SOFA – Secure Object Fieldbus Access

Generic secure access via Fieldbus tunnels

White Paper EmSA-WP-105

Version 1.15
1 Jun 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

Contents

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 change. 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

The aims of SOFA are:

- 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. 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.

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 ac-

tually 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.

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 three-tier base key hierarchy (Provisioning Key, Integrator Key, Operator Key), the Authenticated Encryption with Associated Data (AEAD) primitive, the HMAC-based Key Derivation Function HKDF-SHA-256 and the volatile per-power-cycle salt mechanism. SOFA does not redefine any of these; it consumes them. The session key that protects an individual SOFA exchange is derived from one of the base keys and the active salt, exactly as WP-104 specifies. The cryptographic primitives table is not repeated here; see WP-104, chapter "Cryptographic Primitives" for the authoritative list. In explicit form:

Key derivation with HKDF-SHA-256

Input:

- `IKM` = `base_key` (WP-104 tier in use: Provisioning, 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)

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.

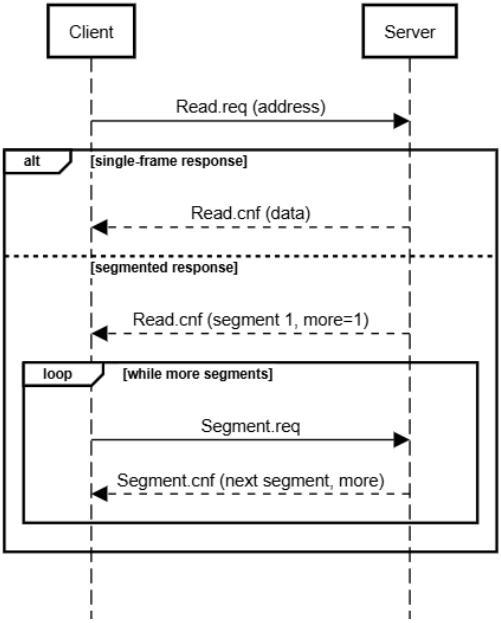

FIGURE 1: GENERIC FIELDBUS READ WITH OPTIONAL SEGMENTATION

As illustrated in Figure 1 and Figure 2, 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.

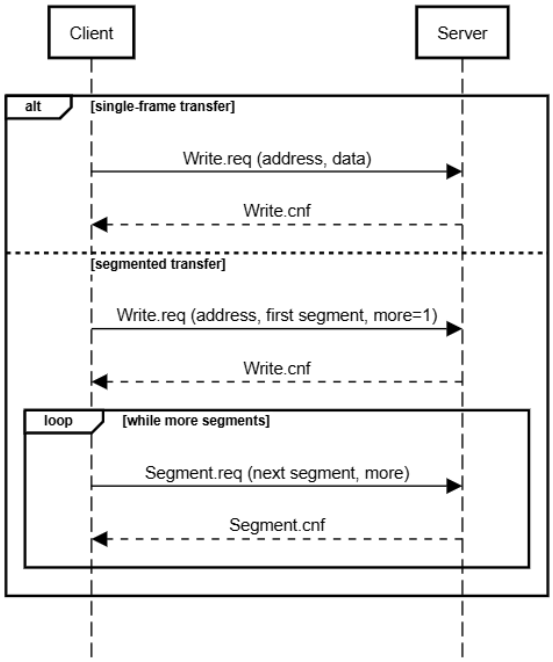

FIGURE 2: GENERIC FIELDBUS WRITE WITH OPTIONAL SEGMENTATION

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:

- 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 and optionally Ascon or ChaCha20-Poly1305 (the latter two are SOFA-level options, not WP-104 inheritances).
- Asymmetric key and algorithm support, where present.
- Session protocols supported: TLS 1.3, TLS-PSK, cTLS or SOFA.
- Certificate support, where applicable.

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.

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 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 may implement either protocol or both. The capability descriptor in the previous chapter announces which one is available. Both protocols derive the on-wire session key from one of the WP-104 base keys and the active 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 (Provisioning, 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 is torn down 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 salt slot and forces a new session on next access.

4.2 Per-Transfer: FBsec

For deployments that cannot afford to keep TLS state per client, SOFA offers FBsec (FieldBus security), a per-transfer challenge-response protocol patterned after the random-challenge mechanism of 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 IEC 9798-2 Mechanism 4 – a three-pass mutual authentication using random challenges – truncated to two passes so that authentication is unilateral 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) (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) (iii) the AEAD nonce is built from a base value and a strictly monotonic counter, a construction 9798 leaves open;
(iv) (iv) entity identifiers and addressing context are carried in the AEAD's Associated Data rather than concatenated into a MAC input;
(v) (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 3 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.

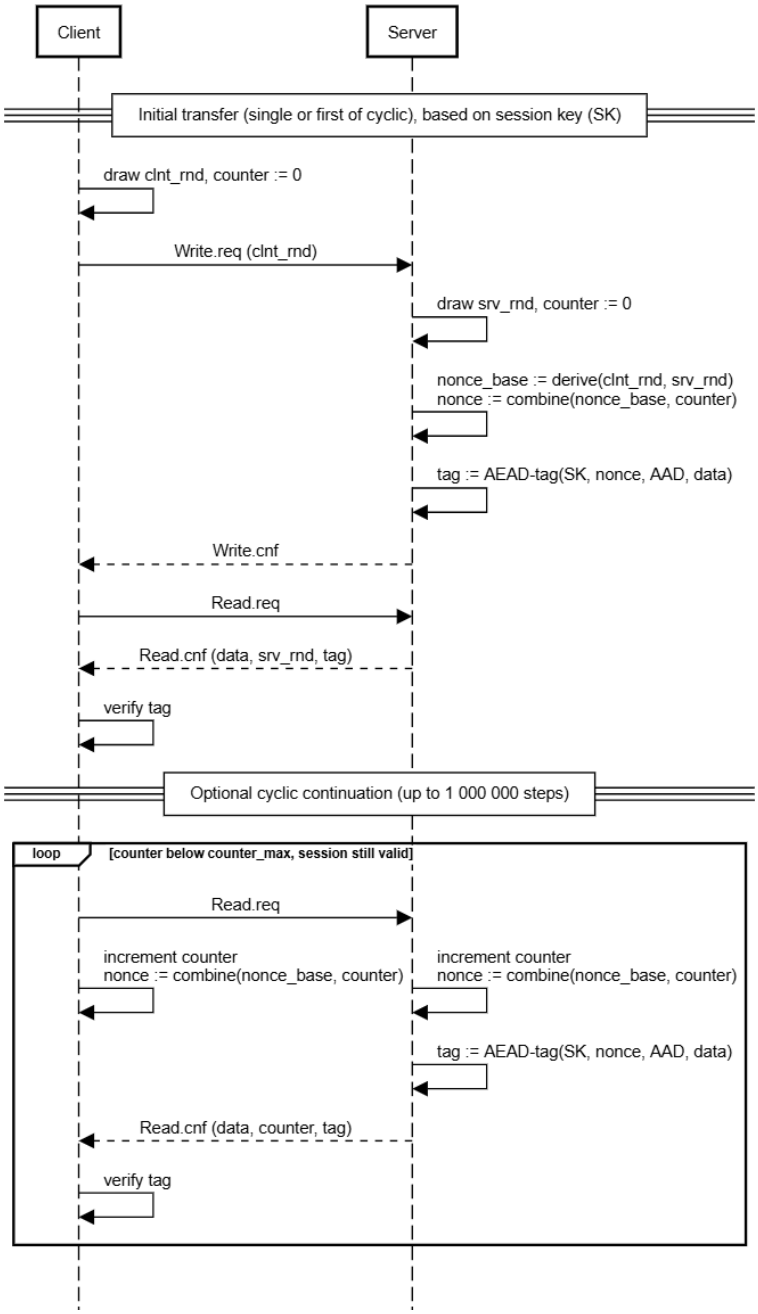

FIGURE 3: SECURE READ WITH OPTIONAL CYCLIC CONTINUATION

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 4 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.

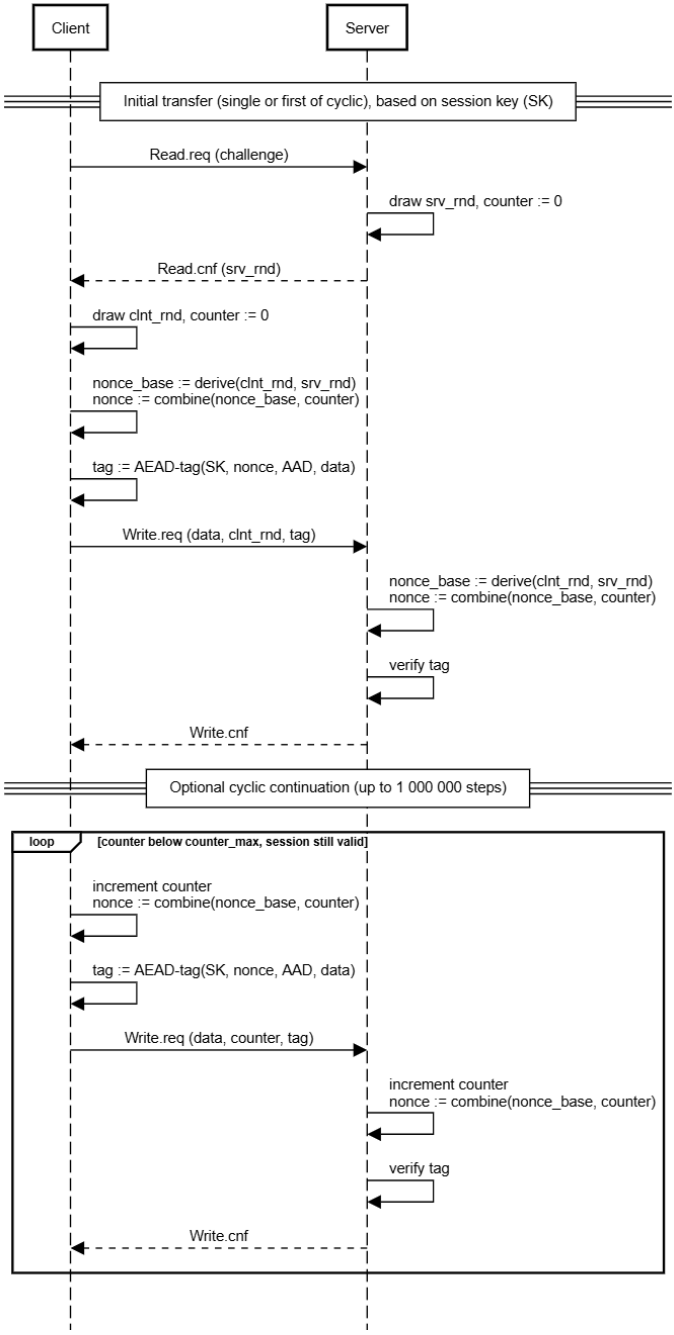

FIGURE 4: SECURE WRITE WITH OPTIONAL CYCLIC CONTINUATION

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 (recommended 1 000 000) 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

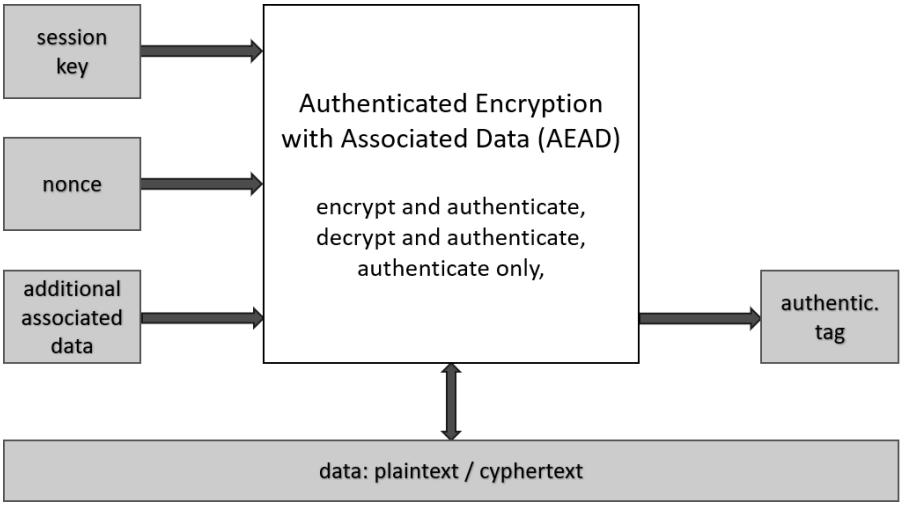

FIGURE 5: AEAD INPUTS AND OUTPUTS

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

profile choice; this document gives a working example below and a per-fieldbus binding document picks 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 32-bit 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 (32 to 64 bits acceptable on both classical CAN and CAN FD; longer tags are an option for deployments that need a wider per-key usage budget).

> **⚠ SECURITY**
> Authentication is mandatory on every secured access. Encryption is optional and negotiated per object, but an exchange without a valid authentication tag 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, preferably 96 bits each where constraints allow. For support of cyclic accesses, a counter of preferably 32 bits is used. All three values need to be mixed together to form a nonce of 96 bits for the AEAD input.

As an example, two random values of 96 bits are combined by XOR to form a 96-bit nonce base:

```
nonce_base = client_random_96 XOR server_random_96
```

The 32-bit message counter is then merged into the low 32 bits of the base:

```
nonce = nonce_base XOR (0^64 || counter_32)
```

For a single transfer the counter is zero, so the working nonce equals the XOR of the two randoms. For cyclic continuations the two random values stay unchanged and the counter increments by one per step, so each continuation produces a fresh nonce without re-running the challenge.

Implementations can hold the per-session nonce state as just two values: `nonce_base` (96 bits, computed once at session start) and the current counter (32 bits). The original client and server randoms can be discarded the moment `nonce_base` has been computed, since neither value is used again – the session key `SK` is derived from the base key and the active salt, not from the randoms. This holds the per-session nonce state to 16 bytes for the full lifetime of the cyclic stream.

An implementation may transmit only the low byte of the counter in each cyclic-continuation response, since the receiver tracks the full 32-bit value locally and uses the low byte as a sanity check. The recommended session limit is 1 000 000 cyclic continuations, after which a fresh challenge must establish new randoms.

This wording is illustrative. A per-fieldbus profile may pick smaller random sizes, a different mix function or a different counter width, as long as the resulting nonce is unique per (`SK`, message) pair across the session.

> **⚠ 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 96-bit randoms used above are sufficient, and a profile that shortens them must justify the smaller width against this bound.

5 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 simply 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.

5.1 Mandatory Objects

A SOFA-capable server should expose at minimum the following fieldbus-visible objects. The names are generic; the actual register numbers or object dictionary indices are profile-specific.

- 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 specified in WP-104.
- 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.

5.2 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.

5.3 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.

5.4 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.

6 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.

6.1 ISO/IEC 9798-2

Section 4.2 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 follows the relevant IETF specifications instead.

6.2 IEC 62443-4-2

SOFA addresses component requirements CR 3.1 (communication integrity) by mandating an authentication tag 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 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.

6.3 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.

6.4 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 authenti-

cated 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.

6.5 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 to 32 to 64 bits is consistent with the guidance for short, time-critical messages, bounded by the uniqueness assured by the nonce construction described in section 4.2.

7 Figures: Sequence Diagram Sources

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

Figure 1:

```
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 2:

```
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 3:

```
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 (up to 1 000 000 steps)==
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 4:

```
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 (up to 1 000 000 steps)==
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
```