An Open-Source SCMS for Secure V2X Infrastructure

IEEE1609.2.1 Standards-Compliant • Privacy-Preserving • Open

White Paper

LGEUS Advanced Security Team

February 12, 2026

1 Executive Summary

The evolution of autonomous and connected vehicles fundamentally reshapes modern transportation systems. Beyond advances in perception and onboard decision-making, the true enabler of large-scale autonomy lies in coordination and integration between entities. Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I), collectively referred to as V2X communication, form the backbone of this ecosystem, enabling cooperative awareness, coordinated maneuvers, traffic optimization, and enhanced road safety. In such an environment, secure, reliable, and privacy-preserving communication is not optional, it is a foundational requirement.

To address these requirements, the IEEE 1609.2 [1] and IEEE 1609.2.1 [2] standards define a comprehensive security architecture for V2X communications. At the core of this architecture lies the Security Credential Management System (SCMS), a public key infrastructure (PKI) responsible for issuing and managing digital certificates used by V2X end entities (EEs). These end entities include onboard units (OBUs), roadside units (RSUs), aftermarket safety devices (ASDs), and other computing platforms that participate in V2X applications. SCMS-issued certificates encode permissions that define the application message types an EE is authorized to transmit, using Provider Service Identifiers (PSIDs) and Service Specific Permissions (SSPs) as defined in IEEE 1609.12.

A defining feature of the SCMS architecture is its strong emphasis on privacy. To prevent long-term tracking of vehicles (mobile OBUs), the SCMS often issues pseudonym certificates, allowing EEs to rotate credentials over time while maintaining authenticated and authorized communication. This approach enables trust without sacrificing anonymity, a critical requirement for public acceptance and regulatory compliance in V2X deployments.

Despite its importance, SCMS remains a complex and challenging system to implement. It is inherently distributed, composed of multiple interacting components, and must address a wide range of concerns beyond cryptography alone. These include performance and scalability, task parallelism, coordination between services, database management, and reliable handling of large volumes of security-critical data. As a result, existing SCMS implementations are proprietary, difficult to audit, and costly to adapt or extend, creating significant barriers to both industrial innovation and academic research.

The OpenSCMS project addresses this gap by providing an open-source implementation of the core components and workflows of an IEEE 1609.2.1-compliant SCMS. Rather than aiming for exhaustive coverage of every optional feature, OpenSCMS focuses on implementing the fundamental building blocks required to support all essential EE–SCMS interaction flows. This "minimum complete" approach enables real-world usability while preserving architectural clarity and extensibility.

OpenSCMS is designed as a scalable distributed system aligned with the architectural principles proposed by the standard. It adopts a microservices-based design in which each SCMS component is isolated and manages its own data, enabling independent scaling and clearer separation of responsibilities. Task parallelism is achieved through a message-driven architecture using task queues (e.g., MQTT), allowing the system to efficiently handle concurrent certificate requests and cryptographic operations. The backend implementation is primarily written in Rust, leveraging its strong guaranties around memory safety, concurrency, and performance; properties that are essential when dealing with security critical infrastructure at scale.

At the heart of OpenSCMS lies a key technical contribution: the oscms-codecs-bridge, a low-level library written in C that implements the core cryptographic logic in addition to the encoding and decoding of IEEE 1609.2.1 SPDUs and ASN.1 structures. This library encapsulates the most sensitive and complex aspects of SCMS operation, including decoding client messages, verifying digital signatures, decrypting requests, constructing compliant responses, encrypting payloads, and generating both explicit and implicit pseudonym certificates exactly as specified by the standard. By centralizing this functionality, the bridge acts as the cryptographic and protocol "engine" of OpenSCMS.

A defining characteristic of the oscms-codecs-bridge is its codec-agnostic design. It abstracts the underlying ASN.1 transpiler behind a clean API, eliminating dependency on proprietary or closed-source toolchains that have historically dominated this space. OpenSCMS currently relies on an open-source transpiler, but the architecture explicitly allows alternative transpilers to be integrated or extended in the future. This design not only avoids vendor lock-in, but also improves auditability, portability, and long-term sustainability of the SCMS ecosystem. Moreover, the oscms-codecs-bridge can be used independently of the Rust backend, making it a standalone contribution that can be embedded in other SCMS implementations or used in dedicated testing and validation tools.

By providing an open, extensible, and production-oriented SCMS implementation, OpenSCMS unlocks new possibilities for both industry and universities. Industrial teams can rapidly prototype and evaluate SCMS-based solutions without the burden of building a complete infrastructure from scratch. Academic researchers gain access to a realistic and modifiable SCMS platform for experimentation, including the evaluation of alternative cryptographic strategies (e.g., evaluating the impact of transitioning to post-quantum cryptography), methods for misbehavior detection and reporting (e.g., machine learning approaches [3]), efficient management, and maintenance of revocation lists, etc. Equally important, developers implementing V2X client software can use OpenSCMS as a reference and validation backend, enabling comprehensive testing of client–SCMS interactions without requiring a separate, proprietary SCMS deployment.

In summary, OpenSCMS lowers the barrier to entry for secure V2X infrastructure development by combining a modern, safe systems backend with a high-performance, codec-agnostic cryptographic core. It represents both a practical tool and a foundational contribution to the open V2X security ecosystem. Overall, OpenSCMS demonstrates that secure, privacy-preserving V2X infrastructures can be both standard-compliant and openly accessible, enabling innovation across industry and academia.

2 SCMS and the Challenges of Secure V2X Infrastructure

Secure V2X communication relies on a carefully designed trust infrastructure capable of supporting large-scale, privacy-preserving, and performance-critical interactions between heterogeneous entities. Within the IEEE 1609.2 and IEEE 1609.2.1 framework, this role is fulfilled by the Security Credential Management System (SCMS). While conceptually defined as a public key infrastructure (PKI), the SCMS extends far beyond traditional PKI systems, introducing unique architectural, cryptographic, and operational challenges driven by the scale and privacy requirements of V2X ecosystems.

2.1 The Role of SCMS in V2X Security

The SCMS is responsible for creating and managing digital certificates that enable authenticated application-level interactions between end entities (EEs), such as on-board units (OBUs), roadside units (RSUs) and other computing platforms. These certificates encode authorization information using Provider Service Identifiers (PSIDs) and Service Specific Permissions (SSPs), allowing fine-grained control over which applications and activities an end entity (EE) is permitted to perform.

A defining characteristic of the SCMS architecture is its emphasis on privacy preservation at scale. Rather than relying on long-lived identifiers, the SCMS frequently provisions pseudonym certificates, allowing end entities to rotate credentials over time. This approach ensures that messages remain authenticated and authorized while preventing long-term tracking of vehicles or infrastructure nodes, an essential requirement for public trust and regulatory compliance.

2.2 Core SCMS Interaction Flows

The IEEE 1609.2.1 standard defines two primary use cases governing the interaction between end entities and the SCMS. These use cases form the backbone of any compliant SCMS implementation.

2.2.1 Enrollment Certificate Provisioning

Before an EE can participate in secure V2X communication, it must first obtain an enrollment certificate. This certificate establishes the EE's identity within the SCMS ecosystem and is used exclusively for secure communication with SCMS components.

In the enrollment certificate provisioning flow, an enrollment certificate authority (ECA) provisions the EE with an enrollment certificate. The request may be sent either directly by the EE or indirectly via a Device Configuration Manager (DCM) acting on its behalf. Upon receiving a valid request, the ECA verifies the request according to policies defined by the SCMS manager and responds with an enrollment certificate.

This use case represents the foundational trust-establishment step in the SCMS lifecycle and serves as the prerequisite for all subsequent interactions.

2.2.2 Authorization Certificate Provisioning

Once an EE possesses a valid enrollment certificate, it may request authorization certificates, which are used in application-level V2X communications. This use case is significantly more complex and involves multiple SCMS components.

In the authorization certificate provisioning flow, the EE interacts primarily with the Registration Authority (RA), either directly or via a Location Obscurer Proxy (LOP) when location privacy is required. The EE submits a request for authorization certificates, which, if valid, results in an acknowledgment from the RA specifying the time and location from which the certificates can be downloaded.

Behind the scenes, the RA coordinates with several other SCMS components to generate the requested certificates. These include the Authorization Certificate Authority (ACA) and, for pseudonym certificates, the Linkage Authorities. Optionally, a Supplementary Authorization Server (SAS) may also be involved. This multi-party interaction ensures that no single component can independently link certificates to a specific EE, reinforcing the privacy guarantees of the system.

The standard supports two approaches to authorization certificate provisioning. In the first, the EE requests and receives a single authorization certificate. In the second, more scalable approach, the EE employs butterfly key expansion, allowing a single request to result in a set of certificates covering multiple time periods and, potentially, multiple certificates per period. This mechanism significantly reduces communication overhead while maintaining strong cryptographic separation between certificates. In addition the butterfly mechanism allows for pseudonymity of the issued authorization certificates in a way that no single SCMS component can track EE devices.

2.3 Architectural and Operational Challenges

Implementing an SCMS that correctly supports these interaction flows presents a number of non-trivial challenges.

First, SCMS is inherently a distributed system, composed of multiple logical components that must coordinate securely and efficiently. Each component has distinct responsibilities, data models, and trust assumptions, making isolation and clear interface definitions essential.

Second, the system must handle high volumes of cryptographic operations, including signature verification, encryption, decryption, and certificate generation, often under strict latency constraints. This must be achieved while preserving privacy properties such as unlinkability and resistance to correlation attacks.

Third, SCMS implementations must address traditional systems challenges that are largely orthogonal to cryptography but equally critical to correctness and scalability. These include task parallelism, asynchronous processing, reliable message exchange, database consistency, fault tolerance, and operational observability.

Finally, from a development and research perspective, SCMS implementations have historically suffered from limited accessibility. Proprietary implementations and closed toolchains make it difficult for researchers to experiment with alternative cryptographic strategies—such as SCMS providers from certification consortiums like Omniair [1], or for industry teams to prototype and validate client-side implementations without substantial upfront investment.

2.4 Addressing SCMS Complexity with OpenSCMS

The challenges outlined above—ranging from cryptographic correctness and privacy preservation to scalability, system coordination, and operational complexity—highlight the need for an SCMS implementation that is both architecturally sound and practically usable. The OpenSCMS project was designed specifically to address these challenges by providing a clean, modular, and validated implementation of the core IEEE 1609.2.1 SCMS workflows.

Rather than attempting to replicate every optional component defined in the standard, OpenSCMS focuses on delivering a compact yet extensible architecture that correctly implements the essential interaction flows between end entities and the SCMS. This approach ensures correctness and interoperability while significantly lowering the barrier to adoption for both industrial and academic users.

A key design goal of OpenSCMS is to remain faithful to the logical structure and security guarantees of the standard, while simplifying deployment, experimentation, and extension. This balance is achieved through clear component boundaries, explicit

[1] `https://www.omniair.org`

interfaces, and a strong separation between protocol logic, cryptographic processing, and system orchestration.

These design choices directly influence the architectural decisions presented in the next section, which describes how OpenSCMS realizes these concepts through a layered and distributed implementation.

2.4.1 Compact yet Complete SCMS Architecture

OpenSCMS implements the two fundamental EE interaction flows as defined by IEEE 1609.2.1: enrollment certificate provisioning and authorization certificate provisioning. These flows are supported through direct interaction between the EE and the Enrollment Certificate Authority (ECA) and the Registration Authority (RA), respectively.

To reduce architectural overhead while preserving functional correctness, OpenSCMS does not implement certain auxiliary components defined in the standard, such as the Device Configuration Manager (DCM), the Location Obscurer Proxy (LOP), and the Supplementary Authorization Server (SAS). Instead, the system enables the correct execution of both provisioning flows through a streamlined architecture in which:

- End entities interact directly with the ECA for enrollment certificate provisioning.
- End entities interact directly with the RA for authorization certificate provisioning.
- The RA additionally exposes endpoints responsible for certificate distribution, effectively subsuming the role of the Distribution Center (DC).
- Essential bootstrapping functionality for end entities—traditionally associated with the DCM—is provided directly by OpenSCMS.

This design preserves the security and privacy properties mandated by the standard while avoiding unnecessary complexity. As a result, OpenSCMS provides a practical foundation that can be deployed, tested, and extended without requiring a full-scale SCMS deployment.

Importantly, the correctness of these flows has not been validated solely through specification compliance, but also through end-to-end integration with a conformant EE client. The client implements the IEEE 1609.2.1 protocol, was originally validated against one of Omniair SCMS providers, and successfully completes enrollment and authorization provisioning flows against OpenSCMS, demonstrating protocol-level interoperability and validating the architectural choices made by the project.

2.4.2 Establishing the Root of Trust

Before an EE can securely interact with the SCMS, it must establish a root of trust by being provisioned with certificates for electors, root certificate authorities (root CAs), or both. The IEEE 1609.2.1 standard explicitly leaves the mechanisms and security requirements for this provisioning outside its scope.

OpenSCMS addresses this gap by providing a flexible and extensible mechanism for initializing trust anchors. During system startup, OpenSCMS loads Certificate Chain Files (CCFs) and Certificate Trust Lists (CTLs), which define the trust relationships used throughout the system.

To support testing, experimentation, and rapid deployment, OpenSCMS includes a standalone utility, *scms-manager*, implemented on top of the *oscms-codecs-bridge* library. This tool enables the generation of elector and root CA key pairs and certificates, as well as the creation of CCF and CTL artifacts. These generated payloads can be directly consumed by OpenSCMS during initialization.

While *scms-manager* serves as a reference and support tool, OpenSCMS does not impose a fixed trust model. Users remain free to generate and manage trust anchors according to their own operational and security requirements, ensuring that the system can be adapted to a wide range of deployment scenarios.

2.4.3 Deployment and Operational Considerations

In addition to protocol correctness and cryptographic soundness, OpenSCMS was designed with operational practicality in mind. Secure infrastructure must be deployable, observable, and maintainable in real-world environments.

To this end, OpenSCMS provides a unified deployment strategy based on Docker, Kubernetes, and Skaffold. This setup enables reproducible builds, simplified deployment and redeployment workflows, and seamless integration with modern container orchestration and monitoring ecosystems.

By adopting cloud-native deployment practices, OpenSCMS supports scalable experimentation, automated testing, and continuous integration workflows. This operational flexibility further reinforces the project's goal of serving as a practical foundation for both research and industrial prototyping.

3 OpenSCMS Architecture Overview

The OpenSCMS architecture was designed not only to implement the IEEE 1609.2.1 standard, but to do so in a way that remains auditable, extensible, and practical for real-world deployment. Rather than concentrating protocol logic, cryptography, orchestration, and infrastructure concerns into a monolithic system, OpenSCMS adopts a layered and service-oriented architecture that cleanly separates responsibilities across well-defined components.

At its core, OpenSCMS is composed of two primary implementation layers: a Rust-based backend responsible for SCMS endpoint interfaces, orchestration, persistence, and deployment concerns, and a low-level C library, *oscms-codecs-bridge*, which implements the cryptographic and protocol-specific core logic. This separation is a deliberate architectural choice that enables both safety and performance without sacrificing compliance or extensibility.

3.1 Layered Design and Responsibility Separation

The top layer of OpenSCMS is implemented in Rust and represents the system's control plane. This layer is responsible for exposing the SCMS interfaces to end entities, coordinating interactions between SCMS components, and managing state, concurrency, and persistence. From the perspective of an external client, this layer is the SCMS.

The Rust backend implements the Registration Authority (RA), Enrollment Certificate Authority (ECA), Authorization Certificate Authority (ACA), and Linkage Authority (LA) as independent microservices. Each component operates autonomously, maintains its own local database, and exposes a well-defined set of HTTP endpoints corresponding to its role in the standard. This service-level separation closely mirrors the conceptual SCMS architecture defined in IEEE 1609.2.1, while allowing practical deployment flexibility.

Rust was chosen for this layer due to its strong guarantees around memory safety, concurrency, and predictable performance. These properties are particularly important in a security-critical system that processes cryptographic material and handles concurrent requests from potentially large fleets of end entities. Rust's native support for C interoperability also enables seamless integration with the lower-level protocol core.

3.2 *oscms-codecs-bridge*: Cryptographic and Protocol Core

All cryptographic operations and protocol-specific data manipulation are implemented in *oscms-codecs-bridge*, a standalone library written in C. This library encapsulates the full complexity of IEEE 1609.2.1 data handling, including ASN.1 SPDU encoding and decoding, digital signature verification, certificate chain validation, encryption and decryption, and certificate issuance.

The library implements the generation of enrollment certificates, successor enrollment certificates, and authorization certificates, supporting both explicit and implicit certificates via ECDSA and ECQV mechanisms, respectively. For authorization certificates, it supports butterfly key expansion in its original, unified, and compact unified variants, as well as non-butterfly certificates in plain and encrypted forms. All cryptographic and encoding logic required by the standard is concentrated in this layer.

The Rust backend interacts with *oscms-codecs-bridge* exclusively through a defined API. Incoming protocol messages are passed to the library for decoding and validation, and responses are constructed by invoking the appropriate handlers. This allows the Rust services to remain agnostic of ASN.1 structures and cryptographic primitives, focusing instead on orchestration and policy enforcement.

3.3 Codec and Transpiler Abstraction

A key architectural decision in OpenSCMS is the abstraction of ASN.1 tooling. *oscms-codecs-bridge* does not hard-bind protocol logic to a single ASN.1 compiler or codec. Instead, it introduces a codec/transpiler layer that allows concrete implementations to be plugged in behind a stable API interface.

This design avoids dependency on proprietary or closed ASN.1 toolchains and enables experimentation, auditing, and future migration without architectural disruption. Protocol logic remains stable even if the underlying ASN.1 implementation changes.

This separation allows OpenSCMS to remain safe, extensible, and performant without coupling protocol logic to ASN.1 tooling or cryptographic primitives.

3.4 Microservices, Concurrency, and Persistence

Each SCMS component in OpenSCMS runs as an independent service with its own database and filesystem storage. This isolation simplifies reasoning component behavior and aligns with the distributed nature of real-world SCMS deployments.

Concurrency is handled explicitly at the architectural level. The Registration Authority (main front-end component) are internally split into frontend and worker roles, enabling asynchronous processing of long-running tasks. Task queues are used to decouple request handling from certificate generation, allowing the system to scale horizontally by adding worker instances.

3.5 Deployment and Operational Considerations

OpenSCMS provides a unified deployment strategy based on containerization. All components can be deployed using Docker, enabling consistent environments across development, testing, and production. For orchestration at scale, Kubernetes is supported, providing automated deployment, scaling, and lifecycle management.

Skaffold is used to streamline development and deployment workflows, enabling rapid iteration, monitoring, and redeployment of services. Together, these tools allow OpenSCMS to be deployed in environments ranging from local testbeds to cloud-based or hybrid infrastructures.

The following section demonstrates how this architecture is exercised through concrete SCMS workflows, illustrating the interaction between components and validating the design through real protocol execution.

4 Implemented SCMS Workflows

This section describes the SCMS workflows implemented by OpenSCMS and demonstrates how the architecture presented in the previous section is exercised and validated through concrete protocol interactions. Rather than aiming for full coverage of every optional component defined in IEEE 1609.2.1, OpenSCMS focuses on the minimal and sufficient set of workflows required to enable secure, privacy-preserving V2X communication, while preserving extensibility for future evolution.

The workflows presented here correspond to the most critical interactions between end entities (EEs) and the SCMS: end entity (EE) registration, enrollment certificate provisioning, authorization certificate provisioning, and successor enrollment. For each workflow, architectural diagrams are used to make explicit relationships between components, the separation of responsibilities, and the asynchronous processing model adopted by OpenSCMS.

These workflows represent the essential interactions required for secure V2X operation and demonstrate that the OpenSCMS is not merely architectural scaffolding, but a functional and compliant SCMS implementation.

4.1 EE Registration, Bootstrapping and Enrollment Workflow

EE registration and enrollment constitute the foundation of trust in the SCMS. Before an EE can request authorization certificates or participate in V2X communication, it must establish a canonical identity and obtain an enrollment certificate.

The following diagrams provide an architectural view of these workflows, explicitly showing component boundaries, responsibilities, and interaction patterns adopted by OpenSCMS.

In this flow, the Registration Authority (RA) front-end is responsible for handling initial interaction with the EE. The client submits a registration request containing its canonical public key. Based on this key, the RA generates a canonical identifier and a device identifier, which get persisted in the RA database. These identifiers form the basis for all subsequent authorization and enrollment checks.

The registration process is a synchronous operation, providing immediate feedback to the client. In addition to registration, the RA exposes endpoints to check registration status, allowing other SCMS components to validate an EE's eligibility before proceeding with certificate issuance for example.

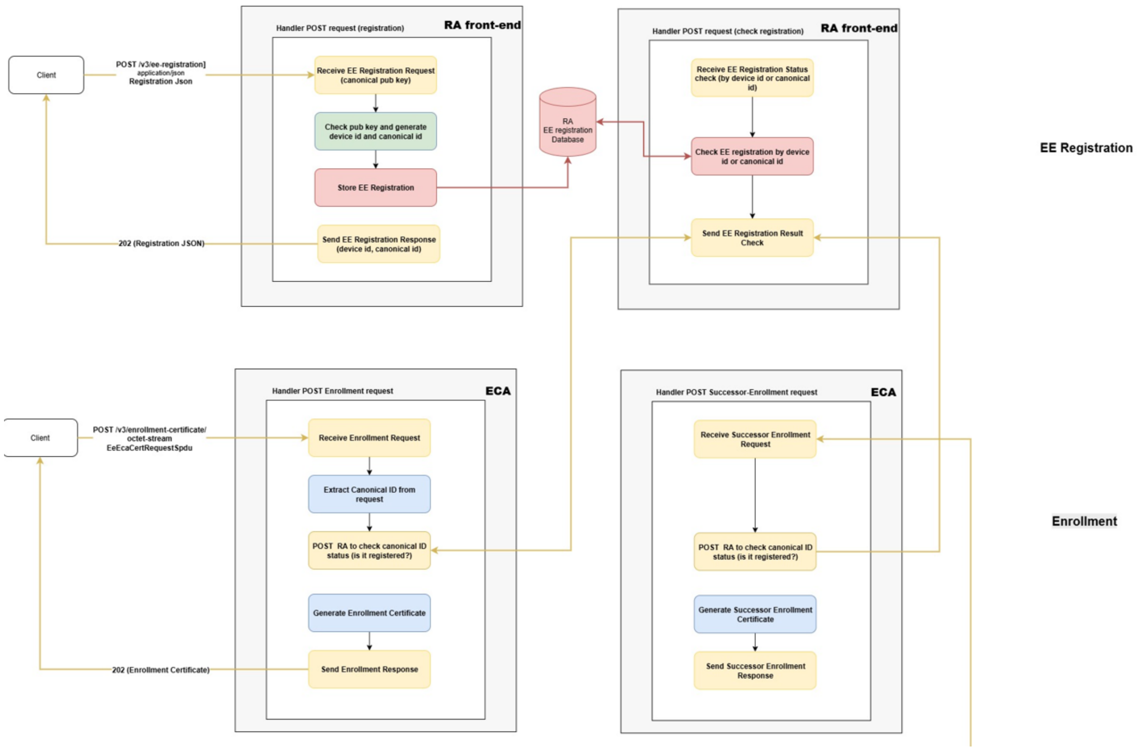
Client
POST /v3/ee-registration]
application/json
Registration Json
Handler POST request (registration)
RA front-end
Receive EE Registration Request (canonical pub key)
Check pub key and generate device id and canonical id
Store EE Registration
Send EE Registration Response (device id, canonical id)
202 (Registration JSON)
RA EE registration Database
Handler POST request (check registration)
RA front-end
Receive EE Registration Status check (by device id or canonical id)
Check EE registration by device id or canonical id
Send EE Registration Result Check
EE Registration
Client
POST /v3/enrollment-certificate/
octet-stream
EeEcaCertRequestSpdu
Handler POST Enrollment request
ECA
Receive Enrollment Request
Extract Canonical ID from request
POST RA to check canonical ID status (is it registered?)
Generate Enrollment Certificate
Send Enrollment Response
202 (Enrollment Certificate)
Handler POST Successor-Enrollment request
ECA
Receive Successor Enrollment Request
POST RA to check canonical ID status (is it registered?)
Generate Successor Enrollment Certificate
Send Successor Enrollment Response
Enrollment

Figure 1: *Illustrates the end-to-end registration and enrollment flow as implemented in OpenSCMS*

Once registered, the EE may request an enrollment certificate. As shown in Figure 1, the Enrollment Certificate Authority (ECA) receives the enrollment request, extracts the canonical identifier from the decoded SPDU, and queries the RA to verify the registration status. Upon successful verification, the ECA generates the enrollment certificate and returns it directly to the client.

This flow demonstrates the clear separation of responsibilities: identity management and status verification remain centralized in the RA, while certificate generation is isolated within the ECA. The synchronous nature of enrollment simplifies client logic and establishes a trusted communication channel for subsequent SCMS interactions.

In addition to registration, OpenSCMS provides, through RA, bootstrapping functionality that would traditionally be handled by a Device Configuration Manager (DCM). The RA exposes endpoints to distribute global and end-entity policy files, which define protocol parameters such as cryptographic algorithms, timing constraints, and operational policies. Consolidating these responsibilities within the RA simplifies deployment while preserving full functionality.

Together, these steps establish the root operational trust required for all subsequent SCMS interactions.

4.2 Authorization Certificate Provisioning and Download

Authorization certificate provisioning is the most complex and resource-intensive workflow implemented in OpenSCMS. It is also the workflow that most clearly demonstrates the benefits of the system's layered and asynchronous architecture requiring coordination between the RA, ACA, and LA.

The flow begins when an EE submits an authorization certificate request to the

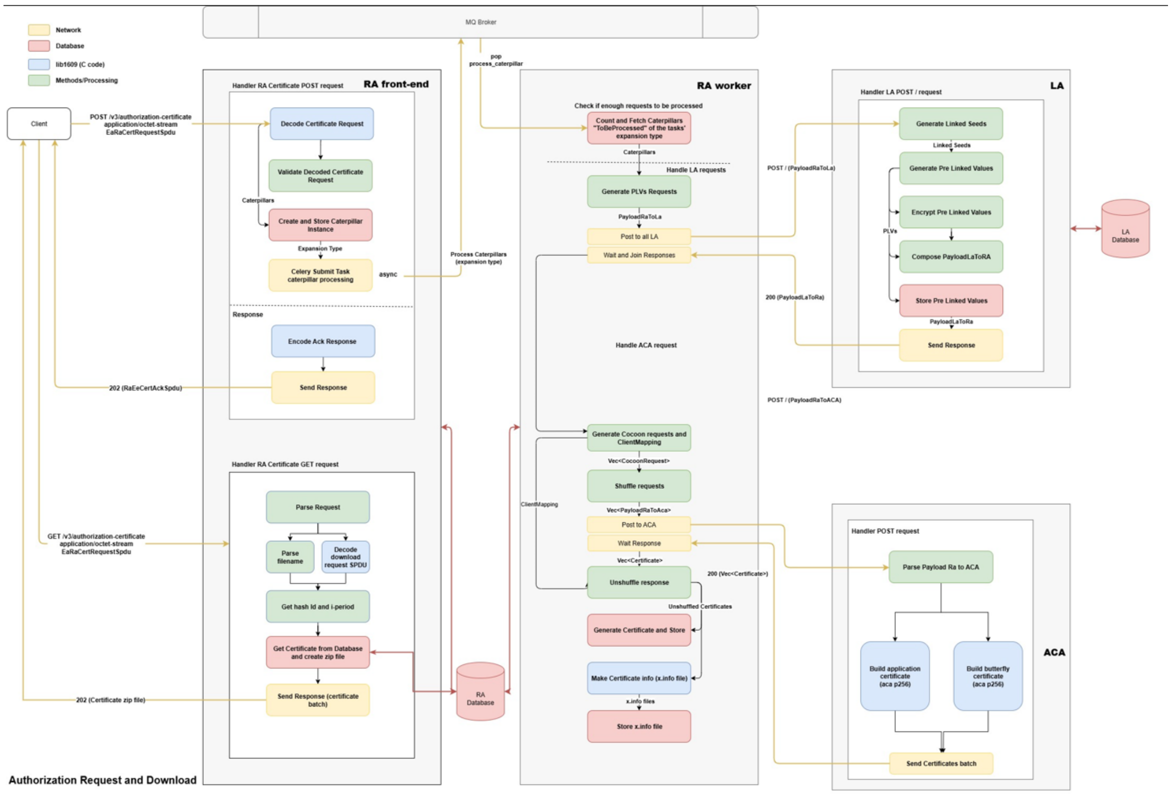
MQ Broker
Network
Database
lib1609 (C code)
Methods/Processing
Client
POST /v3/authorization-certificate application/octet-stream EaRaCertRequestSpdu
RA front-end
Handler RA Certificate POST request
Decode Certificate Request
Validate Decoded Certificate Request
Caterpillars
Create and Store Caterpillar Instance
Expansion Type
Celery Submit Task caterpillar processing
async
Process Caterpillars (expansion type)
pop process_caterpillar
Response
Encode Ack Response
Send Response
202 (RaEeCertAckSpdu)
Handler RA Certificate GET request
GET /v3/authorization-certificate application/octet-stream EaRaCertRequestSpdu
Parse Request
Parse filename
Decode download request SPDU
Get hash Id and i-period
Get Certificate from Database and create zip file
Send Response (certificate batch)
202 (Certificate zip file)
RA Database
Authorization Request and Download
RA worker
Check if enough requests to be processed
Count and Fetch Caterpillars "ToBeProcessed" of the tasks' expansion type
Caterpillars
Handle LA requests
Generate PLVs Requests
PayloadRaToLa
Post to all LA
Wait and Join Responses
Handle ACA request
Generate Cocoon requests and ClientMapping
Vec<CocoonRequest>
Shuffle requests
ClientMapping
Vec<PayloadRaToAca>
Post to ACA
Wait Response
Vec<Certificate>
Unshuffle response
Unshuffled Certificates
Generate Certificate and Store
Make Certificate info (x.info file)
x.info files
Store x.info file
POST / (PayloadRaToLa)
200 (PayloadLaToRa)
POST / (PayloadRaToACA)
200 (Vec<Certificate>)
LA
Handler LA POST / request
Generate Linked Seeds
Linked Seeds
Generate Pre Linked Values
PLVs
Encrypt Pre Linked Values
Compose PayloadLaToRA
Store Pre Linked Values
PayloadLaToRa
Send Response
LA Database
ACA
Handler POST request
Parse Payload Ra to ACA
Build application certificate (aca p256)
Build butterfly certificate (aca p256)
Send Certificates batch

Figure 2: *Presents the full authorization request and download workflow, including internal coordination between the RA, LA, and ACA.*

RA front-end. This request includes parameters defining the desired certificate type, validity periods, and butterfly key expansion strategy, as dictated by SCMS policies. The RA front-end decodes and validates the request using *oscms-codecs-bridge*, persists request metadata, and immediately responds with an acknowledgment. This acknowledgment specifies when and where the EE should later download the generated certificates.

As illustrated in Figure 2, the RA front-end does not perform certificate generation directly. Instead, it enqueues a task into a message queue, enabling asynchronous processing and parallelism. RA workers consume these tasks and orchestrate the certificate generation process.

The diagram shows how the RA worker interacts with the Linkage Authority (LA) to generate linkage seeds and pre-linked values, which are required for efficient revocation checking. These values are generated, encrypted, and persisted by the LA before being returned to the RA worker. This workflow represents the most complex interaction supported by OpenSCMS, involving multiple SCMS components while preserving strict separation of responsibilities and asynchronous execution.

Once linkage material is available, the RA worker composes a request to the Authorization Certificate Authority (ACA). The ACA is responsible for generating the actual authorization certificates, supporting both explicit and implicit certificates as well as multiple butterfly key expansion variants. All cryptographic operations and ASN.1 encoding are delegated to the *oscms-codecs-bridge* library.

After receiving the certificate batch from the ACA, the RA worker stores the generated artifacts and associated metadata. When the EE later issues a download request, the RA front-end retrieves the certificates from persistent storage, packages them into a response, and delivers them to the client.

This workflow exemplifies the decoupling between client-facing interfaces and compute-intensive operations. It also shows how OpenSCMS can scale horizontally by increasing the number of RA workers without modifying protocol logic.

4.3 Successor Enrollment Request and Download

Successor enrollment supports a certificate lifecycle continuity by allowing an already enrolled EE to obtain a successor enrollment certificate. While conceptually simpler than authorization provisioning, it follows a similar asynchronous processing model.

In this flow, the EE submits a successor enrollment request to the RA front-end. The request is decoded, validated, and persisted, and an acknowledgment is returned to the client. As with authorization provisioning, the RA front-end delegates certificate generation to background workers via a message queue.

RA workers retrieve pending successor enrollment tasks, coordinate with the ECA to generate a successor enrollment certificate, and store the resulting certificate in the RA database. The client later retrieves the certificate through a dedicated download endpoint.

Figure 3 highlights how OpenSCMS reuses the same architectural patterns across different workflows. The consistent split between front-end request handling and worker-based processing simplifies reasoning about system behavior and ensures predictable performance under load.

4.4 Distribution Center Functionality (DC)

In addition to its primary responsibilities, the Registration Authority also implements distribution endpoints that would traditionally be handled by a Distribution Center. This allows end entities to retrieve certificates and related artifacts from a single, consistent interface, further simplifying deployment.

4.5 Architectural Consistency Across Workflows

Taken together, Figures 1 through 3 demonstrate how OpenSCMS applies a uniform architectural model across all critical SCMS workflows. In each case, HTTP-facing Rust services implement the standard-defined interfaces, enforce policy, and coordinate interactions, while cryptographic correctness and ASN.1 manipulation are delegated to the *oscms-codecs-bridge* library.

Taken together, these workflows demonstrate that OpenSCMS is not a partial or conceptual implementation, but a fully executable SCMS capable of supporting real end-entity interactions while remaining aligned with the IEEE 1609.2.1 specification.

5 *oscms-codecs-bridge*: Cryptographic and Protocol Core

The *oscms-codecs-bridge* is a low-level library fully implemented in C that concentrates the cryptographic, encoding, and protocol-specific logic defined by IEEE 1609.2.1. While OpenSCMS provides a complete SCMS backend, *oscms-codecs-bridge* stands as an independent, reusable, and technically mature contribution to the V2X security ecosystem.

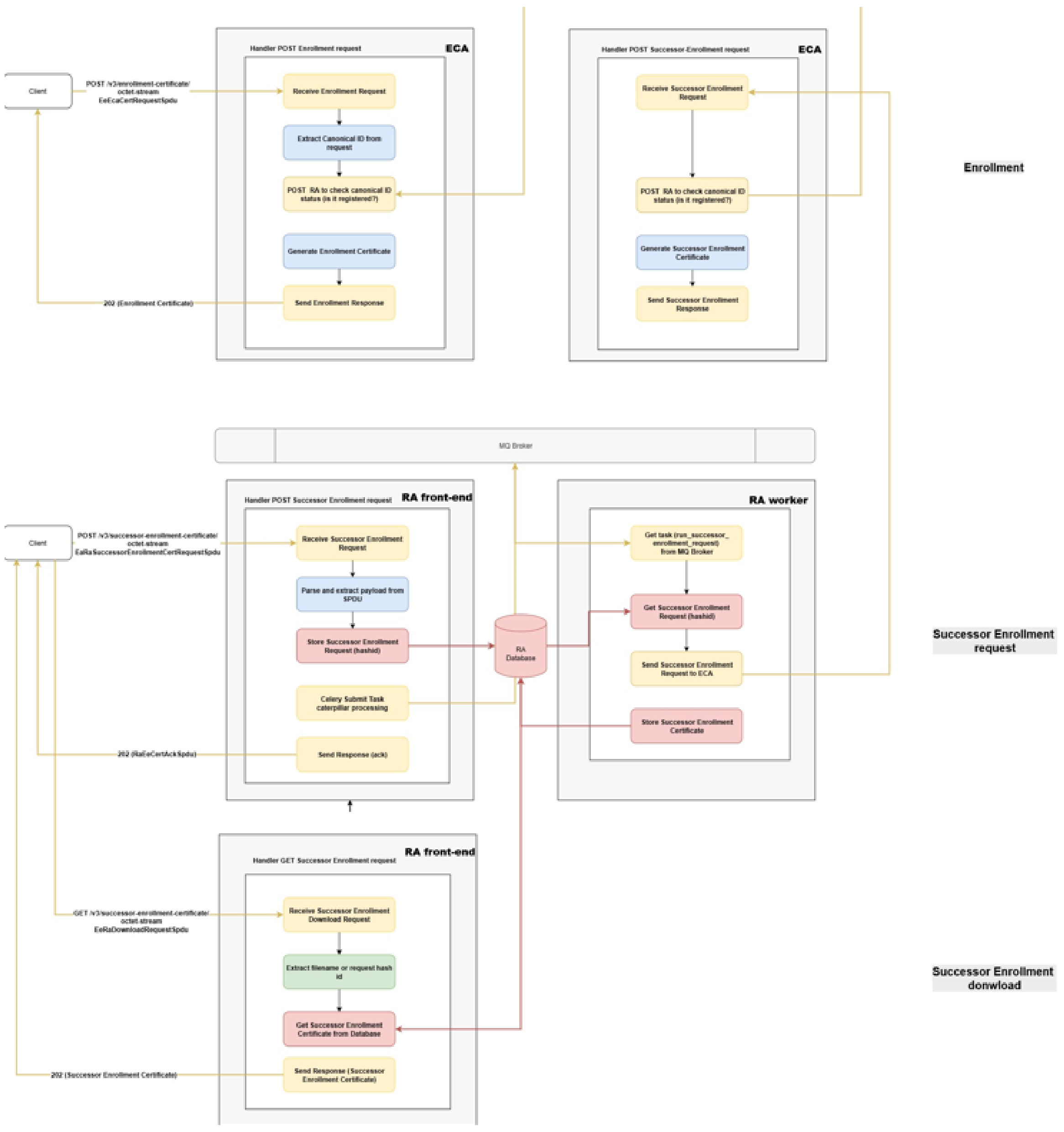
Handler POST Enrollment request
ECA
Client
POST /v3/enrollment-certificate/
octet-stream
EeEcaCertRequestSpdu
Receive Enrollment Request
Extract Canonical ID from request
POST RA to check canonical ID status (is it registered?)
Generate Enrollment Certificate
Send Enrollment Response
202 (Enrollment Certificate)
Handler POST Successor-Enrollment request
ECA
Receive Successor Enrollment Request
POST RA to check canonical ID status (is it registered?)
Generate Successor Enrollment Certificate
Send Successor Enrollment Response
Enrollment
MQ Broker
Handler POST Successor Enrollment request
RA front-end
Client
POST /v3/successor-enrollment-certificate/
octet-stream
EaRaSuccessorEnrollmentCertRequestSpdu
Receive Successor Enrollment Request
Parse and extract payload from SPDU
Store Successor Enrollment Request (hashid)
Celery Submit Task caterpillar processing
Send Response (ack)
202 (RaEeCertAckSpdu)
RA Database
RA worker
Get task (run_successor_enrollment_request) from MQ Broker
Get Successor Enrollment Request (hashid)
Send Successor Enrollment Request to ECA
Store Successor Enrollment Certificate
Successor Enrollment request
Handler GET Successor Enrollment request
RA front-end
GET /v3/successor-enrollment-certificate/
octet-stream
EeRaDownloadRequestSpdu
Receive Successor Enrollment Download Request
Extract filename or request hash id
Get Successor Enrollment Certificate from Database
Send Response (Successor Enrollment Certificate)
202 (Successor Enrollment Certificate)
Successor Enrollment donwload

Figure 3: *Depicts the successor enrollment request and download workflow..*

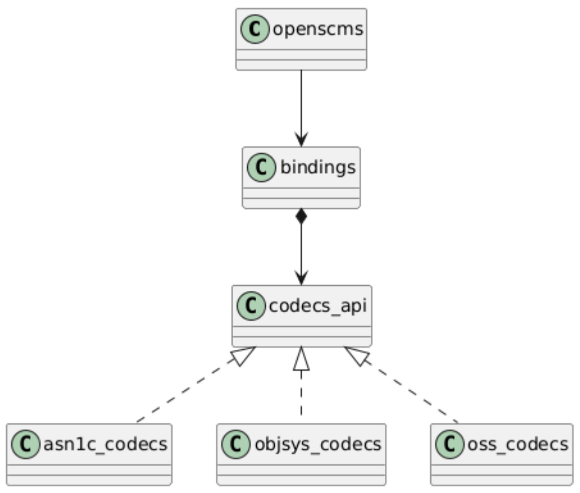

Figure 4: *Layered Architecture of* oscms-codecs-bridge

The library was designed to address a set of long-standing challenges faced by both industry and academia: the lack of open, auditable implementations of IEEE 1609.2.1 codecs; tight coupling between protocol logic and proprietary ASN.1 toolchains; and the difficulty of reusing cryptographic and SPDU-processing logic across different systems.

The decision to implement *oscms-codecs-bridge* entirely in C was deliberate and driven by technical requirements intrinsic to cryptographic systems. Fine-grained control over memory allocation, byte-level manipulation, and deterministic performance is critical when implementing certificate construction, signature verification, encryption, and ASN.1 serialization.

In addition, C provides natural compatibility with widely used cryptographic libraries such as OpenSSL and with code generated by ASN.1 transpilers, including ASN1C. This compatibility allows *oscms-codecs-bridge* to operate efficiently while remaining portable and interoperable across different environments.

Rather than embedding this complexity into the Rust backend, OpenSCMS isolates it in a dedicated core library. This separation ensures that cryptographic correctness and protocol fidelity are handled in a single, well-defined place, while higher layers remain focused on orchestration and system concerns. While developed as part of OpenSCMS, this library was intentionally designed as a standalone and reusable component, suitable for integration beyond a single SCMS implementation.

5.1 Layered Design of *oscms-codecs-bridge*

The *oscms-codecs-bridge* follows a layered architecture that mirrors the separation-of-concerns philosophy applied at the system level. This internal structure is illustrated in Figure 4.

At the lowest layer, oscms-asn1c-generated contains the raw C source files produced by a fork of the open-source ASN1C transpiler, applied to patched IEEE 1609.2.1 ASN.1 specifications. This layer exposes the full structural complexity of the standard and preserves transparency over generated code, which is essential for auditing and long-term maintenance.

Building on top of these generated sources, oscms-asn1c-codecs provides a concrete implementation of the codec layer. This module adapts the raw ASN1C structures into a controlled interface suitable for higher-level use, while remaining strictly compliant with the 1609.2 standard encoding rules.

Above the concrete codecs sits *oscms-codecs-api*, a pure abstraction layer that defines how SPDUs are encoded and decoded, independently of the underlying transpiler. This API is intentionally narrow and explicit: it defines how protocol data is represented at the API boundary, while hiding codec-specific representations and memory layouts. As a result, alternative codec implementations can be introduced without modifying protocol logic.

At the top of the stack, *oscms-codecs-bridge* binds this abstract codec interface to the cryptographic and protocol workflows required by SCMS components. This layer implements the handlers that perform SPDU decoding, signature verification, trust-chain validation, encryption, certificate construction, and response encoding. Importantly, this layer is completely codec-agnostic: all ASN.1 manipulation is performed through *oscms-codecs-api*.

5.2 Abstract Codec API and Extensibility

The *oscms-codecs-api* defines a small and disciplined set of functions that form the contract between protocol logic and codec implementations. These functions fall into four categories:

- Encoding API functions that serialize API-level structures into COER-encoded buffers;
- Decoding API functions that parse COER-encoded data into API-level structures;
- Internal helpers that convert API structures into codec-specific internal representations; and
- Internal helpers that perform the inverse conversion.

This design enforces a strict boundary between protocol semantics and serialization mechanics. While the current API and concrete implementation support only the SPDUs required by the SCMS server, this limitation is intentional rather than structural. The API can be extended to cover additional SPDUs such as those required by V2X client implementations, without architectural changes.

This controlled scope ensures that *oscms-codecs-bridge* remains focused, auditable, and correct, while preserving a clear path toward broader reuse.

5.3 The bridge: Cryptographic Core and SPDU Processing

The defining feature of *oscms-codecs-bridge* is its role as the cryptographic execution engine for OpenSCMS. It implements the handlers used by all SCMS components: Registration Authority, Enrollment Certificate Authority, Authorization Certificate Authority, and Linkage Authority, to process protocol messages.

These handlers perform, in tightly controlled sequence, the decoding of incoming SPDUs, verification of digital signatures, validation of certificate trust chains, decryption of protected payloads, construction of responses according to IEEE 1609.2.1 rules, encryption, and final encoding.

A defining characteristic of *oscms-codecs-bridge* is that protocol processing is expressed as explicit, readable execution flows, rather than as opaque codec invocations. Each handler corresponds to a well-defined protocol operation, and combines decoding, cryptographic verification, data transformation, and encoding in a controlled and auditable sequence.

Rather than exposing low-level cryptographic primitives directly to the server, the bridge presents semantic operations that reflect the intent of the IEEE 1609.2.1 protocol. A typical handler follows a pattern like the following:

```

// 0. Process input and compose internal CMIS representation
OscmsCertManagementPduArgs pdu_args = {0};
...

// 1. Encode CMIS pdu
OscmsOctetBuffer encoded_cert_mngt_pdu = {0};
int rc = oscms_encode_cert_mngt_pdu(&pdu_args, &encoded_cert_mngt_pdu);
...

// 2. Sign and encode Dot2Data-Signed SPDU
rc = sign_encode_dot2_data_signed_nist(...)
...

// Clean up
oscms_empty_octet_buffer(&encoded_cert_mngt_pdu);

// Success
```

Listing 1: *Cryptographic and protocol composition in a SPDU handler - The CMIS Handler*

This pattern illustrates a core design principle: cryptographic correctness and protocol semantics are interleaved intentionally, not layered as an afterthought. Verification always precedes interpretation, and cryptographic protections are applied only after the response structure is fully constructed and validated.

This approach is particularly important for complex protocol messages. The CertificateManagementInfoStatusSpdu (CMIS), for example, is among the most structurally rich SPDUs defined in IEEE 1609.2.1. It contains multiple nested substructures, including certificates, signatures, status codes, and optional fields, each of which must be assembled and protected according to precise rules. In *oscms-codecs-bridge*, the construction of such SPDUs is expressed step by step, making the code both reviewable and traceable back to the standard.

The sequence diagram shown in Figure 5 highlights this orchestration clearly. It shows how encoding and decoding operations are woven together with cryptographic steps, resulting in a final SPDU that is both syntactically valid and cryptographically sound. This explicit composition model is a strong indicator of technical maturity: the code does not rely on hidden side effects or implicit assumptions but instead encodes protocol intent directly into execution flow.

5.4 Cryptographic Utilities and Practical API Design

Supporting the SPDU handlers is a set of carefully designed cryptographic utilities that encapsulate recurring low-level operations while remaining close to the needs of the protocol. Rather than exposing raw OpenSSL calls throughout the codebase, *oscms-*

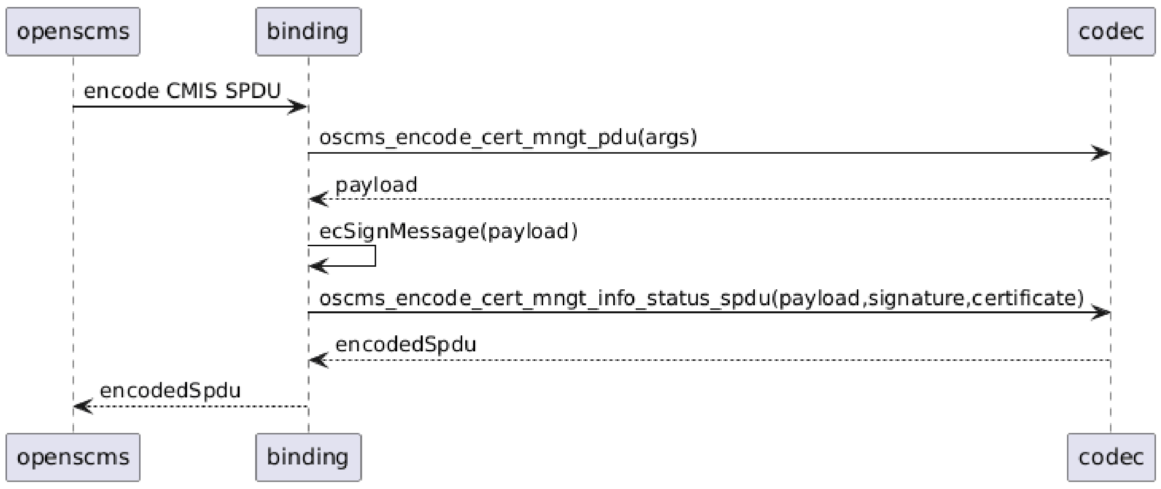

Figure 5: *SPDU Construction Sequence Using* oscms-codecs-bridge

codecs-bridge defines purpose-specific helper functions that reflect IEEE 1609.2.1 concepts.

For example, elliptic curve operations, key handling, and signature generation are exposed through focused interfaces that make intent explicit:

```

int ec_implicit_private_reconstruction_p256_value(
        const uint8_t cat_hash[64], BIGNUM *bn_k, EVP_PKEY *private_key,
            uint8_t *sigma);

int ec_sign_message(
        const OscmsOctetBuffer *message, EVP_PKEY *evp_private_key,
            OscmsHashAlgorithm hash_algorithm, uint8_t *rs);
...
```

Listing 2: *High-level elliptic curve cryptographic utilities*

Encryption and decryption are similarly abstracted, with APIs that clearly distinguish between plaintext buffers, protected payloads, and associated metadata:

```
/**
     *  Decrypt data using ECIES scheme
     ...
     * @return 0 on success, -1 on failure
     */
int ecies_decrypt(...);

/**
     * ECIES encryption:
     *  - Generate a random symmetric key (k)
     *  - Encrypt the plaintext with the symmetric key
     *  - Generate an ephemeral key pair (v, V)
     *  - Generate a shared secret (SS = v*R)
     *  - Compute KDF + HMAC and compose the encrypted key
     * ...
     * @return 0 on success, -1 on failure
     */
int ecies_encrypt(...)
```

Listing 3: *Payload encryption abstraction*

These utilities serve two important purposes. First, they centralize cryptographic correctness, ensuring that sensitive operations are implemented consistently and reviewed in one place. Second, they raise the abstraction level of the codebase, allowing protocol handlers to read more like protocol descriptions than cryptographic implementations.

This design is especially valuable in a standard as intricate as IEEE 1609.2.1. Aligning utility APIs with protocol concepts rather than with cryptographic library primitives, the code becomes easier to audit, easier to extend, and less error-prone.

From a usability perspective, this also makes *oscms-codecs-bridge* approachable for researchers and engineers who may not be experts in low-level cryptography, but who need to reason precisely about protocol behavior. The API guides correct usage by construction, reducing the likelihood of misuse or subtle security flaws.

5.5 Memory Safety and Buffer Management

A notable indicator of the library's maturity is the attention given to memory safety in a low-level language context. The library provides a dedicated abstraction for managing byte buffers, including allocation, resizing, copying, and ownership transfer.

```
SO_EXPORT OscmsOctetBuffer *oscms_octet_buffer_new(void);
...
SO_EXPORT int oscms_octet_buffer_init_from_buffer(
        OscmsOctetBuffer *octet_buffer, const uint8_t *buffer, size_t
           length);
SO_EXPORT OscmsOctetBuffer *oscms_octet_buffer_duplicate(const
    OscmsOctetBuffer *octet_buffer);
...
```

Listing 4: *Octet Buffer Abstraction*

Rather than relying on ad hoc pointer manipulation, *oscms-codecs-bridge* uses these utilities to enforce disciplined handling of binary data. This is particularly important when dealing with encoded SPDUs, cryptographic payloads, and ASN.1 structures, where subtle memory errors can lead to security vulnerabilities or undefined behavior.

By providing explicit buffer utilities, the library reduces the cognitive load on developers and establishes consistent patterns for safe memory handling across the codebase.

This disciplined approach to buffer management is particularly important in security-critical codebases, where memory handling errors can undermine otherwise correct cryptographic designs.

5.6 Reusability Beyond OpenSCMS

Although *oscms-codecs-bridge* was developed as the cryptographic core of OpenSCMS, it was intentionally architected as a standalone library. Its codec-agnostic design, abstract API, and clean separation between protocol logic and system orchestration make it suitable for reuse in other contexts.

For industry, this enables integration into existing SCMS implementations or validation infrastructures without adopting the full OpenSCMS backend. For academic researchers, it provides an open, auditable, and extensible codebase for experimenting with alternative cryptographic strategies, privacy mechanisms, and protocol extensions.

By lowering the barrier to working with IEEE 1609.2.1 at a low level, *oscms-codecs-bridge* contributes not only to an implementation, but also to enabling a foundation for innovation in V2X security.

6 Validation and Practical Usage

The architectural and protocol decisions described throughout this document have been validated through end-to-end interaction with a compliant end-entity (EE) client implementing the IEEE 1609.2.1 specification. All core workflows supported by OpenSCMS, including registration, enrollment certificate provisioning, authorization certificate provisioning, and successor enrollment have been exercised using real protocol messages and cryptographic material.

This validation confirms not only syntactic compliance with the standard, but also semantic correctness of the implemented flows. It demonstrates that OpenSCMS can be used as a functional SCMS backend for testing, validating, and iterating on V2X client implementations.

From both an industrial and academic perspective, this capability is critical. Implementers of V2X devices require a reliable SCMS environment to validate client behavior, certificate handling, and error conditions. openSCMS eliminates the need to implement or license a separate SCMS for this purpose, significantly reducing development and experimentation costs.

7 Current Limitations and Assumptions

While OpenSCMS implements the core IEEE 1609.2.1 workflows required for end-entity interaction and certificate provisioning, certain components, features, and operational assumptions are intentionally out of scope in the current implementation. These limitations reflect both the evolving nature of the V2X standards landscape and the design goal of maintaining a compact, auditable, and extensible SCMS foundation.

7.1 Registration Authority Scope and Missing MA Endpoints

The current implementation of the Registration Authority does not include endpoints related to Misbehavior Authority (MA) interactions. In particular, the MA certificate download interface (Table 22 of the standard) and the misbehavior report submission interface (Table 15) are not implemented.

This limitation exists because OpenSCMS does not currently implement a Misbehavior Authority component. While the SCMS architecture assumes the existence of such functionality, there is no universally standardized and interoperable misbehavior reporting protocol defined for North America, which limits the feasibility of a portable implementation.

7.2 Supported End Entity Types and Certificate Variants

A running instance of OpenSCMS can support multiple types of end entities and certificate requests simultaneously. The following combinations are currently supported:

- OBU clients may request pseudonym certificates or identification certificates

- RSU clients may request identification certificates
- End entities may request either implicit or explicit pseudonym certificates

Implicit certificates rely on Elliptic Curve Qu-Vanstone (ECQV[2])-based keys, reducing certificate size by approximately 64 bytes per certificate. Explicit certificates rely on standard ECDSA P-256 keys and signatures. Both implicit and explicit certificates are generated in full compliance with IEEE 1609.2.1.

All end-entity and certificate authority certificates are assumed to follow the IEEE 1609.2 format, rather than X.509. This aligns with the standard requirements and simplifies cryptographic handling within the SCMS context.

7.3 ASN.1 Version Support and Migration

The OpenSCMS currently supports the IEEE 1609.2.1 ASN.1 specification tagged as 2022-published. The project includes transpiling utilities, based on a fork of the open-source ASN1C transpiler, to convert the IEEE 1609.2.1 ASN.1 files into C code used by the *oscms-codecs-bridge* library.

If newer official ASN.1 specifications are published in the future, these utilities can be reused to regenerate codecs and migrate the implementation with minimal architectural impact.

7.4 Policy Files and Regional Scope

OpenSCMS policy files closely follow the policies defined by the SaesolTech SCMS, which is one of the SCMSs eligible for client certification under the OmniAir Consortium's IEEE 1609.2.1 program.

Current limitation: OpenSCMS does not yet support per-region policy files. Support for multiple regional policy configurations within a single deployment is considered a future enhancement.

7.5 Bootstrapping Trust Anchors and Server Certificates

For testing and experimentation purposes, OpenSCMS provides a bootstrapping script that generates sample elector certificates, a sample certificate authority chain, and a Certificate Trust List (CTL) signed by those electors. These artifacts are generated in COER format and stored on the filesystem, where they can be loaded by SCMS components such as the ECA, ACA, and RA during initialization.

Current limitations: certificates for Misbehavior Authorities and Linkage Authorities are not generated as part of this process, as those components are not fully implemented in the current system.

7.6 CTL Issuance and Elector Assumptions

In operational SCMS deployments, CTLs are typically signed by a quorum of multiple electors. These electors are independent entities responsible for notarizing decisions made by the SCMS Manager, rather than validating the CTL contents themselves.

[2]Elliptic Curve Qu-Vanstone (ECQV) implicit certificate scheme developed by Certicom Research (now a part of BlackBerry).

The SCMS Manager represents a consortium of stakeholders responsible for the long-term operation of the V2X ecosystem, including automotive OEMs, roadside infrastructure operators, traffic management systems, and governmental agencies.

Status: OpenSCMS is not yet integrated with an operational SCMS Manager. The existing CTL bootstrapping process generates sample electors that sign the CTL through a local bootstrapping script, enabling testing and validation but not representing a production governance model.

7.7 Misbehavior Handling and Revocation

The current OpenSCMS implementation does not support misbehavior reporting or certificate revocation based on misbehavior detection. This reflects the lack of a precise and universally adopted misbehavior reporting standard in North America.

While the European ETSI TS 103 759 standard defines interoperable data structures and protocols for misbehavior reporting, North American standards assume the presence of misbehavior handling within the SCMS architecture without defining a concrete message format or protocol.

Although the Linkage Authority (LA) component is implemented and supports deployment of multiple instances, it currently operates as a standalone module. The LA is not integrated into the authorization certificate generation workflow, since OpenSCMS does not yet implement revocation or misbehavior-driven linkage mechanisms.

This limitation is consistent with the fact that misbehavior detection and revocation workflows are out of scope for IEEE 1609.2.1, but it should be clearly noted that the LA integration remains incomplete.

Future direction: misbehavior support is a desired feature for OpenSCMS, ideally implemented in a manner that enables interoperability across North American, European, and other regional standards.

8 Impact, Reusability, and Future Directions

OpenSCMS lowers the barrier to entry for secure V2X infrastructure development by providing an open, standard-compliant, and extensible SCMS implementation. Its impact extends beyond a single deployment or use case.

For industry, OpenSCMS enables rapid prototyping and validation of SCMS-based architectures without reliance on proprietary implementations or toolchains. It provides a concrete foundation for internal products, testing environments, and interoperability efforts.

For academia, OpenSCMS offers a realistic and auditable platform for research on V2X security, privacy, and cryptographic protocols. Researchers can focus on advancing techniques such as alternative pseudonym strategies or enhanced privacy mechanisms, rather than reimplementing foundational infrastructure.

The *oscms-codecs-bridge* library further amplifies this impact by offering a reusable cryptographic and protocol core that can be integrated into other SCMS implementations or extended to support client-side protocol processing.

Future work includes extending codec support to additional SPDUs, enabling direct reuse in V2X clients, supporting alternative ASN.1 transpilers, and implementing additional SCMS components as required by specific deployment scenarios as shown in the next topics.

8.1 Roadmap and Planned Enhancements

This roadmap is intended to provide transparency regarding current TODOs, guide community contributions, and clarify the long-term direction of the project. No specific timeline is implied; priorities may evolve based on standardization progress, deployment feedback, and community involvement.

8.1.1 Architectural Decomposition and New SCMS Components

One of the primary roadmap items is the architectural decomposition of responsibilities currently concentrated in the Registration Authority (RA).

Planned enhancements include:

- Implementation of a dedicated Device Configuration (DC) component responsible for distributing trusted SCMS artifacts, such as certificates and configuration data of other SCMS components.
- Implementation of a Device Configuration Management (DCM) component responsible for managing device lifecycle information, including registration, updates, and operational metadata of end entities.

This separation aims to reduce RA complexity, improve scalability, and better align OpenSCMS with reference SCMS architectures.

8.1.2 Operational Frontend and Management Interface

OpenSCMS currently exposes REST APIs as its primary interaction mechanism and operates strictly as a backend service.

Planned enhancements include the development of an optional frontend interface to:

- Register and manage end entities and devices
- Query device and certificate status
- Download certificates and trusted artifacts
- Visualize system health and operational status
- Monitor quantitative metrics such as throughput, latency, and resource utilization

This interface would significantly improve operational usability for integrators and system operators.

8.1.3 Multi-Region and Multi-PSID Support

The current OpenSCMS implementation assumes a single operational region and a fixed set of PSIDs.

Planned enhancements include:

- Support for multiple regions within a single OpenSCMS deployment
- Support for multiple, region-specific PSID sets

- Association of policy files and certificate constraints with specific regions and PSID groups

These capabilities are essential for deploying OpenSCMS in more complex, real-world environments.

8.1.4 Performance Evaluation and Scalability Analysis

A structured performance and stress evaluation of OpenSCMS has not yet been completed.

Planned activities include:

- Quantitative performance benchmarking under different load scenarios
- Measurement of throughput and response times for core certificate workflows
- Identification of bottlenecks and scalability limits

Special attention is expected to be given to synchronous workflows, such as authorization certificate and successor enrollment certificate downloads requests, which may represent potential performance constraints.

8.1.5 Secure Internal Component Communication

OpenSCMS currently assumes a trusted internal network and does not enforce authentication or encryption between internal components.

Planned enhancements include introducing secure internal communication mechanisms, such as:

- Mutual TLS (mTLS) using internal service certificates
- OAuth2-based service-to-service authentication
- Or any other suggestion

These mechanisms would strengthen the security posture of OpenSCMS, particularly in distributed or cloud-based deployments.

8.1.6 Key Management and HSM/KMS Integration

Private keys and certificates are currently stored on the filesystem of the hosting servers.

Planned enhancements include:

- Integration with Hardware Security Modules (HSMs)
- Support for external Key Management Systems (KMS), such as cloud-based or on-premise solutions

This evolution would ensure stronger key protection guarantees and align OpenSCMS with best practices for cryptographic material management.

8.1.7 Misbehavior Handling and Linkage Authority Integration

Although a Linkage Authority (LA) component is implemented and supports multiple instances, it is not yet integrated into the authorization certificate issuance workflow.

Planned enhancements include:

- Integration of the LA into certificate generation flows once revocation mechanisms are defined
- Exploration of interoperable misbehavior reporting and revocation strategies compatible with multiple regional standards

This work is expected to depend on future standardization efforts, particularly regarding misbehavior reporting protocols in North America.

References

[1] Institute of Electrical and Electronics Engineers, “IEEE Standard for Wireless Access in Vehicular Environments (WAVE)—Security Services for Applications and Management Messages.” IEEE Std 1609.2-2022, Oct. 2022.

[2] Institute of Electrical and Electronics Engineers, “IEEE P1609.2.1 — Draft Standard for Wireless Access in Vehicular Environments (WAVE): Certificate Management Interfaces.” IEEE Standards Association Project P1609.2.1, 2022. Draft standard, not yet ratified.

[3] F. R. Lone, A. Nazir, A. Gupta, S. Sawhney, and S. Ahmed, “Machine learning for misbehavior detection in intelligent transportation systems using BSM data,” in *2024 International Conference on Computational Intelligence and Network Systems (CINS)*, pp. 1–6, 2024.