}$$

2. Then it takes the same steps as during the initialisation phase to create a new Session:

$$\begin{aligned} STK_G &:= \texttt{TRNG}(128\,\text{bit}) \neq 0 \\ s_G &:= \texttt{currenttime}() \\ m_G &:= s_G \\ N_G^{ctr} &:= 0 \end{aligned}$$

From now on, $STK_G^{old}, N_G^{ctr,old}$ are not delivered to Clients within Response messages any more, STK_G, N_G^{ctr} are given instead.

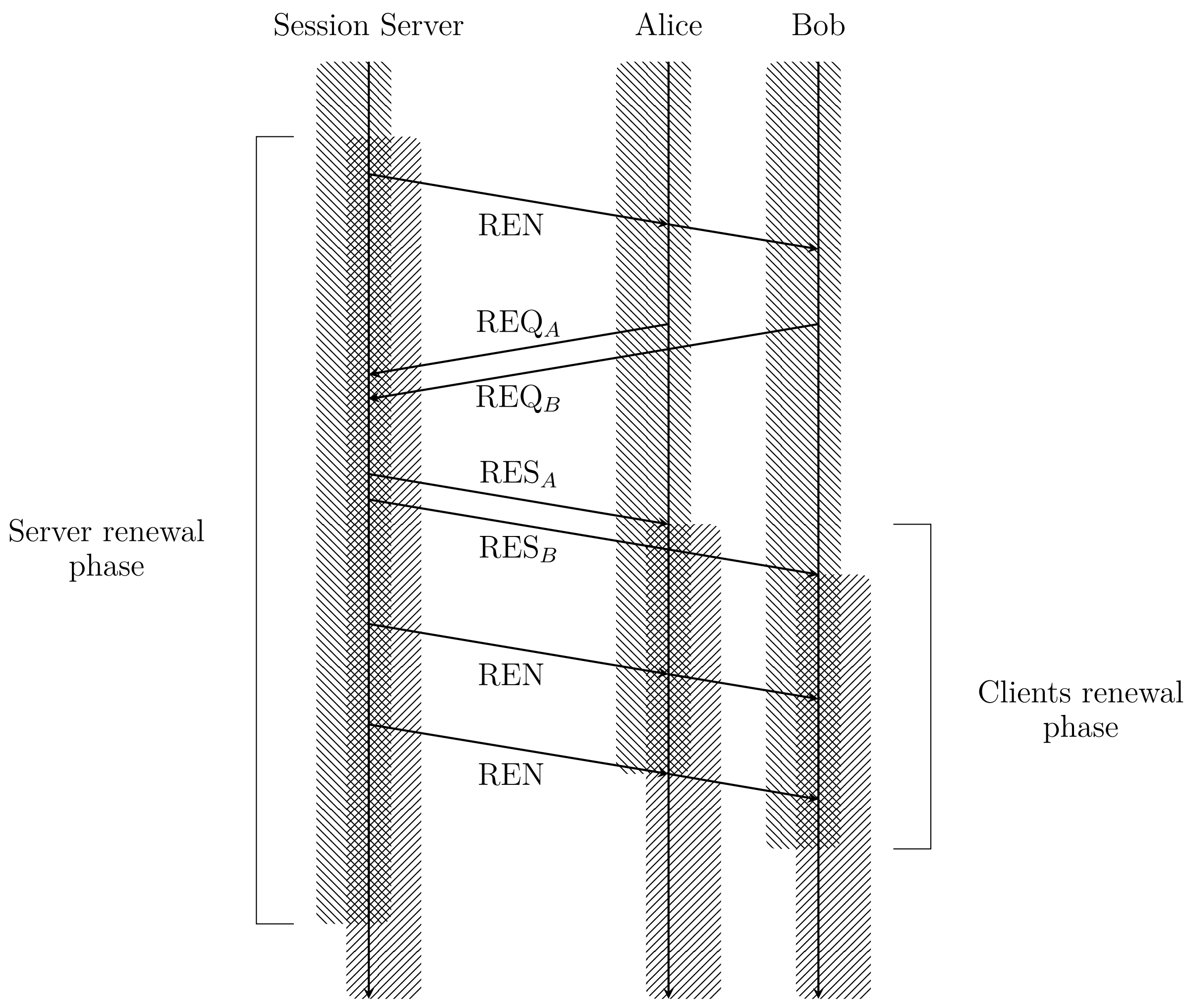


Figure 20: Timeline of the Session renewal phases on the Session Server and on the Clients. In this example Alice and Bob react on the first REN message to request the new Session information. The patterned areas with diagonal lines going from top-left to bottom-right (\\\\\\) indicate the time interval when the Party is still able to use STK_G^{old}, while the top-right to bottom-left diagonal pattern (///) indicates when the Party possesses the new STK_G. Not to scale.

3. The Session Server has to transmit 3 distinct Session Renewal Notification messages, as defined in Section 5.7.3, each built anew just before transmission, with a time delay of t_G^{ntf} between each transmission.

4. While the Session Renewal Notifications are being transmitted and until the Session renewal phase is over[18], received messages using STK_G^{old} are still processed and accepted, if valid. They could be easily distinguished from the messages using the new STK, because the ones with the old STK have large Counter Nonces $\approx N_G^{ctr,old}$,

[18]The implementation can easily verify if the Session renewal phase is still ongoing by checking whether $STK_G^{old} \neq 0$ holds, as STK_G^{old} is cleared during the Session renewal phase termination.

while the new ones are closer to 0. In particular if:

$$ctrnonce \geq \left\lfloor \frac{N_G^{ctr} + N_G^{ctr,old}}{2} \right\rfloor$$

then the message uses STK_G^{old}. The $N_G^{ctr,old}, m_G^{old}$ are also updated for each received Secured Application Data message that uses STK_G^{old}, just like they usually would (see Section 5.6.2).

5. Just like after the Server initialisation (see Section 5.1), as long as $m_G = s_G$ for a Group G, no Client in that Group has yet obtained the new STK_G, N_G^{ctr} values enabling it to validate and decrypt a Secured Application Data message. This should discourage the Session Server to transmit such messages within the Group, as no Client would be able to process them successfully.

6. The Server-side Session renewal phase terminates when at least one of the following conditions is satisfied:

   - when enough Secured Application Data messages that use the new STK_G have been received or transmitted by the Session Server. The threshold is twice the Counter Validity Delay:
   $$N_G^{ctr} \geq 2\, D_G^{ctr}$$
   - when enough time has passed since the start of the new Session. The threshold is six times the delay between Session Renewal Notifications since the Session start:
   $$\texttt{timedelta}(s_G, \texttt{currenttime}()) > 6\, t_G^{ntf}$$

   At termination of the Server-side Session renewal phase, the Session Server erases the old STK, Counter Nonce and Message Timer and thus cannot accept old messages any more:

$$\begin{aligned} STK_G^{old} &:= 0 \\ N_G^{ctr,old} &:= 0 \\ m_G^{old} &:= 0 \end{aligned}$$

5.7.3 Session Renewal Notification message format

The fields of the message are defined in Figure 21, including their data types, names and values.

REN-Header			REN-Payload	
GID	*SID*	*PTY*	`uint24` *ctrnonce*	`uint8[16]` *tag*
id_G	$id^S = 0$	REN	$N_G^{ctr,old}$	(see note below)

Figure 21: Diagram of a Session Renewal Notification message

The value of the *tag* field is the keyed (with the old STK), labelled hash of the Header and *ctrnonce* fields:

$$tag \coloneqq \texttt{Hash}(STK_G^{old}||m_{ren}||GID||SID||PTY||ctrnonce, 16\,\mathrm{B})$$

The Label m_{ren} is the constant `uint8[11]` array:

$$m_{ren} \coloneqq (99, 98, 115, 95, 114, 101, 110, 101, 119, 97, 108)$$

being the ASCII encoding of the string `cbs_renewal` without null-terminations. The *GID*, *SID* and PTY fields in the concatenated hashing input are each encoded into a `uint8`. If no *GID* field is present in the Header, it's assumed to have the value 0.

5.8 Client processing Session Renewal Notification

The Client on its own is not aware of the Session's expiration, so it gets notified by the Session Server with a Session Renewal Notification message. If the message is valid, the Client then reacts by transmitting a Request and expecting a Response to get the new STK.

5.8.1 Session Renewal Notification validation

1. The *GID* field, if present, must indicate a known Group.

2. The PTY field must indicate a valid type (in this case REN).

3. The *SID* field must be 0 (being the Session Server), thus $id^S = 0$. Otherwise, the CBS implementation must provide a SOM-warning to the application.

4. The REN-Payload must have length $\geq$ 19 B (sizes of *ctrnonce* + *tag* fields). Longer Payloads are allowed because additional bytes may be used by future CBS versions.

5. The validating Client must have a non-zero STK_G, thus be enabled to validate and decrypt the message.

6. The validating Client must be in the proper state to accept Session Renewal Notifications, i.e. when it's not in the Client-Side Session renewal phase and when it's not currently waiting for a Response to a previously transmitted Request message:

   $$STK_G^{old} = 0 \land N_G^{req} = 0$$

   This check effectively drops repeated Session Renewal Notifications.

7. The received Counter Nonce must be smaller than its maximum value, according to its data type:

$$\mathit{ctrnonce} < \max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$

where the *ctrnonce* is the field of the received message. Otherwise, the CBS implementation must provide a RON-warning to the application.

8. The validating Client's local variable of the Counter Nonce must be smaller than its maximum value, according to its data type:

$$N_G^{ctr} < \max \texttt{uint24} = 2^{24} - 1 = \texttt{0xFFFFFF}$$

In other words, once the Counter Nonce reaches the maximum, every secured message is discarded. The Session should anyway be renewed way before such limit from the Session Server.

9. The received Counter Nonce must be fresh. This means that the following expression must be true:

$$\mathit{ctrnonce} \geq N_G^{ctr} - \texttt{ctrdelay}(m_G, \texttt{currenttime}(), S_G^{max}, D_G^{max})$$

where the *ctrnonce* is the field of the received message, N_G^{ctr} is the local Counter Nonce variable of the validating Client, S_G^{max}, D_G^{max} are taken from the validating Client's configuration and m_G from its current state. Otherwise, the CBS implementation must provide a OLD-warning to the application. This check effectively drops the messages that are not fresh.

10. The *tag* must be valid (when evaluated with STK_G, not STK_G^{old}). Otherwise, the CBS implementation must provide a INV-warning to the application.

In case any of these checks fails, the message is discarded and the warning, if any, is provided to the application. Otherwise the receiving Client:

1. Saves the timestamp of the received valid message:

$$m_G := \texttt{currenttime}()$$

2. Increments the local Counter Nonces variable considering the largest between the received and the local variable:

$$N_G^{ctr} := max\{N_G^{ctr}, \mathit{ctrnonce}\} + 1$$

3. Enters the Client-side Session renewal phase as per next Section.

5.8.2 Client-side Session renewal phase

1. Just like during the Client initialisation, the Client must transmit a Request and wait for a Response.

2. While waiting for a Response, the (current for the Client, old for the Server) STK and Counter Nonce are still in the variables of the current session STK_G, N_G^{ctr} and can still be used both for reception and transmission of Secure Application Data messages.

   Until a Response is received and until the end of the Client-side Session renewal phase, further Session Renewal Notifications are ignored, to avoid triggering multiple Request transmissions in a row from the same Client. This is handled by the check at Point 6 in Section 5.8.1.

3. Upon receiving a valid Response, similarly to the Session Server, the Client copies the old STK, Counter Nonce and Message Timer into temporary variables, to keep available a bit longer to handle pending messages:

$$\begin{aligned} STK_G^{old} &:= STK_G \\ N_G^{ctr,old} &:= N_G^{ctr} \\ m_G^{old} &:= m_G \end{aligned}$$

   Then the just-received, new Counter Nonce and STK are saved into STK_G and N_G^{ctr}, while the Request and Message Timers are updated, all as per a regular reception of a Response message, as indicated in Section 5.4.

4. Until the Client-side Session renewal phase is over[19], received messages using STK_G^{old} are still processed and accepted, if valid. They could be easily distinguished from the messages using the new STK, because the ones with the old STK have large Counter Nonces $\approx N_G^{ctr,old}$, while the new ones are closer to 0. In particular if:

$$ctrnonce \geq \left\lfloor \frac{N_G^{ctr} + N_G^{ctr,old}}{2} \right\rfloor$$

   then the message uses STK_G^{old}. The $N_G^{ctr,old}, m_G^{old}$ are also updated for each received message that uses STK_G^{old}, just like they usually would (see Section 5.6.2).

5. The Client-side Session renewal phase terminates when at least one of the following conditions is satisfied:

   - when enough Secured Application Data messages that use the new STK_G have been received or transmitted by the Client. The threshold is twice the Max Counter Validity Delay:

$$N_G^{ctr} \geq 2\, D_G^{ctr}$$

[19]The implementation can easily verify if the Session renewal phase is still ongoing by checking whether $STK_G^{old} \neq 0$ holds, as STK_G^{old} is cleared during the Session renewal phase termination.

- when enough time has passed since the reception of a valid Response:

$$\texttt{timedelta}(r_G, \texttt{currenttime}()) > t_G^{ren}$$

At termination of the Client-side Session renewal phase, the Client erases the old STK, Counter Nonce and Message Timer and thus cannot accept old messages any more:

$$\begin{aligned} STK_G^{old} &:= 0 \\ N_G^{ctr,old} &:= 0 \\ m_G^{old} &:= 0 \end{aligned}$$

If no Response is received, the same behaviour as during the Client initialisation phase in case of a missing Response can be applied (see Section 5.4.2).

6 Checks against replays, spoofs and DoS attacks

Because the Parties communicate over a bus, every Party is able to receive messages from anyone else, including active attackers transmitting replayed and spoofed messages. In this section a few countermeasures are listed.

6.1 Server processing messages only the Server may transmit

Upon receiving a Response or Session Renewal Notification message (which only the Session Server itself may transmit), regardless if they are valid or not, the Session Server must notify the application about it with a MFM-warning, even if the message is addressed as coming from any Client instead of the Session Server. No further processing on the message is performed and it can be discarded. The Session Server must not filter out these messages, i.e. it must always be able to receive them to perform this check.

6.2 Client processing Requests

For performance reasons, Clients may be configured to drop received Request messages a priori without validating them, as they are anyway destined to the Session Server and not to other Clients. For higher security, a Client A can *optionally* validate any received Request message as follows to detect spoofing and replay attacks. Note that in the case when the Client is in a non-receiving low-power mode, such check cannot be performed.

6.2.1 Request message validation

1. The PTY field must indicate a valid type (in this case REQ).
2. The *SID* field must be different than the validating Party's Source Identifier id^A. Otherwise, the CBS implementation must provide a MFM-warning to the application.

In any case, the Request message is discarded at the end of the validation and the warning, if any, is provided to the application. Note that there is no check of the GID field as the validating Party generally does not know all the Groups other Parties belong to.

6.3 A note on Denial of Service (DoS) attacks

The Request message contains no proof of timeliness, thus it can be replayed and the Session Server would simply create and transmit Responses to them. Clients reject unexpected Responses, so there is no actual impact on the states of any Parties. On the other hand, such mechanism could be an attack vector for a Denial of Service (DoS) attack towards the Session Server, forcing it into performing cryptographic computations and keeping its processing load high. The choice not to include timeliness information into Request messages is for performance reasons, as this would require more messages and thus increased latency for the Session Server to distribute the Session information to the Clients.

The protocol is defined under the assumption that DoS attacks are **not performed on the CBS layer**, as it would be much easier for an attacker to perform them on lower layers. To block the CAN bus completely, it is sufficient to constantly transmit a stream of dominant values (logical zeros) or to actively invalidate all the frames sent by others.

Nevertheless the protocol includes minimal and optional protection against DoS attacks on the CBS layer with Clients checking for spoofed/replayed Requests (see Section 6.2).

The implementation of CBS may also contain additional custom countermeasures against such DoS attacks, for example by preventing the Session Server to respond to many successive Requests from the same Party or by still responding but notifying the application with DOS-warnings that an anomalous and potentially nefarious activity was detected.

7 Low-power modes of the Parties

7.1 Client low-power mode

Some Clients may not be active 100% of the time and may deactivate some peripherals (including the CAN transceiver) while waiting for some interrupts in order to spare power, which could make them lose messages that appeared on the bus in the meantime.

Before entering a non-receiving low-power mode, the Client must invalidate its Session information[20] for all Groups:

$$\begin{aligned} STK_G \coloneqq STK_G^{old} &\coloneqq 0, \forall G \\ N_G^{ctr} \coloneqq N_G^{ctr,old} &\coloneqq 0, \forall G \\ N_G^{req} &\coloneqq 0, \forall G \end{aligned}$$

Once the Client awakes again, it must transmit a Request and wait for a Response for each Group of interest, just like during the Client initialisation phase (see Section 5.2.1). This is a requirement to either get a new STK and current Counter Nonce, if the Session was renewed while the Client was in low-power mode, or at the very least to get the current Counter Nonce from a trusted source (the Server), in case the Session is still the same.

While waiting for a Response, **no** Secure Application Data message must be transmitted **or** received. This is effectively prevented by the check 6 of Section 5.6.2, given that the local STK variables are cleared.

Rationale: The transmission of Secured Application Data messages after the awakening, but before getting the up-to-date Session information, would most probably force the receiving Parties to reject them because their old Counter Nonce or even old STK. Receiving any Secured Application Data messages during the same time-frame, on the other hand, opens a window for replay attacks: any message that appeared in the time-frame between the entering and exiting of the low-power mode could be replayed and accepted by the just-awakened Client, as it would contain a Counter Nonce that is still newer than what the Client had since before the low-power phase.

7.1.1 Client caching Request before low-power

Just before entering a low-power mode, a Client may generate a new Request Nonce and Request message for each desired Group as per Section 5.2.1, except the messages are not

[20]Because this makes the other Client variables (see Section 3.2) unusable, as the various checks would anyhow prevent the transmission or reception of Secured Application Data messages, the implementation may decide to simply zero-out all the variables for all of the Groups, effectively clearing the state as if the Client had just booted up for the first time.

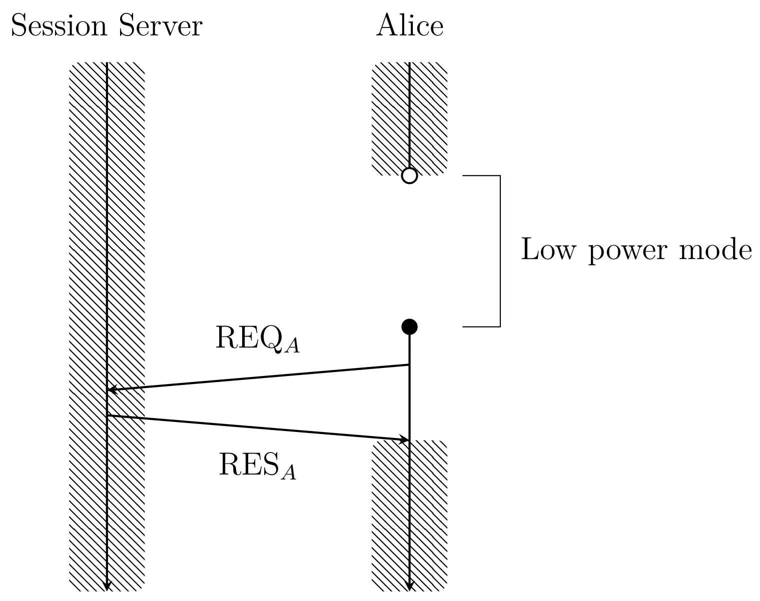


Figure 22: Timeline of the Alice entering a low-power mode (white dot), awaking again (black dot) and requesting the current Session information to the Session Server. The diagonal-patterned area indicates the time interval when the Party possesses the up-to-date STK_G, N_G^{ctr} and is thus able to communicate securely with others. Not to scale.

sent, but cached in a buffer. Once the Client awakes again, the Requests that is supposed to transmit are the cached ones, already prepared and can be directly transmitted. The Client must only take care to store the current timestamps for each Group before the transmission instead of storing them at message construction:

$$r_G \coloneqq \texttt{currenttime}()$$

This behaviour is *optional* and may have a performance advantage, to let the Client obtain the Responses and get up-to-speed with the rest of the Groups faster, as it does not have to generate the Request messages after awaking.

7.2 Session Server low-power mode

The Session Server is allowed to go only into such low-power modes that guarantee **no** message on the bus being lost. For example, deactivating non-CAN related peripherals is thus allowed, as long as any message over CAN activates again all the required components to process said message in time.

References

[1] 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.

[2] Roger M. Needham and Michael 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.

[3] 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.

[4] *Alice and Bob notation.* URL: https://en.wikipedia.org/wiki/Alice_and_Bob (visited on 2020-07-14).

[5] Christoph Dobraunig et al. *Ascon v1.2.* 2019. URL: https://ascon.iaik.tugraz.at/files/asconv12-nist.pdf.

[6] *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.

Appendices

A Alternative AEAD ciphers and hash functions

The protocol user may prefer using an alternative definition of the `AEAD` or `Hash` function rather than Ascon128 and Ascon-XOF, for example because a cryptographic library with AES-GCM and HMAC-SHA256 is already available on the used platform, possibly including hardware support, or LTKs with more than 128 bits are a system requirement.

This is possible, provided that every Party within the same Group uses the alternative `AEAD`/`Hash` and that the cryptographic properties described in Sections 2.10.1 and 2.10.2 are still valid. Additionally:

- In case some fields would require padding (e.g. the plaintext or the AEAD-nonces to fit a block cipher), it should be a padding with zeros added on the most-significant side[21] of the value, i.e. the value does not change if interpreted as an integer. For example, assuming the AEAD-nonce as used in the Response message (see Section 5.3.2) needs to be 160 bit instead of 128:

$$N_{aead} \coloneqq \overbrace{N_G^{req}||N_G^{res}}^{128\,\text{bit}}\,||\,\overbrace{0\ldots0}^{32\,\text{bit}}$$

- In case the required tag or digest is shorter than the actual output of the `AEAD`/`Hash` function, the latter should be truncated and only the first bytes (those with smaller indices in the array of bytes) should be considered.
- In case the LTK must be longer than 128 bits, the `AEAD` function must be changed to an implementation supporting such key length. The STK size is fixed at 128 bits to avoid changing the format and size of the Response message.
- The `Hash` function must be resistant to length-extension attacks; otherwise the used keyed hash scheme (prepending the key to the hashed message) can be exploited.

[21] Considering that little Endian is the encoding choice (see Section 2.7), this means a right-side zero-padding of the encoded values.

B Reconfiguring Long Term Keys

In case a Client needs to be reconfigured (e.g. the hosting device is replaced or a configuration reset is required), a new[22] LTK_{AS} must be generated by an external third party, like a human operator, and installed on both the Client and the Session Server. In case the Session Server needs to be reconfigured, the same operation must be performed for each Client-Server pair.

Rationale: The system is designed to allow a simple replacement of the Clients, which are assumed to be more vulnerable and exposed devices, e.g. peripherals on a vehicle which could be damaged by a light traffic accident. The Session Server is assumed to be a more powerful, central, less exposed, controlling device, e.g. the body control module of a vehicle. The assumption is that the replacement of such a central device requires anyhow a reconfiguration of many subsystems on the bus, so reconfiguring also the cryptographic keys should not be an issue.

[22]The assumption here is that the old LTK_{AS} cannot be read out from one of the two Parties holding it as it's kept in a secure subsystem or module. In any case, it's good security practice to avoid reusing the same keys after a reconfiguration.