

EmSA's Open Source FBsec CANopen Implementation

SOFA – Secure Object Fieldbus Access – for
CANopen FD

User Manual EmSA-UM-105-COP

Version 1.001

1 June 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	4
2	Prerequisites.....	5
3	Building from Source	6
4	FBsec CANopen FD Capabilities	7
5	Secure Object Dictionary Entries	8
5.1	Secure Object Entries Provided.....	8
5.2	Object Access Implementation	8
6	Running the Examples	10
6.1	Key File Format	10
6.2	Optional CAN FD bridge to a real network.....	11
6.3	Reading the trace output	11
6.4	Shutting down	12
7	Integration Guide.....	13
7.1	Port Hooks.....	13
7.2	Integrator Entry Points.....	14
7.3	Key Model	14
7.4	Master Keys vs Session Keys	16
7.5	Provisioning Workflows	16
7.6	MCU Key Storage Options	17
7.7	RNG Hook Requirements	18
7.8	Key Rotation Policy	19
7.9	OD Entry Placement	20
7.10	USDO Transport Mapping	21
7.11	Callback Wiring.....	22
7.12	Abort Code Mapping.....	23
7.13	Cyclic Mode Notes	23

8	Threat Model	25
---	--------------------	----

1 Scope

SOFA (Secure Object Fieldbus Access) demonstrates application-layer secure read and write of CANopen FD object dictionary entries on top of unmodified USDO (Universal Service Data Object) transfers. The package ships source code for a CANopen FD bus simulator, a server and a client, all running on a Windows PC and driven primarily from the command line. An optional bridge to a PEAK PCAN-Basic interface lets the simulated bus carry traffic onto and off of a real CAN FD network for live tracing or interaction.

Key handling in the simulation follows the WP-104 vocabulary but skips the full lifecycle. WP-104 specifies that each layer (Provisioning, Integrator, Operator) holds a master key and per-session keys are derived per access via `HKDF(layer_master, salt, info)`. This implementation skips the derivations: what is loaded by `--key-file` and used on the wire are the already-derived session keys for each role.

Real embedded targets must implement the full lifecycle (state machine, NV storage, authenticated install / overwrite, factory restore); this simulation leaves those concerns to the integrator's port layer.

SECURITY

This package is a simulator and demonstrator, not a secure product. It follows the WP-104 vocabulary but skips the key lifecycle: it loads already-derived session keys from a file and implements no provisioning state machine, no non-volatile key storage, no authenticated key install or overwrite and no factory restore. A real embedded target must implement that lifecycle in its port layer. Do not ship the simulator key handling as production security.

2 Prerequisites

Read first:

- EmSA-WP-105 "Secure Object Fieldbus Access" — the conceptual model and protocol shape SOFA implements.
- EmSA-WP-104 "Key Provisioning for Minimal Fieldbus Systems" — the key lifecycle that the simulator skips but that a real deployment must implement.

To build and run:

- Windows 10 or 11.
- CMake 3.20 or later.
- MSVC 14.50 (Visual Studio 2026), /W4 /WX /permissive- clean. The bundled executables statically link the MSVC C runtime so they run on any Windows 10+ machine without the Visual C++ Redistributable.
- mbedtls is vendored in-tree under `third_party/mbedtls/` (a 7-file subset: AES, GCM, SHA-256, plus glue). No separate mbedtls install is required.

Optional, for live CAN FD bridging:

- PEAK PCAN-Basic Windows driver (PEAK-Drivers 5 or newer).
- A CAN FD-capable PEAK interface (PCAN-USB FD, PCAN-USB Pro FD, PCAN-PCI Express FD; classical-CAN-only adapters are not sufficient).
- PCANBasic.dll is resolved at run time via LoadLibrary. Machines without it still build and run the simulator; only the bridge feature is skipped.

Cross-fieldbus portability. The variant-agnostic layers, namely `shared/` (crypto and the state machine), `client_common/`, `server_common/` and `bus_common/`, are written so a future variant (CANopen CC, EtherCAT, ...) can drop into `variants/<name>/` without touching the common code.

3 Building from Source

The package ships prebuilt executables, so running the demo needs no build. To rebuild from source, or to port the code to another target, configure and build with CMake 3.20 or later and MSVC. With the Visual Studio 2026 generator, run `cmake -S . -B build -G "Visual Studio 18 2026" -A x64`, then `cmake --build build --config Release`; the bus, server and client executables appear under `build\`.

Compile-time options live in two places. The CMake cache variables, set with `-D` at configure time, are `FBSEC_FEATURE_READ`, `FBSEC_FEATURE_WRITE` and `FBSEC_FEATURE_CYCLIC` (each 0 or 1, default 1, with at least one of read or write set) and `FBSEC_AEAD_DEV_ID_SIZE` (1 for the CANopen FD node-id AAD layout, 2 reserved for future variants). The AEAD algorithm and tag length are not CMake variables; they are edited in `shared/fbsec_config.h`: set `FBSEC_AEAD_AES256_GCM` to 1 and `FBSEC_AEAD_AES128_GCM` to 0 for AES-256, and change `FBSEC_AEAD_TAG_LEN_BYTES` (default 8, range 4 to 16 for AES-GCM) for the tag length. `FBSEC_AEAD_ENCRYPTION` selects encrypt-and-authenticate (1, the default) or authenticate-only (0).

4 FBsec CANopen FD Capabilities

WP-105 fixes the shape of secure object access but leaves the specific cryptographic primitive, key and tag sizes, and on-wire AAD layout to the implementer. SOFA's CANopen FD variant selects:

- AEAD primitive: AES-128-GCM (NIST FIPS 197 + SP 800-38D). 16-byte session key, 12-byte GCM nonce, 8-byte truncated tag (SP 800-38D Appendix C). AES-256-GCM is alternatively selectable at build time by editing `shared/fbsec_config.h` (set `FBSEC_AEAD_AES256_GCM` to 1 and `FBSEC_AEAD_AES128_GCM` to 0), which makes the session key 32 bytes.
- KDF: HKDF-SHA-256 (RFC 5869) with info string "FBSEC-SK-v1"\0 || keyid (13 bytes). The derivation lives in the integrator's port layer; the simulator installs already-derived session keys directly.
- Three role-bound session keys: keyid 1 = Provisioning Session Key, keyid 2 = Integrator Session Key, keyid 3 = Operator Session Key (read-only under default policy).
- Wire keyid byte: bit 7 = encryption flag, bit 6 = cyclic-arm flag, bits 5..4 reserved (must be 0), bits 3..0 = base key id (1..15). The full byte is in the AAD prefix so downgrade attempts (strip encryption, flatten cyclic, swap keyid) fail the tag verify.
- Associated Data (AAD) prefix: 12 bytes on the CANopen FD variant (`FBSEC_AEAD_DEV_ID_SIZE = 1`): four fixed bytes (protocol version, mechanism, direction and the wire keyid byte, which therefore sits at offset 3), then the 1-byte server and client node ids, a 4-byte data_id and a 2-byte data length, matching WP-105's "client node ID, server node ID" wording for CANopen FD.
- Entity authentication: ISO/IEC 9798-2 Mechanism 4 truncated to two passes (the third pass that would close mutual auth is dropped, leaving unilateral authentication, in which the party that verifies the tag, the client on a read or the server on a write, authenticates the data as coming from a holder of the shared session key; both peers still contribute a fresh random per single-shot transfer, XOR-mixed to form the GCM nonce).
- Per-(key, session) frame budget: `FBSEC_AEAD_KEY_USE_LIMIT = 1 000 000` frames per cyclic session, well under the SP 800-38D 2³² invocations-per-key guidance for the 64-bit truncated tag. Reaching the limit forces re-arming with a freshly rotated key.
- Cyclic mode (§11.1 of the secure-tunnel spec): the cyclic-capable single read or write arms a session that survives up to 60 s of idle time and is separately capped at the `FBSEC_AEAD_KEY_USE_LIMIT` total-frame budget.

5 Secure Object Dictionary Entries

5.1 Secure Object Entries Provided

The following table shows the four secure object dictionary entries implemented.

Index, Sub	Access	Size	Purpose
0x2010, 0	SECURE_WO	4 bytes	Write target; the value written mirrors into 0x2020,0.
0x2020, 0	SECURE_RO	4 bytes	Auto-incrementing 32-bit counter (increments after each read).
0xC016, 0	SECURE_WO	16 bytes	Free-form 16-byte write target (last write kept).
0xC018, 0	SECURE_RO	16 bytes	Identity pattern: device_id (big-endian) followed by 0x02..0x0F.

5.2 Object Access Implementation

The reference server implements four secure entries against four operation modes.

- SRD (Secure Read)
- SWR (Secure Write)
- SRDPOLL (Secure Read, continuous polling)
- SWRPOLL (Secure Write, continuous polling)

Two checks gate every secure request that reaches one of these entries.

First, the AEAD verification confirms that the request carries a valid authentication tag computed with one of the loaded session keys; a forged or tampered request fails here. (An entry can optionally insist on a single specific session key at this layer, for example an entry that only accepts the Provisioning Session Key, but the four demo entries do not, so any of the three role keys passes the AEAD step.)

Second, the server consults its role-policy function, which decides whether that role is allowed to do that operation on that address. This implementation default policy: the Provisioning Session Key and the Integrator Session Key may read and write all four entries; the Operator Session Key may read but not write. A secure write attempted under the Operator role therefore passes the cryptographic check cleanly but is then refused

with abort code `0x06010000` (UNSUPPORTED). This is a deliberate refusal at the policy layer, not a crypto failure. Integrators replace this single function (`fbsec_sod_port_role_allowed()`) to install their own per-entry, per-role mapping without touching any of the cryptographic code. The two gates emit different aborts: an entry-level lock or write-protect (the access-allowed hook) returns `LOCKED` (`0x08000022`), while a role-policy denial (the role-allowed hook) returns `UNSUPPORTED` (`0x06010000`).

A note on the four operations: The `SRD` / `SWR` / `SRDPOLL` / `SWRPOLL` distinction is a client-side choice, not a property of the entry: every read-capable entry in this demo supports both single-shot reads (`SRD`) and cyclic poll-reads (`SRDPOLL`), and every write-capable entry supports both `SWR` and `SWRPOLL`. `SRDPOLL` is not a separate kind of entry; it begins with the same two-pass challenge-response that a single `SRD` uses, with one extra bit in the first request that asks the server to arm a session for follow-up polls against that address. The cyclic session survives until 60 s of idle time elapse, or until the per-(key, session) frame budget of 1 000 000 frames is reached, at which point a fresh arming exchange is required.

6 Running the Examples

Three launcher batches live under `run\`:

- `start_fd_hub.bat`
starts the simulated CAN FD bus (TCP listener on 127.0.0.1:5810; optionally bridges to a PCAN-FD interface via the auto-detect prompt at start-up).
- `start_fd_server.bat`
starts a CANopen FD server, loading session keys from `run\keys-demo.txt`.
- `start_fd_client.bat`
starts the client in interactive `--menu` mode, also loading session keys from `run\keys-demo.txt`.

At start-up the client prints how many session keys were loaded from the key file, then prompts for own node id, target node id, encryption mode (encrypt+auth or auth-only / MAC), and the role to use for this session (Provisioning, Integrator, or Operator Session Key).

The menu then offers six options:

- Single 16-byte secure read (`srd 0xC01800`).
- Single 16-byte secure write (`swr 0xC01600`).
- Single 4-byte secure read (`srd 0x202000`).
- Single 4-byte secure write (`swr 0x201000`).
- 300 cyclic 4-byte reads on `0x202000` at 200 ms cadence (one arming `srd + 299` poll-reads against the same session).
- 300 cyclic 4-byte writes on `0x201000` at 200 ms cadence.

Bus, server and client each emit a color-coded trace on stdout (blue = request, magenta = response, red = abort, grey = metadata). Per-(key, session) frame use is hard-capped at `FBSEC_AEAD_KEY_USE_LIMIT = 1 000 000` regardless of menu choice; reaching the limit tears the cyclic session down and the client must re-arm with a fresh challenge.

6.1 Key File Format

The server and the client load session keys from a text file given with `--key-file` (the launchers use `run\keys-demo.txt`), or take a single key inline with `--key` or `--main-key`. The two inline flags differ: `--key`, after `--keyid N`, installs an already-derived session key directly into slot `N`, while `--main-key` sets a per-device master key from which session keys are derived together with `--salt`, mirroring the master-plus-salt provisioning pattern. The file holds one row per key; blank lines and lines starting with a hash are ignored. Each row is the key id, a free-form role label and the key as hex bytes, separated by

whitespace. The key id runs from 1 to 15 (the demo uses 1, 2 and 3 for the Provisioning, Integrator and Operator Session Keys); the role label is informational, since the parser keys on the id; the hex value is the session key, 16 bytes for AES-128-GCM or 32 bytes for AES-256-GCM. For example, the row 1 prov-session 00112233445566778899AABBCCDDEEFF installs a 16-byte key as key id 1.

6.2 Optional CAN FD bridge to a real network

With no arguments, the bus probes for a PCAN device at start-up. If a CAN FD-capable channel is found in the AVAILABLE or PCANVIEW state, the bus prints a prompt of the form "PCAN device on PCAN_USBBUS1 detected. Connect at 500k/2M? (y/n):". Answering y bridges every USDO frame seen on the simulated bus onto the real CAN FD network at 500 kbit/s arbitration / 2 Mbit/s data, and broadcasts every CAN FD frame received from the real bus back to all simulated peers. A hardware CANopen FD server or client can join the simulated mesh on equal footing.

Simulator-internal control frames (peer announce, peer loss, frame error; CAN IDs in the $0 \times 1 \text{FFFFFFEx}$ range) are not bridged; they have no meaning on a physical bus.

Useful command-line flags accepted by the bus:

- `--pcan PCAN_USBBUS1` bridges non-interactively to the named channel. Accepts symbolic names (`PCAN_USBBUS1..8`, `PCAN_PCIBUS1..8`) or hex literals (`0x51`, `0x41`, ...).
- `--no-pcan` skips the auto-probe prompt entirely.
- `--port N` selects an alternate TCP port (default 5810).
- `--color / --no-color` force ANSI color on or off.
- `--quiet` suppresses the trace on stdout.
- `--help` prints usage and exits.

If the detected PCAN device is classical-CAN only (no FD), the bus prints "device(s) detected but none support CAN FD" and continues in TCP-loopback-only mode. SOFA USDO frames carry up to 64 payload bytes and are too long for classical CAN.

6.3 Reading the trace output

Each program emits a color-coded trace on stdout. The direction tints are: blue for requests (TX from client, RX at server), magenta for responses (TX from server, RX at client), red for aborts, and grey for metadata (peer announce/loss, ACK envelopes).

The bus trace columns are:

- frame number

- timestamp
- [src->dst] node-id pair, 2-digit hex
- CAN ID
- payload hex

The client and server per-request traces use a similar layout:

- [timestamp] TX or RX
- verb tag (srd / swr / poll variant)
- src->dst node ids, 2-digit hex
- [data_id] 32-bit identifier in hex
- payload hex, followed by a Plain: ... line on the receiving side that shows the decrypted plaintext

The first byte after the verb tag in every client request that carries one is the wire keyid byte. Its four fields are:

- bit 7 - encryption flag (0 = authenticate only, 1 = encrypt and authenticate)
- bit 6 - cyclic-capable (0 = single-shot, 1 = arm session for follow-up polls)
- bits 5..4 - reserved, must be 0
- bits 3..0 - base key identifier (1 = Provisioning Session Key, 2 = Integrator Session Key, 3 = Operator Session Key)

Worked examples that appear in the bus trace:

- 0x82 - encrypt + Integrator Session Key, single-shot
- 0xC2 - encrypt + Integrator Session Key, cyclic-capable single
- 0x01 - auth-only + Provisioning Session Key, single-shot
- 0xC1 - auth-only + Provisioning Session Key, cyclic-capable single

6.4 Shutting down

Press Q in the client menu to quit the client, then Ctrl-C in the server and bus windows. The bus uninitializes any open PCAN channel automatically on Ctrl-C; no manual cleanup is needed.

7 Integration Guide

This chapter is the porting guide for engineers taking SOFA off the Windows demonstrator and onto a real CANopen FD device. It covers the two surfaces an integrator wires up: key handling (where keys come from, how they get into the device, and where they live on the MCU) and the Object Dictionary side (where secure entries sit, how the two-pass srd/swr flow maps onto USDO, and how aborts surface).

The cryptographic protocol itself (AAD construction, nonce derivation, AEAD direction tags and the wire keyid byte semantics) is out of scope here; that contract is fixed in `fieldbus_sim_secure_tunnel_spec.txt` and is what every variant in the SOFA family must honor. The address mapping between SOFA's (`device_id`, `data_id`) and CANopen's (`node id`, `index`, `subindex`) is documented in `fieldbus_sim_canopen_fd_spec.txt`; the protocol-flow visualization is in `flow_diagrams.txt`. These three reference files ship in the package `doc/` folder alongside this manual. The reader is assumed to have AEAD basics (key/nonce/tag, why nonce reuse is fatal under GCM) and CANopen FD experience with a USDO-capable stack.

7.1 Port Hooks

Integration into a real device's stack happens through a small set of port hooks. The server-side dispatch in `shared/fbsec_secure_od.c` calls these hooks; the variant adapter never touches the AEAD or OD bookkeeping directly. The seven hooks the server requires:

- `uint16_t fbsec_sod_port_get_device_id(void)` - returns the local CANopen node id (1..127).
- `uint16_t fbsec_sod_port_get_time_ms(void)` - millisecond clock used for cyclic-session idle timeout.
- `bool fbsec_sod_port_random(uint8_t *buf, uint16_t len)` - fills `buf` with `len` bytes of cryptographic-quality random for `server_random`; returns false on a random-source failure, which the server surfaces as `FBSEC_SOD_ABORT_GENERAL`.
- `bool fbsec_sod_port_access_allowed(fbsec_sod_op_t op, uint32_t data_id)` - application-level access gate (lock policy, write-protect); returns false to abort with `LOCKED`.
- `uint32_t fbsec_sod_port_read_before(uint32_t data_id, uint8_t *dst, uint16_t *len)` - fills `dst` with the plaintext for a `SECURE_RO` entry just before AEAD seal; `*len` is the maximum on entry and the actual length on return; returns 0 or an abort code.
- `uint32_t fbsec_sod_port_write_after(uint32_t data_id, const uint8_t *src, uint16_t len)` - commits `len` verified plaintext bytes for a `SECURE_WO` entry just after AEAD open; returns 0 or an abort code.

- `bool fbsec_sod_port_role_allowed(fbsec_sod_op_t op, uint8_t key_id_base, uint32_t data_id)` - per-keyid role policy (default-allow read; default-deny write for the Operator Session Key); returns false to abort with `UNSUPPORTED`. The `op` argument is the enum `fbsec_sod_op_t` (`FBSEC_SOD_OP_READ = 0`, `FBSEC_SOD_OP_WRITE = 1`).

The client side requires a single hook, `bool fbsec_secure_port_random(uint8_t *buf, uint16_t len)` (declared in `shared/fbsec_secure_proto.h`), which fills `buf` with `len` bytes of `client_random` for each secure verb under the same contract as the server random hook. Key storage in the simulator: slots 1..15 live in RAM only and are populated at start-up from `--key / --main-key` arguments or from `run/keys-demo.txt`. A real target replaces this with one of the storage topologies described in MCU Storage Options below.

7.2 Integrator Entry Points

Where the hooks above are functions the secure-OD calls into the host, the host drives the secure-OD through a small API declared in `shared/fbsec_secure_od.h` and `shared/fbsec_hkdf.h`:

```
void fbsec_sod_init()
```

Resets the registry, key slots and challenge state.

```
bool fbsec_sod_register_entry()
```

Registers a secure entry. The `fbsec_sod_entry_t` struct carries `data_id`, `key_id` (1 to 15, or `FBSEC_SOD_KEY_NONE` for any), `access_flags` (`FBSEC_SOD_ACCESS_SECURE_RO` or `FBSEC_SOD_ACCESS_SECURE_WO`), `value_type` and `data_len`.

```
bool fbsec_sod_set_key()
```

Installs a session key into a slot and refuses an already-populated slot.

```
fbsec_sod_status_t fbsec_sod_dispatch()
```

Handles one inbound request and fills either the reply payload or `out_abort`.

```
bool fbsec_hkdf_sha256()
```

Derives session keys for hosts that derive them rather than install them directly.

7.3 Key Model

Implementation note. SOFA is a simulator. On a real embedded target, WP-104 section 3.4 specifies that each layer (Provisioning, Integrator, Operator) has a master key, and for each communication a per-session key is derived via `HKDF(layer_master, salt,`

info). SOFA simulates both the masters and the derivation step out: what the CLI loads, what `fbsec_sod_set_key()` installs, and what the wire keyid byte selects are the already-derived session keys.

The wire keyid byte (defined in `shared/fbsec_aead.h`) is one byte carrying four fields. The whole byte is baked into the AAD prefix at offset 3, so any downgrade attempt fails AEAD verification:

- bit 7 - encryption flag (0 = authenticate only / plaintext on the wire, 1 = encrypt-and-authenticate)
- bit 6 - cyclic-arm flag (meaningful only on `READ_CHALLENGE` / `WRITE_CHALLENGE`; ignored on the poll directions)
- bits 5..4 - reserved, must be 0 (verifier rejects non-zero with `FBSEC_SOD_ABORT_UNSUPPORTED` so future bit assignments do not silently collide with older peers)
- bits 3..0 - base key identifier (1..15)

Macros in `shared/fbsec_aead.h` extract the fields: `FBSEC_AEAD_KEYID_ENCRYPT`, `FBSEC_AEAD_KEYID_IS_CYCLIC`, `FBSEC_AEAD_KEYID_RESERVED`, `FBSEC_AEAD_KEYID_BASE`, and the constant `FBSEC_AEAD_KEYID_MAX = 15`. The protocol version is `FBSEC_AEAD_PROTOCOL_VERSION = 0x01`: a single in-the-field revision, no version history to be backward-compatible with.

Per-data_id key binding. The server's secure-OD registry binds each registered entry to a numeric key id. At dispatch time the server compares the wire keyid's base 4 bits against `entry->key_id` and aborts with `FBSEC_SOD_ABORT_UNSUPPORTED` on mismatch. The sentinel `FBSEC_SOD_KEY_NONE = 0x00` means "this entry accepts any provisioned key", useful for status / capability objects. Every other registered entry is bound to exactly one key id; that binding is the policy hook. Change the binding and you change which role can access what, without touching crypto code.

Role conventions inherited from `MCOPSecureAccess` and used in the demo: keyid 1 = Provisioning Session Key (factory / integrator install, read + write); keyid 2 = Integrator Session Key (site-deployment, read + write); keyid 3 = Operator Session Key (runtime operation, read only); keyid 4..15 are available for application use. The cryptographic core does not interpret these numbers - only the role hook in `server_common/server_common_hooks.c` does. Replace the hook to install a different mapping (per-entry roles, multi-key bitmaps, lock-state-aware policy); no changes to the OD library or wire protocol are needed.

7.4 Master Keys vs Session Keys

SOFA is wire-agnostic about derivation but ships a derivation helper for hosts that prefer not to store final session keys directly. The derivation is HKDF-SHA-256 with `IKM = master_key`, `salt = session_salt` (host-set), `info = "FBSEC-SK-v1" || NUL || keyid` (13 bytes), and output length `L = FBSEC_AEAD_KEY_SIZE` (16 or 32 bytes). The info string is exactly the literal "FBSEC-SK-v1" (11 ASCII chars) plus its trailing NUL plus the wire keyid byte; either side that derives keys must use the same info string and salt. A mismatch surfaces cleanly as "tag verify failed" on the first secure verb.

Two install patterns exist at the API level:

- Direct session-key install. Host code calls `fbsec_sod_set_key(keyid, key)` once at boot per role, with the fully-derived session key bytes. HKDF is bypassed. Appropriate when keys are stored in a per-device secure store (factory-injected at production) or are pushed by an operator-controlled provisioning tool that already did the derivation off-device.
- Master-plus-salt derivation. Host code holds a per-device master key (often in OTP or a secure element), reads a salt from operator-supplied configuration on boot, calls `fbsec_hkdf_sha256()` to derive the session key per role, and then `fbsec_sod_set_key()` to install it. Useful when a single master is provisioned per device and per-deployment salts rotate the session key without re-touching the master.

Both sides treat each key slot as write-once-per-process: `fbsec_sod_set_key()` refuses to overwrite a populated slot. To rotate, tear down the process or session and provision again.

7.5 Provisioning Workflows

Three patterns cover most fielded use cases; pick one or combine.

Factory inject. Per-device session keys are programmed at production time and installed into the device's persistent store (flash, OTP or secure element). On boot the firmware reads the keys back and calls `fbsec_sod_set_key(keyid, key)` once per role. Simplest model; no master to leak. Cost: keys must be uniquely generated and tracked per device, and field re-key requires a physical operation or a separately-secured channel that itself uses another key. Often the right pattern for small, low-rotation deployments.

Derive-from-master. Each device holds a single per-device master key. Session keys used on the wire are derived at boot as `session_key_i = HKDF-SHA-256(master, device_id || salt_i, info_i, L)`. Only the master ever leaves the factory floor; the salt is rotated independently (the deployment toolchain pushes a new salt each maintenance window).

Pros: small per-device persistent footprint; session keys rotate without re-provisioning the master. Cons: master compromise rotates every session key derived from it.

Scheduled re-key. Session keys are swapped on a planned cadence. The device tears down its secure-OD session, re-runs whichever provisioning path applies (factory inject or derive-from-master), and resumes. Both client and server treat key slots as write-once-per-process, so a session-tear-down boundary is the natural rotation boundary. Hot-reload (key swap without a session tear-down) is deliberately not implemented: rotating a key while a cyclic-mode session is in flight risks tag forgery on the first frame after the swap. Deployments that need sub-second rotation should model it as a re-arm: tear down the cyclic session, install the new key, arm a fresh session.

7.6 MCU Key Storage Options

SOFA's port-hook surface (`fbsec_sod_set_key`, `fbsec_sod_port_random`, `fbsec_sod_port_get_device_id`) deliberately does not mandate where keys live on the target. The host owns the storage decision. Five common topologies, with what SOFA needs from each:

- **Volatile SRAM only.** Keys are derived or fetched at boot and held in SRAM for the process lifetime. Power cycle wipes them; provisioning is required on the next boot. Use when keys are derived from a hardware root of trust at boot (e.g. PUF + HKDF), or when the device is paired to an attested trusted controller that pushes keys after each reset. Caveat: any debug-port or DMA-readback exposure during the live session is the host's problem; SOFA keeps the keys in C arrays inside `g_keys[]` (private to `shared/fbsec_secure_od.c`).
- **Internal MCU flash.** Keys live in a dedicated flash region read on boot. Cheapest persistent option; mandatory protection is to enable the MCU's read-out-protection (RDP / CRP / lock bits) so a JTAG / SWD attacker cannot dump the page. On parts that can fall back to factory-reset state, also enable the anti-rollback fuse if available. Caveat: a single bad write cycle can corrupt the slot; treat the region as a small append-only store with CRC and a generation counter, or write a known-good "boot key" alongside the active one and prefer the boot key on CRC fail.
- **OTP / fuses.** One-time programmable region (Cortex-M, Renesas RA, ESP32, etc.). Truly write-once, factory-set. Suits a master key in the derive-from-master pattern: provision the master once at production, derive everything else at boot. Caveat: capacity is small (often 1 to 4 keys); a master leak is permanent; do not put session keys you intend to rotate in OTP.
- **External secure element / TPM-equivalent.** Separate die (ATECC608, OPTIGA, Microchip TA100, ST33) over I2C / SPI. The key never leaves the chip; the application asks the secure element to perform the AEAD operation. Strongest topology for high-assurance deployments. Integration cost: the SOFA AEAD shim

(`shared/fbsec_aead.c` calling `mbedtls_gcm_crypt_and_tag`) becomes a remote call through the secure-element's command set. Replace the body of the AEAD seal/open with the SE's GCM primitives; keep the AAD construction and `fbsec_aead.h` API intact so callers do not change.

- TrustZone-isolated SRAM (Cortex-M23/M33 and similar). Keys live in a Secure-world SRAM region; the Non-secure-world application calls a secure-side service for every AEAD operation. Practical sweet spot when the silicon already has TrustZone but no separate secure element: the integrator pays the cost of a thin secure-world driver and gains key-isolation against a compromised application. The HKDF in `shared/fbsec_hkdf.c` should also live in the secure world if session keys are derived at boot.

7.7 RNG Hook Requirements

SOFA consumes randomness on both peers, mutually-contributed per single-shot transfer ($R = \text{FBSEC_AEAD_RANDOM_SIZE}$, default 12).

- Server: every `READ_CHALLENGE` response and every `WRITE_CHALLENGE` issues a fresh `server_random[R]` via `fbsec_sod_port_random()` (the Windows reference impl in `server_common/server_common_hooks.c` uses `CryptGenRandom`).
- Client: every `srd` Pass 1 generates `client_random[R]`; every `swr` Pass 2 also generates a fresh `client_random[R]`, sent alongside the keyid byte and cipher-text+tag. The reference port hook lives in `shared/fbsec_secure_proto.h` as `fbsec_secure_port_random()`; the Windows reference impl is in `client_common/client_common_platform.c`.
- Per-frame GCM nonce: `nonce[NS] = client_random[0..NS-1] XOR server_random[0..NS-1]`, where $NS = \text{FBSEC_AEAD_NONCE_SIZE} = 12$. The nonce stays unique and unpredictable as long as at least one side draws a sound random, one that is unpredictable and does not repeat within the key-use budget; XOR with such a value carries the freshness even when the other side is weak, and only a simultaneous collapse of both RNGs degrades the nonce space. This assumes both peers are honest; an adversarial peer that chooses its random can force a collision and is treated as a compromised endpoint. Both randoms are bound into the AAD tail, so an in-flight tamper of either contribution fails AEAD verification.
- Cyclic mode uses the same XOR + counter nonce construction: both peers retain `nonce_base = client_random XOR server_random` across the session and increment a 32-bit counter per poll. Per-poll uniqueness derives from the full mutual-random base (96 bits) plus the counter.

SECURITY

Both peers must use a cryptographic-quality RNG. The mutual-random construction tolerates a low-entropy but honest RNG on one side only while the other side stays

sound; a peer that deliberately chooses its random to force a nonce collision is a compromised endpoint, out of scope here; if entropy fails on both peers, the GCM nonce space collapses and the tunnel breaks under a motivated attacker. Treat a random-source failure as fatal and stop, rather than continue with predictable nonces.

Requirements: a cryptographic-quality RNG on both sides. A weak RNG (e.g. rand()) seeded from time(NULL)) collapses GCM uniqueness and breaks the tunnel under any motivated attacker. On MCU targets without a hardware RNG, seed mbedtls_ctr_drbg from the most physical entropy source available (ADC last bits, RC oscillator drift) and call the DRBG from fbsec_sod_port_random() and the client's RNG hook. Demo / lab builds may use OS RNGs but should still drive entropy from a real source where possible. fbsec_sod_port_random returning false surfaces as FBSEC_SOD_ABORT_GENERAL = 0x08000000 on the server, and as FBSEC_SECP_RANDOM (exit 1) on the client; treat that exit as "RNG broken, do not continue".

7.8 Key Rotation Policy

Single-shot verbs. Each srd / swr invocation generates its own per-frame freshness: both peers contribute an R-byte random and the per-frame GCM nonce is the XOR of their first 12 bytes. There is no per-frame counter to overflow. Rotate session keys on the schedule operations policy allows; the cryptography places no hard ceiling. NIST SP 800-38D section 8 limits AES-GCM to 2^{32} encryptions per (key, IV-prefix) pair, but the mutual-random XOR construction means that bound is reached only if both peers' RNGs collapse to a 96-bit collision space simultaneously; a sound RNG on either side keeps the IV space full-width.

Cyclic-mode sessions. When a client arms cyclic mode (sets bit 6 of the wire keyid byte), the server allocates a session bound to (client_dev, data_id) and computes nonce_base from the two arm-time randoms. A 32-bit frame counter increments per poll frame and is XOR'd into the low 32 bits of nonce_base to form each frame's GCM nonce. The per-(key, session) budget is enforced as a hard wall:

FBSEC_AEAD_KEY_USE_LIMIT = 1 000 000 frames (defined in shared/fbsec_config.h). When slot->counter >= FBSEC_AEAD_KEY_USE_LIMIT both peers refuse further frames; the server tears the slot down and aborts with FBSEC_SOD_ABORT_TRANSFER (0x08000020). The client side fbsec_secure_poll_read / fbsec_secure_poll_write return FBSEC_SECP_PROTOCOL when the counter reaches it.

Why 1 000 000 and not 2^{32} : NIST SP 800-38D's invocation cap for AES-GCM under one (key, IV-prefix) pair is on the order of 2^{32} , but staying that close to the bound erodes

the cryptographic margin. One million frames keeps the aggregate forgery probability near 2^{-39} with the default 8-byte tag (using the conservative NIST SP 800-38D Appendix C single-attempt bound of about 2^{-59} , times the million-frame budget; for these short messages the true per-attempt bound is closer to 2^{-64}) and forces a key refresh as a normal operational event rather than a once-in-a-device-lifetime ceremony. At a 1 ms poll period the budget is about 16 minutes; at 100 ms it is about 28 hours. Either way the operator decides their re-arm cadence well below the wall. Hitting the limit is not a failure; it is the protocol telling the host "rotate the key now".

⚠ SECURITY

AES-GCM breaks if the same nonce is reused under one key: a repeated key-and-nonce pair can expose the authentication key and void both confidentiality and authenticity, so nonce uniqueness is not optional. SOFA keeps every nonce unique by XORing a fresh client random and server random into each single-shot nonce and by advancing a strictly increasing counter over that base in cyclic mode, and the per-key frame budget forces a re-key well before the counter space is stressed. Stay within the budget and never replay or roll back a counter.

Cyclic-mode sessions also expire on the server if they go idle for `FBSEC_SOD_SESSION_IDLE_TIMEOUT_MS` (default 60 000 ms, in `shared/fbsec_secure_od.h`). Tune at compile time if the use case requires longer-lived sessions.

7.9 OD Entry Placement

Place secure entries in the manufacturer-specific area of the CANopen Object Dictionary (CiA-301 section 7.4.7, indices `0x2000..0x5FFF`). Within that range the integrator picks any layout that fits the device's existing OD. The SOFA demo uses two paired ranges as a reference convention: `0x2010 / 0x2020` for paired 4-byte entries (write/read shadow) and `0xC016 / 0xC018` for paired 16-byte entries. These are not normative. What matters is that each (index, subindex) the integrator calls "secure" is registered in the secure-OD with `fbsec_sod_register_entry`, and that the matching CANopen OD descriptor marks the entry with the right access flags for the device profile.

The secure-OD's two access modes line up with classical CANopen RO / WO:

`FBSEC_SOD_ACCESS_SECURE_RO` is read-only via the secure tunnel;

`FBSEC_SOD_ACCESS_SECURE_WO` is write-only. An entry may carry both bits (read AND write); the demo's four entries are pure RO or pure WO for clarity. Plain (non-secure) reads or writes against a `SECURE_RO / SECURE_WO` entry must be rejected by the OD-access layer; this rejection sits below the secure tunnel and is separate from the se-

cure-OD abort codes listed under Abort Code Mapping. The exact vehicle is stack-specific: the CiA-301 codes 0x06010002 (attempt to write a read-only object) and 0x06010001 (attempt to read a write-only object) fit the two directions.

Entry data length. The secure-OD enforces a fixed `data_len` per entry (`fbsec_sod_entry_t.data_len`). On every read the `port_read_before` callback must return exactly that many bytes; on every write the request must carry exactly that many bytes of ciphertext. Variable-length entries are out of scope today. Maximum supported plaintext is `FBSEC_AEAD_MAX_PROTECTED`, default 32 bytes, in `shared/fbsec_aead.h`.

7.10 USDO Transport Mapping

Each SOFA flow becomes a small sequence of USDO expedited download / upload exchanges. The byte sequences inside the USDO data block are exactly what the secure-tunnel layer hands to the transport; they are documented byte-for-byte in `fieldbus_sim_secure_tunnel_spec.txt` and visualized in `flow_diagrams.txt`.

Single secure read (srd). Pass 1 is a USDO download request from client to server carrying `keyid[1] || client_random[R]`, answered by an empty-data USDO download response. Pass 2 is an empty USDO upload request, answered by a USDO upload response carrying `server_random[R] || cipher[N] || tag[T]`.

Single secure write (swr). Pass 1 is an empty USDO upload request, answered by a USDO upload response carrying `server_random[R]`. Pass 2 is a USDO download request carrying `keyid[1] || client_random[R] || cipher[N] || tag[T]`, answered by an empty USDO download response.

Cyclic-capable single (srdpoll / swrpoll iteration 1). Same exchange as the corresponding single, with bit 6 of the `keyid` byte set. The server retains a session bound to (`client_node`, `data_id`); the data-bearing response is byte-identical to the single-shot response (no extra `session_id` on the wire). Subsequent iterations use the cyclic-poll exchanges.

Cyclic-poll read. USDO download request from client carries `counter_low[1]`; the USDO upload response from the server carries `counter_low[1] || cipher[N] || tag[T]`.

Cyclic-poll write. USDO download request from client carries `counter_low[1] || cipher[N] || tag[T]`; the USDO download response from the server is empty.

`R` = `FBSEC_AEAD_RANDOM_SIZE` (default 12). `T` = `FBSEC_AEAD_TAG_SIZE` (default 8, the runtime alias of the `FBSEC_AEAD_TAG_LEN_BYTES` knob in `shared/fbsec_config.h`). `N` = entry plaintext length.

USDO command specifiers used (CiA-aligned): 0x01 = download request (client->server, has data); 0x21 = download response / ACK (server->client, optional data); 0x11 = upload request (client->server, no data); 0x31 = upload response (server->client, has data); 0x80 = abort (either direction, 4-byte code). These match what CANopen Magic decodes natively; the SOFA FD bus encodes and decodes them via variants/canopen_fd/common/fbsec_co_fd_usdo.{c,h}.

Falling back to classical SDO. Where USDO is not available, the same byte sequences fit inside classical CiA-301 SDO segments. Each Pass becomes one initiate-SDO plus zero or more segment-SDO transfers per the SDO rules (expedited up to 4 bytes, segmented above). The wire bytes in the SDO data fields are identical to the USDO data block; only the framing differs.

7.11 Callback Wiring

The secure-OD calls into the host application via two callbacks declared in `shared/fbsec_secure_od.h`:

- `fbsec_sod_port_read_before(data_id, dst, len)` - called whenever the secure-OD needs the latest plaintext for a `SECURE_RO` entry (single SRD Pass 2 and every cyclic poll). The callback writes up to `*len` bytes into `dst` and updates `*len` to the number actually written. Returns 0 on success or an `FBSEC_SOD_ABORT_*` code on failure.
- `fbsec_sod_port_write_after(data_id, src, len)` - called whenever the secure-OD has decrypted and tag-verified an inbound write (single SWR Pass 2 and every cyclic write poll). The callback applies the plaintext to the device's state. Returns 0 on success or an abort code on failure.

In a CANopen FD stack the natural place for the hooks is alongside the existing OD-read / OD-write callbacks. A secure entry's CANopen access callback typically: validates that the access came over a USDO (or SDO) transport, not a PDO or vendor-specific service; hands the bytes into the secure-OD via `fbsec_sod_dispatch`; sends the dispatch's reply back over the same USDO. The secure-OD calls `read_before` / `write_after` at the right moment - the integrator does not call them directly.

Threading and re-entrancy: `fbsec_sod_dispatch` is single-threaded (one client request, one reply). If the host stack is itself multi-threaded, the integrator serialises calls into the dispatch (mutex around the call) and ensures `read_before` / `write_after` do their own locking against device state.

What integrators do NOT need to wire: the AEAD itself (AES-GCM, HKDF, the AAD prefix layout, the nonce derivation); key derivation (provisioned session keys go in once via

fbsec_sod_set_key and the secure-OD picks the right one per incoming keyid byte); and cyclic session bookkeeping (the secure-OD keeps a per-(client_node, data_id) slot, tracks the 32-bit frame counter that is XOR'd into the nonce, and tears the session down on overflow).

7.12 Abort Code Mapping

FBSEC_SOD_ABORT_* codes (shared/fbsec_secure_od.h) are CiA-301-aligned 32-bit values: an integrator can pass them straight into a USDO abort frame's 4-byte data block (little-endian) without translation. The SOFA demo encoder fbsec_co_fd_usdo_encode_abort does exactly that. The current mapping:

- 0x05030000 - FBSEC_SOD_ABORT_TAGFAIL - AEAD tag verification failed (treat as generic "data invalid" on stacks that do not carry a finer-grained code)
- 0x06010000 - FBSEC_SOD_ABORT_UNSUPPORTED - service or command not supported, and the role-policy denial from the role-allowed hook
- 0x06020000 - FBSEC_SERVER_ABORT_NO_ENTRY - object does not exist; emitted by the server dispatch layer for an unknown data_id, not by fbsec_sod_dispatch
- 0x06070010 - FBSEC_SOD_ABORT_TYPEMISMATCH - data length / type wrong
- 0x08000000 - FBSEC_SOD_ABORT_GENERAL - generic catch-all
- 0x08000020 - FBSEC_SOD_ABORT_TRANSFER - transfer failed (poll desync, key-use limit, session expired)
- 0x08000022 - FBSEC_SOD_ABORT_LOCKED - access denied by the access-allowed hook (lock or write-protect)

When the OD-access layer receives any of these from fbsec_sod_dispatch's out_abort, it places the value in the USDO abort frame's data field and sends.

7.13 Cyclic Mode Notes

Cyclic mode amortizes the per-frame challenge-response cost: one full single read or write establishes a session, and follow-up polls run as one USDO request + response carrying only the counter byte plus cipher+tag. The session is torn down by the server when the counter reaches FBSEC_AEAD_KEY_USE_LIMIT (default 1 000 000 frames, in shared/fbsec_config.h) - the server emits FBSEC_SOD_ABORT_TRANSFER on the offending poll - or when the session goes idle for longer than FBSEC_SOD_SESSION_IDLE_TIMEOUT_MS (default 60 000 ms without traffic), in which case the server silently drops the session and the next poll fails with FBSEC_SOD_ABORT_TRANSFER. The client recovers from either by re-arming via a fresh cyclic-capable single read or write (bit 6 set in the keyid).

Cyclic mode is appropriate for fast liveness or freshness verification of a SECURE_RO entry, or streaming a slowly-changing setpoint to a SECURE_WO entry without per-frame round-trip overhead. It is not appropriate for one-shot configuration changes (use single SWR for a cleaner audit trail and no session bookkeeping) or pub/sub broadcast (SOFA is strictly client-server).

8 Threat Model

In scope - the cryptography defends against these:

- **Replay.** Every single-shot secure verb carries mutual-random freshness (`client_random[R]` AND `server_random[R]`) in its AAD on both `srd` and `swr`; cyclic poll frames inherit the same mutual-random base via `nonce_base` and add a monotonically increasing counter bound into the GCM nonce. A captured tag does not authorise a second request: one side's contribution is regenerated on every new arming, so the nonce-base never repeats.
- **Tag forgery.** 8-byte truncated GCM tag at default config; bound by NIST SP 800-38D's tag-length analysis. Configurable up to 16 bytes via `FBSEC_AEAD_TAG_LEN_BYTES` in `shared/fbsec_config.h`.
- **Mechanism / key-id downgrade.** The full mechanism byte and the full wire keyid byte are in the AAD prefix; flipping bit 7 (encryption) or bit 6 (cyclic-arm), repurposing the reserved bits, or rerouting to a different keyid base fails AEAD verification.
- **Direction / addressing tampering.** `device_id`, `data_id`, direction code and payload length are all in AAD. An attacker cannot redirect a captured tag to a different device, `data_id`, or direction.

Out of scope - the cryptography does not defend against these; address them in the platform or process:

- **Simulation Hub trust.** The SOFA hub is a dumb broadcast relay (`hub/fieldbus_sim_hub.c`). A malicious hub sees every secure frame and can drop or reorder them. Confidentiality is at the cipher (when bit 7 is set); availability is not protected.
- **Hardware side channels.** Timing attacks against the AES core, power-analysis against the GCM tag, EM emanation. The mbedTLS AES path used here is not constant-time on all MCUs. High-assurance deployments should switch to a hardware AES engine via External secure element or TrustZone-isolated SRAM as listed in MCU Storage Options.
- **Key extraction from a compromised MCU.** If an attacker reads `g_keys[]` via debug port, JTAG, fault injection or a kernel-level exploit on a hosted platform, the tunnel is over. This is what the storage topologies in MCU Storage Options are for; the cryptography itself assumes the keys are confidential.
- **Coercion / supply chain at provisioning time.** Factory-inject is only as strong as the factory's process. SOFA does not address supply-chain attestation.