SOFA – Secure Object Fieldbus Access

Generic secure access via Fieldbus tunnels

White Paper EmSA-WP-105

Version 1.24
21 July 2026

Jointly authored by

Embedded Systems Academy GmbH
Bahnhofstraße 17
30890 Barsinghausen
Germany

Embedded Systems Academy, Inc.
84 W. Santa Clara St., Suite 700
San Jose, CA 95113
United States

www.em-sa.com

© Embedded Systems Academy, 2026. Permission is granted to copy, distribute, and use this material for non-commercial educational purposes with attribution. Commercial use requires explicit permission.

The authors are not liable for defects or indirect, incidental, special, or consequential damages, including loss of anticipated profits or benefits, arising from the use of this document or warranty breaches, even if advised of such possibilities.

The information presented in this document is believed to be accurate. Responsibility for errors, omission of information, or consequences resulting from the use of this information cannot be assumed by the authors.

Contents

1 Scope and Objective 4
1.1 Terms and Definitions 5
2 Prerequisites 7
2.1 Symmetric Keys Used 7
2.2 Read and Write Accesses 8
3 Secure Communication Configurations 11
3.1 Capability Descriptor 11
3.2 Status Descriptor 12
3.3 Object Selection Determines Direction and Meaning 12
4 Protocol Description 13
4.1 Session-Based: TLS-PSK and cTLS 13
4.2 Per-Transfer: FBsec 14
4.2.1 Secure Read 14
4.2.2 Secure Write 16
4.2.3 Repeated Accesses 18
4.2.4 Authentication Tag Calculation 18
4.2.5 Authentication Tag Calculation Example 19
4.2.6 Nonce Composition Example 20
5 Optional Asymmetric Identity 21
5.1 Identity and Trust Model 21
5.2 Announcing and Selecting the Asymmetric Mode 22
5.3 Handover Exchanges 22
5.3.1 Identity Object and Signed Identity Read 22
5.3.2 Ownership Claim 22
5.3.3 Anchored Provisioning Key Install 23
5.3.4 Local Device Identity Generation 23

5.4 Signed FBsec: Signature-Protected Access 23
5.5 Client Obligations 26
5.6 Segmentation of Signatures 26
6 Implementation Guidance 27
6.1 Mandatory Objects 27
6.2 Salt Slot Behavior 28
6.3 Session Key Reuse Across Servers 29
6.4 Threats Against the Salt Write 29
6.5 Example Code 30
6.6 Outlook to CANopen CC and CANopen FD 30
7 Alignment With Standards and Regulations 31
7.1 ISO/IEC 9798-2 31
7.2 IEC 62443-4-2 31
7.3 EU Cyber Resilience Act 31
7.4 NIS2 32
7.5 BSI TR-02102 32
8 Figures: Sequence Diagram Sources 33

1 Scope and Objective

Many fieldbus deployments still run with no transport security. The devices on those buses are often resource-constrained, the protocols predate modern cryptographic guidance and the application code on each node has no room for a full TLS stack or the required communication bandwidth. At the same time, the EU Cyber Resilience Act (CRA) requires manufacturers to ship secure-by-design products now, and standards bodies have not yet delivered a fieldbus-level security profile that suits this device class.

SOFA (Secure Object Fieldbus Access) closes that gap. SOFA is a symmetric-key security layer that runs entirely on top of an existing client-server fieldbus protocol, without modifying the fieldbus stack itself. SOFA tunnels short, authenticated request-response sequences through ordinary register reads and writes (Modbus) or object dictionary accesses (CANopen, EtherCAT and others), so that a constrained device can offer selective secure access without re-engineering its communication stack.

SOFA is aimed at securing single, selected accesses: service parameters, configuration objects and infrequently invoked functions such as firmware activation, configuration locking or mode switch. It is not designed for protecting the high-rate cyclic process data (periodic measurements, drive-loop updates) that dominates a running fieldbus. For that traffic class a dedicated security sublayer sitting directly above the data link layer is the better fit. The SPsec (Small-Packet Network Security Sublayer) specifications address it explicitly. The SPsec specifications are available at EmSA's technical library at: https://www.esacademy.com/en/library/spsec.html

SOFA aims to:

- Add per-parameter or per-parameter-group secure access on top of the existing fieldbus, with no changes below the application layer.
- Allow secure triggering of well-defined functions such as bootloader activation, configuration locking or transitions into restricted operating modes.
- Provide unilateral authenticity for both directions of access, by AEAD key possession for operational access.
    - On a secure client-to-server write, the server is convinced that the data it received is authentic.
    - On a secure server-to-client read, the client is convinced that the data it received is authentic.
- Optionally, provide authenticity by an Ed25519 signature for the infrequent identity and configuration accesses.

We call the guarantee unilateral because only the receiving side of any single exchange gains an authenticity proof; the sending side does not learn whether the receiver acted

on the message. This is sufficient for the use cases that constrained fieldbus devices actually face: pushing trusted configuration into a device, reading back a trusted measurement or identifier and triggering protected functions. Mutual session-level authenticity belongs to TLS-class protocols, not to a constrained fieldbus application.

> ⚠ **SECURITY**
> Authenticity here means possession of the relevant base key. A valid tag proves that the message came from a holder of that key and was bound to the agreed nonce; it does not by itself identify which device produced it. Whether a key maps to a single device or to a group is a system-level decision taken by whoever provisions the keys: the provisioning authority may assign a distinct key per device or share one key across several nodes, according to the system requirements and the other protocols that rely on those keys. SOFA authenticates the key domain; the meaning of that domain is fixed by the key-assignment policy, not by SOFA.

SOFA is not a standard. The CiA (CAN in Automation) Higher Layer Protocol (HLP) Cybersecurity working group and other bodies are preparing formal profiles that may eventually subsume SOFA. Until those profiles ship, and until existing field devices can be updated to support them, SOFA gives manufacturers, integrators and operators a usable security layer that can be deployed today and that meets the obligations the CRA already places on shipping products.

1.1 Terms and Definitions

This section fixes the vocabulary specific to SOFA and its optional asymmetric identity. Terms inherited from the key model are defined in EmSA-WP-104, and cansecurity.net carries the wider CAN security vocabulary.

- **Terms adopted from EmSA-WP-104.** This document builds on the key model of EmSA-WP-104 and uses its vocabulary without restating it. The following are used here with the meaning that document fixes: base key, and the Provisioning, Integrator and Operator Keys beneath it; security profile; uncommissioned state and commissioning environment; ownership voucher and owner epoch; owner public key and ownership credential; delegation; trust anchor and trust domain; re-provisioning; and factory restore. The device capability level, the factory device identity (IDevID) and the local device identity (LDevID) keep their WP-104 meaning too, but are listed separately below because this document adds a SOFA-specific nuance to each.
- **FBsec (FieldBus security).** SOFA's per-transfer challenge-response protocol, patterned after ISO/IEC 9798-2; see Protocol Description.

- **Session key.** The key that protects each exchange, derived from a base key and the session salt. It is never transmitted; both peers derive it independently.
- **Session salt.** The non-secret derivation input the client writes once per power cycle to the salt slot, combined with a base key to derive the session key.
- **Protection primitive.** The cryptographic construction that protects one secured access in one direction: either the AEAD tag or an Ed25519 signature. It is a property of the object and the direction, fixed by the device and announced in the descriptors. An access is protected by one primitive, never both.
- **Control plane and data plane.** The control plane is the set of infrequent accesses that establish identity, ownership and configuration: identity read, commissioning, ownership claim, mode switch and configuration lock. The data plane is operational read and write access, including cyclic push and pull. Signature protection applies to the control plane only.
- **Device capability level.** The classification C0 to C3 of a device's cryptography, defined in EmSA-WP-104. C0 is symmetric only, C1 can also verify a signature, C2 can also sign, C3 can also agree a key. It bounds which protection primitives and profiles an object can use.
- **Transcript.** The ordered set of values a signature or tag is computed over: a version tag, a role-and-direction byte, both challenge randoms and the addressing context. Signing and verifying are always over the transcript, never over the payload alone.
- **Factory device identity (IDevID).** The manufacturer-installed Ed25519 keypair; permanent, and not erased by factory restore. See EmSA-WP-104.
- **Local device identity (LDevID).** The integrator-installed Ed25519 keypair; erased by factory restore. See EmSA-WP-104.
- **Key management system (KMS).** The integrator's registry of device public keys and key material, from which local device identity public keys are distributed to peers.
- **Signed FBsec.** An FBsec access whose protection primitive is an Ed25519 signature over the challenge transcript, replacing the AEAD tag for that access. It authenticates the sending entity and provides no confidentiality.
- **Signed identity read.** The exchange in which a tool writes a fresh random to the identity object and reads back the factory device identity certificate with a signature over the transcript, proving the device is genuine and is answering now. See Handover Exchanges.

2 Prerequisites

SOFA targets fieldbus systems that already provide a client-server transaction model. A SOFA exchange is implemented as a short sequence of ordinary reads and writes against a single fieldbus address, so the underlying fieldbus must be able to transport request-response transactions of arbitrary payload size between a client and a named server. The protocols we have validated against this requirement are Modbus (where the address is a register number), CANopen (where the address is an object dictionary index and subindex) and EtherCAT mailbox access (which uses the same CoE addressing). Any other fieldbus that exposes equivalent semantics fits the same model.

2.1 Symmetric Keys Used

SOFA assumes that the key model described in white paper EmSA-WP-104 is in place. From WP-104, a SOFA implementation inherits the base key hierarchy (Provisioning Key, Integrator Key, Operator Key) and the cryptographic primitives, the Authenticated Encryption with Associated Data (AEAD) primitive and the HMAC-based Key Derivation Function HKDF-SHA-256. SOFA does not redefine those; it consumes them. SOFA itself defines how a base key becomes the working key on the wire: the session key that protects an individual SOFA data-plane exchange is derived from an operational base key, the Integrator or Operator Key, and a per-power-cycle salt, using HKDF-SHA-256. The Provisioning Key is the ownership credential and does not protect operational traffic. The cryptographic primitives table is not repeated here; see WP-104, chapter "Cryptographic Primitives" for the authoritative list. Figure 1 shows this derivation; in explicit form:

Key derivation with HKDF-SHA-256

Input:

- `IKM` = `base_key` (WP-104 tier in use: Integrator or Operator Key)
- salt = `active_salt` (the per-power-cycle value in the salt slot)
- info = `context_label` (application-specific label)

Output:

- `SK` (the session key required by the AEAD primitive)

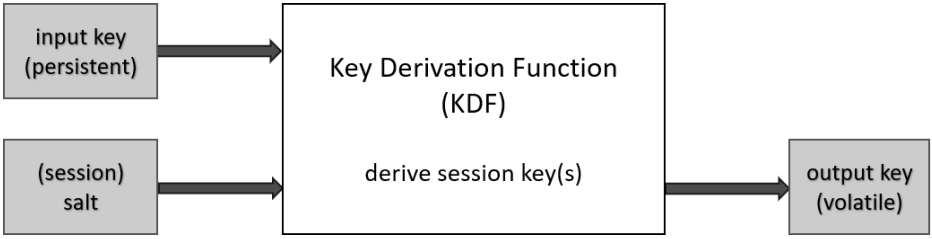


FIGURE 1: KEY DERIVATION WITH HKDF-SHA-256

2.2 Read and Write Accesses

A note on read-write semantics is worth flagging up front, because it can be confusing. At the fieldbus layer, every register or object dictionary entry that carries SOFA traffic must be read-write. The reason is that a SOFA exchange is itself a sequence of fieldbus operations: a client write to deliver a nonce or challenge, a server read to return the data and the authentication tag, then further reads or writes if the exchange requires them. The fieldbus has no idea that these reads and writes are part of a higher-level protocol, so it must permit them in both directions. At the SOFA layer above, however, that same entry exposes only one direction (unilateral security): it is either secure-read-only (the client reads authenticated data from the server) or secure-write-only (the client writes authenticated data to the server). The fieldbus-level read-write surface and the SOFA-level unilateral surface are not in conflict; they are two views of the same object at two different layers.

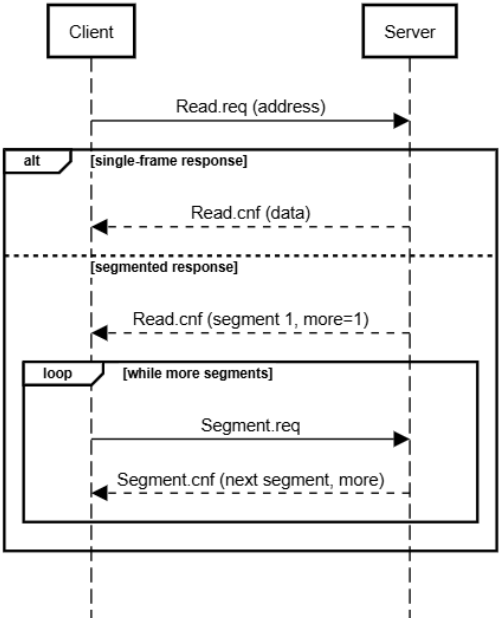


FIGURE 2: GENERIC FIELDBUS READ WITH OPTIONAL SEGMENTATION

As illustrated in Figure 2 and Figure 3, the underlying fieldbus carries any read or write as a single transaction that may be completed in one exchange (single transfer) or split across several segments. SOFA exchanges are layered on top of these primitives without changing them.

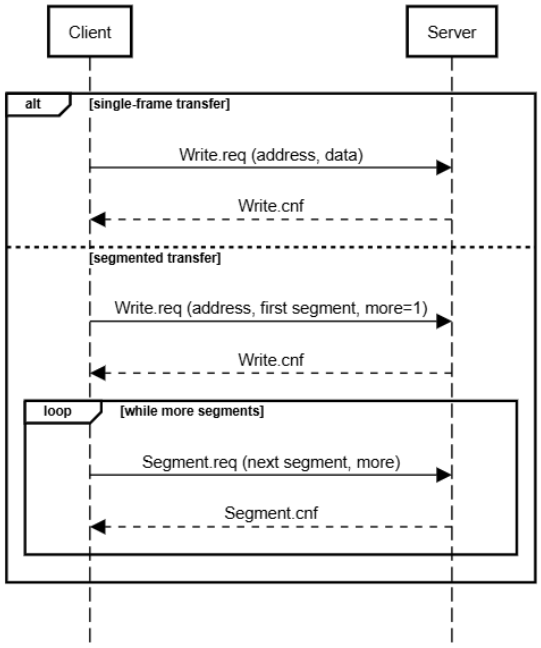


FIGURE 3: GENERIC FIELDBUS WRITE WITH OPTIONAL SEGMENTATION

3 Secure Communication Configurations

Before two devices can establish a secure exchange, they need to discover what the other side can do. SOFA addresses this with two fieldbus-visible objects that every SOFA-capable server should expose. The first carries the server's static security capabilities, the features it could in principle support. The second carries the server's current security state, the features that are actually configured for use right now. Together they let a client decide which protocol variants, which key tiers and which cryptographic primitives are usable against this particular server in this particular state.

3.1 Capability Descriptor

The capability descriptor is read-only and unauthenticated. It lists the cryptographic and protocol features the device implements, regardless of whether any keys are currently provisioned. The descriptor covers at minimum:

- Device capability level and security profile, as defined in EmSA-WP-104, so that a client learns in one read what class of device it faces.
- Symmetric key support: which of the three base-key tiers (Provisioning, Integrator, Operator) are available.
- Key and tag sizes.
- Symmetric algorithms: which AEAD constructions are implemented, typically AES-128-GCM, AES-256-GCM or for very constraint devices ChaCha20-Poly1305.
- Session protocols supported: TLS-PSK, cTLS or SOFA.
- Asymmetric key and algorithm support, where present.
- Certificate support, where applicable.
- Per-object protection: for each secured object and each direction, which primitive applies, the AEAD tag or an Ed25519 signature, and whether AEAD encryption is enabled.

The capability descriptor is unauthenticated because a client must be able to discover it before any key material is in place. The descriptor reveals nothing that an attacker could not also learn from the device datasheet.

Because the descriptor is unauthenticated, a client must never let the device choose the security level for it. The descriptor states what the device can do, not what will be used; the client holds its own minimum profile and refuses a device that declares less. At capability level C2 and above the device echoes its declared level and profile inside the signed identity read, so a client compares the signed values against the descriptor it read in the

clear and detects a forged declaration. At C0 and C1 the declaration cannot be authenticated and the client treats it as advisory, relying on the controlled commissioning environment.

3.2 Status Descriptor

The status descriptor is read-only and unauthenticated and mirrors the capability descriptor field-for-field, but reports what is currently active rather than what is theoretically possible. A bit that is set in the capability descriptor but cleared in the status descriptor means the function is implemented but not yet configured. For example, a device that supports multiple algorithms in its capability descriptor can only report one as currently active.

3.3 Object Selection Determines Direction and Meaning

SOFA does not carry a separate "direction" or "meaning" field in its protocol header. The fieldbus address, the register number or the object dictionary index and subindex, is itself the binding between the secured exchange and the data or function being accessed. Selecting an object determines everything else: the selected object identifies which data is exchanged or which function is triggered, and the same selection fixes the SOFA direction at secure-read-only or secure-write-only. This keeps the on-wire format minimal and pushes all configuration into the device's per-object access table.

4 Protocol Description

SOFA offers two protocols. The first is session-based, using TLS-PSK or cTLS, tunneled over the fieldbus as a sequence of register or object reads and writes. The second is per-transfer, called FBsec, derived from ISO/IEC 9798-2, in which each individual read or write carries its own freshness material. The two protocols address different deployment trade-offs: the session-based protocol pays a one-time handshake cost and then makes repeated accesses cheap; the per-transfer protocol pays a constant per-access cost and avoids any session state on the server.

A server must implement at least one of these protocols. The capability descriptor in the previous chapter announces which ones are available. Both protocols derive the on-wire session key from one of the WP-104 base keys and the active session salt; they differ only in how that session key is used.

4.1 Session-Based: TLS-PSK and cTLS

In this protocol the client opens a short pre-shared-key session against the server, using TLS-PSK as defined in RFC 4279 or cTLS (the compact TLS variant currently being standardized in the IETF). The pre-shared key supplied to the handshake is the SOFA session key derived from the selected base key and the salt; the choice of base key (Integrator or Operator) is encoded in the PSK identity field of the handshake, so the server can look up the right key without a separate selection round. Once the handshake completes, the same TLS record-layer key protects every subsequent SOFA access against the chosen object until the session ends, times out or the device is reset.

After the session is open, an access to a secured object is a normal fieldbus read or write whose payload is wrapped by the TLS record layer. Repeated accesses against the same object are explicitly supported and are the expected pattern for cyclic polling of a status register, for periodic re-reads of an authenticated device identifier or for streaming configuration updates. Each individual record carries a monotonically increasing sequence number, which the TLS record layer manages; that sequence number is what gives every repeated access its own freshness without re-running the handshake.

A session is bound to the power cycle in which it was opened. Resetting the device clears the session salt slot and forces a new session on next access.

4.2 Per-Transfer: FBsec

For deployments that cannot afford to maintain TLS ssessions, SOFA offers FBsec (FieldBus security), a per-transfer challenge-response protocol patterned after the random-challenge mechanism of ISO/IEC 9798-2. Each secure read or write is a self-contained exchange of two fieldbus operations, in which the data carries its own authentication tag and a fresh nonce is contributed by both sides.

FBsec follows the shape of ISO/IEC 9798-2 Mechanism 4, a three-pass mutual authentication using random challenges, truncated to two passes so that authentication is unilateral instead of mutual by construction. Client and Server still both contribute a random value, which preserves the mutual-randomness pattern that diversifies the nonce. SOFA deviates from 9798-2 in five points:

(i) the third pass that would close mutual authentication is dropped, leaving a unilateral exchange;
(ii) the bare MAC of 9798-2 is replaced by an AEAD primitive that subsumes the MAC, adds optional confidentiality and requires an explicit nonce 9798 does not specify;
(iii) the AEAD nonce is built from a base value and a strictly monotonic counter, a construction 9798 leaves open;
(iv) entity identifiers and addressing context are carried in the AEAD's Associated Data rather than concatenated into a MAC input;
(v) a counter extension lets a single challenge cover a long run of cyclic continuations without a fresh round trip, an extension 9798-2 does not describe.

4.2.1 Secure Read

A secure read is a two-step exchange in which the client validates an authenticated payload supplied by the server. Figure 4 shows the exchange end-to-end, including the optional cyclic continuation.

1. The client writes its random portion to the secured object as a challenge. On receipt the server samples its own random portion, sets the message counter to zero, generates the nonce, computes the authentication tag over the requested data and the resulting nonce, then prepares the response payload internally before acknowledging the write. Preparing the response before the write completes lets the client issue the follow-up read immediately, with no extra round trip or delay.
2. The client reads the secured object. The server returns the data, its own random portion, the (possibly truncated) counter and the authentication tag. The client reconstructs the nonce locally and accepts the data only if the tag matches.

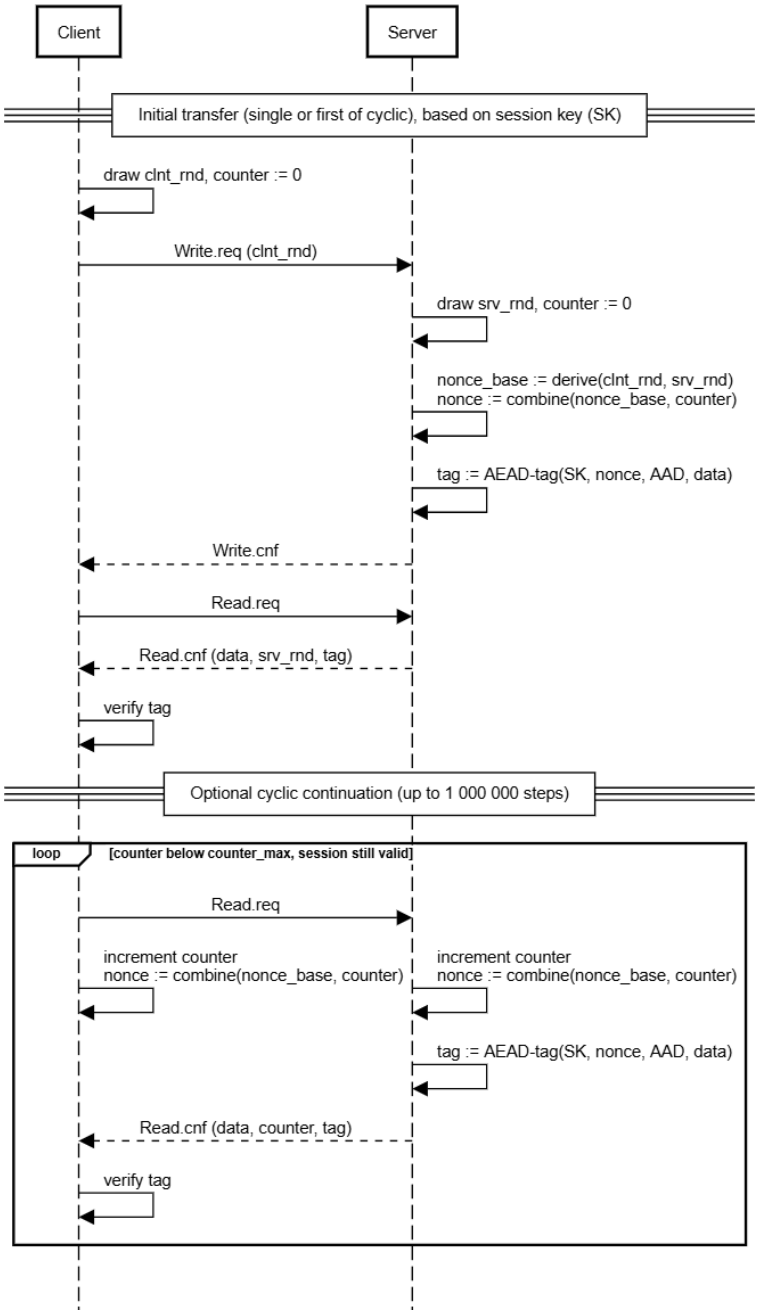


FIGURE 4: SECURE READ WITH OPTIONAL CYCLIC CONTINUATION

The client emerges from this exchange with cryptographic confidence that the data it just read came from a peer holding the same session key and that the data was bound to the nonce the client itself helped form. The server learns nothing it did not already know.

> ⚠ **SECURITY**
> The client random value acts as the freshness challenge for the read. The client reconstructs the nonce from its own current random and accepts the tag only if it matches, so a response replayed from an earlier exchange fails verification. This holds as long as the client random is unpredictable and does not repeat within the life of a session key.

4.2.2 Secure Write

A secure write is the mirror exchange in which the server validates an authenticated payload supplied by the client. Figure 5 shows the exchange end-to-end, including the optional cyclic continuation.

1. The client reads the secured object to receive the challenge. The server returns its own random portion. The server refreshes this value on every read so that two reads in succession return different challenges.
2. The client generates its own random portion, sets the message counter to zero, computes the authentication tag over the data, the associated data and the resulting nonce, then writes the data, its random portion, the (truncated) counter and the authentication tag to the secured object. The server reconstructs the nonce and accepts the data only if the tag matches.

The server emerges from this exchange with cryptographic confidence that the data it just accepted came from a peer holding the same session key and that the data was bound to a nonce the server itself helped form. The client learns nothing it did not already know.

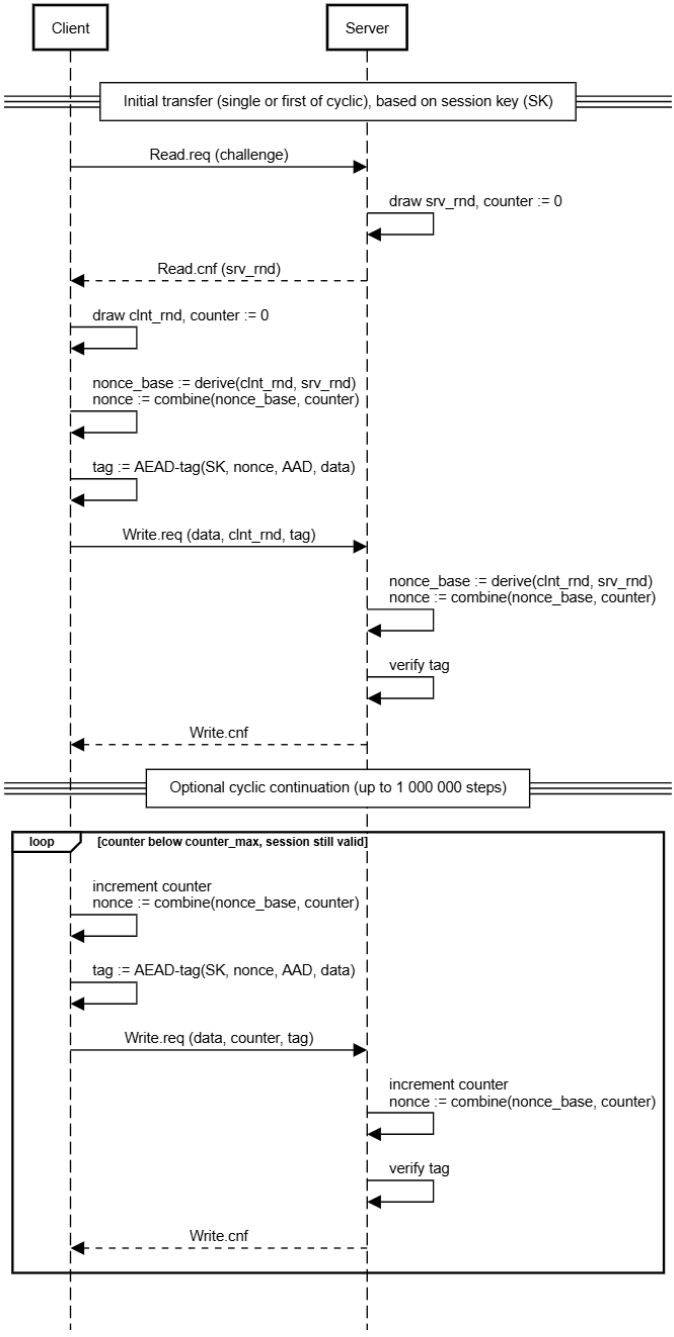


FIGURE 5: SECURE WRITE WITH OPTIONAL CYCLIC CONTINUATION

4.2.3 Repeated Accesses

For a single transfer the counter is zero and both peers contribute fresh randoms through the challenge step of the secure read or secure write.

A client and server may continue an established secure read or secure write cyclically without repeating the challenge. On every continuation the two randoms are reused unchanged and the counter increments by one. The combined nonce stays unique because the counter is strictly monotonic, so each continuation produces a fresh authentication tag with no extra round trip.

A cyclic continuation ends when:

1. either side draws fresh randoms,
2. the counter reaches its profile-specific maximum or
3. the active salt is invalidated by a reset on either node.

After any of these conditions the next access starts a fresh challenge.

4.2.4 Authentication Tag Calculation

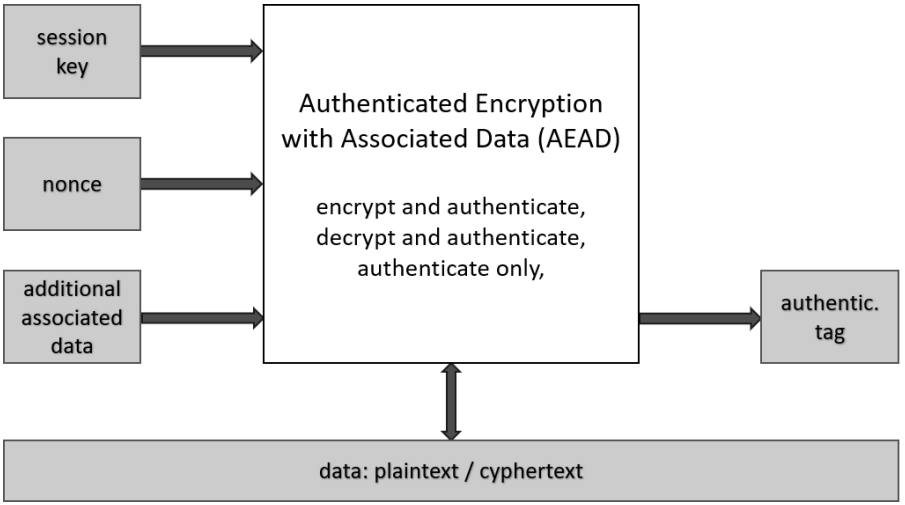


FIGURE 6: AEAD INPUTS AND OUTPUTS

The authentication tag is the output of the AEAD primitive over four inputs, shown in Figure 6: the session key SK derived as described in Prerequisites, the nonce, the Associated Data and the plaintext payload. The nonce is the freshness input. Its exact layout is a

fieldbus-profile choice; this document gives the composition structure below and the per-fieldbus binding fixes the concrete sizes.

The parameters of FBsec are:

- Key: the SOFA session key `SK` derived from the selected base key and the active salt; see Prerequisites for the explicit KDF formula.
- Authentication: mandatory. Every access carries an authentication tag computed over the data and the nonce.
- Encryption: optional, negotiated per object via the capability and status descriptors. Authenticity is the minimum guarantee SOFA provides; confidentiality is added on top where the object's content warrants it.
- Nonce: a value combining a client-contributed random, a server-contributed random and a message counter. The counter is zero for a single transfer and increments by one for each cyclic continuation. The exact layout is fieldbus-specific; see "Nonce Composition Example" below for one composition.
- Authentication tag length: per WP-104 guidance; the concrete width is fixed by the per-fieldbus binding.

> **⚠ SECURITY**
> Authentication is mandatory on every secured access. Encryption is optional and negotiated per object, but an exchange without a valid authentication tag or signature is never accepted; authenticity is the minimum guarantee SOFA provides.

4.2.5 Authentication Tag Calculation Example

Tag generation with AEAD:

Input:

- Key: the SOFA session key `SK` derived (as in Prerequisites)
- Nonce: the freshness value composed from the client random, the server random and the message counter (see "Nonce Composition Example")
- Associated data (AAD): the addressing context, client node ID, server node ID and the data register or object identifier addressed by the exchange, together with the key-select indicator and the encryption-enabled flag. The encryption-enabled flag is derived locally on both sides from the object's capability and status descriptors and is not transmitted per message.
- Plaintext payload: the data covered by the tag, and where the object is encrypted also enciphered by the AEAD primitive

Output:

- Ciphertext (present only when encryption is enabled, same length as the plaintext payload)

- Authentication tag (length per WP-104 guidance)

4.2.6 Nonce Composition Example

Both peers contribute a random value during the challenge step, and both derive the same nonce base from those two randoms. A message counter is then folded into that base so that cyclic accesses stay fresh without a new challenge:

- nonce_base = mix(client_random, server_random)
- nonce = combine(nonce_base, counter)

The counter is zero for a single transfer and increments by one for each cyclic continuation, so the two randoms stay unchanged while every continuation still produces a fresh nonce. The widths of the randoms and the counter, and the mix and combine functions, are fixed by the per-fieldbus binding, not by this document. The one invariant this document fixes is that the resulting nonce is unique per (SK, message) pair across the life of the session key.

> ⚠ **SECURITY**
> Nonce uniqueness under a session key is mandatory. The session key is fixed for the whole power cycle, so every exchange shares it, and with AES-128-GCM a single nonce reuse under one key is catastrophic: it can expose the authentication subkey and allow forgery. The client and server randoms must therefore be wide enough that a collision in the nonce base is negligible over the session; the per-fieldbus binding sets that width, and a binding that narrows the randoms must justify the smaller width against this bound.

5 Optional Asymmetric Identity

SOFA is symmetric at its core, and the protocols described so far authenticate by possession of a base key. Where a device can afford public-key operations, SOFA offers Ed25519 as an alternative protection primitive, selectable per object and per direction. An access so configured is protected by an Ed25519 signature over the challenge transcript instead of by the AEAD tag, not in addition to it. A signature binds the access to a specific identity rather than to mere key possession, which is what an identity read, an ownership claim, a mode switch or a configuration lock needs. The primitive replaces the tag; it does not layer on top of it. It's processing time is far slower to verify than an AEAD tag. Therefore, it is confined to the infrequent control-plane accesses named below and is never used for cyclic or high-rate data. A device that implements none of this remains conformant with the symmetric-only profiles.

The partition between the two primitives is deliberate. The control plane, identity, commissioning, ownership, mode switch and configuration lock, is infrequent, needs attribution and needs no secrecy, so it uses the signature primitive. The data plane, operational access and every cyclic push or pull, is frequent, may need confidentiality and must fit the frame budget, so it uses the AEAD primitive. Where the operational base keys are device-unique, the AEAD primitive already attributes a data-plane access to a single node, so signatures are not needed there.

5.1 Identity and Trust Model

A device may carry the two long-term signature keypairs defined in EmSA-WP-104. The factory device identity, an IEEE 802.1AR IDevID, is installed at manufacturing, is permanent, and survives a factory restore; it proves what the device is. The local device identity, an LDevID, is installed by the integrator when it claims the device and is erased by a factory restore; it is the identity the integrator's own tooling trusts at run time. Both use Ed25519 as specified in RFC 8032.

There is no certificate chain. A device identity is a single certificate, a public key bound to the device by one signature, verified against one trust anchor rather than through a path of intermediate certificates: the factory device identity against the manufacturer's trust anchor, and each local device identity against the record the integrator's key management system (KMS) distributes. This is the minimal-PKI model; a device too constrained for a full certificate path can still verify one signature against one anchor.

Every signed transcript is domain-separated. Before signing, the transcript is prefixed with a version tag and a role-and-direction byte, so a signature produced for one role or

one direction can never be verified as another. This domain separation is independent of, and additional to, the AEAD associated data.

5.2 Announcing and Selecting the Asymmetric Mode

The capability and status descriptors announce, per object and per direction, which protection primitive applies, the AEAD tag or an Ed25519 signature, and whether the identity layer is active. Selection is a property of the object, fixed by the device, and is fail-closed and never negotiated down: a client that does not implement the announced primitive cannot complete the access. The primitive identifier is bound into the signed or authenticated transcript, so a peer that assumes the wrong primitive forms a different transcript and verification fails; there is no downgrade path and no separate negotiation round. The encryption flag of the wire key identifier has no meaning under a signature primitive, which offers no confidentiality: on a signature-protected access it is reserved, must be zero, and a non-zero value is rejected.

5.3 Handover Exchanges

EmSA-WP-104 defines the anchored provisioning handover and states that SOFA specifies its on-wire construction. This section gives that construction in SOFA's read-and-write-tunnel terms. Each exchange below is an ordinary secure read or secure write whose payload carries a certificate or a signature, segmented where it exceeds one frame.

5.3.1 Identity Object and Signed Identity Read

The device exposes a readable identity object that returns its factory device identity certificate. A tool retrieves it without authentication, as WP-104 allows. Retrieval alone proves nothing, because an impostor could serve any certificate, so genuineness rests on the signed identity read, not on the object read. In the signed identity read the tool writes a fresh random value to the identity object and reads back the certificate together with an Ed25519 signature over the domain-separated transcript that includes that random. The tool verifies the certificate against the manufacturer trust anchor and the signature against the certified public key. The fresh random is what makes the answer current: without it, a response recorded from a genuine device could be replayed by an impostor. This is the genuineness step of the WP-104 handover.

5.3.2 Ownership Claim

The device exposes a write-only claim object that accepts an ownership voucher, and only while the device is uncommissioned. The tool writes the manufacturer-signed

voucher, which names one device serial, one integrator public key and an owner epoch. The device verifies the voucher against the manufacturer trust anchor it already holds, checks that the serial names this device, that it is still uncommissioned and that the epoch is above the highest it has stored, then stores the integrator public key as its owner and the new epoch. The claim gates the install that follows: after it, the device accepts a first Provisioning Key only from the named owner.

5.3.3 Anchored Provisioning Key Install

With ownership claimed, the tool installs the Provisioning Key as a secure write signed by the integrator private key. The device verifies that signature against the owner public key it stored during the claim, so the install is accepted only from the entitled integrator, and acknowledges with a signature under its factory device identity that lets the tool confirm the key reached the device it verified. From this point the symmetric key hierarchy of WP-104 proceeds as normal, now rooted in a verified handover.

5.3.4 Local Device Identity Generation

After the Provisioning Key is in place, the device generates its own local device identity keypair, signs the new public key with its factory device identity, and exports the public key. The signature lets the integrator confirm the key was generated on the device it has just verified rather than injected, and the integrator's key management system records the local device identity for use at run time.

> **⚠ SECURITY**
> The private half of either device identity must never be readable and must never leave the device. Only public keys and signatures cross the bus. The signed identity read and every signed install must carry a fresh random, or a recorded exchange can be replayed; a signed identity read without a fresh challenge proves only that some genuine device once answered, not that this device is answering now.

5.4 Signed FBsec: Signature-Protected Access

An FBsec access whose protection primitive is Ed25519 is called signed FBsec. It follows the same two-pass challenge-response shape as the symmetric FBsec of the Protocol Description, but the authentication tag is produced by signing the domain-separated transcript with the sender's identity key rather than by the AEAD. On a secure read the server signs; on a secure write the client signs; the receiver verifies against the sender's public key, distributed by the integrator key management system keyed by node identity. Because the transcript binds both challenge randoms and the addressing context, and

the verifier holds the sender's public key by node identity, the access authenticates the specific node rather than merely a holder of a base key.

The challenge is mandatory for a signed access and is what makes it safe. A signature over a static payload would replay forever, so the signed transcript must cover the verifier's fresh challenge random. A node therefore performs a signature verification only when it has an outstanding challenge that it issued itself, and it bounds how many challenges are outstanding and for how long. An attacker cannot make a node spend an Ed25519 verification without first obtaining a challenge from it, so the two-pass exchange is both the anti-replay mechanism and the rate limiter, and the cost of the verify does not widen the denial-of-service surface. Ed25519 signing is deterministic, so the signed primitive adds no per-signature randomness requirement beyond the challenge randoms FBsec already draws.

Selection is per direction. A device that can verify but not sign, capability level C1, accepts and verifies a signed inbound access but cannot sign its response; that response is either unprotected or, where the device holds an operational key, protected by the AEAD. This is the pattern for a control-plane write such as a configuration lock or a key install: only an authorized party may perform it, the device holds no secret for it, and the client confirms the outcome by reading state back rather than by trusting an authenticated acknowledgement. Signed FBsec provides entity authentication and integrity, not confidentiality; a signed payload travels in the clear. It is used for the control plane, which needs attribution and not secrecy, while operational and cyclic data stays on the AEAD primitive. A single signed access is a two-pass exchange in one direction; signing in each direction, shown in Figure 7, yields mutual entity authentication and adds a third pass for the returning signature.

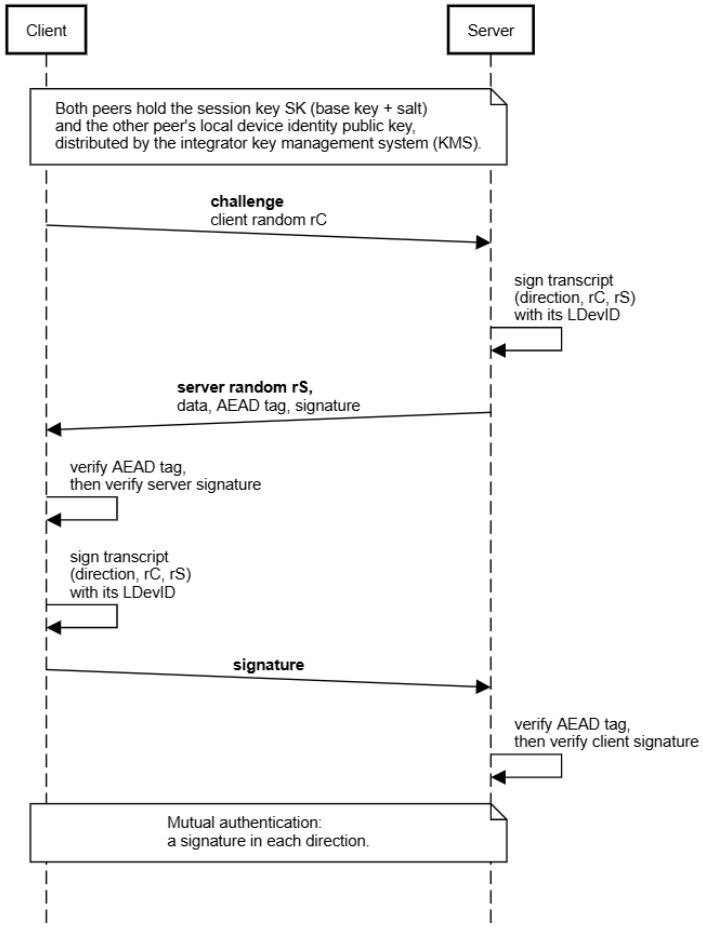


FIGURE 7: SIGNED FBSEC, SIGNATURE-PROTECTED ACCESS

⚠ SECURITY

A protection primitive is a property of the object and its direction, and an access is protected by the AEAD tag or by an Ed25519 signature, never by both and never by neither. Signature protection is confined to infrequent control-plane accesses and must not be used for cyclic or high-rate data, where a per-frame signature neither fits the frame nor meets the rate. An implementation that accepts a signed access without a fresh challenge, or that treats the signature as a reason to skip the freshness the challenge provides, reintroduces replay and must not do so.

5.5 Client Obligations

The security of the asymmetric layer rests on work the client performs before it speaks, and this section states that work normatively. In a fieldbus deployment the constraint is on the server; the client is a manager, configurator or engineering tool with a full stack, a real trust store and a network path to the manufacturer, so the open-ended validation belongs to it and the node carries only the irreducible residue of one signature check and one fresh challenge. Before a client asserts anything to a node it shall have done the following:

- Verified genuineness. Validate the device factory identity certificate against the manufacturer trust anchor, and the signed identity read against a fresh challenge, before trusting that the device is genuine.
- Checked revocation. Confirm, out of band, the revocation status of the device and of any authority whose signature it will present, because the node runs no certificate revocation list and no OCSP.
- Validated ownership. Verify the voucher chain and the owner epoch before claiming or re-provisioning a device.
- Enforced its own minimum. Confirm that the device's declared capability level and security profile meet the client's own minimum policy, and refuse a device that declares less.
- Confirmed the expected identity. Check the device serial against the bill of materials, because the manufacturer anchor proves a genuine product, not the specific product ordered. Substituted genuine hardware passes every check the node itself can make, and only this client-side check against the expected serial catches it.
- Supplied freshness. Present a fresh challenge on every signed access, so that no signed exchange can be replayed.

Where the client delegates authority to a field tool rather than acting directly, it issues a scoped delegation as defined in the Delegation of Authority chapter of EmSA-WP-104, bound to the operations and the device serial or group the tool needs and no wider.

5.6 Segmentation of Signatures

An Ed25519 signature, and the certificate returned by the identity read, exceed the payload of a single fieldbus frame. The signed exchanges therefore use the same arbitrary-payload transport SOFA already assumes in the Prerequisites, segmented over as many frames as the payload needs, exactly as EmSA-WP-104 anticipates for provisioning traffic. The per-fieldbus binding document fixes the concrete segmentation, as it does for the nonce and tag sizes.

6 Implementation Guidance

Until SOFA is taken up by a standards body, we recommend implementing it as a manufacturer-specific security layer above the existing fieldbus stack. Doing so requires no modification to the fieldbus implementation itself: every SOFA-specific behavior lives in application code that reads and writes the same register or object dictionary slots the fieldbus stack already serves. Devices in the field can therefore add SOFA without re-validating their fieldbus stack or running a fresh interoperability test campaign.

The split between data access and function access matters for the implementation. When SOFA is used to read or write fieldbus data that already lives in a register or an object dictionary entry, the application layer gates that register's read or write through the SOFA authentication routine before completing the access. When SOFA is used to trigger a function such as entering bootloader mode, locking configuration against further fieldbus changes or transitioning into a restricted operating mode, the application layer must implement the function itself; the fieldbus stack contributes only the transport. The same SOFA exchange that protects a data write is reused to protect a function trigger; only the semantics of the underlying register or object differ.

6.1 Mandatory Objects

The object names below are generic; the actual register numbers or object dictionary indices are profile-specific. A SOFA-capable server exposes at minimum the following objects:

- Key storage:
  One secure-write-only object per base key. Implementations expose a slot for the Provisioning Key, a slot for the Integrator Key and, in deployments that use one, a slot for the Operator Key. Keys are stored in non-volatile memory and can never be read back over the fieldbus, even by an authenticated peer. The store accepts writes only under the SOFA authentication rule appropriate to the key being installed; the relevant rules are spelled out in WP-104.
- Salt slot:
  One volatile object that holds the active salt for the current power cycle. The slot reads zero immediately after reset. While it reads zero, the slot accepts exactly one unauthenticated write, after which it is locked and becomes read-only for the remainder of the power cycle. This is the slot-and-lock pattern described under Salt Slot Behavior.
- Capability descriptor:
  Read-only, unauthenticated; contents as defined in chapter "Secure Communication Configurations".

- Status descriptor:
  Read-only, unauthenticated; contents as defined in chapter "Secure Communication Configurations".
- Authenticated identity:
  Secure read-only object that returns the device's identity information (in CANopen vendor identifier, product code, revision, serial number) wrapped in a SOFA secure-read exchange, so that a client can confirm it is talking to the device it expects. This confirmation is only as strong as the key behind it: where the responding device shares its base key with other nodes, a valid identity response proves that some holder of that key answered, not that this particular device did.

> **⚠ SECURITY**
> Base keys can never be read back over the fieldbus, even by an authenticated peer. They are write-only and held in non-volatile memory; any read-back path, including a debug or diagnostic one, breaks the security model.

6.2 Salt Slot Behavior

The salt slot deserves its own paragraph because its lifecycle is unusual. From the application's point of view, the slot has three states. Immediately after reset it is empty; in this state it reads zero and accepts a single plain write. Once written, it is active; in this state it carries the per-power-cycle salt, reads back its current value to anyone who asks and refuses further writes. On reset, hard or soft, it returns to the empty state. There is no in-band command to clear the slot; if a client needs a fresh salt, it must cause a reset by a documented service action or a power cycle.

The slot is non-secret. The session key depends jointly on the slot value and the base key, and an attacker who learns the salt without learning the base key gains nothing.

The implementation references the WP-104 KDF call directly. Only three values feed it: the chosen base key, the active salt and the per-purpose context label. No other inputs are mixed in at the WP-105 layer.

> **⚠ SECURITY**
> The salt slot accepts exactly one unauthenticated write after reset, then locks read-only for the rest of the power cycle; clearing it requires a reset. The salt is non-secret, because the session key depends jointly on the salt and the base key, so an attacker who learns the salt but not the base key gains nothing.

6.3 Session Key Reuse Across Servers

The derivation deliberately does not bind any peer identifier. The same base key and the same salt always yield the same session key, regardless of which client or server runs the KDF. Where the base keys are shared across a machine, a client with constrained resources can therefore write the same salt to several servers and reuse one derived session key across all of them: one base key, one salt, one derived key, one set of session buffers on the client. This works only under shared keys; with device-unique keys each server needs its own context.

The trade-off is the loss of per-server independence: a compromise of one server exposes the session key that protects every other server the client talks to under the same salt. Where the base keys are shared across a machine and no peer identity is bound, two servers given the same salt derive the identical session key, so a compromise of one server lets it impersonate the other to the client for that power cycle. Deployments that need per-server independence must either give the client the budget for a per-server security context or bind a server-specific value, such as the node identifier, into the KDF context label so that each pairing yields a distinct session key; that value need not be secret, but the client must obtain it over a trustworthy channel because the fieldbus does not authenticate it.

6.4 Threats Against the Salt Write

The salt write is intentionally unauthenticated. An attacker on the bus can write anything to the salt slot, but only once per power cycle and only before the legitimate client has written. The resulting threats are bounded:

- Pre-positioned salt. An attacker that writes the salt before the legitimate client comes online has not learned anything useful. The derived session key is still secret because it depends on the base key the attacker does not hold. The attacker cannot forge frames, cannot decrypt traffic and cannot recover the base key. The session is unusable to the legitimate client, which is the denial-of-service case.
- Salt replay forcing nonce reuse. Within a power cycle this is prevented by construction: the slot accepts exactly one write, so an attacker cannot replay an old salt to force the server into a previously used session-key regime during that cycle. Across power cycles the salt may recur, because it is client-generated and the server applies no uniqueness check, so the salt is not required to be unique across cycles. Nonce uniqueness for the protected traffic is guaranteed by the per-frame nonce construction, not by the salt mechanism.

- Denial of service. An attacker that wins the race to write the salt locks the slot for the rest of the power cycle, so the legitimate client cannot communicate until the server is power-cycled. This deserves only a side note: any attacker with wire access to a fieldbus already has near-unlimited denial-of-service options, from frame flooding and bit-level corruption to physical disruption. The salt-write race is one more of these; it adds no new class of risk and, like the rest, is contained by the physical and operational trust boundary rather than by the protocol.

The salt write does not give the attacker any path to forgery, decryption or base-key recovery. An attacker who does not hold a base key gains only a single denial-of-service primitive per power cycle; an attacker who holds the relevant base key can additionally impersonate the intended server to the client, as noted above.

6.5 Example Code

We are preparing reference implementations of both protocols described above and intend to release them as open source. Until that release is ready, integrators who would like early access to working example code can contact us through www.em-sa.com.

6.6 Outlook to CANopen CC and CANopen FD

SOFA is deliberately stack-agnostic and is positioned as a stopgap, not as a final answer. Inside the CiA community, the HLP Cybersecurity working group is preparing a fieldbus-native security profile for both CANopen CC and CANopen FD. Where SOFA places its security entirely above the application layer so that legacy devices can adopt it without firmware changes to the fieldbus stack, the CiA profile is expected to push security one or two layers deeper, so that application authors do not need to manage cryptographic state directly. Two functions in particular are likely to move down into the CANopen stack itself in that profile:

- Protected locking and unlocking of device configuration.
- Authenticated access to configured objects without per-call application handling.

A device that ships with SOFA today will need a firmware update to take advantage of the CiA profile when it lands. The investment in SOFA is not wasted by that transition. The key model and the trust domains do not change; only the layer at which the protocol runs does. Migrating from SOFA to the CiA profile is a stack swap, not a re-key.

7 Alignment With Standards and Regulations

This chapter maps SOFA against the standards and regulations most likely to apply to a device that uses it. The alignment is at the protocol and primitive level. The provisioning model SOFA inherits from is covered in EmSA-WP-104.

7.1 ISO/IEC 9798-2

~~Section 4.2 describes FBsec as following~~The Protocol Description describes FBsec as following the shape of ISO/IEC 9798-2 Mechanism 4, a three-pass mutual authentication using random challenges, truncated to two passes so that authentication is unilateral by construction. Five deviations are documented there: the dropped third pass, the substitution of AEAD for MAC, the explicit nonce construction, the use of associated data for entity identifiers and the counter extension for cyclic continuations. TLS-PSK and cTLS sit outside the 9798 family and follow the relevant IETF specifications instead. Where an access uses the signed FBsec primitive, the AEAD tag is replaced by an Ed25519 signature over the same transcript, and a signature in each direction provides mutual entity authentication; the freshness comes from the same two-pass challenge, so the replay protection of Mechanism 4 is preserved under the signature primitive.

7.2 IEC 62443-4-2

SOFA addresses component requirements CR 3.1 (communication integrity) by mandating an authentication tag or, on a signed access, an Ed25519 signature on every secured exchange, and CR 4.1 (information confidentiality) by offering optional AEAD-level encryption negotiated per object. CR 4.3 (use of cryptography) is satisfied by AES-128-GCM as the default AEAD primitive, Ed25519 as the signature primitive and HKDF-SHA-256 as the key-derivation function. The SOFA-level additions Ascon and ChaCha20-Poly1305 are equally suitable. The capability and status descriptors expose a verifiable record of the cryptographic features each device implements and currently activates. Where a device carries the optional asymmetric identity, it further supports CR 1.2 (identification and authentication): the signed identity read and signed FBsec bind an exchange to a specific device rather than to mere key possession. IEEE 802.1AR is the reference for the two-credential identity structure the layer uses.

7.3 EU Cyber Resilience Act

Annex I requires a secure default configuration. A SOFA-capable device refuses every secured exchange until the salt slot has been written and a session key has been derived

from a base key the device holds. There is no path to a protected fieldbus object without the relevant base key, so the device cannot be coerced into a permissive default. Annex II responsibilities are unchanged from those documented in EmSA-WP-104; SOFA is a transport-security layer and inherits the lifecycle obligations from the key model.

7.4 NIS2

Article 21 names cryptography and access control among the risk-appropriate measures an essential or important entity is expected to deploy. SOFA gives the operator authenticated and optionally encrypted access to fieldbus-visible objects, with explicit role separation between integrator and operator credentials when the optional Operator Key is in use. Logging and incident response are deployment concerns and sit outside SOFA proper.

7.5 BSI TR-02102

AES-128-GCM, ChaCha20-Poly1305, Ascon and HKDF-SHA-256 are within current BSI TR-02102 recommendations for embedded systems. Tag truncation is consistent with the guidance for short, time-critical messages, bounded by the uniqueness assured by the nonce construction ~~described in section 4.2~~described under Protocol Description.

8 Figures: Sequence Diagram Sources

The diagrams in this document are produced by https://sequencediagram.org/ from the listings below.

Figure 2:

```
title Generic Fieldbus Read with Optional Segmentation
participant Client
participant Server

Client->Server: Read.req (address)
alt single-frame response
  Server-->Client: Read.cnf (data)
else segmented response
  Server-->Client: Read.cnf (segment 1, more=1)
  loop while more segments
    Client->Server: Segment.req
    Server-->Client: Segment.cnf (next segment, more)
  end
end
```

Figure 3:

```
title Generic Fieldbus Write with Optional Segmentation
participant Client
participant Server

alt single-frame transfer
  Client->Server: Write.req (address, data)
  Server-->Client: Write.cnf
else segmented transfer
  Client->Server: Write.req (address, first segment, more=1)
  Server-->Client: Write.cnf
  loop while more segments
    Client->Server: Segment.req (next segment, more)
    Server-->Client: Segment.cnf
  end
end
```

Figure 4:

```
title Secure Read with Optional Cyclic Continuation
participant Client
participant Server

==Initial transfer (single or first of cyclic), based on session key (SK)==
Client->Client: draw clnt_rnd, counter := 0
Client->Server: Write.req (clnt_rnd)
Server->Server: draw srv_rnd, counter := 0
Server->Server: nonce_base := derive(clnt_rnd, srv_rnd)\nnonce := combine(nonce_base,
counter)
Server->Server: tag := AEAD-tag(SK, nonce, AAD, data)
Server-->Client: Write.cnf
Client->Server: Read.req
Server-->Client: Read.cnf (data, srv_rnd, tag)
Client->Client: verify tag

==Optional cyclic continuation==
loop counter below counter_max, session still valid
  Client->Server: Read.req
```

```
  parallel
    Client->Client: increment counter\nnonce := combine(nonce_base, counter)
    Server->Server: increment counter\nnonce := combine(nonce_base, counter)
  parallel end
  Server->Server: tag := AEAD-tag(SK, nonce, AAD, data)
  Server-->Client: Read.cnf (data, counter, tag)
  Client->Client: verify tag
End
```

Figure 5:

```
title Secure Write with Optional Cyclic Continuation
participant Client
participant Server

==Initial transfer (single or first of cyclic), based on session key (SK)==
Client->Server: Read.req (challenge)
Server->Server: draw srv_rnd, counter := 0
Server-->Client: Read.cnf (srv_rnd)
Client->Client: draw clnt_rnd, counter := 0
Client->Client: nonce_base := derive(clnt_rnd, srv_rnd)\nnonce := combine(nonce_base,
counter)
Client->Client: tag := AEAD-tag(SK, nonce, AAD, data)
Client->Server: Write.req (data, clnt_rnd, tag)
Server->Server: nonce_base := derive(clnt_rnd, srv_rnd)\nnonce := combine(nonce_base,
counter)
Server->Server: verify tag
Server-->Client: Write.cnf

==Optional cyclic continuation==
loop counter below counter_max, session still valid
  Client->Client: increment counter\nnonce := combine(nonce_base, counter)
  Client->Client: tag := AEAD-tag(SK, nonce, AAD, data)
  Client->Server: Write.req (data, counter, tag)
  Server->Server: increment counter\nnonce := combine(nonce_base, counter)
  Server->Server: verify tag
  Server-->Client: Write.cnf
End
```

Figure 7:

```
title: Signed FBsec (Signature-Protected Access)
participant Client
participant Server

note over Client, Server: Both peers hold the other peer's local device
identity public key,\ndistributed by the integrator key management system
(KMS). A signed control-plane\naccess carries no AEAD tag and needs no ses-
sion key.

Client->(1)Server: **challenge**\nclient random rC
Server->Server: sign transcript\n(direction, rC, rS)\nwith its LDevID
Server->(1)Client: **server random rS,**\ndata, signature
Client->Client: verify server signature
Client->Client: sign transcript\n(direction, rC, rS)\nwith its LDevID
Client->(1)Server: **signature**
Server->Server: verify client signature

note over Client, Server: Mutual authentication:\na signature in each di-
rection.
```