University of Genoa

Department of Computer Science, Bioengineering, Robotics and System Engineering

DOCTORAL THESIS
COMPUTER SCIENCE AND SYSTEMS ENGINEERING

Confidential and Permissioned Blockchains for the logistics domain

Supervisor:
Prof. Pierpaolo Baglietto

Author:
Stefano Avola

2026 - Cycle XXXVIII

Borsa di dottorato cofinanziata con risorse dell'Unione europea-*NextGeneration EU*
Piano Nazionale di Ripresa e Resilienza la Missione 4, componente 2 ("*Dalla Ricerca all'Impresa*")

Table of contents

List of Figures

List of Tables

Summary

In a port logistics environment, numerous participants interact while containers and related events flow among them. To enable reliable tracking of these activities, a Blockchain-based solution can be employed. A Permissionless Blockchain is not suitable in this context, as participant identities are typically anonymous. Instead, a Permissioned Blockchain is more appropriate, since each participant operates with a well-defined and verifiable identity.

In this work, the selected implementation of a Permissioned Blockchain is the open-source platform Hyperledger Fabric (HLF). The HLF network instance builds upon an existing Platform that leverages HLF's Private Collections feature in combination with a distributed access control list to restrict access to specific data and events.

Furthermore, to enhance confidentiality, privacy, and trust, a Confidential Computing technique has been integrated into HLF. Specifically, I employed Intel Software Guard Extensions (SGX), which ensures that only authorized entities can access plaintext data, even at the hardware level. SGX protects sensitive information stored in memory—and also files on the filesystem, including the Blockchain itself—by isolating them within protected memory areas known as "enclaves". These enclaves prevent unauthorized access from other users, processes, privileged system components, or even the hypervisor.

Additionally, SGX provides a Remote Attestation mechanism that enables an enclave to prove to another enclave or to a traditional application that it is executing specific code within an SGX environment. This mechanism makes it possible for a member of the HLF network to securely share data or events with an external entity whose identity may not be known in advance.

1. Introduction

In port logistics environments, containers and associated events flow among a diverse set of participants, including Port Authorities, Shippers, Terminal Operators, Shipping Agents, and Transport Companies. Tracking these events in a reliable and tamper-proof manner is a fundamental requirement: stakeholders must be able to verify the history and current state of containers as they move through the supply chain. A Blockchain-based solution addresses these requirements effectively, providing an immutable and distributed ledger of transactions that all authorized participants can consult and trust.

A Permissionless Blockchain is not suitable in this context, since participant identities are typically anonymous and accountability is difficult to enforce. A Permissioned Blockchain is more appropriate: each entity must possess a valid identity before joining the network, and access to data and operations is governed by explicit policies. The selected implementation in this work is Hyperledger Fabric (HLF), an open-source Permissioned Blockchain platform. The HLF network instance builds upon an existing solution [1, 2] that leverages HLF's Private Collections (PCs) feature in combination with a distributed Access Control List (dACL) to restrict access to specific data and events among network participants.

Although HLF's mechanisms—including PCs and ACL-based access control—effectively restrict data visibility at the application layer, they offer no protection against privileged system-level adversaries. A cloud provider, a hypervisor administrator, or a compromised OS kernel can observe memory contents and filesystem artifacts regardless of the application-layer policies enforced by HLF, since HLF components process private data in plaintext in main memory. Addressing this vulnerability requires hardware-level protection mechanisms, which are the subject of this work.

To address the identified hardware-level vulnerability, this work integrates Confidential Computing (CC) into the existing HLF platform. CC is a hardware-based paradigm that protects data in use by executing workloads within hardware-isolated Trusted Execution Environments (TEEs), preventing unauthorized access even from privileged software layers such as the hypervisor or the operating system kernel.

According to [3], CC solutions can be classified into three levels of protection:

1. **Protect keys**: CC is employed to protect cryptographic keys rather than the workload itself, offering a viable alternative to costly and inflexible Hardware Security Modules. This is the approach adopted by Ren et al. [4].
2. **Protect apps or containers**: CC is used to protect the execution of workloads; however, end-to-end protection—encompassing aspects such as Remote Attestation (RA) and key management—is not addressed.
3. **Protect deployments**: all of the above protections are addressed, ensuring that the complete system deployment is protected end-to-end.

The approach adopted in my work corresponds to the third level, as described in further detail in the following sections.

In this work, Intel Software Guard Extensions (SGX) is adopted as the TEE technology, in combination with the Gramine library OS, which enables existing applications to run within SGX enclaves without source code modifications. This combination allows all HLF network components—Peers, Orderers, Chaincode processes, and REST-based client applications—to execute within enclaves, extending the trust boundary to the hardware level. Furthermore, SGX's Remote Attestation (RA) mechanism is leveraged to allow HLF network members to securely share sensitive data with external third-party software (TPS) whose identity and integrity can be cryptographically verified before any data is disclosed.

The port environment has been simplified as illustrated in Fig. 1, and the main participants interacting with the HLF instance are defined as follows:

- The Port Authority administers the HLF network, as described in section 2.2.
- A Shipper creates a shipment associated with one or more containers and tracks their status through the Blockchain.
- Shipping Agents, Transport Companies, and Terminals submit events related to those containers to the Shipper's shipment via the Blockchain, ensuring that such events are verifiable and tamper-proof.
- A Shipper may share selected events with an external TPS—for instance, to support machine learning-based analyses—provided that the TPS can be verified as genuine and as executing within a confidential environment, by means of the RA mechanism.

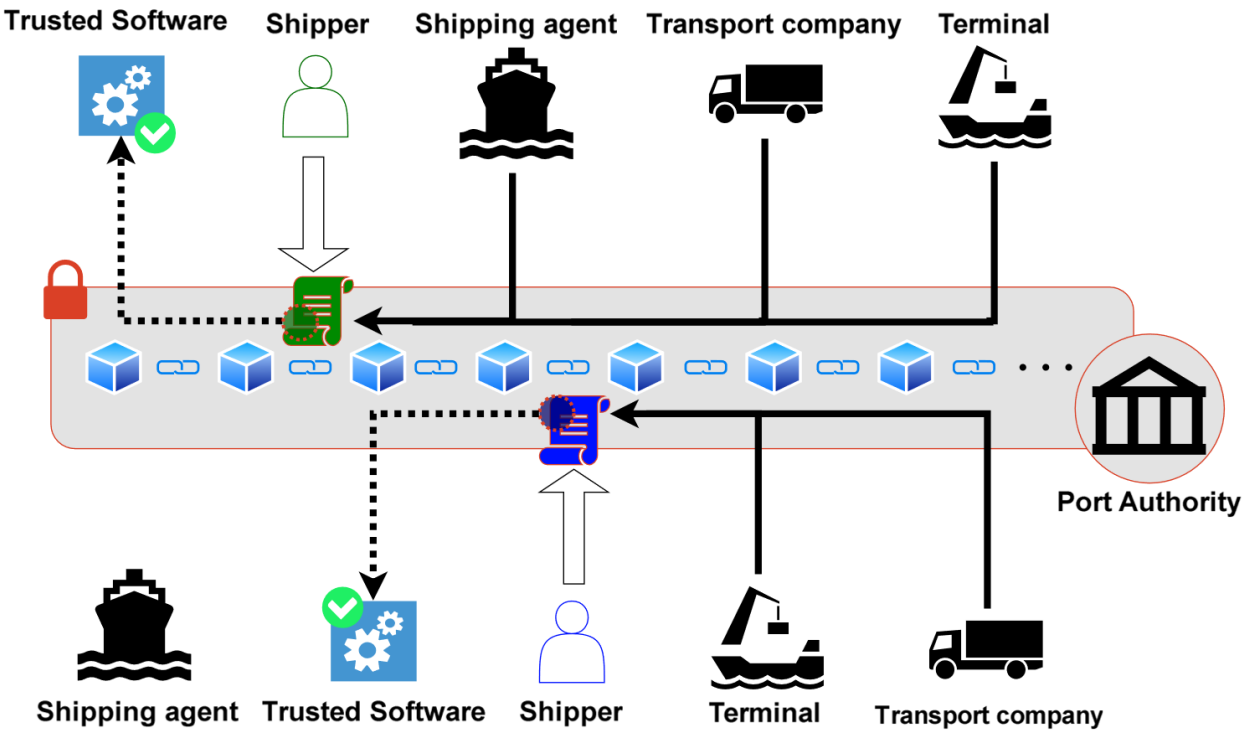

Fig. 1 Confidential HLF network layout

The main contributions of this work are: a Gramine-based deployment strategy (referred to as the Graminized Platform, or GP) that enables all HLF components to execute within SGX enclaves; a novel Peer Operator (PO) component—a RESTful service running inside the enclave—that allows administrative Peer operations to be performed without exposing cryptographic material outside the trusted boundary; a RegisterEnroll component that generates and seals the cryptographic materials required by each HLF component during startup; and a novel RA-REST component that enforces per-request RA for external TPS access to private data. The deployment also required targeted modifications to both Gramine (directory rename support) and HLF (Peer library refactoring, revision of CCAAS deployment scripts, and removal of loopback-address detection logic).

The proposed architecture has been validated on the Port of Genoa scenario. Performance benchmarks conducted with Grafana k6[1] show that the overhead introduced by the graminized deployment is moderate: the mean p95 latency overhead across all tested workloads and concurrency levels is approximately 1.3× compared to a standard HLF deployment, and tends to decrease under higher concurrency. The primary operational limitation is memory consumption, which is substantially higher than in a standard HLF deployment due to Gramine's per-enclave memory model, amounting to approximately 113 GB in total for the experimental setup.

Each goal of this work is addressed by one or more contributions and its achievement is demonstrated in this thesis, as summarized below.

G1 — Extend the confidentiality and privacy guarantees of an HLF deployment beyond the application layer, so that they hold even against a privileged or compromised hosting platform (hypervisor, operating system, or cloud provider) able to read main memory and filesystem artifacts.

Contribution: GP — the integration of Intel SGX into HLF through Gramine, combining enclave-based execution of the HLF components with MRSIGNER-based sealing of their cryptographic material and of the Blockchain state.

Argued in: threat model and security assumptions, including the defense-in-depth analysis of an SGX-guarantee failure (Section 3.2.1-3.2.1.1); residual-gap analysis justifying the layered design (Table 14) and the threat-model summary (Table 16); and the layered privacy guarantee discussed in the Conclusions, where the security properties contributed by each architectural layer are summarized (Table 21).

G2 — Execute all HLF network components (Peers, Orderers, Chaincode, and the REST-based client applications) unmodified within SGX enclaves.

Contribution: the adoption of the Gramine framework (and GSC) together with the targeted modifications I introduced — the PO, the RegisterEnroll component, the per-Organization CCAAS adaptation, and the source-level patches to Gramine and HLF (directory-rename

[1] https://grafana.com/docs/k6/latest/

support, Peer library refactoring, CCAAS deployment scripts, and removal of the loopback-address detection logic).

Argued in: the Graminized architecture and its components (Sections 3.2.2–3.2.7); the component-origin mapping (Table 12) and the per-component graminization/sealing/status summary (Table 13).

G3 — Enable secure and verifiable data sharing between an HLF member and an external TPS whose identity is not known in advance.

Contribution: the RA-REST component — a per-request RA-based REST service leveraging Gramine's RA-TLS library together with the one developed by me.

Argued in: the RA-REST design and behavior (Sections 3.2.8–3.2.9, Fig. 19, Appendix D) and the trusted/untrusted client examples that exercise the privacy-enforcement policy (Section 3.2.8.3).

G4 — Assess the practical cost, in terms of performance and memory overhead, of the graminized deployment on a realistic scenario (the Port of Genoa), relative to a standard HLF deployment.

Contribution: the experimental evaluation and its benchmark methodology (Grafana k6), comparing the fully graminized deployment against the standard HLF deployment.

Argued in: the experimental results on the Port of Genoa use case (Section 3.3; Table 18-Table 20).

The remainder of this thesis is organized as follows. Section 2 presents the state of the art, covering Blockchain technology and the HLF platform (Sections 2.1–2.2), an overview of Confidential Computing (Section 2.3), the Intel SGX technology (Section 2.4), and the Gramine framework together with its main alternatives (Sections 2.5–2.6). Section 3 describes the proposed solution and its evaluation: first the existing HLF-based logistics platform (Section 3.1), then the Graminized architecture and all its components (Section 3.2), and finally the experimental evaluation on the Port of Genoa use case (Section 3.3). Section 4 presents the conclusions and a discussion of current limitations and future work directions (Section 4.1). Additional material — the HLF transaction flows, a worked Intel SGX SDK example, an EGo sealing example, and the flowcharts of the RA library — is provided in Appendices A–D.

This is an industrial doctorate, and the industrial setting provided access to a concrete logistics scenario—the Port of Genoa.

During the preparation of this thesis, I used ChatGPT (versions 5.2 and 5.3) and Claude (Sonnet 4.6 and 5) to improve text presentation, formatting, and linguistic refinement. I have reviewed and edited the output and take full responsibility for the content of this thesis.

2. State-of-the-art

In this section, the technological foundations of my work are introduced. First, Blockchain technology is discussed, including the specific platform adopted. Subsequently, the principles of Confidential Computing, together with the related implementations and tools used in this work, are presented.

2.1. Blockchain

The main technology on which this work is based is the Blockchain. As the name suggests, a Blockchain is a chain of blocks, where each block is a data structure that contains data and, if necessary, related cryptographic material such as public keys, certificates, and digital signatures. In addition, each block contains metadata—including the block number and creation timestamp—as well as the cryptographic hash of both the current block and the previous block. The inclusion of these hashes links the blocks together and provides the Blockchain with tamper-evident properties, thereby ensuring the integrity of the data.

Blockchains (initially known as a "distributed database") are typically not centralized but distributed across a network of peers. Through consensus mechanisms, these peers maintain a consistent and synchronized copy of the Ledger across the network. For this reason, Blockchain systems are often referred to as "distributed ledgers". In some Blockchain platforms like HLF, the Ledger consists of two components: the Blockchain itself and the World State. The Blockchain stores the complete history of all transactions, whereas the World State represents the most recent state of the data derived from those transactions. A graphical illustration of the Blockchain structure is shown in Fig. 2 (taken from [5]).

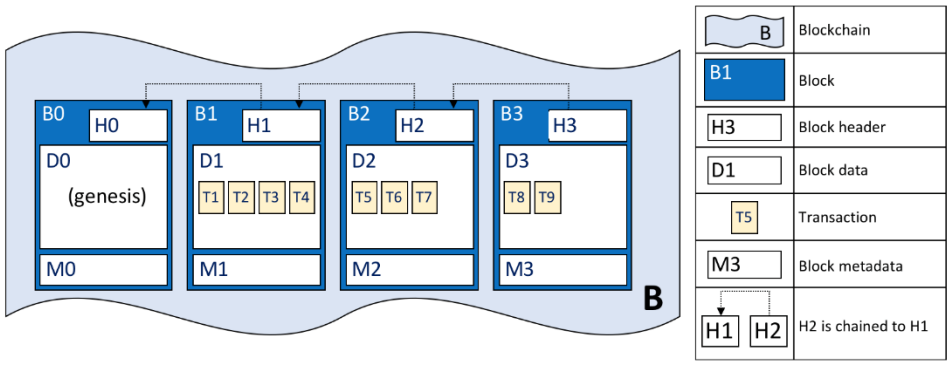

Fig. 2 Blockchain structure

The genesis block is the first to be created and it is a special block that initializes the Blockchain. Unlike subsequent blocks, it typically does not contain transactions. Instead, it includes configuration information required to initialize the network, as is the case in HLF [5] .

Consensus mechanisms enable network nodes to reach agreement on the validity and ordering of operations, commonly referred to as transactions, included in a block [6]. One widely known consensus protocol is Proof-of-Work, used by Bitcoin and by Ethereum (which now also supports Proof-of-Stake). In Proof-of-Work systems, network participants—known as

“miners”—compete to solve a computationally intensive cryptographic puzzle. For example, they must find a nonce such that the resulting hash satisfies a specific condition, such as having a defined number of leading zeros. The first participant to solve the puzzle gains the right to propose the next block in the blockchain. Because the outcome depends on probabilistic competition, this mechanism is generally described as probabilistic consensus [7]. Other consensus mechanisms, including those used by HLF, can instead be considered deterministic. In these systems, specific nodes are responsible for ordering transactions and producing blocks. Once a block has been validated and appended to the chain, it is considered “final and correct” [7], thereby avoiding the possibility of forks—situations in which multiple competing branches of the Blockchain temporarily coexist.

Another important concept associated with Blockchain systems is that of Smart Contracts. Smart Contracts represent the business logic governing interactions with the Blockchain. They define which operations or transactions can be performed, by whom, and under which conditions. Through these rules, Smart Contracts regulate how the World State evolves, effectively enabling its state transitions.

Blockchains can generally be categorized into two types:

- **Permissionless Blockchains**, in which no particular “permission” is required to join the network. Participants do not have a specific identity. An example is Bitcoin, where each participant is represented by an anonymous address rather than a real-world identity.
- **Permissioned Blockchains**, in which participants must be authenticated and authorized before joining the network. In these systems, each entity has a known and verifiable identity. This is the model implemented by HLF, the platform used in the system developed for this thesis.

2.2. Hyperledger Fabric

It is an open-source project founded in 2015 by the Linux Foundation that implements Permissioned Blockchains.

HLF uses a public key infrastructure (PKI) to define participant identities. Consequently, every entity wishing to join an HLF network must possess valid cryptographic keys and digital certificates (X.509) that are recognized by the network. Identity management is handled by a core HLF component called the Membership Service Provider (MSP).

In HLF, a Blockchain network is organized into logical structures called “Channels”. A Channel enables a set of participating nodes to communicate with one another. Each Channel maintains an independent Ledger that is isolated from those of other channels.

Any node that joins a Channel must belong to an Organization. Organizations represent entities participating in the network, such as banks, Port Authorities (PAs), multinational companies, logistics operators, or individual stakeholders.

HLF defines two main types of network nodes: Peers and Orderers. A Peer is a node that maintains a local copy of the Ledger, which it synchronizes with other Peers across the network, including peers belonging to different Organizations. Peers also interact with client applications that invoke Smart Contracts deployed within the Channel. There are 3 types of Peers [6], even if a single Peer may have multiple types at the same time:

- **Anchor Peers**. They serve as communication endpoints for cross-Organization information exchange by interfacing with the Anchor Peers of other Organizations,
- **Endorsing Peers**. They receive transaction proposals from client applications, execute the corresponding Smart Contract, and produce signed endorsements for the transaction results,
- **Leading Peers**. When multiple Peers exist within the same Organization, a leader is selected using the Raft[2] consensus algorithm. Moreover, they act as communication intermediaries between an Organization's Peers and the Ordering Service.

Orderer nodes form the Ordering Service, a specialized component responsible for receiving transaction proposals (more details later), ordering them, and packaging them into blocks that are appended to the Blockchain. The generated blocks are then disseminated to the Leading Peers of the participating Organizations. When multiple Orderers are present, the leader election is also managed using the Raft algorithm.

Consensus in HLF is achieved when Peers validate blocks received from the Ordering Service. Upon receiving a new block, Peers verify that the included transactions satisfy the endorsement policies and correspond to previously endorsed transaction proposals. If the validation is successful, the block is appended to the Peer's local copy of the Blockchain.

In HLF, the Ledger consists of two components: the Blockchain and the World State. The Blockchain stores the immutable sequence of all transactions and can be accessed through appropriate application interfaces or directly from the Peer's filesystem (since it is a file). The World State, instead, represents the current state of the Ledger. It is maintained using a key–value database such as LevelDB or CouchDB. In the system developed for this project, only CouchDB has been used. Specifically, one database instance stores the global Ledger state, while additional databases are created for each Private Collection (more details later).

Client applications interact with the network by invoking transactions defined in Smart Contracts. In HLF terminology, a Chaincode represents a collection of Smart Contracts. Before Chaincode can be used, it must undergo a deployment and approval process involving the participating Organizations. The set of Organizations required to approve the Chaincode and their transactions are determined by configurable policies.

HLF supports multiple types of policies that operate at different levels:

1. **Chaincode-level endorsement policy**. This is the most general policy and defines the endorsement requirements for Chaincode lifecycle operations and transaction

[2] https://raft.github.io

validation. It can specify at Chaincode-deployment-time the subset of Organizations whose approval is required for transactions executed by the Chaincode.
2. **Private data distribution policy**. More than a policy in itself, it is a configuration that defines which Organizations can manage private data of a Private Collection (PC).
3. **Collection-level endorsement policy**. Similar to the Chaincode-level endorsement policy, but applied specifically to transactions involving a particular PC. It is defined at Chaincode-deployment-time.
4. **Key-level endorsement policy**, a.k.a. **State-based endorsement policy**. This policy defines access and endorsement requirements for individual data entries in the Blockchain. It can be dynamically specified during the execution of a Chaincode transaction.

Each policy level introduces additional constraints relative to the preceding level, meaning that policies at lower levels (e.g., Key-level) add requirements to the ones imposed by higher-level policies (e.g., Collection-level).

HLF distinguishes between submit transactions, which modify the Ledger state and follow a multi-phase endorse–order–commit flow involving the client application, the endorsing Peers, and the Ordering Service, and query transactions, which only read the state and are served directly by an endorsing Peer without involving the Ordering Service. As previously mentioned, HLF provides mechanisms for managing private data, allowing certain data to be accessible only to a subset of Organizations. Because of their confidential nature, private data transactions follow a flow that differs from the standard one: only the hash of the private data is disseminated through the Ordering Service and to unauthorized Peers, whereas authorized Peers reconcile the received hash against their locally held plaintext, temporarily kept in a dedicated structure known as the transient data store. The detailed private data transaction flow, together with the standard flow, is reported in Appendix A.

It is worth noting that the privacy mechanisms provided by HLF, including PCs, operate exclusively at the application layer. An adversary with privileged access to the hosting platform—such as a cloud provider, hypervisor, or system administrator with root-level privileges—can observe memory contents, filesystem artifacts, and inter-process communication irrespective of the data segregation policies enforced by HLF. Peer nodes process private data in plaintext in main memory, meaning that even without violating HLF's access control policies, a compromised host can read sensitive data. Addressing this gap requires hardware-level protection mechanisms, which are discussed in the following sections.

2.3. Confidential Computing

Nowadays data that flows through a network ("data in transit" [8]) is typically encrypted to preserve confidentiality. Similarly, data stored in persistent storage ("data at rest" [8]) is often protected through encryption mechanisms. However, when data is actively processed by an application—referred to as "data in use" [8] —it is generally handled in plaintext. During computation, the CPU executes instructions and operates on data that is loaded directly from

RAM, without encryption. Consequently, privileged entities such as a superuser, an OS, or a hypervisor could potentially access sensitive data residing in memory.

A simple practical demonstration of this vulnerability can be performed in a Linux environment. Consider a GoLang[3] program that dynamically generates a TLS certificate and its corresponding private key without storing them on disk, and subsequently exposes an HTTPS server using them.

```
 1. package main
 2.
 3. import (
 4.       "crypto/rand"
 5.       "crypto/rsa"
 6.       "crypto/tls"
 7.       "crypto/x509"
 8.       "crypto/x509/pkix"
 9.       "encoding/pem"
10.       "log"
11.       "math/big"
12.       "net/http"
13.       "time"
14. )
15.
16. func main() {
17.       // 1. Generate private key
18.       key, _ := rsa.GenerateKey(rand.Reader, 2048)
19.
20.       // 2. Make certificate template
21.       template := x509.Certificate{
22.                 SerialNumber: big.NewInt(1),
23.                 Subject:      pkix.Name{CommonName: "localhost"},
24.                 NotBefore:    time.Now(),
25.                 NotAfter:     time.Now().AddDate(1, 0, 0),
26.                 KeyUsage:     x509.KeyUsageKeyEncipherment | x509.KeyUsageDigitalSignature,
27.                 ExtKeyUsage:  []x509.ExtKeyUsage{x509.ExtKeyUsageServerAuth},
28.       }
29.
30.       // 3. Create self-signed certificate
31.       certDER, _ := x509.CreateCertificate(rand.Reader, &template, &template, &key.PublicKey, key)
32.
33.       // 4. Encode cert and key into PEM blocks (in memory)
34.       certPEM := pem.EncodeToMemory(&pem.Block{Type: "CERTIFICATE", Bytes: certDER})
35.       keyPEM := pem.EncodeToMemory(&pem.Block{Type: "RSA PRIVATE KEY", Bytes:
x509.MarshalPKCS1PrivateKey(key)})
36.
37.       // 5. Load into tls.Certificate
38.       cert, _ := tls.X509KeyPair(certPEM, keyPEM)
39.
40.       // 6. HTTPS server
41.       server := &http.Server{
42.                 Addr:    ":8443",
43.                 Handler: http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
44.                           w.Write([]byte("Hello!"))
45.                 }),
46.                 TLSConfig: &tls.Config{Certificates: []tls.Certificate{cert}},
47.       }
48.
49.       log.Println("Serving on https://localhost:8443 ...")
50.       log.Fatal(server.ListenAndServeTLS("", "")) // cert is already in TLSConfig
51. }
52.
```

If the following command is executed with root privileges:

[3] https://go.dev

```
gcore P
```

where P is the PID of the HTTPS server process, the command generates a memory dump of the process. By analyzing the resulting dump file—for instance by searching for the string `"BEGIN RSA PRIVATE KEY"`—the private key stored in the process memory can be retrieved, as illustrated in Fig. 3:

```
00 00 00 00 00 00 00 00 2D 2D 2D 2D 2D 42 45 47 49 4E 20 52 53 41 20 50 ........-----BEGIN RSA P
52 49 56 41 54 45 20 4B 45 59 2D 2D 2D 2D 2D 0A 4D 49 49 45 6F 67 49 42 RIVATE KEY-----.MIIEogIB
41 41 4B 43 41 51 45 41 77 6C 4A 73 56 79 46 73 78 39 58 59 34 5A 48 61 AAKCAQEAwlJsVyFsx9XY4ZHa
79 78 79 68 72 2B 6D 6F 73 56 49 77 71 6E 4A 36 2F 4B 38 4D 5A 6D 30 76 yxyhr+mosVIwqnJ6/K8MZm0v
6C 6A 53 45 51 67 52 4E 0A 4A 57 4C 31 4E 2B 51 44 4B 48 37 39 74 34 34 ljSEQgRN.JWL1N+QDKH79t44
4A 46 4A 64 70 78 68 41 69 6A 4C 31 77 50 71 66 61 38 59 73 72 43 53 7A JFJdpxhAijL1wPqfa8YsrCSz
30 51 4C 4D 51 71 4E 52 45 63 34 78 48 74 6C 34 43 37 41 66 54 4F 45 34 0QLMQqNREc4xHtl4C7AfTOE4
34 0A 67 32 5A 31 32 7A 2F 4D 48 4C 66 35 77 72 41 70 74 31 72 46 70 47 4.g2Z12z/MHLf5wrApt1rFpG
75 42 4C 4A 6B 46 6B 4C 30 47 34 56 76 50 45 61 4B 36 33 31 34 73 65 32 uBLJkFkL0G4VvPEaK6314se2
```

Fig. 3 Private Key dump from RAM

Confidential Computing (CC) is an emerging technology designed to address this limitation by protecting sensitive data in memory and during computation, i.e., “data in use”, as defined by the Confidential Computing Consortium [8], a community focusing on accelerating the adoption of CC technologies. In this threat model, several system components—including other applications on the same host, privileged users, the OS, the hypervisor, and the virtual machine monitor (VMM)—must be considered hostile. These entities can therefore be classified as “unauthorized entities”.

Confidential Computing achieves protection of data in use through the use of Trusted Execution Environments (TEEs). A TEE is defined as an execution environment that provides “a level of assurance of data integrity, data confidentiality, and code integrity” [8]. TEEs are typically implemented in hardware to operate at the lowest possible level of the computing stack. This design choice reflects a fundamental principle of system security: the security of a layer depends on the security of the underlying layers [8]. Implementing TEEs at the hardware level also reduces external dependencies—such as device drivers and peripheral vendors—and consequently minimizes the potential attack surface [8]. Nevertheless, “there is no absolute security” [9].

According to [9], a TEE provides assurance with respect to several key properties:

- **Data confidentiality**: data in use within a TEE cannot be observed by unauthorized entities.
- **Data integrity**: data in use within a TEE cannot be modified, injected, or tampered with by unauthorized entities.
- **Code integrity**: the code executed within a TEE cannot be altered by unauthorized entities.

Some TEEs may also provide additional capabilities, including:

- **Code confidentiality**: the same for data confidentiality but referred to code.
- **Authenticated Launch**: before running an application within a TEE, it can check for authorization or authentication.

- **Programmability**: the ability to execute arbitrary application code, although some TEEs may impose restrictions on the supported instruction set.
- **Attestability**: the capability to prove that a specific workload is executing within a genuine TEE. This concept is particularly important in CC and will be discussed in more detail in the following sections.
- **Recoverability**: mechanisms that allow recovery from a compromised or non-compliant state.

Thus, the hardware ultimately serves as the root of trust, since it provides guarantees ("attest") that a workload—such as an application or even an entire virtual machine—is indeed executing within a TEE [10].

2.3.1. Packaging models

The Confidential Computing Consortium (CCC) has defined several "packaging models" [11], i.e. different granularity levels of isolation. They represent different levels of protection boundaries, corresponding to the Trusted Computing Base (TCB). The TCB consists of a set of hardware and software components whose vulnerabilities could compromise the security of the entire system [10]. In general, a smaller TCB implies stronger security [10]. Within these packaging models, the workload executing inside the TCB remains invisible to entities outside its boundary [10].

The main packaging models defined by the CCC are the following:

- **Confidential library**. In this model, an application is divided into confidential and non-confidential components. The confidential component—referred to as an "enclave" in Intel SGX—executes within a TEE and is protected from other libraries within the same process, whether confidential or not, as well as from the whole hosting environment.
- **Confidential process.** In this model, an entire process is executed inside a TEE. The process is therefore isolated from other processes in the host system and from the host environment itself.
- **Confidential container.** In this configuration, the entry-point process of a container is executed within a TEE. This protects the container from other containers running on the same host and from the host environment.
- **Confidential VM.** In this model, an entire VM is executed inside a TEE. The VM is therefore isolated from other VMs, the hypervisor, and the rest of the host system.

These packaging models are typically shipped with specialized software components designed to operate within the TEE. Such software facilitates the development of applications that run securely inside TEE and provides mechanisms for performing operations such as Remote Attestation and making decision based on Attestation results. As discussed previously, CC ensures that both data and code remain protected not only during execution within the CPU but also while residing in RAM. Various techniques exist to achieve this protection, as summarized in [11] and illustrated in Table 1, where different TEE implementations are

compared according to their supported isolation mechanisms. Implementations that do not support a specific isolation technique are indicated with a red "X".

	CPU Addressability Isolation			Memory Isolation
	Access Control Validation	**Address Translation**	**Paging Control**	**RAM Encryption**
Description	Only certain processes can access memory areas	Outside the TEE, segmented memory areas are not directly addressable	Non-confidential processes are not run in the CPU concurrently with confidential ones	Encrypt the memory area where data and code reside.
Intel SGX	✓	✗	✗	✓
Intel Trust Domain Extensions (TDX)	✗	✓	✓	✓
AMD Secure Encrypted Virtualization - Secure Nested Paging (SEV-SNP)	✗	✗	✓	✓
ARM Confidential Compute Architecture (CCA)	✓	✗	✗	✓

Table 1 TEEs implementations and supported methodologies

A brief overview of these approaches is provided in Table 2. The following technologies represent some of the most relevant implementations currently available. However, my work focused specifically on Intel SGX, which has been studied in greater depth (more details in following sections) and employed as the primary CC technology in the proposed system. The rationale for selecting this technology is discussed in Section 2.6.5.

TEE Implementation	Packaging Model	Description
Intel SGX	Confidential library	Available since 2015, Intel SGX is a set of CPU instruction set extensions that enable user applications to create protected memory areas known as "enclaves".
Intel TDX	Confidential VM	Intel TDX represents Intel's more recent CC technology and extends the enclave concept to entire VMs. In this architecture, protected VMs—referred to as "Trusted Domains"—are isolated from the virtual

		machine monitor (VMM) and the hypervisor. Each virtual machine is assigned a unique memory encryption key [12].
AMD SEV-SNP	Confidential VM	SEV was implemented in 2016 by AMD. It enables the creation of encrypted VMs through a hardware-based mechanism known as Secure Memory Encryption (SME). In this architecture, VM memory is encrypted using keys generated at system boot, with a unique key assigned to each VM and managed by a dedicated microcontroller [13, 14]. Subsequent generations—SEV-ES (Encrypted State) and SEV-SNP (Secure Nested Paging)—introduce additional protections.
Arm (CCA)	Confidential VM	The Arm CCA defines a framework in which the hypervisor retains control over VMs while being prevented from accessing the VM's code, memory, or execution state. This is possible thanks to an isolation space called "Realm", which operates separately from the traditional execution space of the system [15].

Table 2 Main TEEs implementations description

Another relevant implementation is provided by Amazon Web Services through AWS Nitro Enclaves. This technology is available as a feature of Amazon EC2 instances and enables the creation of highly isolated virtual machines, known as "enclaves", derived from an existing EC2 instance. The parent EC2 instance allocates resources to the enclave and acts as its "parent instance". The enclave runs its own kernel and operates with dedicated memory and CPU resources that are partitioned from the parent instance. Importantly, enclaves do not have access to persistent storage, external networking, or interactive access. Communication with the parent instance is possible only through secure local communication channels, typically implemented via local sockets [16].

2.3.2. Attestation

An important concept in CC is Attestation. It is defined as "the validation of a hardware-signed Report (an 'Attestation Report') containing measurements of the TCB" [11]. In practice, it is a mechanism that allows a party—whether operating within a TEE or not—to verify that a remote counterpart is genuine and that its workload is executing inside a trusted environment.

An application running inside a TEE is shipped with its "measurement", which is a cryptographic hash derived from the application's code, data, and build process. This measurement uniquely represents the software components that constitute the TCB of the enclave [17]. During the Attestation process, the TEE generates a Report that includes these measurements along with additional metadata describing the execution environment. The Attestation Report is then cryptographically signed by the underlying hardware, producing an "Attestation Signature" [17]. A verifier can validate this signature to confirm that the Report was generated by genuine trusted hardware. The verifier may also compare the reported measurements against expected

values to determine whether the executing code corresponds to a trusted application. Based on this verification process, the verifier decides whether or not to trust the remote workload.

Notice that Attestation enables the establishment of a trust relationship between previously unknown parties. This capability is particularly relevant in my work, since — as discussed in the following sections— it allows a participant of a HLF network to establish trust with an entity external to the network.

There are 2 kinds of Attestations: Local Attestation and Remote Attestation. Their specific implementation details are discussed later in the section dedicated to Intel SGX.

2.4. Intel SGX

Intel SGX is a set of CPU instructions that enables the creation and management of "enclaves". Enclaves are protected memory areas in which application code and data can be executed and stored confidentially. These areas are isolated from the rest of the system, ensuring that privileged entities—such as an OS, a root user, or a hypervisor—cannot access their contents. As described in [18], this protection is achieved through two main mechanisms:

1. **Controlled enclave entry**. The enclave environment can only be entered through specific SGX instructions that perform strict security and protection checks.
2. **Memory encryption and integrity protection**. The portion of RAM associated with an enclave is transparently encrypted, with replay protection, using a key stored within the CPU. This key is accessible only to the processor and is randomly regenerated at each power cycle.

As a result, unauthorized entities cannot directly access, read, or modify enclave memory. Only code executing inside the enclave itself can interact with the enclave's protected memory area, thereby preserving the confidentiality and integrity of both data and code.

Although my work did not focus on the security analysis of Intel SGX, it is important to note that it has been subject to several classes of attacks, including denial-of-service attacks and cache-based side-channel attacks. Various mitigation techniques have been proposed in the literature, with differing levels of practicality and effectiveness [17, 19, 20].

Before describing how applications can be executed within an enclave and how Attestation mechanisms are performed, it is useful to introduce the memory layout and the main architectural components of Intel SGX. Detailed analyses of the SGX architecture are provided by Costan et al. [17] and [18]. The following section summarizes only the most relevant concepts necessary for understanding the system used in this work. For a comprehensive treatment, the reader is referred to the aforementioned references.

2.4.1. Memory layout

The first relevant component in Intel SGX memory architecture is the Processor Reserved Memory (PRM), which is a protected subset of the RAM. The PRM hosts the code and data of

SGX enclaves and is inaccessible to other software components, including the OS, privileged applications, and DMA.

Within the PRM, enclave data and code are stored in the Enclave Page Cache (EPC). The EPC is divided into fixed-size pages of 4 KB, allowing individual pages to be assigned to different enclaves. This design enables multiple enclaves to coexist on the same host simultaneously. Although the OS is responsible for allocating and freeing EPC pages, direct access to these pages is restricted exclusively to enclaves. After enclave initialization, data and code cannot be arbitrarily loaded into EPC memory anymore. In fact, when an EPC page is allocated, it is immediately initialized, even if the corresponding code or data is loaded from non-PRM memory. Since the OS is not trusted, SGX CPUs verify the correctness of the memory management decisions performed by the OS. For example, the CPU ensures that a single EPC page cannot be allocated simultaneously to multiple enclaves. This verification is performed through the Enclave Page Cache Map (EPCM), which records the OS decisions for each EPC page. The EPCM is implemented as an array containing one entry per EPC page and is used exclusively for security checks. Fig. 4 (taken from [18]) provides an overview of the SGX physical memory layout.

Each EPCM entry contains 3 fields:

- `valid`: indicates whether the corresponding EPC page is allocable or not.
- `PT`: specifies the type of page. Typical values include `PT_REG`, used for pages storing enclave code and data, and `PT_SECS`, which references pages containing the SGX Enclave Control Structures (SECS), discussed below.
- `ENCLAVESECS`: identifies the enclave that owns the EPC page, enabling access control mechanisms.

Because each EPCM entry identifies a single enclave owner, EPC pages cannot be shared among enclaves. Consequently, enclaves cannot communicate through shared EPC memory; instead, inter-enclave communication must occur through untrusted non-EPC memory.

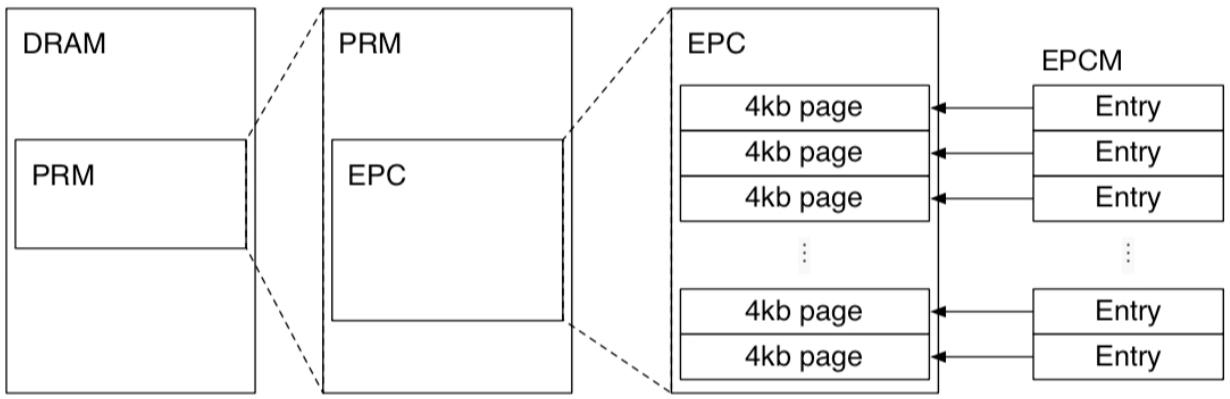

Fig. 4 SGX physical memory layout overview

Each enclave is associated with a SECS, which stores enclave's metadata. This includes configuration parameters such as the debug flag, which determines whether debugging operations on enclave memory are permitted. The SECS is stored in an EPC page of type `PT_SECS`, allowing sensitive information—such as enclave measurements—to be securely stored. Notably, the SECS is not mapped into the enclave's address space and is accessible only by the CPU.

To map enclave code and data stored in EPC pages, each enclave defines an Enclave Linear (or Virtual) Address Range (ELRANGE). ELRANGE is a portion of the enclave's virtual address space that is considered trusted, whereas memory outside this region is treated as untrusted. The boundaries of ELRANGE are specified in the SECS using a base (`BASEADDR` field) and a size (`SIZE` field).

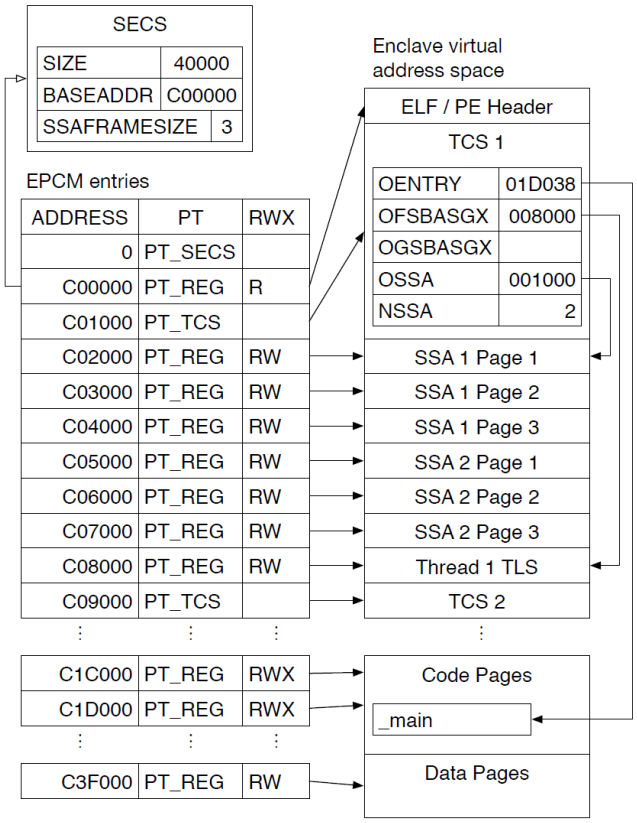

Fig. 5 Example of enclave's virtual address space

Multiple threads may execute concurrently within the same enclave. When creating an enclave, its author must specify the maximum number of threads that may execute within it. This requirement arises because each thread is associated with a Thread Control Structure (TCS). The TCS is stored in an EPC page whose EPCM entry type is `PT_TCS`. Although enclave code cannot directly access the TCS structure, it may be inspected when the enclave operates in debug mode. During enclave execution, hardware exceptions may occur. Handling such exceptions typically requires a transition to a higher privilege level, invoking the related system's hardware exception handler. To prevent the exposure of the enclave's execution state during this transition, the CPU securely stores the thread's execution context in a trusted structure known as the State Save Area (SSA). Each SSA is stored in an EPC page of type `PT_REG`, meaning that enclave code can access it if required. The maximum number of EPC pages available to an enclave for its SSA is specified in the `SSAFRAMESIZE` field of the SECS. In addition, each TCS defines both the number of available SSAs (NSSA) and the location of the first SSA within the enclave's virtual address space. Fig. 5 (taken from [17]) illustrates an example layout of the enclave virtual address space.

An enclave author acts as a Certification Authority (CA) for its enclave. Therefore, the author provides the enclave with a certificate represented by a data structure known as the Signature Structure (SIGSTRUCT). This structure is generated by an enclave build toolchain, which has access to the enclave author's private key. It also sets the fields as reported on the left in Fig. 6 (taken from [17]). Moreover, SIGSTRUCT contains both metadata and a cryptographic signature.

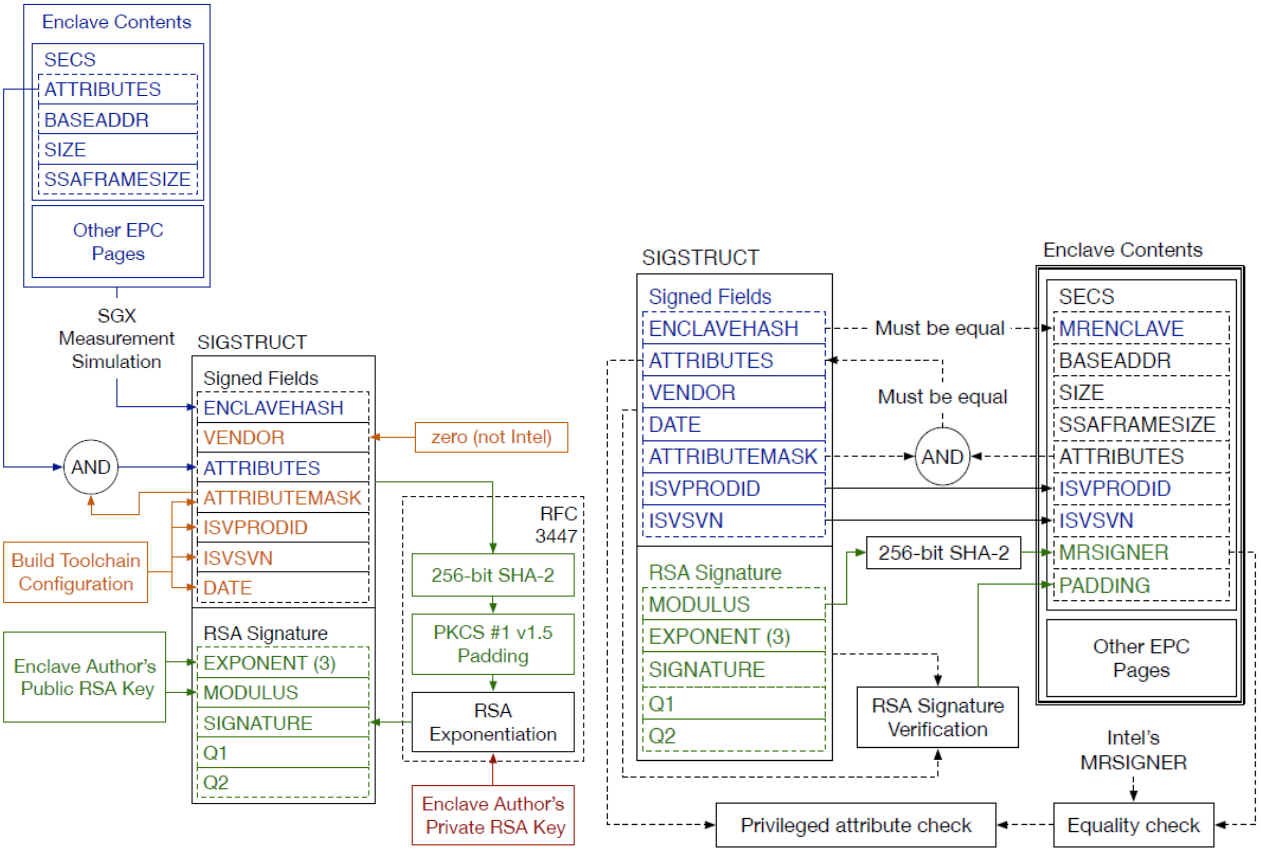

Fig. 6 SIGSTRUCT and its verification

The metadata portion of the SIGSTRUCT includes several important fields:

- `ENCLAVEHASH`: the enclave measurement (MRENCLAVE), representing the hash of its code, data and associated metadata, including build-time information (see 2.4.2),
- `ATTRIBUTES`: a set of flags defining enclave attributes, such as debug mode and supported CPU features,
- `ISVPRODID` (or ISV_PROD_ID or PRODID or PROD_ID): identifier of the software module signed by the enclave author,
- `ISVSVN` (or ISV_SVN or SVN): the security version number associated with the software module.

The signature portion of the SIGSTRUCT contains the following components:

- `EXPONENT`: the RSA public exponent, fixed to the value 3.
- `MODULUS`: the RSA modulus.
- `SIGNATURE`: the effective signature.

- `Q1` and `Q2`: auxiliary fields used to simplify RSA signature verification.

Because the exponent is fixed, enclave certificates are distinguished by their RSA modulus. The key modulus is used to issue enclave certificates. Instead of directly storing the public key modulus, SGX uses its 256-bit SHA-2 hash, known as the signer measurement (MRSIGNER), to identify the enclave author.

During enclave initialization, the SGX instruction `EINIT` is called and it receives a valid SIGSTRUCT as input and performs several verification steps against the enclave's contents (as reported on the right in Fig. 6), including validation of the signature. If the verification succeeds, it creates the SECS structure associated with the enclave, copying values contained in the SIGSTRUCT, as illustrated in Fig. 6.

For a more detailed analysis of the SGX architecture, the reader is referred to the work of Costan et al. [17].

In summary, the main components of the SGX memory layout are:

- **PRM**. A protected subset of the RAM.
- **EPC**. A subset of the PRM, where enclave code and data reside. It is split into pages that can be assigned to different enclaves.
- **EPCM**. An array where each entry contains metadata for each EPC page. It is mainly used for security checks. Each page is assigned a specific type that determines its behavior.
- **SECS**. A structure defining the enclave's metadata. It is stored in an EPC page.
- **ELRANGE**. The trusted portion of the enclave's address space where enclave code and data are mapped.
- **TCS**. A structure associated with a thread executing inside the enclave. It is stored in an EPC page.
- **SSA**. A structure containing the execution context of an enclave thread. It is used to securely store it during hardware exceptions, preventing it from being exposed to the system's hardware exception handler. It is also stored in an EPC page.

2.4.2. Enclave Lifecycle

The lifecycle of an SGX enclave describes how an enclave is created, initialized, executed, and eventually destroyed through a sequence of hardware instructions. Each transition between lifecycle states is performed using specific SGX instructions invoked by system software, as illustrated in Fig. 7 taken from [17].

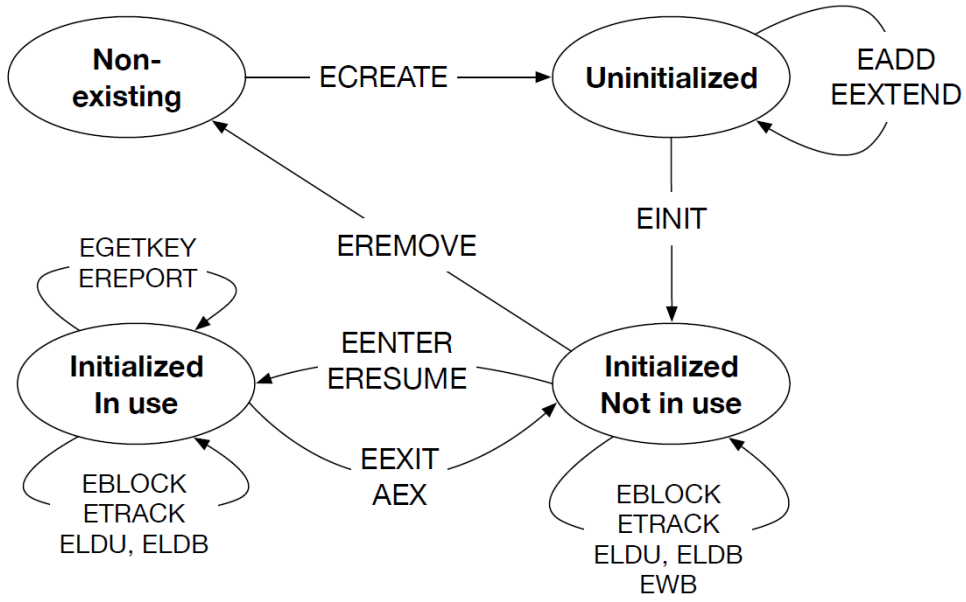

Fig. 7 Enclave lifecycle

In Fig. 7 many instructions are shown, but I highlight only the main ones [17]:

- `ECREATE` creates a new enclave by allocating a free EPC page and initializing the corresponding SECS fields. At this stage, the enclave is not yet runnable because the attribute `ATTRIBUTES.INIT` is set to `0`, indicating that the enclave is still uninitialized. During this step, it also generates a unique enclave identifier that is stored in the SECS and used to track the enclave throughout its lifecycle.
- `EADD` loads enclave's code and data into EPC pages. It also creates TCS and `PT_REG` pages.
- `EEXTEND` updates MRENCLAVE by incorporating the contents of these pages.
- `EINIT` marks the enclave as initialized and prevents any further use of `EADD` or `EEXTEND` for that enclave. It uses an Intel-provided component running within an enclave called "Launch Enclave" in order to do that.
- `EENTER` is invoked by an unprivileged application rather than system software, unlike previous instructions. It performs like a jump into an enclave code, backing up the CPU registers it modifies.
- `EEXIT` returns back to the application, restoring the CPU registers previously saved by `EENTER`.
- In case of faults or interrupts the CPU performs `AEX`. During this operation, the enclave state (like CPU registers) is securely stored in the SSA and then calls the hardware handler. In this way the enclave's state (data, code, etc.) is not exposed to the system software. After the faults or interrupts have been handled, execution can be resumed using `ERESUME`, called by the handler.
- `EREMOVE` deallocates all enclave's EPC pages.

An important aspect of this lifecycle is the computation of the MRENCLAVE. It must be computed up to `EINIT` instruction, because after `EINIT` instruction code and data cannot be added to the enclave (neither `ECREATE` can be called again). MRENCLAVE is calculated by hashing the sequence of instructions and their inputs during enclave creation, specifically across the `ECREATE`, `EADD`, `EEXTEND` and `EINIT` instructions. `ECREATE`, `EADD` and `EEXTEND` progressively update a temporary MRENCLAVE stored in the SECS, while `EINIT` finalizes it. It is important to note that `EADD` only adds pages to the enclave but does not measure their contents; the actual measurement of code and data is performed by `EEXTEND`.

For a more detailed analysis of the SGX architecture, the reader is referred to the work of Costan et al. [17].

2.4.3. Attestation

Software attestation is a mechanism through which a Verifier assesses the trustworthiness of an Attester [9]. The objective of software Attestation (or simply Attestation) is to enable the Verifier to establish trust in the Attester by obtaining an "Attestation Signature", i.e., a Report ("Attestation Data") signed by trusted hardware that includes measurements uniquely identifying the Attester [17]. This ensures that the Attester is executing within an isolated and secure environment and provides verifiable evidence of its behavior.

Two types of Attestation exist: Local Attestation and Remote Attestation. In the following, the Attestation mechanisms implemented by Intel SGX are described. Before discussing them, it is necessary to introduce several key concepts underlying the Attestation process.

Components	Description
Report	A data structure containing enclave information such as MRENCLAVE, MRSIGNER, ISV_PROD_ID, ISV_SVN, ATTRIBUTES, and REPORTDATA, along with a signature that can be verified by the same host [18, 21]. REPORTDATA represents user-defined data. The report is generated by the `EREPORT` instruction [18]. The signing key used is referred to as the Report Key, described below.
Quote	A data structure that includes a Report, additional metadata, and a signature over the Report. The signing key is the Attestation Key, described below.
Intel Provisioning Certification Service (PCS[4])	A service that provides an API for retrieving Provisioning Certification Key (PCK) certificates (described below), certificate revocation lists (CRLs), the Quoting Enclave (QE), and verification collateral associated with the QE.
Intel Provisioning Certificate Caching	A caching service (also deployable on-premises) used to store data retrieved from Intel PCS.

[4] Notice that the acronym PCS (Provisioning Certification Service) is different from PCs (Private Collections)

Service (PCCS)	
Quoting Enclave (QE)	An enclave that receives an SGX Report and produces the corresponding SGX Quote. Its identity (i.e., measurements and attributes) is publicly known and vetted by Intel. The QE runs on the same platform as the application enclave being attested.
Provision Certificate Enclave (PCE)	An enclave that uses the Provisioning Certification Key to sign proofs that Attestation Keys are generated on genuine SGX hardware. Its identity (i.e., measurements and attributes) is publicly known and vetted by Intel. It also runs on the same platform as the application enclave being attested.

Table 3 Key components in attestation

In addition to these components, several keys are involved in the Attestation process.

Keys	Description
Root Provisioning Key (RPK) and Root Seal Key (RSK)	These keys are fused into the CPU during manufacturing and serve as the root material for deriving other platform-specific keys. The RSK is not known to Intel, whereas RPK is.
Report Key	A key primarily derived from the RSK and the target enclave's MRENCLAVE [17]. The target enclave is the intended recipient of the Report and can derive the same key, enabling it to verify the Report's signature. This is possible because the target enclave derives it by using its MRENCLAVE [17].
Provisioning Certification Key (PCK)	A key derived from the RPK and MRSIGNER, with the derivation performed exclusively by the PCE [22]. The corresponding public certificate is signed and distributed by Intel.
Attestation Key (AK)	A key generated by the QE, whose public component is authenticated by the PCE using the PCK.
Attestation collaterals	The set of auxiliary data required to verify a Quote, including the PCK certificate, CRLs, and TCB information such as CPU SVN, etc.

Table 4 Keys in attestation

Fig. 8 (taken from [22]) summarizes the key hierarchy:

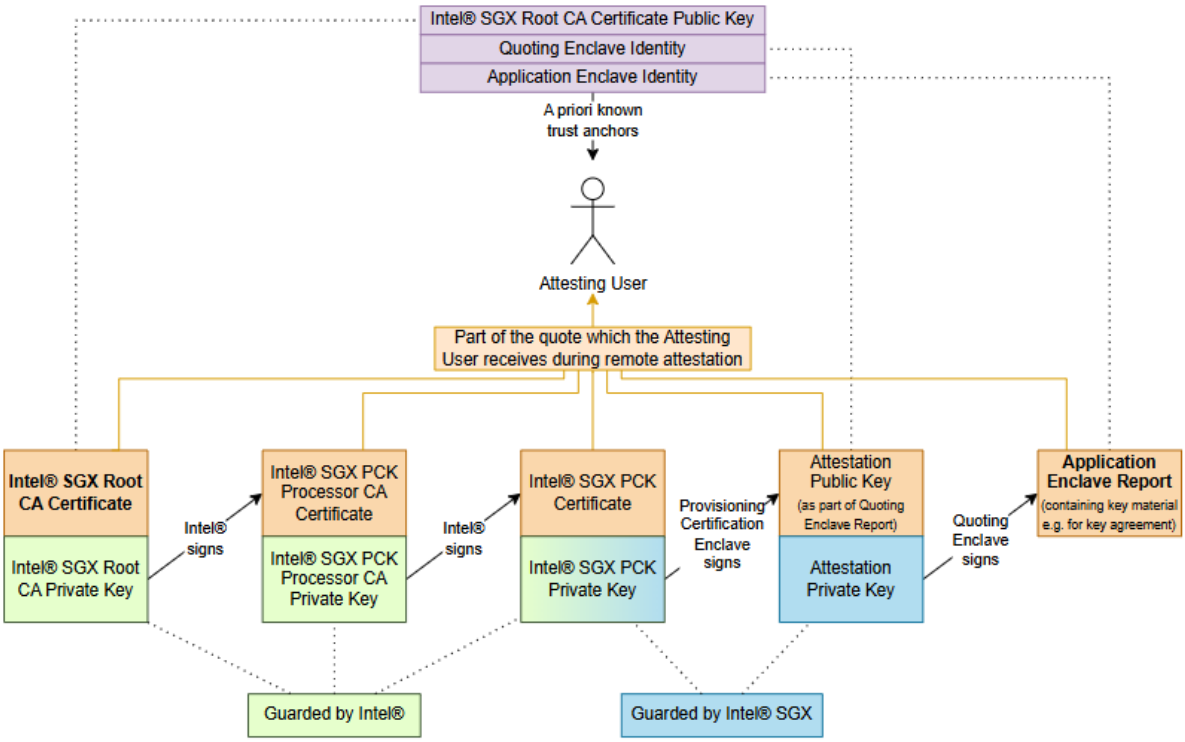

Fig. 8 Key hierarchy

The signature chain (or key hierarchy) underlying the Report signing process is as follows:

1. The chain begins with the Intel SGX Root CA certificate, whose private key signs the Intel SGX PCK Processor CA certificate,
2. Intel SGX PCK Processor CA certificate is only known to Intel, and it is securely stored and used in the documented ways [22]. Its private key is used to sign the Intel SGX PCK Certificate,
3. The PCK certificate, along with its corresponding private key (the PCK), is generated by the PCE. The PCK is then used to sign the QE's Report containing the public key of AK,
4. The QE uses the AK to sign the Report generated by the application enclave being attested.

Each step in this chain is verifiable and rooted in trusted Intel components, since each SGX component is trustworthy and publicly available. Consequently, the Quote can be trusted by an Attesting User AU, even if the AU does not operate within an enclave (in case of Remote Attestation). Furthermore, the AU can enforce policies that restrict interaction to enclaves with a specific MRENCLAVE or associated with a given MRSIGNER, both of which are included in the Report. This enables the AU to verify, in a trustworthy and verifiable manner, the behavior (i.e., the code), the author, and relevant security properties of the application executing within the enclave.

In summary, the trust chain originates from Intel's public root certificate, which guarantees the PCK Processor certificate. This certificate, in turn, guarantees the PCK certificate of the platform hosting the attested application. The PCK certifies the AK generated by the QE, which is ultimately used to sign the application enclave's Report.

2.4.3.1. Local Attestation

Local attestation is performed between two enclaves hosted on the same platform. Fig. 9 (taken from [18]) illustrates this process.

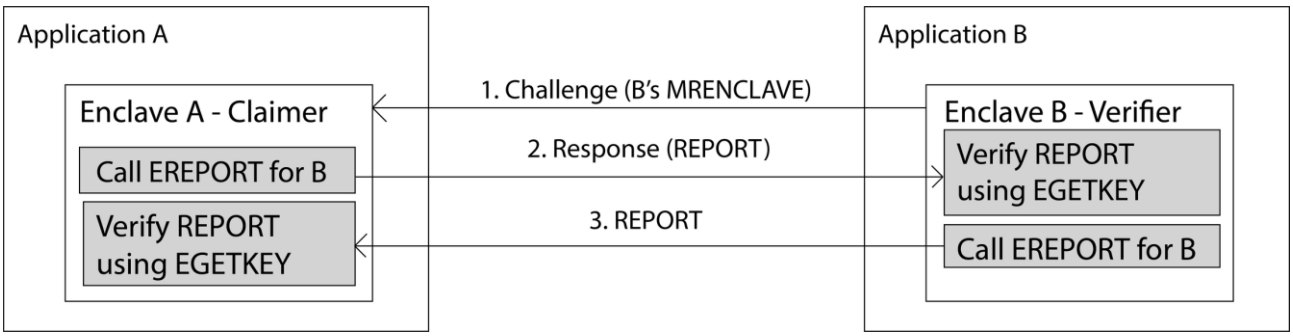

Fig. 9 Local attestation

Let A and B be two enclaves that have already established a communication channel, which does not need to be secure. Enclave B requests enclave A to prove that it is running within an enclave on the same platform. The procedure is as follows:

i. B sends its MRENCLAVE to A (step 1 in the figure),
ii. A generates a signed Report using the `EREPORT` instruction. The Report is signed with a Report Key derived from the RSK and B's MRENCLAVE. The key derivation and report generation are performed atomically by the `EREPORT` instruction [23] (part of step 2 in the figure),
iii. A sends the generated Report to B (part of step 2 in the figure),
iv. B derives the same Report Key using the `EGETKEY` instruction (part of step 2 in the figure),
v. B verifies A's Report using the derived Report Key (part of step 2 in the figure),
vi. To achieve mutual Attestation, B generates a signed Report targeting enclave A, using A's MRENCLAVE, which was included in A's Report (part of step 3 in the figure),
vii. A derives the corresponding Report Key and verifies B's Report (part of step 3 in the figure).

2.4.3.2. Remote Attestation

Remote Attestation (RA) is performed between an application running inside an enclave and a remote Verifier, which may be a standard (non-enclave) application. Fig. 10 (adapted from [24]) illustrates the RA scheme. Three main entities are involved:

- **Untrusted machine**, hosting the enclave being attested, along with QE and PCE.
- **Trusted machine**, where the Verifier resides.
- **Intel**, providing the Intel PCS service.

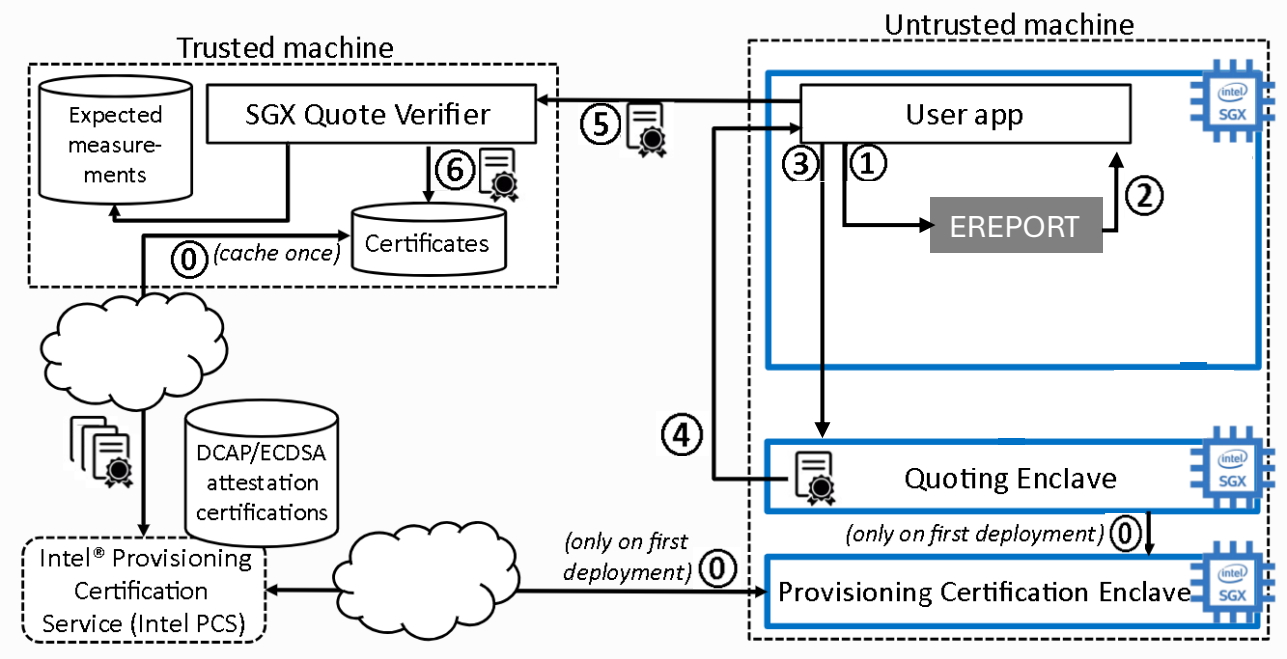

Fig. 10 Remote attestation

Before RA can be executed (denoted as step "0" in the figure), several preparatory steps are required:

- PCE generates the PCK [25],
- An Intel subscription to Intel PCS is required to tie the platform to the PCK certificate, which is signed and distributed by Intel. This certificate is then added to the Verifier's trusted certificate pool.
- QE generates an AK. Its public key, included in the QE's Report, is signed by the PCE using the PCK. Prior to signing, the PCE verifies the QE's Report through Local Attestation.

Once the setup is complete, the verifier V aims to assess the authenticity and the genuineness of the enclave application EA:

1. EA invokes `EREPORT` to generate a Report targeted at QE.
2. EA sends the Report to QE, which verifies it and, if successful, produces the corresponding Quote.
3. QE returns the Quote to EA.
4. EA forwards the Quote to V.
5. V validates the Quote by checking the signature chain described in Fig. 8. Then it can verify the trusted measurements (or hardware/software and security versions) contained in the Quote.

2.4.4.Sealing

Sealing is a feature that enables secure storage of encrypted data on the filesystem. The encryption is performed within the enclave using a key derived via the `EGETKEY` instruction. Two derivation methods are supported:

- **From RSK and enclave's MRENCLAVE**. This ensures that only an enclave with the exact same code can encrypt and decrypt the sealed data. However, any modification to the code results in a different MRENCLAVE, leading to a different derived key and making previously sealed data inaccessible.
- **From RSK and enclave's MRSIGNER**. This allows enclaves signed by the same author to encrypt and decrypt the data. Unlike the previous method, code updates do not prevent data decryption, as long as the signer remains unchanged. Consequently, enclaves sharing the same author can seal and unseal the same data.

2.4.5. Programming model

This section provides an overview of how developers can execute applications within an SGX enclave.

The first step in developing Intel SGX applications is to identify the assets that require protection, determine the code that operates on them, and encapsulate this logic within a trusted library [26].

SGX applications are written in C/C++ and follow a specific architectural model in which the codebase is divided into two main parts [26]:

- **Untrusted**: This portion executes outside the enclave. It is responsible for loading and managing the enclave and can invoke enclave functions through "ECALLs", which are calls transferring execution from the untrusted environment to the trusted one,
- **Trusted**: This portion executes within the enclave and contains the sensitive logic and data. It can invoke external functions through "OCALLs", which are calls transferring execution from the trusted environment to the untrusted one.

To support these interactions, SGX introduces "Edge Routines" (ERs) [26] which are functions that serve to bind ECALLs or OCALLs. It is important to note that the untrusted part (i) controls

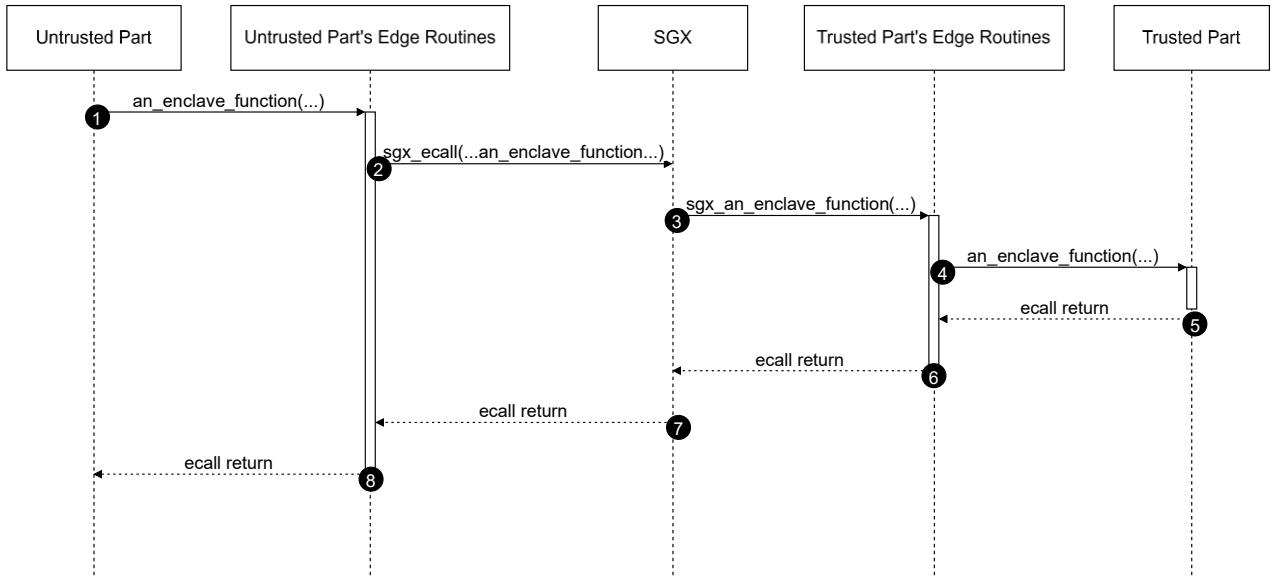

Fig. 11 ECALL sequence diagram

the invocation order of the enclave interface functions, (ii) may provide incorrect inputs to ECALLs or return incorrect results from OCALLs (or perform wrong operations), (iii) can load arbitrary enclaves, potentially allowing an attacker to load a malicious enclave designed to exploit vulnerabilities [26].

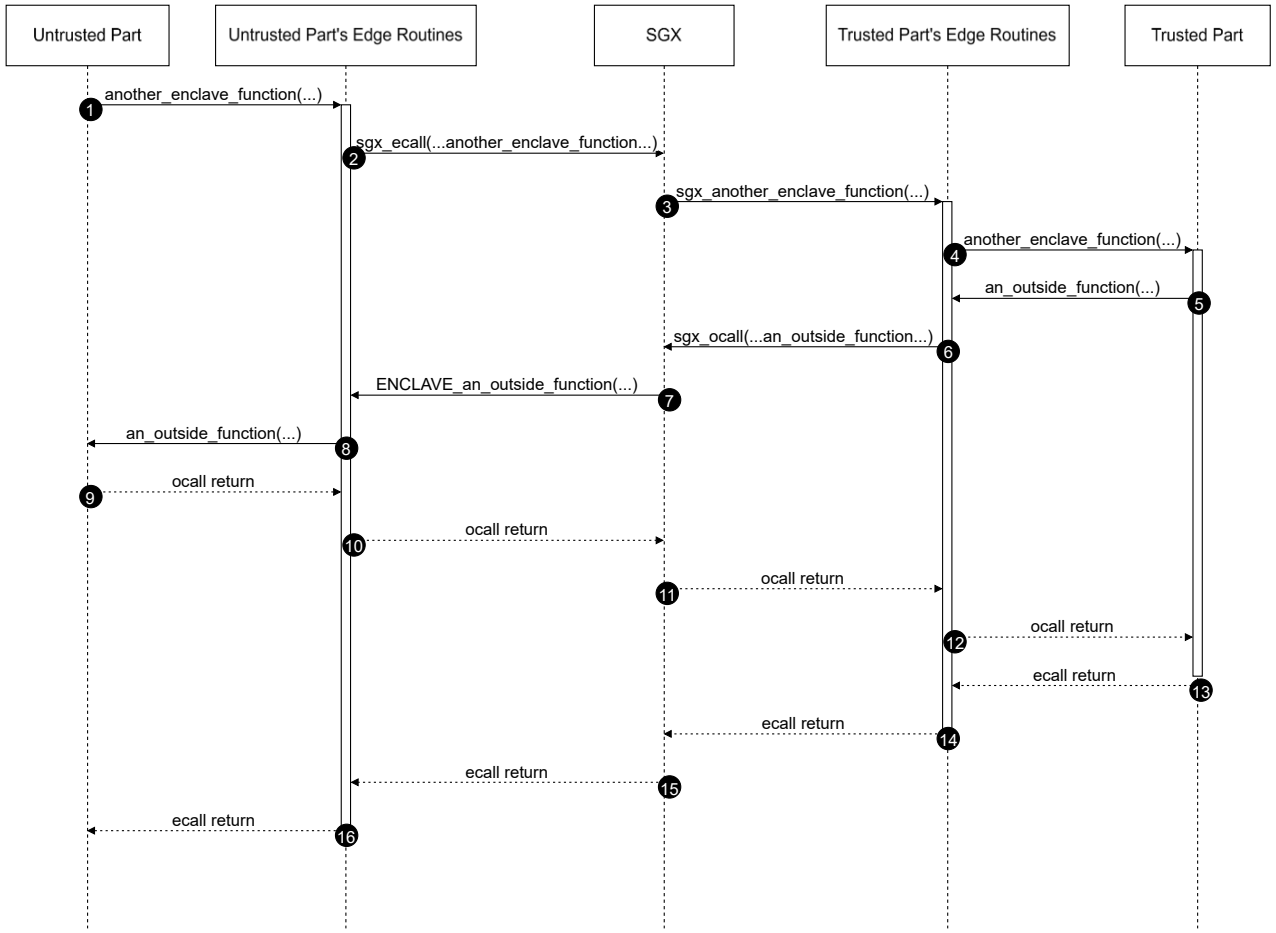

Fig. 12 OCALL sequence diagram

Fig. 11 and in Fig. 12 illustrate an example of a simplified sequence diagrams for ECALL and OCALL interactions, respectively. The application consists of four components, while "SGX" in the figure refers to the SGX SDK. The Trusted and Untrusted parts each include their own ERs. For simplicity, function parameters are omitted. The ECALL execution flow is as follows:

1. The Untrusted Part (UP) invokes `an_enclave_function(…)`, which is intended to execute within the enclave. This function acts as a wrapper that internally calls the corresponding ER.
2. The UP's ER invokes `sgx_ecall(…)`, specifying the target function. This call transfers control to SGX.
3. SGX performs the necessary transitions from the untrusted domain into the enclave and invokes `sgx_an_enclave_function(…)`, defined in the Trusted Part (TP)'s ER.
4. The TP's ER calls the actual TP's function `an_enclave_function(…)`.
5. After execution within the enclave, the result is returned through the ER back to the UP.

The OCALL execution flow follows the same initial steps as ECALL up to step 5, after which the behavior diverges:

5. TP invokes `an_outside_function(…)`, which is intended to execute outside the enclave. Like ECALL, this is a wrapper that calls TP's ER.
6. TP's ER invokes `sgx_ocall(…)`, specifying the target function. This call transfers control to SGX.
7. SGX transitions execution from the enclave to the untrusted domain and invokes the UP's ER's `ENCLAVE_an_outside_function(…)` (`ENCLAVE` is the name of the enclave used as example in this sequence diagram).
8. UP's ER invokes the actual UP's `an_outside_function(…)`.
9. After execution, the result is returned through the ER back to the TP.
10. The remaining steps follow the return path analogous to the ECALL completion.

In Linux environments, an enclave is packaged as a Shared Object (.so) file (or as a Dynamic Library on Windows). This file contains the enclave's code and data, which are loaded into the EPC during enclave creation. It also includes enclave metadata—such as the number of threads and the SIGSTRUCT—which is not loaded into the EPC but is used by the UP to correctly load the enclave.

A complete worked example of an Intel SGX SDK enclave application on Linux — including the project structure, the enclave definition (EDL), and the ECALL/OCALL implementation — is reported in Appendix B. It is worth noting that this low-level development model is precisely what the Gramine framework (Section 2.5) avoids, by executing unmodified applications inside enclaves without requiring the code to be partitioned into trusted and untrusted components.

2.4.6. Overhead

From an assembly-level perspective, the code executed within an enclave (e.g., `Enclave.cpp` in the previous example) is equivalent to that of a traditional function. Indeed, the generated assembly for the enclave shared object is identical to that of a standard, non-SGX application. To illustrate this, consider a file named `nosgx.cpp` with this code:

```
1. #include <stdint.h>
 2. uint32_t foo()
 3. {
 4.     int a = 1;
 5.     int b = 2;
 6.     int c = 3;
 7.     return c - b - a;
 8. }
 9.
10. int main(int argc, char* argv[])
11. {
12.     foo();
13.     return 0;
14. }
15.
```

This follows the traditional programming model. An equivalent implementation can then be defined within an enclave using the same structure as in the previous example, including the files:

`Enclave.edl`:

```
enclave {
    trusted {
        public uint32_t foo();
    };
    untrusted {
    };
};

```

`Enclave.cpp`:

```
#include "Enclave_t.h"
uint32_t foo()
{
    int a = 1;
    int b = 2;
    int c = 3;
    return c - b - a;
}

```

`app.cpp`:

```
#include "sgx_urts.h"
#include "Enclave_u.h"

static sgx_status_t initialize_enclave(const char* enclave_path, sgx_enclave_id_t *eid)
{
    sgx_status_t ret = SGX_ERROR_UNEXPECTED;

    ret = sgx_create_enclave(enclave_path, SGX_DEBUG_FLAG, NULL, NULL, eid, NULL);
    if (ret != SGX_SUCCESS) { return ret; }
    return SGX_SUCCESS;
}

int main(int argc, char* argv[])
{
    sgx_enclave_id_t enclaveId = 0;
    sgx_status_t ret = initialize_enclave("libenclave.signed.so", &enclaveId);
    if (ret != SGX_SUCCESS) { return false; }

    foo(enclaveId, NULL);

    sgx_destroy_enclave(enclaveId);
    return 0;
}

```

With a default enclave configuration. In this setup, the application simply creates the enclave and invokes the function `foo()`, which performs a basic computation.

After compiling both versions (without optimization) and generating the enclave shared object and the `nosgx` executable, the assembly code can be inspected. By executing the enclave using SGX-enabled GDB (allowing enclave debugging) and examining the implementation of `foo()`, or alternatively by directly disassembling the shared object, one obtains the assembly code like:

```
0000000000001122 <foo>:
    1122: f3 0f 1e fa                   endbr64
    1126: 55                            push   %rbp
    1127: 48 89 e5                      mov    %rsp,%rbp
    112a: c7 45 f4 01 00 00 00          movl   $0x1,-0xc(%rbp)
    1131: c7 45 f8 02 00 00 00          movl   $0x2,-0x8(%rbp)
    1138: c7 45 fc 03 00 00 00          movl   $0x3,-0x4(%rbp)
```

```
    113f: 8b 45 fc                  mov    -0x4(%rbp),%eax
    1142: 2b 45 f8                  sub    -0x8(%rbp),%eax
    1145: 2b 45 f4                  sub    -0xc(%rbp),%eax
    1148: 5d                        pop    %rbp
    1149: c3                        ret
    114a: 66 0f 1f 44 00 00         nopw   0x0(%rax,%rax,1)
```

Instead, by inspecting the `foo()` code in the `nosgx` executable one obtains the assembly code like:

```
0000000000001129 <_Z3foov>:
    1129: f3 0f 1e fa               endbr64
    112d: 55                        push   %rbp
    112e: 48 89 e5                  mov    %rsp,%rbp

1138:     c7 45 f8 02 00 00 00      movl   $0x2,-0x8(%rbp)
    113f: c7 45 fc 03 00 00 00      movl   $0x3,-0x4(%rbp)
    1146: 8b 45 fc                  mov    -0x4(%rbp),%eax
    1149: 2b 45 f8                  sub    -0x8(%rbp),%eax
    114c: 2b 45 f4                  sub    -0xc(%rbp),%eax
    114f: 5d                        pop    %rbp
    1150: c3                        ret
```

Which are the same. Thus, this observation highlights that the primary differences introduced by SGX do not lie in the application logic itself, but rather in the additional components required for secure execution. These include the ERs, SGX-specific instructions (e.g., `ECREATE`, `EINIT`), and hardware-level mechanisms such as continuous memory encryption and decryption.

Such mechanisms introduce performance overhead. As reported by Kumar et al. [27]], when the memory footprint exceeds the EPC size, performance degradation can reach up to approximately three times, a result corroborated by Zhao et al. [28]. Similarly, in the context of a multi-threaded HTTP server, performance degradation increases with the number of concurrent threads accessing the enclave. In particular, up to a sevenfold slowdown is observed with 256 threads compared to a non-SGX implementation [27], while with 32 threads the degradation is approximately 2.5x.

A practical demonstration of this overhead can be obtained through a square matrix multiplication executed within an enclave. This example follows the same structure as the previous enclave application example but replaces the simple `foo()` function with a function `compute()`, which takes the matrix size as input:

```
 1. void compute(int size)
 2. {
 3.     int** matrix1 = new int*[size];
 4.     int** matrix2 = new int*[size];
 5.     int** C = new int*[size];
 6.     for (int i = 0; i < size; i++)
 7.     {
 8.         matrix1[i] = new int[size];
 9.         matrix2[i] = new int[size];
10.         C[i] = new int[size];
11.     }
12.
13.     for (int i = 0; i < size; i++)
14.     {
15.         for (int j = 0; j < size; j++)
16.         {
17.             matrix1[i][j] = i*j;
18.             matrix2[j][i] = i*j;
19.         }
20.
```

```
    }
    // Perform operations on the matrix...
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            for (int k = 0; k < size; k++)
            {
                C[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    for (int i = 0; i < size; i++)
    {
        delete matrix1[i];
        delete matrix2[i];
        delete C[i];
    }
    delete[] matrix1;
    delete[] matrix2;
    delete[] C;
}

```

Although the implementation can be further optimized, such optimizations are beyond the scope of this example. Notably, the same code can also be used in a conventional (non-SGX) application.

The enclave-based application operates similarly to the assembly-level example. By executing the application with increasing matrix sizes, the multiplication is performed and the execution time is measured. The results are reported in Fig. 13.

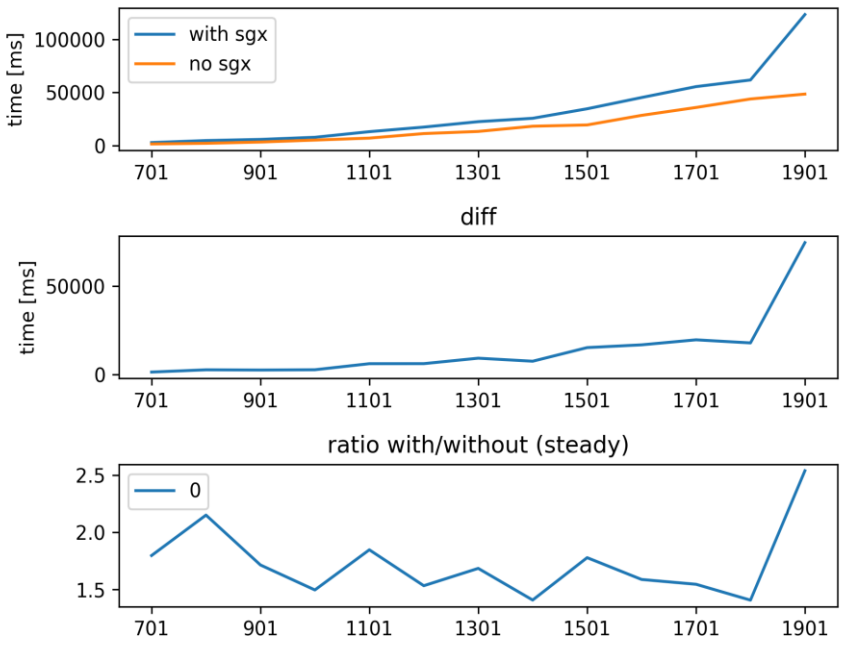

Fig. 13 SGX vs no-SGX matrix multiplication benchmark

The x-axis represents the matrix size. As observed, the ratio between execution within the enclave and outside it remains approximately constant at 2.5, although the absolute performance gap increases with input size. This behavior is consistent with the fact that the

memory footprint remains below the EPC size (98 MB) in this experimental setup, as discussed by Kumar et al. [27].

2.5. Gramine

Gramine is a multi-process Library OS (LibOS) and portability framework [29] that enables the execution of unmodified Linux applications across multiple platforms. It operates as an intermediate layer between the application and the underlying system. Currently, it supports both standard Linux and SGX-enabled Linux (hereafter referred to as Linux-SGX).

2.5.1. Architecture

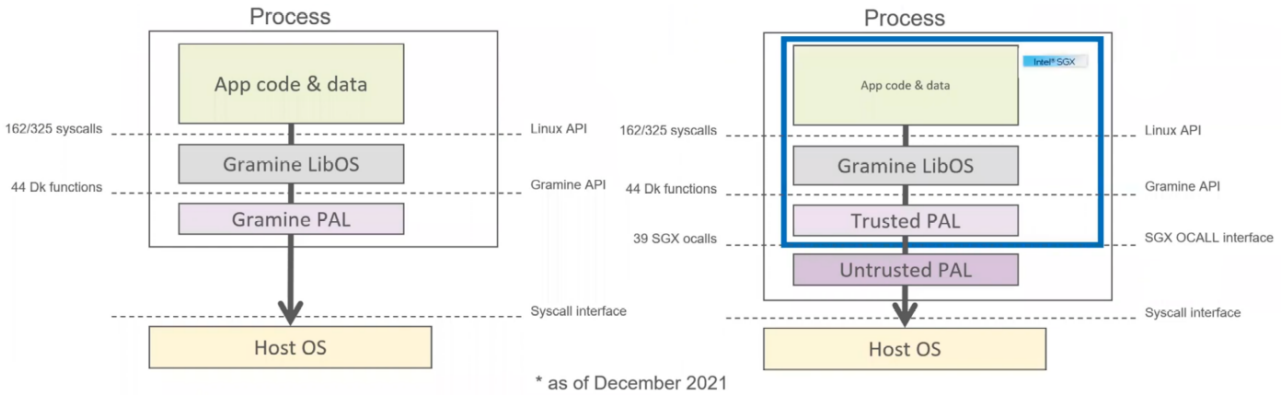

Fig. 14 Gramine architecture

Gramine adopts a modular architecture, illustrated on the left side of Fig. 14Fig. 16 (taken from [29]), in which the OS is decomposed into three layers:

1. **LibOS layer**: the OS is implemented in user space as a library, rather than within the kernel. This design allows reimplementation of system calls, APIs, and interfaces. In [29], 162 system calls were supported; in Gramine v1.8 (used in this work), this number increases to 177 [30].
2. **Platform Adaptation Layer (PAL)**: this layer abstracts system calls into a reduced set of functions, enabling platform independence and improving portability across different execution environments (Linux and Linux-SGX). At the time of [29], PAL provided 44 functions; in version 1.8, it provides 56 functions, including SGX-specific ones [31]. The PAL operates on top of the host OS, to which it performs traditional system calls.
3. **Host OS layer**: this corresponds to the underlying OS running on the host machine.

In the case of Linux-SGX, the PAL is further divided into:

- **Trusted PAL**, which executes within the enclave and acts as a bridge toward the external environment,
- **Unstruted PAL**, which executes outside the enclave.

The Trusted PAL issues SGX OCALLs towards the Untrusted PAL. In [29], 39 SGX OCALLs were defined, whereas in Gramine v1.8 this number increases to 50 [32]. The Untrusted PAL subsequently performs system calls to the host OS.

2.5.2. Manifest

The Gramine configuration file, referred to as the "manifest" [33], is written in TOML syntax and allows specification of both general and SGX-related parameters. The most relevant options used in this work are summarized below. `URI`s follow the format `"file:/path/to/resource"`, whereas `path`s are traditional paths.

Syntax	Description
`loader.log_level = "[none|error|warning|debug|trace|all]"`	This controls the verbosity of logging, ranging from `"none"` to `"all"`, with `"error"` as the default and most suitable setting for production environments.
`libos.entrypoint = "[PATH]"`	This specifies the path, within the enclave Gramine filesystem, of the initial executable launched after enclave creation.

`loader.insecure__use_cmdline_argv = true`
or
`loader.argv = ["arg0", "arg1", "arg2", ...]`
or
`loader.argv_src_file = "file:file_with_serialized_argv"`
	Application arguments can be provided through different mechanisms. The first one allows passing arguments when running the enclave (through `gramine-sgx` or `gramine-direct`) but is considered insecure since inputs are not verified. The others define fixed arguments that are securely hashed as trusted inputs. The third one is generated through a Gramine tool.

`sys.enable_extra_runtime_domain_names_conf = [true|false]`	This forwards the host's DNS resolver configuration to the enclave.
`sys.experimental__enable_flock = [true|false]`	This is still experimental and enables `flocks` system calls[5] in Gramine. Default to `false`.
`loader.env.[ENVIRON] = { passthrough = true }`	This is the way to define the environment variable `ENVIRON` inside the Gramine environment, with value obtained by the host. You can set also a value for them, even through a file.

`fs.mounts = [`
` { type = "[chroot|...]", path = "[PATH]", uri = "[URI]" },`
` { type = "[chroot|...]", path = "[PATH]", uri = "[URI]" },`
`]`
In case of encrypted:
` { type = "encrypted", path = "[PATH]", uri = "[URI]", key_name = "[KEY_NAME]"}`
	This defines how files and directories from the host are mapped into the enclave Gramine filesystem. `uri` is referred to the host filesystem, while `path` to the enclave one. There can be different types:
	• By default it is `chroot`, it is like docker volumes: host's file/directory `uri`s are directly mapped into enclave's `path`.
	• `tmpfs`: it states that the `path` is an in-memory file/directory. `uri` is ignored. Every process has its own isolated `tmpfs`.
	• `encrypted` is like `chroot`, but everything inside `path` is transparently sealed or encrypted with a specific key, and it will be sealed or encrypted if the application creates a

[5] https://man7.org/linux/man-pages/man2/flock.2.html

	file/directory inside it. More specifically, `key_name` can have different values: ○ the name of an encryption key, which can be passed directly into the manifest (only for debugging purposes) or by secret provisioning (described below). In any case, it is expected to be at `/dev/attestation/keys/[KEY_NAME]`, ○ `_sgx_mrenclave` (only for SGX), which seals based on the MRENCLAVE, ○ `_sgx_mrsigner` (only for SGX), which seals based on the MRSIGNER.

Table 5 Gramine configuration in manifest

When running in SGX mode, additional parameters are available, otherwise are ignored:

Syntax	Description
`sgx.debug = [true|false]`	This enables SGX debug features or not. Default to `false`.
`sgx.edmm_enable = [true|false]`	It enables the EDMM feature. EDMM is an SGX2 (a more recent release of SGX) feature that allows dynamic allocation and deallocation of enclave memory and threads [34]. Default to `false`.
`sgx.enclave_size = "[SIZE]"`	It defines the enclave memory size at creation time (or the maximum size when EDMM is enabled. This memory includes both application and Gramine components (PAL and LibOS). Notably, the specified memory is immediately allocated in RAM, and each child process spawned within Gramine allocates an enclave of the same size. So, this means that if an application with `sgx.enclave_size = "2G"` creates 2 children, the effective allocated memory is 6 GB. Consequently, memory usage scales linearly with the number of processes unless EDMM is enabled, in which case memory is allocated on demand.
`sgx.max_threads = [NUM]`	It specifies the maximum number of enclave threads when EDMM is disabled. At least four threads are required, as Gramine internally creates three helper threads: one to facilitate inter-process communication (if there is only 1 process, it sleeps); a thread to manage asynchronous tasks; a thread performing TLS-handshake on pipes creation (once performed, it terminates). When EDMM is enabled, this parameter defines the number of preallocated thread slots, which may be exceeded at runtime.

`sgx.allowed_files = [`
`  "[URI]",`
`  "[URI]",`
`]`
	This tells the host's file/directory `URI`s that can be accessed (opened, read and created) without integrity verification. If it is a directory, all its subdirectories and files are treated in the same way.

`sgx.trusted_files = [`
`  "[URI]",`
`  "[URI]",`
`]`
	Unlike allowed files, the trusted files are hashed at build time. They can be accessed by the enclave only if the integrity check succeeds. Trusted files are immutable within the enclave, preventing external tampering.

`sgx.remote_attestation = "[none|epid|dcap]"`	It specifies the type of RA, default to `"none"`. In this work, the DCAP model is adopted. More information below about attestation.

Table 6 SGX configuration in manifest

2.5.3. Simple interaction

The following example illustrates how to execute a modified version of the official Gramine "Hello World" example [35].

```
# Copyright (C) 2023 Gramine contributors
# SPDX-License-Identifier: BSD-3-Clause

libos.entrypoint = "/helloworld"
loader.log_level = "error"

loader.env.LD_LIBRARY_PATH = "/lib"

fs.mounts = [
  { path = "/lib", uri = "file:{{ gramine.runtimedir() }}" },
  { path = "/helloworld", uri = "file:helloworld" },
  { path = "/example", uri = "file:/home/user/example" },
]


sgx.debug = true
sgx.edmm_enable = {{ 'true' if env.get('EDMM', '0') == '1' else 'false' }}
sgx.enclave_size = "1G"

sgx.trusted_files = [
  "file:helloworld",
  "file:{{ gramine.runtimedir() }}/",
  "file:/home/user/example"
]

sgx.allowed_files = [
  "file:/home/user/example",
]

```

The manifest defines the entrypoint executable, logging level, environment variables, filesystem mounts, and SGX configuration. When executed, the enclave behaves as follows:

- it logs only errors (`loader.log_level = "error"`).
- The first Linux executable run within the enclave, after Gramine's initialization, is `/helloworld` (`libos.entrypoint`). It is mounted here `fs.mounts = […{ path = "/helloworld", uri = "file:helloworld" },…]`, which means that:

    - It comes from the host filesystem located in the same directory where the Gramine tools are used to preprocess the manifest (described below). In fact, `uri = "file:helloworld"` is defined without absolute path,
    - It is mapped inside the Gramine filesystem to `/helloworld` (in fact: `path = "/helloworld"`).
    - It can be run thanks to `sgx.trusted_files = ["file:helloworld",…]`. It could be also run even if specified into `allowed_files`, but without integrity check.
- Inside the Gramine environment, the environment variable `LD_LIBRARY_PATH` is set to `/lib`, and when `/helloworld` runs, it can see that variable.
    - It is mounted by `{ path = "/lib", uri = "file:{{ gramine.runtimedir() }}" }`, which (in Jinja markup) calls the method `runtimedir()` on `gramine` constant, which is globally accessible during manifest preprocessing, and contains the path to Gramine's runtime directory.
    - You can use the environment variable securely, because the host's version is trusted (`sgx.trusted_files = […,"file:{{ gramine.runtimedir() }}/",…]`).
- In the mounted files, there is also `{ path = "/example", uri = "file:/home/user/example" }`, which maps the host's `/home/user/example` to `/example` inside the enclave.
    - It is present in the trusted files. In this way its content is integrity checked, and it can be read securely.
    - However, in order to write files or directories inside `/home/user/example`, the configuration `sgx.allowed_files = ["file:/home/user/example",]` must be specified.
- The enclave runs in debug mode (`sgx.debug = true`).
- EDMM is enabled or not depending on the result of the environment check `env.get('EDMM', '0') == '1'`. If it is the value is `'true'`, else `'false'`.
- The enclave size is configured to 1GB (`sgx.enclave_size = "1G"`).

Assuming the manifest is saved as `helloworld.manifest.template`, the directory structure is as follows:

```
/home/user/
├── example/
│   └── ...
└── helloworld/
    ├── helloworld
    └── helloworld.manifest.template
```

Where `helloworld` is the ELF executable to be run inside the enclave. The execution procedure is as follows:

1. After having installed all Gramine's prerequisites, generate a private key for signing SGX enclaves, identifying the MRSIGNER:

```
gramine-sgx-gen-private-key
```

This produces an RSA-3072 key (required by Intel SGX) stored at `$HOME/.config/gramine/enclave-key.pem`.

2. Preprocess the manifest:

```
gramine-manifest helloworld.manifest.template helloworld
```

This generates the expanded manifest file `helloworld.manifest`, including resolved paths and file (both trusted and allowed) hashes and other manifest configuration.

3. Sign the manifest, that is the enclave, the trusted files and the Gramine environment:

```
gramine-sgx-sign --manifest helloworld.manifest --output helloworld.manifest.sgx
```

It takes the `helloworld.manifest` and produces the signed manifest (`helloworld.manifest.sgx`) and the SIGSTRUCT file (`helloworld.sig`) of `helloworld` enclave.

4. Run the `helloworld` application within Gramine by passing the name of a manifest in the current directory

```
gramine-sgx helloworld
```

The execution may require additional time due to manifest processing and enclave initialization.

Finally, SGX-specific details of the enclave (e.g., MRENCLAVE and MRSIGNER) can be inspected using:

```
gramine-sgx-sigstruct-view helloworld.sig
```

2.5.4.Attestation

In addition to facilitating enclave development while preserving unmodified applications, Gramine provides mechanisms to simplify both Local and Remote Attestation processes, as described in [24]. These mechanisms are organized into three levels of abstraction, offering progressively simplified interfaces at the cost of reduced flexibility:

1. **Low-level interface**: enables both Local and Remote Attestation through direct interaction with a pseudo-filesystem.
2. **Secure Channel (RA only)**: builds upon the low-level interface and simplifies its usage through the RA-TLS library provided by Gramine.
3. **Secret Provisioning (RA only)**: further abstracts the process by leveraging a dedicated Secret Provisioning library.

There are also ways to use Vendor-specific plugins and Third-Party Attestation solutions.

Although both EPID and DCAP Attestation mechanisms are supported, this work exclusively adopts the DCAP approach.

The three abstraction levels are discussed in detail below.

2.5.4.1. Low-level interface

The Attestation interface is exposed through a pseudo-filesystem, as summarized below. Each pseudo-file is process-local, eliminating the need for synchronization mechanisms such as locks.

Pseudo-file	Description
`/dev/attestation/attestation_type`	Read-only; returns the Attestation type in use (`none`, `epid` or `dcap`).
`/dev/attestation/user_report_data`	Readable and writable; in SGX, this is a 64-byte string embedded within the SGX Report or Quote.
`/dev/attestation/target_info`	Readable and writable; contains information about a target enclave and is represented as an opaque 512-byte structure.
`/dev/attestation/my_target_info`	Read-only; provides information about the current enclave and is used in Local Attestation. It is also a 512-byte opaque structure.
`/dev/attestation/report`	Read-only; contains the SGX Report generated via the `EREPORT` instruction. Prior to reading this file, both `/dev/attestation/user_report_data` and `/dev/attestation/target_info` must be populated, as they are required inputs for Report generation.
`/dev/attestation/quote`	Read-only; similar to `/dev/attestation/report`, but returns an SGX Quote. Unlike the report, it does not require prior initialization of `/dev/attestation/target_info`.
`/dev/attestation/keys/[KEY_NAME]`	It provides a 128-bit encryption key obtained from a Secret Provisioning service prior to application execution; this key is used for encrypting files and directories (as mentioned in 2.5.2).

Table 7 Gramine pseudo-files for attestation

Once a Report is generated, it can be forwarded to another enclave, such as the QE, in the case of RA. A common practice is to write a hash of a public key—generated within the enclave—into `/dev/attestation/user_report_data`. When this value is included in the SGX Quote, it binds the enclave identity to the corresponding public key.

It is important to note that Gramine does not provide pseudo-files for Attestation verification. At this abstraction level, verification is delegated to external tools and libraries, such as the Intel DCAP verification libraries.

The following C code snippet (coming from [24]) illustrates how to generate an SGX Quote using this low-level interface:

```
1. sgx_report_data_t user_report_data = {0};
2. memcpy(&user_report_data, "some-dummy-data", sizeof("some-dummy-data"));
3.
4. int fd1 = open("/dev/attestation/user_report_data", O_WRONLY);
5. write(fd1, &user_report_data, sizeof(user_report_data));
```

```
 6.
 7. uint8_t quote[SGX_QUOTE_MAX_SIZE];
 8. int fd2 = open("/dev/attestation/quote", O_RDONLY);
 9. read(fd2, &quote, sizeof(quote));
10.
```

In the first two lines, the user report data is initialized with the value `"some-dummy-data"`. Then, it is written to `/dev/attestation/user_report_data`. Subsequently, the SGX Quote is generated and stored in the `quote` variable. This Quote can then be transmitted to a remote verifier.

2.5.4.2. Secure Channel

This level builds upon the previous one by introducing a higher level of abstraction. It leverages the RA-TLS library, which extends the standard TLS handshake protocol by enforcing the verification of an endpoint's SGX Quote embedded within its certificate. Starting from Gramine "v1.8", RA-TLS is interoperable across different TEE frameworks (e.g., Occlum).

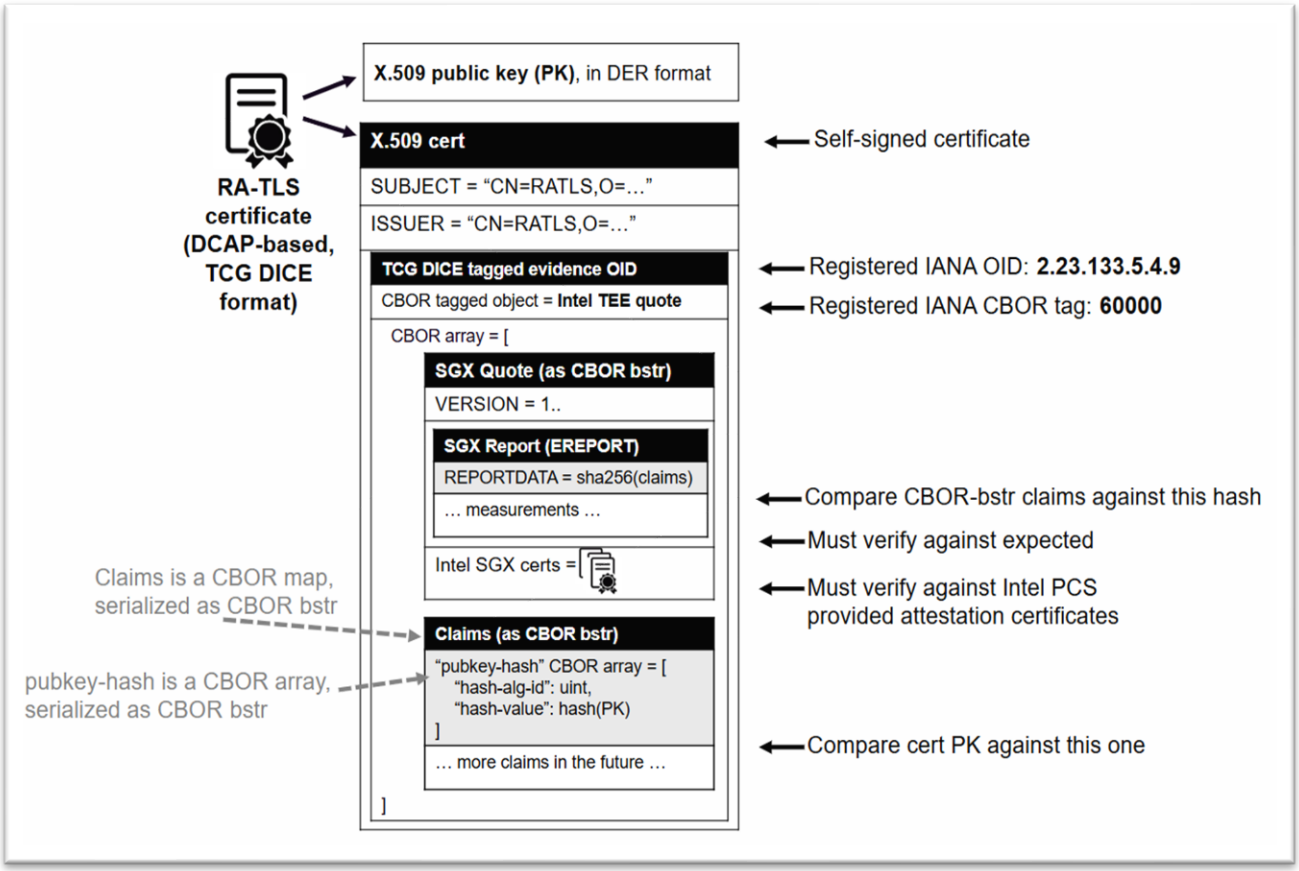

Fig. 15 RA-TLS certificate

Fig. 15 (taken from [24]) illustrates the structure of an RA-TLS certificate. This certificate extends the standard X.509 format by incorporating SGX-related information within a critical extension identified by the OID `2.23.133.5.4.9` (`tcg-dice-conceptual-message-wrapper`). The value of this extension is an array of CBOR-encoded elements containing:

1. The SGX Quote, which embeds the SGX Report along with the complete SGX certificate chain.
2. A set of claims, including the `"pubkey-hash"` claim, which stores the hash of the ephemeral public key (in DER format) generated within the enclave.

To bind the RA-TLS certificate to the enclave application that generated it, the REPORTDATA field of the SGX Report includes the hash of these claims. The certificate is self-signed, as the chain of trust is conveyed through the embedded SGX Quote.

Gramine provides two libraries to facilitate the use of RA-TLS:

- `ra_tls_attest.so` retrieves the SGX Quote via the `/dev/attestation` pseudo-filesystem and embeds it into the RA-TLS certificate. This library requires DCAP to be enabled in the Gramine manifest (`sgx.remote_attestation = "dcap"`).
- `ra_tls_verify_dcap.so` verifies both the RA-TLS certificate and the embedded SGX Quote using the DCAP verification library (`libsgx_dcap_quoteverify.so`). This requires a properly configured DCAP infrastructure on the host system.

In combination with the MBed TLS C library [36] it is possible to implement a complete Attestation-aware communication flow (even mutual): a server generates and presents an RA-TLS certificate, while a client verifies it. Notice that the client is not required to run within an enclave to perform SGX Quote verification.

Verification can be customized by registering a callback function via `ra_tls_set_measurement_callback()` provided by `ra_tls_verify_dcap.so`, with the following signature:

```
int (*callback)(char* mrenclave, char* mrsigner, char* isv_prod_id, char* isv_svn)
```

This callback enables application-specific validation logic based on enclave identity parameters such as MRENCLAVE, MRSIGNER, ISVPRODID, and ISVSVN extracted from the RA-TLS certificate. Alternatively, these values can be specified through environment variables: `RA_TLS_MRENCLAVE`, `RA_TLS_MRSIGNER`, `RA_TLS_ISV_PROD_ID` and `RA_TLS_ISV_SVN`.

Prior to evaluating these parameters, the verification library performs checks on hardware and software security requirements, returning status codes. Additional SGX-specific environment variables can be configured to relax these checks if necessary. By default, these variables are set to `0` (strict mode), but can be set to `1` to relax them:

- `RA_TLS_ALLOW_OUTDATED_TCB_INSECURE`, whether to allow the “outdated TCB” status, which is insecure in production environments. This is because the TCB level of the platform hosting the enclave is not completely secure, due to potential unpatched vulnerabilities [37].
- `RA_TLS_ALLOW_HW_CONFIG_NEEDED`, whether to allow the “HW configuration needed” status, which requires some configuration at hardware-level [37] (not so strict [24]).
- `RA_TLS_ALLOW_SW_HARDENING_NEEDED`, whether to allow the “SW hardening needed” status, which means that the platform of the attesting enclave needs some software hardening to mitigate the attack risk [37].
- `RA_TLS_ALLOW_DEBUG_ENCLAVE_INSECURE`, whether to allow debug enclaves, which is insecure in production environments.

A flowchart illustrating the operation of the Secure Channel abstraction level is provided in Appendix D.

2.5.4.3. Secret Provisioning

The highest level of abstraction provided by Gramine is Secret Provisioning. While the Secure Channel approach offers greater flexibility, many use cases involve a client (Attester) connecting to a server (Verifier) to retrieve one or more secrets. This server is referred to as a "Secret Provisioning Server" (or "Secret Provisioner"). In this model, the server employs a standard X.509 certificate, whereas the client uses an RA-TLS certificate.

Gramine provides two libraries to support this workflow, both built upon the RA-TLS mechanism of the Secure Channel level and inheriting its configuration and environment variables:

- `secret_prov_attest.so` analogous to the attestation library used in the Secure Channel, but extended to support secret retrieval. This can be achieved in two ways:
    1. **programmatically**, as illustrated by the following simplified C example [38]:

```c
#define SEND_STRING "MORE"
#define CA_CRT_PATH "ca.crt"
#include "secret_prov.h"
// ...
uint8_t* secret1 = NULL;
size_t secret1_size = 0;
// ...
struct ra_tls_ctx* ctx = NULL;
int ret = secret_provision_start("server:80;localhost:4433;anotherserver:4433", CA_CRT_PATH, &ctx);
if (ret < 0) { }  // error
// ...
ret = secret_provision_get(ctx, &secret1, &secret1_size);
if (ret < 0) { } // error
if (!secret1_size) { } // error
// ...
ret = secret_provision_write(ctx, (uint8_t*)SEND_STRING, sizeof(SEND_STRING));
if (ret < 0) { } // error
// ...
ret = secret_provision_read(ctx, secret2, sizeof(secret2));
if (ret < 0) { } // error
// ...
```

    Where by including the Gramine secret provisioning library (`#include "secret_prov.h"`):
    - `secret_provision_start()` attempts to connect to one of the specified server URLs and verifies its X.509 certificate against the CA certificate located at `CA_CRT_PATH`.
    - `secret_provision_get()` retrieves the secret from the server and stores it in `secret1` with length `secret1_size`.
    - `secret_provision_write()` sends data to the server.
    - `secret_provision_read()` retrieves additional secret(s).

    The key distinction between `secret_provision_get()` and `secret_provision_read()` lies in the type of secret retrieved: the former returns a variable-length "first secret" (described below in the server section), whereas the latter expects fixed-length secrets distinct from the initial one.

2. **through environment variables**, defined in the manifest prior to application startup:
    - `SECRET_PROVISION_CONSTRUCTOR`: if set to `1/true/TRUE`, initiates the secret retrieval process before application execution. By default, it is not set.
    - `SECRET_PROVISION_SET_KEY`: specifies the name under which the retrieved secret is stored in `/dev/attestation/keys/[KEY_NAME]`. The secret must be a 16-bytes AES-GCM key used for encrypted files. If set to `default`, it is the default key for encrypting files.
    - `SECRET_PROVISION_SECRET_STRING`: if `SECRET_PROVISION_CONSTRUCTOR` is set to `1/true/TRUE`, `SECRET_PROVISION_SET_KEY` is not set, and the secret is a string of maximum 4k characters, you can get the secret by reading this environment variable.
    - `SECRET_PROVISION_SERVERS`: defines a semicolon-separated list of Secret Provisioning Server URLs.
    - `SECRET_PROVISION_CA_CHAIN_PATH`: specifies the path to the server's CA certificate chain.

- `secret_prov_verify_dcap.so` provides functionality for implementing a Secret Provisioning Server, including customization of measurement verification and secret provisioning logic. A simplified usage example in C (adapted from [38]) demonstrates that:

```c
#include "secret_prov.h"
#define PORT "4433"
#define EXPECTED_STRING "MORE"
#define FIRST_SECRET "FIRST_SECRET"
#define SECOND_SECRET "42"
#define SRV_CRT_PATH "server.crt"
#define SRV_KEY_PATH "server.key"

static int verify_measurements_callback(const char* mrenclave, const char* mrsigner,
                                        const char* isv_prod_id, const char* isv_svn) {
  if (memcmp(mrenclave, expected_mrenclave, sizeof(expected_mrenclave)) {
    return -1;
  }
  // other checks…
  return 0;
}
static int communicate_with_client_callback(struct ra_tls_ctx* ctx) {
    uint8_t buf[sizeof(EXPECTED_STRING)] = {0};
    int ret = secret_provision_read(ctx, buf, sizeof(buf));
    if (ret < 0) { … } // connection closed or error
    if (memcmp(buf, EXPECTED_STRING, sizeof(EXPECTED_STRING))) { … } // wrong string error
    ret = secret_provision_write(ctx, (uint8_t*)SECOND_SECRET, sizeof(SECOND_SECRET));
    if (ret < 0) { … } // error
    return 0;
}

int main(void) {
  // …
  int ret = secret_provision_start_server((uint8_t*)FIRST_SECRET, sizeof(FIRST_SECRET),
                                          PORT, SRV_CRT_PATH, SRV_KEY_PATH,
                                          verify_measurements_callback,
                                          communicate_with_client_callback);
```

```
33.   if (ret < 0) { … }  // error
34.   // ...
35. }
```

It is the server side of the client described above. By including the Secret Provisioning library, `secret_provision_start_server()` initializes a Secret Provisioning Server:

- listening on `localhost:PORT`,
- using TLS with the server certificate (stored in `SRV_CRT_PATH`) and private key (stored in `SRV_KEY_PATH`).
- performing RA-TLS certificate verification followed by invocation of a user-defined `verify_measurements_callback()` to validate enclave measurements.
- automatically provisioning the initial secret `FIRST_SECRET`,
- executing a user-defined `communicate_with_client_callback()` for further interaction.

Within this callback, the server can process client-provided data (`secret_provision_read()`), apply application-specific validation logic, and return additional secrets using `secret_provision_write()`. The connection is terminated upon completion of this interaction.

It is important to note that the libraries provided by Gramine are not thread-safe. Additionally, the verification ones can be used outside the enclave, as they are distributed as part of the Gramine framework.

2.5.5. Encrypted Files

By specifying the `encrypted` type in filesystem mount, each file is transparently and independently encrypted. The encrypted file mechanism in Gramine provides the following guarantees [39]:

1. **Data confidentiality**, achieved through encryption of file contents stored in untrusted storage.
2. **Data integrity**, ensured via a Message Authentication Code (MAC) verified during file read and decryption operations.
3. **File path match**: enforced through metadata checks that ensure the file path at access time matches the one used during creation, thereby preventing file-swapping attacks.

However, certain limitations remain. Specifically, data freshness is not guaranteed after a file is closed, and some metadata—such as file name, file size, and access timestamps—are not protected [39].

Encrypted files follow a specific format, with distinct representations in host storage and within the SGX enclave. On the host, such files are identified by a special magic string (`GRAFS_PF`). For further details, the reader is referred to [39].

2.5.6. Gramine Shielded Containers

Gramine Shielded Containers (GSC) [40] is a tool employed in this work to integrate the Gramine environment into existing Docker images, enabling their execution within SGX enclaves. This integration process, referred to as "graminization", transforms a program or container image into a "graminized" version, meaning that it is executed through Gramine. The process consists of three main steps:

1. **Build Gramine**. In this step, the Gramine source code is compiled with a specified configuration. This step can be omitted if a pre-built Docker image containing Gramine is provided, as is the case in this work (as better described in following sections).
2. **Build the graminized image**. This step produces an intermediate Docker image with the suffix `-unsigned`. Inside it Gramine artifacts are copied from step 1. Environment variables are then configured by combining user-defined variables, GSC-specific variables, and those from the original Docker image, with precedence given in this order. Exceptions include `LD_LIBRARY_PATH`, `PATH`, and `LD_PRELOAD`, which are concatenated rather than overridden. Additionally, this step generates a list of trusted files, which includes all files from the original Docker image. These environment variables and the trusted file list are incorporated into a newly generated manifest. The container entry point is modified to invoke a helper script that executes `gramine-sgx`. The arguments passed to this command are derived from the original Docker image's `ENTRYPOINT` and `CMD` fields. They are stored in a trusted arguments file. Alternatively, an option is available to provide arguments at runtime.
3. **Sign the enclave**. In the final step, the unsigned image is signed using a specified key. This process generates the SGX-specific manifest file and the corresponding SIGSTRUCT files required by SGX. The resulting image is prefixed with `gsc-`.

Consequently, starting from a base image `img`, GSC produces a new image `gsc-img`, which is executed within an SGX enclave.

The graminized Docker image can be executed using standard Docker tools (e.g., Docker CLI or Docker Compose), provided that SGX devices and sockets are correctly exposed to the container.

To enable DNS resolution within the enclave, the manifest should include the option `sys.enable_extra_runtime_domain_names_conf = true`.

The build process reported above requires a configuration file in YAML format, which includes the following key parameters:

- `Distro`: specifies the Linux distribution used to build Gramine. It should match the base Docker image. When set to `auto`, GSC automatically detects the appropriate distribution.
- `Registry`: defines the repository hosting the Linux distribution image.
- `Gramine.Repository`: indicates the source repository of Gramine.
- `Gramine.Branch`: specifies the branch to use (default: `master`).
- `Gramine.Image`: optionally provides a pre-built image containing Gramine, in which case `Gramine.Repository` and `Gramine.Branch` are ignored.

As an illustrative example, the following steps demonstrate how to graminize the official Python 3 Docker image (taken from [40]):

1. Clone the official GSC repository,
2. Inside it copy the template configuration file. In this case, no modifications are required:

```
1. Distro: "auto"
2. Registry: ""
3. Gramine:
4.     Repository: "https://github.com/gramineproject/gramine.git"
5.     Branch:     "master"
```

3. Generate the signing key, for example using the Gramine utility:

```
gramine-sgx-gen-private-key
# or via openssl
openssl genrsa -3 -out enclave-key.pem 3072
```

The key may be stored securely or generated on a separate machine for signing purposes.

4. Pull the Python 3 Docker image

```
docker pull python:bullseye
```

5. Perform graminization using the GSC `build` command:

```
./gsc build --insecure-args python:bullseye test/generic.manifest
```

The `--insecure-args` option allows passing runtime arguments instead of relying on trusted arguments. The file `test/generic.manifest`, included in the official repository, serves as the Gramine manifest template used by GSC, and it is reported below:

```
1. sgx.enclave_size = "4G"
2. sgx.max_threads = 8
3.
4. sgx.trusted_files = [
5.   "file:/gramine/app_files/entrypoint.manifest",  # unused entry, only to test merging of manifests
6. ]
7.
```

Notably, `libos.entrypoint` does not need to be explicitly defined, as it is automatically configured by GSC.

6. Sign the graminized image:

```
./gsc sign-image python:bullseye enclave-key.pem
```

7. Execute commands within the container as usual. Just as an example:

```
1. docker run --device=/dev/sgx_enclave \
2.     -v /var/run/aesmd/aesm.socket:/var/run/aesmd/aesm.socket \
3.     gsc-python:bullseye -c 'print("HelloWorld!")'
```

Scripts can also be mounted as trusted files via the manifest and executed within the enclave. Anyway, scripts or commands are run within an SGX enclave.

It is important to note that GSC does not provide image encryption. This represents a limitation, along with others documented in [40]. For instance, only a limited set of base Linux distributions (e.g., Ubuntu and CentOS) is supported. Furthermore, mounting Docker volumes as `trusted_files` at container startup is not supported, as trusted files must be resolved at build phase. As a workaround, such files can be embedded directly into the image during that phase or specified as `allowed_files`.

2.6. Other Solutions

In this section, other frameworks based on or supporting Intel SGX that were analyzed but not adopted in this work are presented. The primary reason is that Gramine, together with its GSC tool, provides a more comprehensive and user-friendly solution, offering features that are not fully supported by alternative frameworks (as discussed in Section 2.6.5).

2.6.1. Occlum

Occlum is a memory-safe, multi-process LibOS that, similarly to Gramine, enables the execution of unmodified applications within an SGX enclave [41]. It is typically distributed as a Docker container, although it can also be installed via package managers.

Regarding Attestation, Occlum provides shared libraries supporting a low-level interface, including SGX Quote generation and verification. However, it does not offer higher-level abstractions comparable to Gramine's RA-TLS (Secure Channel) or Secret Provisioning mechanisms [42]. Nevertheless, RA can be integrated into the initialization phase `init` of the Occlum boot flow (more details below).

At the time of writing, Occlum supports 161 syscalls [43], which is slightly fewer than those supported by Gramine.

Occlum is included here only for comparison and is not used in the proposed system

2.6.1.1. Overview

Occlum operates through the concept of an "instance", which represents the execution environment. An instance is essentially a directory with a specific structure:

```
myinstance/
├── image
│   ├── bin
│   ├── dev
│   ├── etc
│   │   ├── hosts
│   │   └── localtime
│   ├── ext2
│   ├── host
│   ├── lib
│   │   ├── ld-musl-x86_64.so.1
```

```
│   │   ├── ...
│   │   └── libstdc++.so.6
│   ├── lib64
│   │   └── ld-linux-x86-64.so.2
│   ├── opt
│   │   └── occlum
│   │       └── glibc
│   │           ├── etc
│   │           │   └── localtime
│   │           └── lib
│   │               ├── ld-linux-x86-64.so.2 -> /lib64/ld-linux-x86-64.so.2
│   │               ├── ...
│   │               └── libstdc++.so.6
│   ├── proc
│   ├── root
│   ├── sbin
│   └── tmp
├── initfs
│   ├── bin
│   │   └── init
│   ├── dev
│   ├── etc
│   │   ├── hosts
│   │   └── localtime
│   ├── lib
│   │   ├── ld-musl-x86_64.so.1
│   │   ├── ...
│   │   └── libstdc++.so.6
│   └── proc
└── Occlum.json
```

It can be created using either `occlum init` within a directory (`myinstance`) or `occlum new myinstance` from its parent directory. Within an instance, the `initfs` and `image` directories emulate a traditional Linux filesystem:

- The `initfs` directory is the Occlum init filesystem, that is the filesystem used during the `init` phase of the boot flow (more details later).
- The `image` directory is the Occlum root filesystem seen by the application at runtime. It contains both application-specific files and Occlum-related components.

Application files can be copied into the `image` directory either manually or using the `copy_bom` utility [44]. This tool reads a BOM file and performs the copy to the destinations. A BOM file is written in YAML and this is an example:

```
1. includes:
2.   - base.yaml
3. targets:
4.   - target: /bin
5.     copy:
6.       - files:
7.         - ../hello_world
```

And to perform the copy:

```
copy_bom -f PATH_TO_BOM_FILE --root image --include-dir INCLUDE-DIRs
```

This means that under `image` destination path, all files and directories specified in targets will be copied from their source to the corresponding targets under `image`. It can include some other BOM files (`base.yaml` in the example) contained in `INCLUDE-DIRs`, which can include other targets. In this example the target executable is placed under `image/bin/hello_world` (and also all files and directories related to `base.yaml`).

Once all required files are in place under `image`, the command `occlum build` is used to package the environment. This process generates the Occlum filesystem image and the Occlum SGX enclave. Additionally, the `build` and `run` directories are created:

```
myinstance/
├── build
│   ├── .Occlum_sys.json
│   ├── .Occlum_sys.json.protected
│   ├── bin
│   │   ├── ...
│   │   └── occlum-run -> /opt/occlum/build/bin/occlum-run
│   ├── Enclave.xml
│   ├── image_config.json
│   ├── initfs
│   │   └── __ROOT
│   │       ├── 0200000000000000000000000000000
│   │       ├── ...
│   │       ├── 1000000000000000000000000000000
│   │       └── metadata
│   ├── lib
│   │   ├── libocclum-libos.signed.so
│   │   ├── ...
│   │   └── libocclum-pal.so.0.31.0
│   └── mount
│       └── __ROOT
│           ├── 0200000000000000000000000000000
│           ├── ...
│           ├── 1100000000000000000000000000000
│           └── metadata
├── image
│   └── ...
├── initfs
│   └── ...
├── Occlum.json
└── run
    ├── initfs
    │   └── __ROOT
    └── mount
        └── __ROOT
```

- The `build` directory is used to perform the boot flow (more details below).
- `run/mount/__ROOT` represents the root filesystem visible to the application during execution. This filesystem is encrypted by default (see below for details).

To execute an application within an SGX enclave, the following command is used:

```
occlum run PATH_TO_EXECUTABLE [ARGS]
```

where `PATH_TO_EXECUTABLE` refers to the path of the application within the Occlum filesystem image, and `ARGS` represents optional arguments.

Occlum also supports a simulation mode, which can be enabled during the build phase using:

```
occlum build --sgx-mode SIM
```

This mode allows testing without requiring SGX hardware support.

2.6.1.2. Occlum Filesystems

Occlum supports multiple filesystem types, each providing different levels of security and functionality:

Filesystem	Description

SEFS	It is a filesystem based on an Intel SGX library designed to protect filesystem data. It supports two operating modes:
	• **Integrity-only**: ensures the integrity of the filesystem.
	• **Encryption**: extends the integrity-only mode by providing confidentiality through encryption. The encryption key can be derived from:
	  ○ MRSGINER (together with PRODID and hardware information), which is the default option.
	  ○ A user-provided key.

UnionFS	It combines files and directories from multiple filesystems into a single unified view and is used as the root filesystem of the LibOS. It consists of two SEFS instances: 1. One integrity-only, initialized from the user-provided `image` directory at enclave build time. On the host, it is located at `./build/mount/__ROOT`, while inside the LibOS process it is mounted at `/`. 2. One encrypted, generated at runtime while the enclave is executing. On the host, it is located at `./run/mount/__ROOT`, while inside the LibOS process it is also mounted at `/`.
Async-FS	It is an asynchronous filesystem mounted at `/sfs` within the Occlum image, designed to improve performance. However, it is limited to files with a maximum size of 4 GB.
HostFS	A filesystem that maps the host filesystem into the Occlum image without providing protection or validation mechanisms.
RamFS	An in-memory filesystem.

Table 8 Occlum supported filesystems

Filesystems can be mounted or unmounted dynamically at runtime.

2.6.1.3. Configuration

Occlum configuration is defined in the `Occlum.json` file, which allows specifying the following parameters:

Configuration	Example

The total memory size allocated to theLibOS process and the number of threads.	`{`
	`  // Resource limits`
	`  "resource_limits": {`
	`  // The total size of enclave memory`
	`  // available to LibOS processes`
	`  "user_space_size": "512MB",`
	`  ...`
	`  // The max number of LibOS`
	`  // threads/processes`
	`  "max_num_of_threads": 256`
	`  },`
	`  ...`
	`}`

The valid entrypoints for `occlum run`,	`{`
	`  ...`
	`  "entry_points": [`
	`  "/bin"`
	`  ],`
	`  ...`
	`}`

Environment variables, which can either be hard-coded in the configuration and passed to the application (`"default"`) or inherited from the host system (`"untrusted"`).	`{ ... "env": { "default": [ "MY=VAR" ], "untrusted": [ "HOSTEXAMPLE" ] }, ... }`
ISVPRODID, ISVSVN, debug.	`{ ... "metadata": { // ISVPRODID "product_id": 0, // ISVSVN "version_number": 0, "debuggable": false, ... }`
Mount points and filesystem configurations. For each filesystem, it is possible to define its type, its mount point within the LibOS process, and additional parameters (e.g., the source path for HostFS).	`{ ... "mount": [ { "target": "/example", "type": "hostfs", "source": "./example" }, ] ... }`

Table 9 Occlum configuration

This configuration is processed during the `occlum build` phase.

2.6.1.4. Boot Flow

Occlum follows a well-defined boot flow, illustrated in Fig. 16 (taken from [45]):

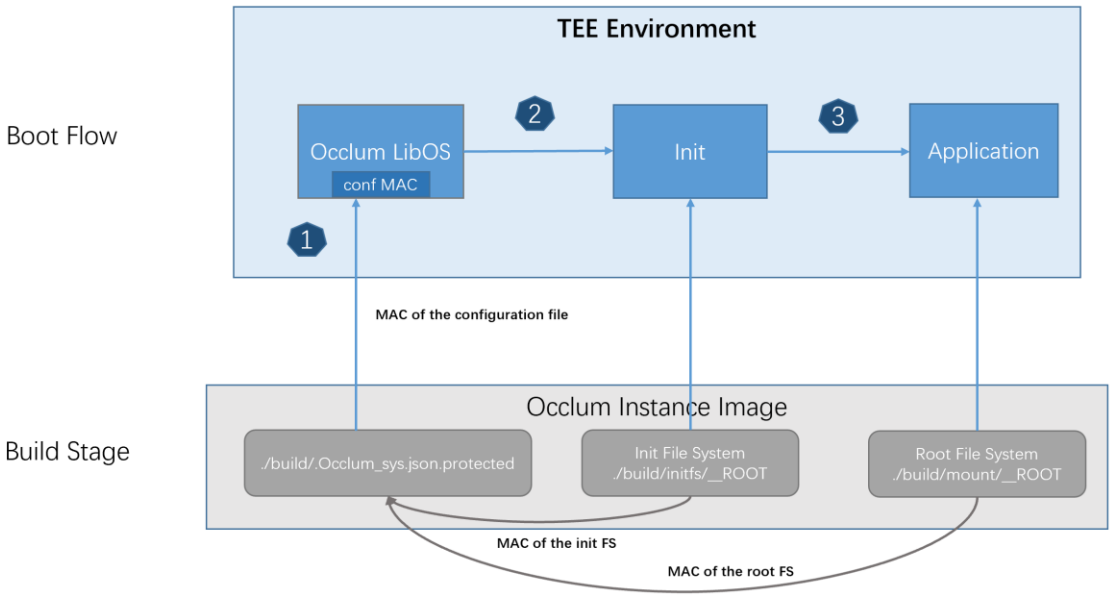

Fig. 16 Occlum boot flow

After executing `occlum build`, the `build` directory contains several files and directories, including:

- the **init filesystem**, which is mounted during the init stage (step 2 in Fig. 16).
- the **root filesystem**, which is mounted during the application stage (step 3 Fig. 16).
- the `.Occlum_sys.json.protected` file, which stores the measurements of the aforementioned filesystems and is itself measured.

Executing `occlum run` initiates the boot process, which consists of three stages, as reported in Fig. 16:

1. **Occlum LibOS stage**. The Occlum LibOS kernel is executed within the enclave. It loads `.Occlum_sys.json.protected` and verifies the integrity of the configuration. If the verification succeeds, it performs initialization steps as specified in the configuration. It then verifies the integrity of the init filesystem. Upon success, the filesystem is mounted and the `init` process is launched.
2. **Init stage**. The `init` process may perform preliminary operations, such as RA. It subsequently mounts the root filesystem for the application, provided that the integrity check is successful.
3. A**pplication stage**. The application is executed.

2.6.2. EGo

EGo is a framework that enables the execution of Go (GoLang) applications within Intel SGX enclaves [46]. It supports running unmodified Go applications, while also providing the EGo module [47], which simplifies the use of SGX features such as RA and sealing.

The framework includes a modified Go compiler that remains compatible with the standard one. As a result, applications compiled with EGo can be executed both inside an enclave and in a conventional (non-enclave) environment. Additionally, EGo provides a dedicated runtime environment for enclave execution.

Ego is configured through a JSON file `that` defines the configuration for both the EGo framework and the SGX environment, for example:

```
{
    "exe": "main",
    "key": "private.pem",
    "debug": false,
    "heapSize": 512,
    "executableHeap": false,
    "productID": 1,
    "securityVersion": 1,
    "mounts": [
        {
            "source": "/tmp/",
            "target": "/tmp",
            "type": "hostfs",
            "readOnly": false
        }
    ],
    "env": null,
    "files": null
}

```

Specifically, it specifies that:

- The application to be run is `main` (`"exe": "main"`),
- The enclave signing key is provided in the file `"private.pem"` (`"key": "private.pem"`) from which MRSIGNER is derived,
- Debug mode is disabled (`"debug": false`),
- The enclave is allocated 512 MB of memory (`"heapSize": 512`), and the heap is non-executable (`"executableHeap": false`), which is relevant for libraries relying on just-in-time compilation,
- ISVPRODID (`"productID": 1`), ISVSVN (`"securityVersion": 1`),
- The host directory `/tmp` is mounted within the enclave filesystem at `/tmp` with write permissions,
- No environment variables (`"env": null`) or additional files (`"files": null`) are included in the enclave measurement.

The EGo module also provides mechanisms for generating X.509 certificates that embed SGX Quotes. However, it is not compliant with the RA-TLS standard. Specifically, it uses the extension `1.3.6.1.4.1.311.105.1` [48] , rather than `2.23.133.5.4.9`, which is adopted by Gramine.

A minimal sealing example using the EGo API is reported in Appendix C.

2.6.3. Confidential Containers (CoCo)

This project, under the Cloud Native Computing Foundation (CNCF), enables Kubernetes to run containers within confidential environments referred to as Confidential Virtual Machines (CVMs) [49]. They are currently backed by Intel SGX, Intel TDX, and AMD SEV-SNP. The framework leverages both the containerd and Kata Containers runtimes [50]. The latter is an open-source project that enhances isolation and security by running lightweight Virtual Machines instead of traditional containers.

Within each CVM—hosting a Pod that contains the workload—several additional components are introduced:

- **Kata agent**: manages the container lifecycle.
- **Attestation Agent (AA)**: generates the SGX Quote and handles the RA process.
- **Confidential Data Hub (CDH)**: provides secrets and keys to the workload after successful attestation.

Externally, trust is established through the so-called "Trustee stack", which includes:

- **Key Broker Service (KBS)**: manages sessions and provisions secrets and keys.
- **Attestation Service (AS)**: performs Attestation verification against defined policies.
- **Reference Value Provider Service (RVPS)**: stores policies and trusted measurements.

From the user's perspective, Kubernetes workloads operate as usual. However, prior to container execution, CoCo performs an Attestation workflow to ensure workload integrity and authenticity. The process consists of the following steps:

1. The container image is encrypted (or signed) and pushed to a registry.
2. You provision the encryption (or signing) key to the KBS.
3. When a Pod using the image is deployed, the AA initiates a session request with the KBS.
4. The KBS returns a nonce to the AA.
5. The AA generates and sends its SGX Quote to the KBS.
6. The KBS forwards the Quote to the AS.
7. The AS verifies the Quote against policies and trusted measurements obtained from the RVPS.
8. The verification result is returned to the KBS.
9. If successful, the KBS delivers the key to the CDH.
10. The key is then used to decrypt the image and proceed with execution.

However, for enclave-based containers, the framework requires container images integrated with a LibOS [51], although this integration is already handled by GCS. Furthermore, future releases are expected to discontinue Intel SGX support in favor of Intel TDX and AMD SEV-SNP.

2.6.4. Fabric Private Chaincode (FPC)

By default, HLF executes components within standard Docker containers. Consequently, these components do not operate in confidential environments, and both private data and

Blockchain state remain exposed in plaintext in RAM. FPC is a prototype project that introduces CC capabilities into HLF by enabling Peer nodes to execute Chaincode within a TEE [52]. Specifically, it leverages Intel SGX through the Intel SGX SDK. To support this functionality, four main components are introduced:

- **Chaincode enclave**. This component runs within an enclave and executes a single Chaincode instance. Upon creation by a Peer, it generates a public–private key pair and uses the public key as its identity. It then attempts to register with the enclave registry, which accepts the registration upon successful RA.
- **Ledger enclave**. This enclave maintains metadata related to the latest Blockchain state. For instance, when data are retrieved from the Ledger, the Chaincode verifies consistency by comparing the retrieved value with the corresponding value stored in the Ledger enclave (properly hashed and encrypted with the private key of the Ledger enclave).
- **Enclave registry**. Implemented as a Chaincode outside the enclave, this component tracks all registered Chaincode enclaves. It stores Attestation results obtained through RA, allowing clients and Peers to verify whether a given Chaincode is executing within a trusted enclave.
- **Enclave transaction validator**. This component operates outside the enclave and assists Peers in validating transactions involving Chaincodes running inside an enclave.

The transaction and validation flows largely resemble those of standard HLF, with the addition of mandatory RA steps. Before invoking a transaction, a client retrieves both the public key of the Chaincode enclave and its Attestation result. Using this information, the client verifies that the Chaincode is running inside a trusted enclave with the expected code. If verification succeeds, the client encrypts the transaction invocation using the Chaincode enclave's public key and submits it to the Peer. Before executing the Chaincode, the Peer also performs RA with the Chaincode. Upon successful verification, the Chaincode enclave is invoked and returns an encrypted result, optionally using a key provided by the client.

It is important to note that data are processed in plaintext within the Chaincode enclave but are encrypted before being written to the Ledger and decrypted upon retrieval. The key used for this process can be provided by the client or the system administrator.

To develop FPC-enabled Chaincodes, developers must use the FPC Shim [53], a middleware layer that connects Fabric Peers with enclaves and provides an interface for interacting with the Ledger. However, the current implementation presents several limitations, including the lack of support for PCs and CouchDB rich queries. This limitation renders the starting platform used in this work incompatible with FPC. In contrast, frameworks such as Gramine can support PCs without requiring code modifications (in principle).

Finally, while the Ledger itself is not sealed, the Chaincode state is encrypted by the enclave before being persisted [54].

2.6.5. Why Intel SGX and Gramine

Following a review of the aforementioned technologies and frameworks, a selection was required among different TEE implementations, each characterized by a distinct confidential packaging model. The comparison is summarized below:

TEE Implementation	Packaging Model	Pros	Cons
Intel SGX	Confidential library / Confidential container (via frameworks)	1. Available on consumer hardware for older releases, making it suitable for development and testing. 2. Provides fine-grained isolation, resulting in a smaller TCB and reduced attack surface. Additionally, not all applications within a VM are required to run confidentially. 3. Existing applications or container images can be executed without modification when using appropriate supporting frameworks.	1. Cloud and on-premises deployments for recent hardware generations can be costly. 2. The Intel SGX SDK is not suitable for many practical use cases.
Intel TDX	Confidential VM	1. All applications executed within the VM are automatically protected, requiring no modifications.	1. Both cloud and on-premises solutions can be expensive. 2. Limited adoption at the time of evaluation.
AMD SEV-SNP	Confidential VM	1. All applications executed within the VM are automatically protected, requiring no modifications.	1. Both cloud and on-premises solutions can be expensive.

Table 10 TEE implementations comparison

Based on this comparison, Intel SGX was selected due to: (i) its broader availability, (ii) its finer-grained security model, (iii) the ability to run unmodified applications through supporting frameworks, and (iv) a more favorable trade-off between advantages and limitations compared to alternative TEE implementations.

Subsequently, it was necessary to identify the SGX-based framework best suited for our work based on HLF. The following comparison summarizes the evaluated frameworks:

Framework	Type	Source change	Sealing	Remote attestation	RA-TLS	Containerization support	Limitations
Gramine [29]	LibOS	No	Yes (MRSIGNER and MRENCLAVE)	Yes (via libraries)	Yes	Yes (via GSC)	Partial support for system calls; GSC does not encrypt container images, supports a limited set of Linux distributions, and does not allow Docker volumes mounted at runtime to be used as trusted files.
Occlum [41]	LibOS	No	Yes (MRSIGNER only)	Yes (via libraries)	No	Yes	Partial support for system calls; requires manual creation of application images with Occlum integration; sealing does not support MRENCLAVE.
EGo [46]	Go library	Partially	Yes, programmatically	Yes, programmatically	No	No	Restricted to Go applications.
CoCo [49]	Container runtime (Kubernetes-based)	No	Yes, transparently	Only within the CoCo ecosystem	No reference to it	Yes	Requires LibOS-based images; Intel SGX support is expected to be discontinued in future releases
FPC [52]	HLF-specific framework	Yes	No	Limited to FPC-specific workflows	No reference to it	Yes	Restricted set of supported ledger operations. Only C/C++ supported.

Table 11 SGX-based frameworks comparison

In this work, Gramine was selected to execute the Platform within SGX enclaves for the following reasons:

1. It enables the execution of (in principle) unmodified applications written in any programming language.
2. It provides GSC, a unique feature among the analyzed frameworks, which allows existing Docker images to be "Graminized" with minimal effort.
3. It supports automatic sealing of directories and their subdirectories through configuration. Moreover, it avoids sealing the entire enclave filesystem, thereby

mitigating potential performance overhead. It supports multiple sealing policies, including MRENCLAVE-, MRSIGNER-, and key-based approaches.
4. It offers multiple abstraction levels for RA, simplifying integration depending on application requirements, and includes support for RA-TLS.

3. Graminized HLF-based logistics solution

This work builds upon an existing HLF-based solution developed for the logistics domain, which has been extended to operate within an Intel SGX confidential environment using the Gramine framework.

3.1. The Existing Solution

The original solution leverages HLF to provide controlled data access among Organizations participating in a channel [1, 2]. This is achieved through a distributed Access Control List (dACL) mechanism built on top of PCs. For further details, refer to [1, 2].

The system is based on two main roles: **Shipment Owners (SOs)** and **Shipment Handlers (SHs)**. SOs are responsible for creating shipments that track one or more containers, while SHs generate events associated with these containers and forward them to the respective SOs. A more detailed description of the workflow is provided in the following sections.

The architecture consists of a Channel representing the logistics hub. The Channel includes:

- Three Orderer nodes belonging to the Orderer Organization.
- One Peer belonging to the Port Authority Organization, configured with two PCs: one for authorizations (PC_A) and one for events (PC_E), with the Chaincode installed.

Each Organization operates its own CA, responsible for generating the cryptographic material required to join the Channel and for managing Organization's user identities.

At runtime, i.e. when a Channel is already up and running, Organizations may dynamically join or leave the Channel. Upon joining, an Organization is provisioned with its own infrastructure, analogous to the Port Authority setup: one Peer, two PCs, one CA, and the installed Chaincode. Notably, each Organization's single Peer fulfills multiple roles simultaneously, acting as an Anchor Peer, Endorsing Peer, and Leading Peer (without followers).

3.1.1. PC for Authorizations

PC_A stores dACL records written by SOs within the SHs' collections, thereby authorizing SHs to write events into the SOs' PC_E. While this design may appear vulnerable—since an SH could attempt to authorize itself by modifying its own PC_A—several safeguards mitigate this risk:

1. The Chaincode is governed collectively and cannot be modified unilaterally by a single Organization; any change requires consensus among participating Organizations.
2. When an SO creates a shipment, it is recorded on the global Ledger and associated with that SO. Through State-based endorsement policies, only the owning SO is permitted to

modify it. Unauthorized modification attempts are rejected. General information is stored on the global Ledger, while sensitive data reside in the SO's implicit PC.

3. Authorization records written by an SO can only be modified by that SO, even if stored within an SH's PC_A.
4. A useful function enables verification of whether a given event exists within an Organization's PC_E. This is achieved by hashing the provided plaintext event and comparing it with the stored hash retrieved from the hashed representation of the PC_E.

3.1.2. PC for Events

PC_E is the collection used to store events and is owned by SOs (each SH also maintains its own PC_E). Only authorized SHs and the owning SO are permitted to write to it. By design, only the owning Organization can read and modify its PC_E, while other Organizations can only submit write requests. The dACL mechanism enforces authorization checks prior to any write operation. If an SH is not authorized, the failed attempt is explicitly recorded on the Blockchain.

3.1.3. ACL

The access control mechanism is implemented as a distributed ACL, meaning there is no centralized authority managing access requests. Instead, ACL records are distributed across the relevant PCs where they are required. Currently the ACL only works for writing operation.

In HLF, user identities can include an affiliation field within their certificates; this field is leveraged to represent user roles within an Organization. Each ACL record consists of two components: a **Principal** and a set of associated container identifiers. A Principal is defined as a triple `⟨User, Role, Organization⟩`, inspired by access control models in Unix-like systems. While the User and Role fields are optional, the Organization field is mandatory. The semantics are as follows:

- `⟨U,R,O⟩`: authorizes a specific user `U` within Organization `O` having at least role `R`,
- `⟨-,R,O⟩`: authorizes all users in Organization `O` with at least role `R`,
- `⟨U,-,O⟩`: authorizes a specific user `U` within Organization `O`,
- `⟨-,-,O⟩`: authorizes all users within Organization `O`.

User names are unique within each CA and, consequently, within each Organization.

Once defined, a Principal is associated with a set of container IDs. This association grants write access to events related to those containers for users belonging to the Organization hosting the dACL record and satisfying the Principal constraints.

3.1.4. Chaincode and Client Application

The Chaincode is designed to support multiple operations, including:

- creation of shipments and events,
- retrieval of events by container ID,
- event verification,
- creation and retrieval of domain-specific entities (e.g., Location, Position, Document).

These entities are derived from the BiTAS (Blockchain in Transport Alliance Standard) [55], which forms the basis of the data model used in this Platform.

While the Chaincode adheres strictly to the BiTAS data structure, client applications must interface both with BiTAS and with the potentially heterogeneous data models used by individual Organizations. To address this, an Adapter pattern is employed, enabling bidirectional translation between Organization-specific data formats and the BiTAS standard.

The client application supports user management operations, including user creation and re-enrollment. It also provides functionality to retrieve detailed information about Blockchain blocks associated with a given transaction ID, such as the submitting entity, the hash of the previous block, and the timestamp of creation. Two types of client applications are provided:

- A command-line Java application, primarily intended for administrative tasks (e.g., user management) and testing (e.g., creation of shipments and synthetic events).
- A REST-based application, which enables the creation of shipments and events through well-defined APIs, facilitating integration with external systems while adhering to the required data structures.

Table 12 reports a clear mapping of all system components to their respective origins, explicitly distinguishing between elements inherited from the existing solution and those introduced as novel contributions in my work.

Component	Description	Origin
PC configuration	Specification of explicit (PC_A, PC_E) and implicit PCs per Organization, including distribution and endorsement policies, as well as configuration parameters.	Inherited
dACL mechanism	Implementation of a Chaincode-level access control mechanism that allows Organizations to grant and revoke write permissions on their PC_E to designated Organizations.	Inherited
Authorization model (⟨U,R,O⟩)	Design of a fine-grained authorization model based on user, role, and Organization, enabling multiple levels of access granularity within the dACL.	Inherited
Chaincode implementation	Chaincode functionalities for shipment and event creation, querying, and hash-based integrity verification, ensuring that private data remain undisclosed.	Inherited
REST-based application	REST-based interface that abstracts interactions with the Chaincode for authorized users.	Inherited
Gramine-based deployment strategy	Methodology for graminizing all HLF components — Peers, Orderers, Chaincodes (via CCAAS), and REST-based client applications — allowing their execution within SGX enclaves without requiring source code modifications.	New

RegisterEnroll	Component responsible for generating and sealing cryptographic materials for each HLF component at startup within its enclave, thereby preserving confidentiality and integrity at the filesystem level.	New
Peer Operator	RESTful component co-located with the Peer within the same enclave, enabling administrative Peer operations to be executed in an environment where cryptographic materials remain sealed.	New
RA-REST	RESTful component that enforces RA of external TPS prior to disclosing selected private collection data, thereby enabling controlled and verifiable data sharing with entities outside the blockchain network.	New
Sealing strategy	Configuration-driven sealing of all cryptographic materials and Blockchain state using Gramine's sealing policies.	New
Go/cgo attestation module	Go module interfacing with Gramine's C/C++ RA libraries via cgo, enabling native integration of RA within Go-based HLF components.	New
Gramine and HLF source patches	Targeted modifications to Gramine (directory rename support) and to HLF Peer/Orderer source code (library refactoring, CCAAS script refactoring, loopback-address detection removal).	New
CCAAS Chaincode per-Organization	Adaptation of the Chaincode-as-a-Service (CCAAS) model to deploy one graminized Chaincode container per Organization, preventing private data from traversing non-confidential containers; additionally, Peer deployment scripts have been refactored into libraries to avoid failures in Gramine's checkpoint-and-restore mechanism.	New

Table 12 Mapping of system components to their origin. Components labeled as "Inherited" are reused from the existing solution [1, 2], whereas components labeled as "New" are introduced in my work.

3.2. The Graminized solution

The existing Platform was adapted to run within Intel SGX enclaves using the Gramine framework. The graminization process of Docker images—such as those used by HLF Peers and Orderers—was significantly simplified through the GSC tool. The graminized Platform (GP) is based on the software stack and implementation reported by Table 13.

Component	Version	Gramine Manifest Strategy	Sealing Strategy	Status
Gramine	v1.8	–	–	Supported
HLF Peer node	v2.5.11	GSC-based graminization; the Peer binary is refactored to support both library usage (for integration within the PO process) and CLI execution; CCAAS deployment scripts are refactored as libraries;	MRSIGNER — Peer keys and Blockchain state	Supported

		RegisterEnroll is executed at startup alongside the PO.		
HLF Orderer	v2.5.11	GSC-based graminization; minor source-level patches are applied to address unsupported system calls; RegisterEnroll is executed at startup.	MRSIGNER — Orderer keys	Supported
CCAAS	v2.5.11	GSC-based graminization of a custom image; one container is deployed per Organization; RegisterEnroll is executed at startup.	MRSIGNER — Chaincode keys	Supported
REST Client Application	v2.5.11	GSC-based graminization of a custom image; RegisterEnroll is executed at startup.	MRSIGNER — HLF client keys	Supported
RA-REST	Custom	GSC-based graminization of a custom image; integration of a Go/cgo module enabling interaction with Gramine's RA mechanisms.	MRSIGNER — HLF client keys	Supported
Peer Operator	Custom	Co-located with the Peer under the same Gramine configuration; exposes a RESTful interface.	MRSIGNER (shares Peer enclave)	Supported
RA between PO and clients	–	Planned feature; the administrative plane is not yet subject to RA.	N/A	Future work
CouchDB	v3.1.1	Not graminized; communicates with the Peer via TLS.	N/A	Known limitation

Table 13 Summary of software versions, graminization and sealing strategy, and current support status for each HLF component

The GP is founded on four key mechanisms—PCs, Sealing, RA, and enclave-based execution—whose combined use is essential. Table 14 illustrates the residual threat exposure associated with the omission of any individual mechanism, thereby justifying the layered design adopted in this section.

Mechanism Removed	Residual Gap
PCs removed	Private data are sent in plaintext to all channel participants, thereby violating application-level privacy requirements among HLF Organizations, irrespective of underlying hardware protections.

Sealing removed	Cryptographic materials and Blockchain state are stored in plaintext on the filesystem, allowing a privileged host to access them directly, despite encryption of data in memory during execution.
RA removed	An external TPS may access private data without any verifiable assurance of its identity or execution environment; consequently, a malicious or compromised TPS cannot be distinguished from a legitimate one based solely on network credentials.
Enclave-based execution removed	Data in use remain in plaintext within main memory, enabling a privileged host (e.g., hypervisor or cloud provider) to access them during processing, even if they are sealed on the filesystem and logically isolated at the application layer.

Table 14 Residual threat exposure resulting from the omission of each privacy mechanism, highlighting the necessity of the GP architecture.

Hereinafter "Remote Attestation" or "Attestation" are used interchangeably.

Architecture overview. Before the individual components are described in detail, Fig. 17 presents a high-level overview of the proposed architecture for a single Organization. Every HLF component — the Peer (co-located with the PO and RegisterEnroll), the Orderer nodes, the Chaincode (deployed via CCAAS), and the REST client application — executes unmodified (in principle) within its own SGX enclave through Gramine, and the RA-REST server runs in a dedicated enclave that mediates data sharing toward external TPS. The only component that remains outside the enclave boundary is CouchDB, which stores the world state and communicates with the Peer over TLS; this is a known limitation, discussed in 3.2.1 and 4.1. Cryptographic material and Blockchain state are sealed through MRSIGNER. External interactions cross the enclave boundary at three points: the administrator drives Peer operations through the PO over HTTPS; an external TPS is granted data only after per-request RA over RA-TLS; and clients and the Peers of other Organizations interact through the standard Fabric gRPC channel. The same graminized stack is replicated by every Organization participating in the channel. Notice that an Orderer node belongs only to the administrative Organization, but for simplicity, in figure it is reported together with a Peer node.

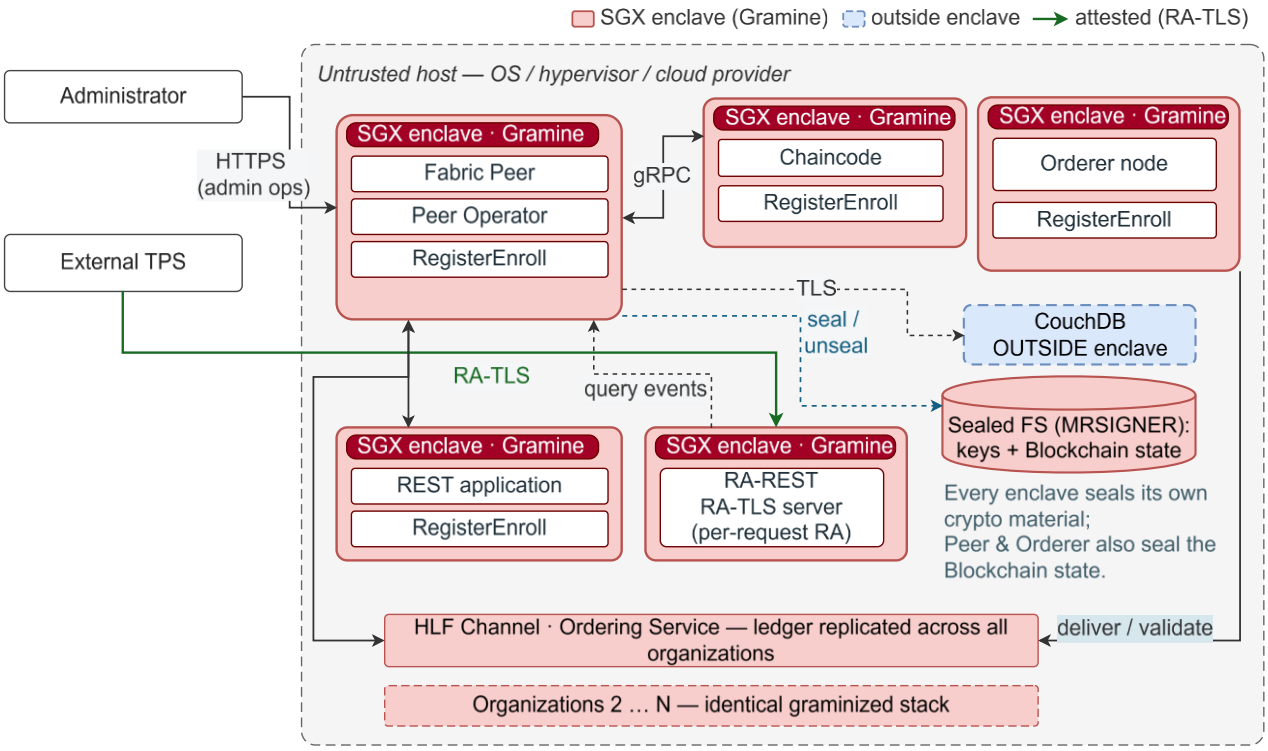

Fig. 17 GP architecture per Organization. All HLF components, except CouchDB, run inside SGX enclaves via Gramine.

TCB and design rationale. Executing entire HLF components inside enclaves, rather than only the Chaincode, enlarges the TCB with respect to a minimal-enclave design such as FPC [52]. The enclave-resident TCB of the proposed system comprises, per Organization, five enclaves: (i) the Chaincode together with RegisterEnroll; (ii) the Peer together with the PO and RegisterEnroll; (iii) the Orderer together with RegisterEnroll; (iv) the REST application together with RegisterEnroll; and (v) the RA-REST server. Each enclave additionally includes the Gramine LibOS. Thus, the TCB encompasses all of the components listed above, whereas the TCB of FPC is limited to the Chaincode and the trusted ledger enclave. The host-interaction surface of each enclave — the set of files and host services that cross the enclave boundary — is explicitly enumerated by the corresponding Gramine manifest (the `trusted_files`, `allowed_files`, and `fs.mounts` entries reported throughout Sections 3.2.3–3.2.8) and corresponds to the interface serviced by Gramine, which is broader than the narrow ECALL/OCALL interface of FPC.

This design decision is deliberate and is defended on three grounds:

- First, the minimal TCB of FPC is small precisely because its protection scope is narrow: only the Chaincode executes inside the enclave, whereas the Peer process runs outside it. Under the threat model of this work (Goal G1), in which a privileged host can read the Peer's main memory and filesystem, a minimal-enclave design does not, by itself, protect the Peer's in-memory private data, the ledger, or the application-level access-control machinery of the inherited platform. The whole-component-in-enclave decision therefore trades a larger TCB for a strictly broader confidentiality guarantee:

the increased TCB is the cost of protecting the entire component rather than the Chaincode alone.

- Second, the graminized approach preserves the full feature set of the inherited platform. In particular, it retains PCs and CouchDB-based rich queries, which are not supported by FPC and on which the inherited dACL design directly depends. It also retains language compatibility: existing Fabric components and Chaincode run unmodified, whereas adopting FPC would require re-implementing the Chaincode against the FPC programming model.
- Third, from an engineering-economics standpoint, the inherited components are graminized almost as-is through GSC, and the implementation effort is concentrated on the new components (the PO, RegisterEnroll, and RA-REST) and on the targeted Gramine and HLF patches, rather than on a re-implementation of the application logic.

The explicit and acknowledged cost of this decision is a larger enclave-resident TCB and a correspondingly larger residual attack surface.

Table 15 contrasts the whole-application-enclave approach adopted in this work with the minimal-enclave approach of FPC.

Dimension	Whole-application-enclave (this work)	Minimal-enclave (FPC)
TCB size	Large: full component (Peer/Orderer/Chaincode/REST) + Gramine LibOS + new components	Minimal: Chaincode + trusted ledger enclave
Protection scope	Entire component: data-in-use in memory + MRSIGNER-sealed crypto material and ledger	Chaincode execution and its state; Peer runs outside the enclave
Feature compatibility	Preserves PCs, CouchDB rich queries	Not supported
Language compatibility	Existing components/Chaincode run unmodified	Chaincode must target the FPC programming model (C/C++)
Development effort (existing platform)	Low (GSC graminization) + focused effort on new components	High (Chaincode re-implementation and deployment methods)
Residual attack surface	Larger (more enclave code; Gramine host-interaction surface)	Smaller (narrow ECALL/OCALL interface)

Table 15 Whole-application-enclave (this work) vs. minimal-enclave (FPC) design trade-off.

The impact of potential SGX vulnerabilities is discussed in Section 3.2.1.1.

3.2.1. Security assumptions

The threat model is intentionally scoped to a specific and practically relevant adversarial setting: a malicious or compromised hosting platform. This includes cloud infrastructure providers, hypervisors, and privileged system operators with unrestricted access to the physical machine or virtual machine on which HLF components execute. Within this model, the adversary is assumed to be capable of reading and modifying main memory, inspecting filesystem contents, and observing network traffic at the host level. However, it is assumed that hardware-level guarantees provided by Intel SGX, as exposed through the Gramine framework, remain intact and cannot be compromised.

Within this model, several additional trust assumptions are made. First, Intel SGX and the Gramine framework are assumed to provide their specified security guarantees; their internal security mechanisms are not the focus of my work. Second, HLF Organizations trust each other and Channel components are mutually trusted once deployed and verified through the standard HLF lifecycle process. Third, The governance mechanisms of the HLF Channel, including the Chaincode endorsement policy and the dACL mechanism, are assumed to function correctly; Byzantine behavior among channel participants is considered out of scope. Fourth, the hosting platform is explicitly untrusted and represents the primary adversary targeted by my work. Finally, External TPSs are not trusted by default; trust is established on a per-request basis through RA.

The sealing strategy adopted throughout the system is MRSIGNER-based, meaning that sealed cryptographic material can be decrypted by any enclave signed with the same author key, regardless of its internal code. This design choice deliberately prioritizes operational continuity across software updates—an essential requirement in HLF systems where Peer and Chaincode binaries evolve through a governed lifecycle—over the stronger isolation guarantees of MRENCLAVE-based sealing, which would invalidate all sealed data upon any modification of the enclave code.

Table 16 provides a structured analysis of the main threat classes relevant to the proposed system, categorizing each as mitigated, known limitation, or explicitly out of scope.

Threat Class	Treatment in the Proposed System	Status
Privileged-host memory access	The memory of all HLF components is encrypted within SGX enclaves through Gramine, ensuring that a compromised host operating system, hypervisor, or cloud provider cannot access plaintext data during execution.	Mitigated

Plaintext key material on the filesystem	All cryptographic materials are protected via MRSIGNER-based Gramine sealing. Sealed artifacts are stored in encrypted form on the filesystem and can only be decrypted by enclaves signed with the same author key.	Mitigated
Untrusted TPS accessing private data	The RA-REST component enforces RA at every request. A TPS is granted access to PC data only after successful verification of its RA-TLS certificate, including validation of MRENCLAVE against a predefined set of trusted measurements.	Mitigated
Tampered Chaincode	Modifications to Chaincode require approval through the HLF lifecycle process, which is governed by a quorum of Channel Organizations and executed within enclaves. In addition, Gramine performs binary integrity verification of Chaincode binary during enclave initialization.	Mitigated
CouchDB outside enclave	CouchDB operates outside the SGX trust boundary and is not executed within an enclave in the current implementation. Communication with CouchDB relies on TLS to ensure transport-layer confidentiality. This limitation is acknowledged, and enclave-based deployment of CouchDB is considered future work.	Partial / future work
Administrative plane	The PO exposes a REST interface for administrative Peer operations. In the current implementation, RA between the PO and requesting clients is not enforced, meaning that the administrative plane does not yet benefit from attestation-based trust guarantees. This is an acknowledged limitation and a candidate for future work.	Partial / future work
SGX side-channel attacks	Side-channel attacks on SGX enclaves are well documented in the literature [17, 19, 20], and various mitigation strategies have been proposed. This class of attacks is explicitly considered out of scope in this work.	Out of scope
Rollback and replay attacks on sealed data	Gramine-based sealing does not inherently provide freshness guarantees; a privileged adversary may replace a sealed artifact with an older version. The attack cannot forge history: the ledger is replicated across the Peers of several Organizations and anchored by the Ordering Service, so blocks are delivered in a fixed, signed, hash-chained sequence, a rolled-back node re-synchronizes via the gossip protocol, and any gap or fork is detected through the block hash chain. The residual risk is thus confined to availability and to stale-but-authentic local reads served by a compromised node before re-synchronization, and to the freshness of secrets held	Residual risk

	only in the local sealed store. Strengthening this risk is left as future work.	
Compromised shared CA	The experimental setup employs a single CA shared across all Organizations, which is suitable for a controlled evaluation environment. The compromise of the CA is considered out of scope.	Out of scope
Compromised Ordering Service	Orderers are assumed to be trustworthy and are operated under the administrative governance of the Channel. Byzantine behavior in the Ordering Service is out of scope; the system relies on HLF's Raft-based consensus, which provides crash fault tolerance but not Byzantine fault tolerance [56].	Out of scope
Malicious enclave author (MRSIGNER trust)	MRSIGNER-based sealing implies that any enclave signed with the same author key can decrypt sealed data, regardless of its internal implementation. Consequently, the security of the author's signing key is a critical trust assumption, and its compromise is considered out of scope.	Out of scope
Post-attestation data misuse by trusted TPS	RA ensures the identity and integrity of a TPS enclave at the time of interaction; however, it does not prevent a correctly attested TPS from misusing data after it has been received — for example, by exfiltrating it through a covert channel. This limitation is inherent to attestation-based secure data-sharing architectures.	Residual risk

Table 16 Threat model summary: principal threat classes, their treatment in the proposed system, and their scope status

Threat classes classified as out of scope should not be interpreted as negligible or unimportant; rather, they correspond to orthogonal research problems or engineering challenges that lie outside the primary contributions of this work. Potential extensions addressing these aspects are discussed in Section 4.1 as future work. Inference attacks based on on-chain metadata or cross-chain information leakage are also considered out of scope in the present study.

3.2.1.1. Defense-in-depth considerations

A security analysis of Intel SGX or Gramine themselves is outside the scope of this work, which focuses on their integration with HLF; what follows is therefore not an assessment of SGX's internal robustness, but a structured reflection on the ramifications for the integrated deployment should an SGX guarantee fail. SGX is consequently treated as one layer of a defense-in-depth architecture rather than the sole defense. If enclave confidentiality fails, data-in-use on the affected host may be exposed. One solution could be to run each Organization's node on premises. Otherwise, the application-layer PCs and dACL still restrict data distribution, TLS still protects data in transit, and the integrity, immutability, non-repudiation and multi-Organization replication of the ledger remain unaffected, so the breach is scoped to the compromised host rather than to the whole network. In RA, compromised platforms can be revoked. Moreover, data requiring confidentiality over very long horizons should not rely on a hardware TEE alone: this is a residual risk whose solution can be planned in future.

3.2.2. Revisions and Extensions

Several modifications were required to enable the execution of GP. The following issues were encountered and addressed. Items 1, 2, 4, and 7 involve modifications to the source code of HLF and/or Gramine, while the remaining items concern Platform-level adaptations or newly introduced components:

1. **Peer network address resolution**
   HLF Peers require a network address to bind to, typically provided via environment variables. However, Peers attempt to retrieve the host loopback address using the Netlink interface with the `AF_UNSPEC` address family. Gramine does not support this address family, instead supporting only `AF_INET`, `AF_INET6` and `AF_UNIX` [57]. Additionally, the Peer configuration flag `addressAutoDetect` does not prevent this behavior. In fact, even if it is defined as "Whether the Peer should programmatically determine its address" and "When set to true, will override Peer address" [58], regardless of its value, the Peer tries to get the loopback address, and checks if the provided address is `0.0.0.0` or `::` (and in this case it returns the auto detected address), then it checks the flag. Since explicitly provided addresses in this work never use such wildcard values, the loopback detection step was removed from the Peer logic, thereby avoiding the need to extend Gramine with additional system call support, which may not be immediate.
2. **Directory rename limitation in Gramine**
   During startup, an Orderer attempts to rename a directory. As reported in [59], Gramine's implementation of the `rename` system call supports only regular files. To address this limitation, the Gramine source code (approximately 5 lines of code) was modified to support directory renaming in this specific context.
3. **Sealing and crypto file management**
   A key requirement of GP is to ensure that HLF nodes' cryptographic material and Blockchain data remain inaccessible outside enclaves. While Gramine provides robust sealing mechanisms, sealed files are subject to path integrity checks [60], meaning that files cannot be accessed if their path changes after creation. This restriction prevents the use of Docker-mounted volumes, as file paths are altered by Docker's internal filesystem management. To address this, cryptographic material must either be provisioned securely or generated within the same node's container as the enclave which sealed them.
   However, the Peer CLI requires access to these files in plaintext for operations such as Channel management and Chaincode lifecycle commands. To resolve this, a new component—referred to as the "Operator" or "Peer Operator" (PO) when associated to a Peer—was introduced. This component:
   - Generates cryptographic material within an enclave.
   - Exposes an HTTPS interface to receive external commands and forward them to the Peer CLI (bound to the Peer node).

- Executes within the same enclave[6] as the Peer node.

A similar approach is adopted for Orderers, Chaincodes, and the Platform's REST server, although in these cases the Operator is only responsible for cryptographic material generation.

4. **Peer as a library for performance optimization**
   The HLF Peer is originally provided only as a CLI executable and not as a reusable Go library. Consequently, each request would require spawning a new enclave process via Gramine, introducing significant performance overhead (as better described in following sections). To mitigate this, the Peer source code was modified in a way that it can be used both as a CLI executable and exposes its functionality as a Go library. In contrast, this modification was not necessary for the Fabric CA, which already provides a library interface.
5. **Genesis block generation**
   Orderer initialization requires a genesis block, which in turn depends on Orderer certificates. After generating cryptographic material via the Operator, certificates are copied into the environment where modified HLF scripts generate the genesis block. Since the genesis block contains only public information, such as those related to Organizations joining, it is not sealed. Once generated, it is provided to the Orderer for startup.
6. **Chaincode execution via CCAAS**
   In standard HLF deployments, Peers dynamically build and run Chaincode containers through Docker. However, these containers are not executed in a confidential environment and require access to sealed cryptographic material. To address this, every Chaincode is deployed using the Chaincode-as-an-external-Service (CCAAS) model, with graminized container images. Each image includes an Operator for generating sealed TLS credentials (for communication with its Peer) and the Chaincode logic itself. Since CCAAS supports only Go and JavaScript, the original Chaincode (implemented in Java) was migrated to Go.
7. **Fork emulation and process management**
   During CCAAS deployment, auxiliary scripts are executed as child processes of the Peer. In standard systems, this relies on the fork mechanism with copy-on-write semantics. However, SGX does not support memory sharing or copy-on-write. Gramine emulates this behavior using a checkpoint-and-restore mechanism, where the parent process state is serialized into an encrypted blob and restored in the child process [61]. This approach introduces substantial performance overhead (up to three orders of magnitude slower than native fork, as reported in [61]). Nevertheless, due to the complexity of HLF Peer, the checkpoint-and-restore mechanism fails in PAL handle serialization, because in that blob there are objects like gRPC sockets with sealed TLS files that are strictly related to the parent and Gramine refuses to serialize them for

[6] Notice that in this thesis I state "two processes run within the same enclave" for better understanding, since two processes, even in Gramine, run each within its own enclave. I mean that they run under the same Gramine environment configuration, which differs from running them in separate ones.

security reasons. To address these issues, the Peer source code was further modified to incorporate deployment scripts as Go libraries, eliminating the need to spawn child processes.

8. **REST application adaptation**
   The REST application was also graminized and adapted to include the Operator component, which seals the identity used to connect to the HLF Channel. Additionally, for compatibility with the modified HLF components, the application was migrated from Java to Go.
9. **CA simplification**
   For simplicity, the architecture uses a single CA for the Orderer Organization and one shared CA for all other Organizations, instead of maintaining one CA per Organization.
10. **RA-REST component for external interaction**
    A new component, referred to as "RA-REST", was introduced to enable secure data sharing with TPS outside the HLF network (as illustrated in Fig. 1). This component is implemented in Go and leverages a library written by me and built on top of Gramine's RA-TLS support. Since RA-TLS integration requires C code, interoperability with Go is achieved through the cgo tool [62]. Upon establishing communication between RA-REST and TPS, RA-REST verifies not only the RA-TLS certificate provided by TPS but also its enclave identity attributes, including MRENCLAVE, MRSIGNER, ISVPRODID, and ISVSVN. This enables trust to be established either at the code level (MRENCLAVE) or at the enclave author level (MRSIGNER). Further details are provided in the following sections.

On the HLF side, approximately 300 lines of code were changed across approximately 17 files in the Peer source code, spanning three categories of modification: (i) refactoring of the Peer binary to be used as both a standalone CLI executable and a Go library (PeerLib), enabling the PO to invoke Peer operations in-process; (ii) refactoring of CCAAS deployment scripts and corresponding Peer logic to execute as Go library calls, avoiding fork-based child processes that are incompatible with Gramine's checkpoint-and-restore mechanism; and (iii) removal of the unconditional loopback-address detection step that relies on the `AF_UNSPEC` address family, which is not supported by Gramine.

3.2.3. Graminized Peer

Given the aforementioned assumptions, the graminization of the Peer follows the standard GSC workflow, with the exception of the base Gramine image. Instead of using the default image from the official repositories, a patched version by me was employed to ensure compatibility with HLF according to the items in previous section.

To be graminized, the Peer requires a Gramine manifest. It is reported below:

```
1. loader.log_level = "debug"
2.
3. sgx.enclave_size = "4G"
4. sgx.max_threads = 256
5.
6. sys.enable_extra_runtime_domain_names_conf = true
```

```
sys.experimental__enable_flock = true

loader.env.FABRIC_CFG_PATH = { passthrough = true }
loader.env.FABRIC_LOGGING_SPEC = { passthrough = true }
loader.env.CORE_PEER_TLS_ENABLED = { passthrough = true }
loader.env.CORE_PEER_PROFILE_ENABLED = { passthrough = true }
loader.env.CORE_PEER_TLS_CERT_FILE = { passthrough = true }
loader.env.CORE_PEER_TLS_KEY_FILE = { passthrough = true }
loader.env.CORE_PEER_TLS_ROOTCERT_FILE = { passthrough = true }
loader.env.CORE_PEER_ID = { passthrough = true }
loader.env.CORE_PEER_ADDRESS = { passthrough = true }
loader.env.CORE_PEER_LISTENADDRESS = { passthrough = true }
loader.env.CORE_PEER_CHAINCODEADDRESS = { passthrough = true }
loader.env.CORE_PEER_CHAINCODELISTENADDRESS = { passthrough = true }
loader.env.CORE_PEER_GOSSIP_BOOTSTRAP = { passthrough = true }
loader.env.CORE_PEER_GOSSIP_EXTERNALENDPOINT = { passthrough = true }
loader.env.CORE_PEER_LOCALMSPID = { passthrough = true }
loader.env.CORE_PEER_MSPCONFIGPATH = { passthrough = true }
loader.env.CORE_OPERATIONS_LISTENADDRESS = { passthrough = true }
loader.env.CORE_METRICS_PROVIDER = { passthrough = true }
loader.env.CHAINCODE_AS_A_SERVICE_BUILDER_CONFIG = { passthrough = true }
loader.env.CORE_CHAINCODE_EXECUTETIMEOUT = { passthrough = true }

loader.env.CORE_LEDGER_STATE_STATEDATABASE = {passthrough = true}
loader.env.CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS = {passthrough = true}
loader.env.CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME = {passthrough = true}
loader.env.CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD = {passthrough = true}

loader.env.CORE_PEER_KEEPALIVE_TIMEOUT = { passthrough = true }
loader.env.CORE_PEER_KEEPALIVE_CLIENT_TIMEOUT = { passthrough = true }

loader.env.CORE_PEER_GOSSIP_DIALTIMEOUT = { passthrough = true }
loader.env.CORE_PEER_GOSSIP_CONNTIMEOUT = { passthrough = true }
loader.env.CORE_PEER_GOSSIP_ALIVETIMEINTERVAL = { passthrough = true }
loader.env.CORE_PEER_GOSSIP_ALIVEEXPIRATIONTIMEOUT = { passthrough = true }

loader.env.CA_URL = { passthrough = true }
loader.env.PEER_OPERATOR_PORT = { passthrough = true }

loader.env.CORE_VM_ENDPOINT = { passthrough = true }
loader.env.CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE = { passthrough = true }


sgx.trusted_files = [
  "file:/gramine/app_files/entrypoint.manifest",
  "file:/etc/hyperledger/",
  "file:/var/",
]

sgx.allowed_files = [
  "file:/var/hyperledger/production/",
  "file:/etc/hyperledger/",
  "file:/tmp/",
]

fs.mounts = [
  { path = "/etc/hyperledger/", uri = "file:/etc/hyperledger/" },


  { type = "encrypted", path = "/etc/hyperledger/fabric/", uri = "file:/etc/hyperledger/fabric/",
key_name = "_sgx_mrsigner" },
  { type = "encrypted", path = "/var/hyperledger/production/", uri =
"file:/var/hyperledger/production/", key_name = "_sgx_mrsigner"},

  { path = "/tmp/", uri = "file:/tmp/" }
]
```

This means that:

- The logging level is set to debug,

- The enclave is configured with a maximum memory size of 4 GB and up to 256 threads. These parameters are necessary, as the default values are insufficient to support Peer execution.
- The option `enable_extra_runtime_domain_names_conf = true` ensures that the enclave uses the host's domain name resolution configuration.
- The option `sys.experimental__enable_flock = true` enables support for the `flock` system call.
- The `loader.env.*` entries that environment variables defined on the host are propagated into the enclave.
- The `sgx.trusted_files` entries specify files and directories (and their subdirectories) that are subject to runtime cryptographic integrity checks, including `"file:/gramine/app_files/entrypoint.manifest"`, `"file:/etc/hyperledger/"`, `"file:/var/"`.
- The `sgx.allowed_files` entries define directories (`"file:/var/hyperledger/production/"`, `"file:/etc/hyperledger/"` and `"file:/tmp/"`) that are not subject to integrity checks and can be modified by the enclave. Combined with `sgx.trusted_files`, this allows verification of existing files at startup while still permitting subsequent modifications to those directories (like `"file:/etc/hyperledger/"`).
- Filesystem mappings are defined through `fs.mounts`, enabling host directories to be mounted within the enclave. In particular:
  - `/etc/hyperledger/` and `/tmp/` are mapped directly, allowing visibility of changes through `sgx.allowed_files`.
  - `/etc/hyperledger/fabric/` and `/var/hyperledger/production/` (and their subdirectories) are sealed using the enclave's MRSIGNER key (`key_name = "_sgx_mrsigner"`). Although MRENCLAVE could be used (as Peer code is expected to remain stable), MRSIGNER was preferred to facilitate development, allowing code updates without invalidating sealed data.

  This configuration enables the Peer to operate normally while ensuring that the Blockchain data and associated cryptographic material remain sealed and inaccessible outside the enclave.

When using GSC, it is not necessary to explicitly define the entrypoint in the manifest, as GSC automatically executes the container's original entrypoint within the enclave. The Peer container entrypoint is implemented as a Bash script, as reported below:

```
1. #!/bin/bash
2.
3. ./peerOperator/peerRegisterEnroll
4.
5. ./peerOperator/peerOperator &
6.
7. peer node start
8.
```

It launches three processes within the same enclave:

1. Registration and enrollment of the identities used by the Peer through PO.
2. Execution of the PO in the background.
3. Startup of the Peer node.

To generate a graminized Docker image, a modified Peer image (`fabric-peer`) is first built, including the modifications described in the previous section. The GSC tool is then invoked with the following commands:

```
1. gsc build -c config.yaml fabric-peer $PATH_TO_PEER_MANIFEST
2. gsc sign-image fabric-peer $PATH_TO_GRAMINE_ENCLAVE_KEY
```

The first one graminizes the `fabric-peer` image with the GSC configuration (`-c config.yaml`) and the Peer's Gramine manifest (`$PATH_TO_PEER_MANIFEST`). The second one signs with the enclave key (`$PATH_TO_GRAMINE_ENCLAVE_KEY`) the newly generated intermediate image. If successful, it produces the graminized Peer image `gsc-fabric-peer`.

The resulting container image can be deployed using Docker Compose, like:

```
 1.   peer0.portauthority.portauthoritydomain.com:
 2.     container_name: peer0.portauthority.portauthoritydomain.com
 3.     image: gsc-fabric-peer
 4.     labels:
 5.       service: hyperledger-fabric
 6.     environment:
 7.       - FABRIC_CFG_PATH=/etc/hyperledger/fabric/
 8.       - FABRIC_LOGGING_SPEC=INFO
 9.       - CORE_PEER_TLS_ENABLED=true
10.       - CORE_PEER_PROFILE_ENABLED=true
11.       - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
12.       - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
13.       - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
14.       - CORE_PEER_ID=peer0.portauthority.portauthoritydomain.com
15.       - CORE_PEER_ADDRESS=peer0.portauthority.portauthoritydomain.com:7051
16.       - CORE_PEER_LISTENADDRESS=0.0.0.0:7051
17.       - CORE_PEER_CHAINCODEADDRESS=peer0.portauthority.portauthoritydomain.com:7052
18.       - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
19.       - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.portauthority.portauthoritydomain.com:7051
20.       - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.portauthority.portauthoritydomain.com:7051
21.       - CORE_PEER_GOSSIP_USELEADERELECTION=true
22.       - CORE_PEER_GOSSIP_ORGLEADER=false
23.       - CORE_PEER_GOSSIP_ELECTION_LEADERALIVETHRESHOLD=10s
24.       - CORE_PEER_LOCALMSPID=PortAuthorityMSP
25.       - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/fabric/msp
26.       - CORE_OPERATIONS_LISTENADDRESS=peer0.portauthority.portauthoritydomain.com:9444
27.       - CORE_METRICS_PROVIDER=prometheus
28.       - CHAINCODE_AS_A_SERVICE_BUILDER_CONFIG={"peername":"peer0_portauthority"}
29.       - CORE_CHAINCODE_EXECUTETIMEOUT=300s
30.       - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
31.       - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb0:5984
32.       - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=admin
33.       - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=adminpw
34.       - CORE_PEER_KEEPALIVE_TIMEOUT=120s
35.       - CORE_PEER_KEEPALIVE_MININTERVAL=1200s
36.       - CORE_PEER_KEEPALIVE_CLIENT_TIMEOUT=120s
37.       - CORE_PEER_KEEPALIVE_CLIENT_MININTERVAL=120s
38.       - CORE_PEER_KEEPALIVE_DELIVERYCLIENT_INTERVAL=1200s
39.       - CORE_PEER_KEEPALIVE_DELIVERYCLIENT_TIMEOUT=120s
40.       - CORE_PEER_GOSSIP_DIALTIMEOUT=90s
41.       - CORE_PEER_GOSSIP_CONNTIMEOUT=80s
42.       - CORE_PEER_GOSSIP_ALIVETIMEINTERVAL=150s
43.       - CORE_PEER_GOSSIP_ALIVEEXPIRATIONTIMEOUT=350s
44.
45.       - CA_URL=ca-portauthority:7054
46.       - PEER_OPERATOR_PORT=8443
```

```
    privileged: true
    volumes:
        - ../organizations/fabric-ca/portauthority/ca-cert.pem:/tmp/ca.crt
        - peer0.portauthority.portauthoritydomain.com:/var/hyperledger/production
        - /dev/sgx_enclave:/dev/sgx_enclave
        - /var/run/aesmd/aesm.socket:/var/run/aesmd/aesm.socket
    working_dir: /
    ports:
      - 7051:7051
      - 8443:8443
      - 9444:9444
    networks:
      - portchain

```

The configuration includes:

- Use of the `gsc-fabric-peer` image,
- Execution in privileged mode (`privileged: true`) to enable access to SGX device (`/dev/sgx_enclave`) and socket (`/var/run/aesmd/aesm.socket`).
- Volumes, environment variables and the remaining configuration are referred to Peer and PO.

Additional environment variables have been introduced for each Peer and include:

- `CA_URL`: address of the CA used for registration and enrollment.
- `PEER_OPERATOR_PORT`: port on which the PO exposes its HTTPS interface.

For additional Peer Organizations, only Peer- and PO-specific configuration parameters need to be modified, as the same image, SGX device, and socket can be reused.

3.2.4. Graminized Orderer

The graminization process for the Orderer closely mirrors that of the Peer. It involves defining a Gramine manifest, adapting the container entrypoint, and applying the GSC workflow to produce a graminized image.

This is the manifest:

```
loader.log_level = "debug"

sgx.enclave_size = "4G"
sgx.max_threads = 256

sys.enable_extra_runtime_domain_names_conf = true
sys.experimental__enable_flock = true

loader.env.FABRIC_LOGGING_SPEC = { passthrough = true }
loader.env.FABRIC_CFG_PATH = { passthrough = true }
loader.env.ORDERER_GENERAL_LISTENADDRESS = { passthrough = true }
loader.env.ORDERER_GENERAL_LISTENPORT = { passthrough = true }
loader.env.ORDERER_GENERAL_LOCALMSPID = { passthrough = true }
loader.env.ORDERER_GENERAL_LOCALMSPDIR = { passthrough = true }
loader.env.ORDERER_GENERAL_GENESISMETHOD = { passthrough = true }
loader.env.ORDERER_GENERAL_GENESISFILE = { passthrough = true }
loader.env.ORDERER_GENERAL_TLS_ENABLED = { passthrough = true }
loader.env.ORDERER_GENERAL_TLS_PRIVATEKEY = { passthrough = true }
loader.env.ORDERER_GENERAL_TLS_CERTIFICATE = { passthrough = true }
loader.env.ORDERER_GENERAL_TLS_ROOTCAS = { passthrough = true }
loader.env.ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE = { passthrough = true }
loader.env.ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY = { passthrough = true }
```

```
loader.env.ORDERER_GENERAL_CLUSTER_ROOTCAS = { passthrough = true }
loader.env.ORDERER_GENERAL_BOOTSTRAPMETHOD = { passthrough = true }
loader.env.ORDERER_CHANNELPARTICIPATION_ENABLED = { passthrough = true }
loader.env.ORDERER_ADMIN_TLS_ENABLED = { passthrough = true }
loader.env.ORDERER_ADMIN_TLS_CERTIFICATE = { passthrough = true }
loader.env.ORDERER_ADMIN_TLS_PRIVATEKEY = { passthrough = true }
loader.env.ORDERER_ADMIN_TLS_ROOTCAS = { passthrough = true }
loader.env.ORDERER_ADMIN_TLS_CLIENTROOTCAS = { passthrough = true }
loader.env.ORDERER_ADMIN_LISTENADDRESS = { passthrough = true }
loader.env.ORDERER_OPERATIONS_LISTENADDRESS = { passthrough = true }
loader.env.ORDERER_METRICS_PROVIDER = { passthrough = true }

loader.env.CA_URL = { passthrough = true }
loader.env.ORDERER_ID = { passthrough = true }

sgx.trusted_files = [
  "file:/gramine/app_files/entrypoint.manifest",
  "file:/var/",
  "file:/tmp/",
]

sgx.allowed_files = [
  "file:/var/hyperledger/production/",
  "file:/var/hyperledger/orderer/",
  "file:/tmp/",
]

fs.mounts = [
  { path = "/var/hyperledger/production/", uri = "file:/var/hyperledger/production/"},
  { path = "/var/hyperledger/orderer/", uri = "file:/var/hyperledger/orderer/"},
  { type = "encrypted", path = "/var/hyperledger/orderer/msp", uri =
"file:/var/hyperledger/orderer/msp", key_name = "_sgx_mrsigner" },
  { type = "encrypted", path = "/var/hyperledger/orderer/tls", uri =
"file:/var/hyperledger/orderer/tls", key_name = "_sgx_mrsigner" },
  { type = "encrypted", path = "/var/hyperledger/production/orderer/chains/", uri =
"file:/var/hyperledger/production/orderer/chains/", key_name = "_sgx_mrsigner"},
  { path = "/tmp/", uri = "file:/tmp/" },
]

```

The same configuration principles apply for the log level, enclave size, maximum number of threads, `enable_extra_runtime_domain_names_conf`, `flock` system call, environment variables, trusted and allowed files and mounts (referred to the Orderer). However, sealing is applied selectively. Only directories containing cryptographic material and Blockchain data are sealed using MRSIGNER. Specifically, the following directories (and their subdirectories) are sealed:

- `/var/hyperledger/orderer/msp`,
- `/var/hyperledger/orderer/tls`,
- `/var/hyperledger/production/orderer/chains/`.

This selective sealing is necessary because the orderer creates and modifies temporary directories under `/var/hyperledger/orderer/`. As noted in [59], directory renaming within encrypted paths is not fully supported, which would otherwise result in runtime errors.

The Orderer entrypoint is implemented as a Bash script:

```
#!/bin/bash
/operator/ordererRegisterEnroll
orderer
```

Which performs identity registration and enrollment through the Operator before starting the Orderer node.

The GSC workflow is identical to that used for the Peer, but with a different image name: `fabric-orderer` and the resulting.

`gsc-fabric-orderer` can be deployed using Docker Compose, like:

```
  orderer0.ordererdomain.com:
    container_name: orderer0.ordererdomain.com
    image: gsc-fabric-orderer
    labels:
      service: hyperledger-fabric
    environment:
      - FABRIC_CFG_PATH=/var/hyperledger/production/
      - FABRIC_LOGGING_SPEC=INFO
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_LISTENPORT=7050
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
      - ORDERER_GENERAL_TLS_ENABLED=true
      - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
      - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
      - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
      - ORDERER_KAFKA_TOPIC_REPLICATIONFACTOR=1
      - ORDERER_KAFKA_VERBOSE=true
      - ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE=/var/hyperledger/orderer/tls/server.crt
      - ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY=/var/hyperledger/orderer/tls/server.key
      - ORDERER_GENERAL_CLUSTER_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
      - ORDERER_ADMIN_TLS_ENABLED=true
      - ORDERER_ADMIN_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
      - ORDERER_ADMIN_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
      - ORDERER_ADMIN_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
      - ORDERER_ADMIN_TLS_CLIENTROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
      - ORDERER_ADMIN_LISTENADDRESS=0.0.0.0:7053

      - CA_URL=ca-ordererdomain-com:8054
      - ORDERER_ID=orderer0.ordererdomain.com
    privileged: true
    volumes:
        - ../organizations/fabric-ca/ordererdomain/ca-cert.pem:/tmp/ca.crt
        - orderer0.ordererdomain.com:/var/hyperledger/production/orderer
        - /dev/sgx_enclave:/dev/sgx_enclave
        - /var/run/aesmd/aesm.socket:/var/run/aesmd/aesm.socket
    ports:
      - 7050:7050
      - 7053:7053
    networks:
      - portchain

```

Deployment via Docker Compose follows the same principles for the Peer, except that no PO configuration is required. An additional environment variable, `ORDERER_ID`, is introduced to define the Common Name used in TLS certificates and the Orderer's identity.

3.2.5. Graminized Chaincode

As previously discussed, the Chaincode is deployed using the CCAAS model and has been migrated from Java to Go.

The graminization process again follows the GSC workflow, with a dedicated Gramine manifest reported here:

```
loader.log_level = "debug"

sgx.enclave_size = "4G"
sgx.max_threads = 256

sys.enable_extra_runtime_domain_names_conf = true
sys.experimental__enable_flock = true

loader.env.CHAINCODE_SERVER_ADDRESS = { passthrough = true }
loader.env.CHAINCODE_ID = { passthrough = true }
loader.env.CHAINCODE_NAME = { passthrough = true }
loader.env.CORE_CHAINCODE_ID_NAME = { passthrough = true }

loader.env.CHAINCODE_TLS_DISABLED = { passthrough = true }
loader.env.CHAINCODE_TLS_KEY = { passthrough = true }
loader.env.CHAINCODE_TLS_CERT = { passthrough = true }
loader.env.CHAINCODE_CLIENT_CA_CERT = { passthrough = true }

loader.env.CORE_PEER_LOCALMSPID = { passthrough = true }
loader.env.CA_URL = { passthrough = true }

sgx.trusted_files = [
  "file:/gramine/app_files/entrypoint.manifest",
  "file:/home/chaincode/",
  "file:/var/",
]

sgx.allowed_files = [

  "file:/home/chaincode/",
  "file:/tmp/",
]

fs.mounts = [
  { path = "/home/chaincode/ca.crt", uri = "file:/home/chaincode/ca.crt"},
  { type = "encrypted", path = "/home/chaincode/crypto/", uri = "file:/home/chaincode/crypto/",
key_name = "_sgx_mrsigner"},
  { path = "/tmp/", uri = "file:/tmp/" }
]


```

Its entrypoint is the following Bash script:

```
#!/bin/bash
/operator/ccRegisterEnroll
./chaincode
```

It invokes the Operator, which generates TLS cryptographic material required for secure communication with its Peer, and then starts the Chaincode.

The `gsc-${CC_NAME}_ccaas_image` Docker image can be executed through:

```
docker run -d --name peer0_portauthority_${CC_NAME}_ccaas  \
  --network network_portchain \
  --privileged \
  -v /var/run/aesmd/aesm.socket:/var/run/aesmd/aesm.socket -v /dev/sgx_enclave:/dev/sgx_enclave \
  -v ./organizations/fabric-ca/portauthority/ca-cert.pem:/home/chaincode/ca.crt \
  -e CHAINCODE_SERVER_ADDRESS=0.0.0.0:${CCAAS_SERVER_PORT} \
  -e CHAINCODE_ID=$PACKAGE_ID -e CORE_CHAINCODE_ID_NAME=$PACKAGE_ID \
  -e CHAINCODE_NAME=$CC_NAME -e CORE_PEER_LOCALMSPID=PortAuthorityMSP \
  -e CHAINCODE_TLS_DISABLED=false -e CHAINCODE_TLS_KEY=/home/chaincode/crypto/server.key \
  -e CHAINCODE_TLS_CERT=/home/chaincode/crypto/server.crt \
  -e CHAINCODE_CLIENT_CA_CERT=/home/chaincode/crypto/ca.crt \
  -e CA_URL=ca-portauthority:7054 \
    gsc-${CC_NAME}_ccaas_image:latest
```

Where:

- `${CC_NAME}` is the Chaincode name,
- Execution in privileged mode (`--privileged`) to enable access to SGX device and socket.
- Some Chaincode-related environment variables have been defined, including:
  - Again, the `CA_URL` used for TLS material enrollment.
  - `${CCAAS_SERVER_PORT}` which is the port on which the CCAAS server listens,
  - `$PACKAGE_ID` which is the package ID of the Chaincode (obtained during Chaincode deployment workflow),

3.2.6. Graminized Application

To complete GP, the (already existing in the Platform) REST application was also adapted for execution within SGX enclaves. Originally implemented in Java, it was migrated to Go to ensure compatibility with the Chaincode data model.

The application follows the same GSC-based graminization process as other components, with a similar manifest configuration, as reported below:

```
loader.log_level = "warning"

sgx.enclave_size = "4G"
sgx.max_threads = 256

sys.enable_extra_runtime_domain_names_conf = true
sys.experimental__enable_flock = true

loader.env.FABRIC_CFG_PATH = { passthrough = true }
loader.env.FABRIC_LOGGING_SPEC = { passthrough = true }
loader.env.CORE_PEER_LOCALMSPID = { passthrough = true }
loader.env.CA_URL = { passthrough = true }
loader.env.USERNAME = { passthrough = true }

sgx.trusted_files = [
  "file:/gramine/app_files/entrypoint.manifest",
  "file:/var/",
  "file:/home/rest/",
  "file:/home/application/",
]

sgx.allowed_files = [
  "file:/home/rest/",
  "file:/home/application/",
  "file:/tmp/",
]

fs.mounts = [
  { path = "/home/rest/", uri = "file:/home/rest/" },
  { path = "/home/application/", uri = "file:/home/application/" },
  { path = /home/rest/application.properties", uri =
"file:/home/application/application.properties" },
  { type = "encrypted", path = "/home/rest/users/", uri = "file:/home/application/users/", key_name
= "_sgx_mrsigner" },

  { path = "/tmp/", uri = "file:/tmp/" }
]

```

It is similar to the previous graminized components.

The entrypoint is the following Bash script:

```
#!/bin/bash
/operator/appRegisterEnroll
/home/rest/rest-api
```

It first performs user registration and enrollment (based on the `USERNAME` environment variable, reported below) and then launches the REST server.

The resulting graminized Docker image `gsc-fabric-rest` can be deployed using Docker Compose, like:

```
  rest-${ORG_NAME}-${ORG_DOMAIN}:
    container_name: rest-${ORG_NAME}-${ORG_DOMAIN}
    image: gsc-fabric-rest
    environment:
      - USERNAME=${ORG_NAME}
      - CORE_PEER_LOCALMSPID=${ORG_NAME}MSP
      - CA_URL=${CA_URL}
    privileged: true
    volumes:
      - ${APPLICATION_PATH}:/home/application/
      - ${APPLICATION_PATH}/application.properties:/home/rest/application.properties
      - /dev/sgx_enclave:/dev/sgx_enclave
      - /var/run/aesmd/aesm.socket:/var/run/aesmd/aesm.socket
    ports:
      - ${REST_PORT}:${REST_PORT}
    networks:
      - portchain
```

It follows the same configuration principles as other components. To connect to the HLF Channel, the application requires:

- `CORE_PEER_LOCALMSPID`: the Organization's MSP identifier,
- `USERNAME`: the identity used for interaction with the Channel,
- `APPLICATION_PATH` the directory containing the REST application and its configuration files,
- `REST_PORT` the port on which the REST service listens.

3.2.7. Operator

As previously introduced, the **Operator** is a newly designed component that executes within the same enclave as the Peer, Orderer, Chaincode, and REST application. For the latter three components, the Operator is responsible solely for generating cryptographic material. In contrast, when co-located with the Peer (the PO), it additionally exposes an HTTPS server that accepts and processes Peer-related commands.

Fig. 18 illustrates a simplified sequence diagram for a Peer Channel join request. The process can be summarized as follows:

1. The administrator issues a GET request to the PO endpoint `/channelJoin`, providing the `blockfile` query parameter, which specifies the path to the genesis block file. This

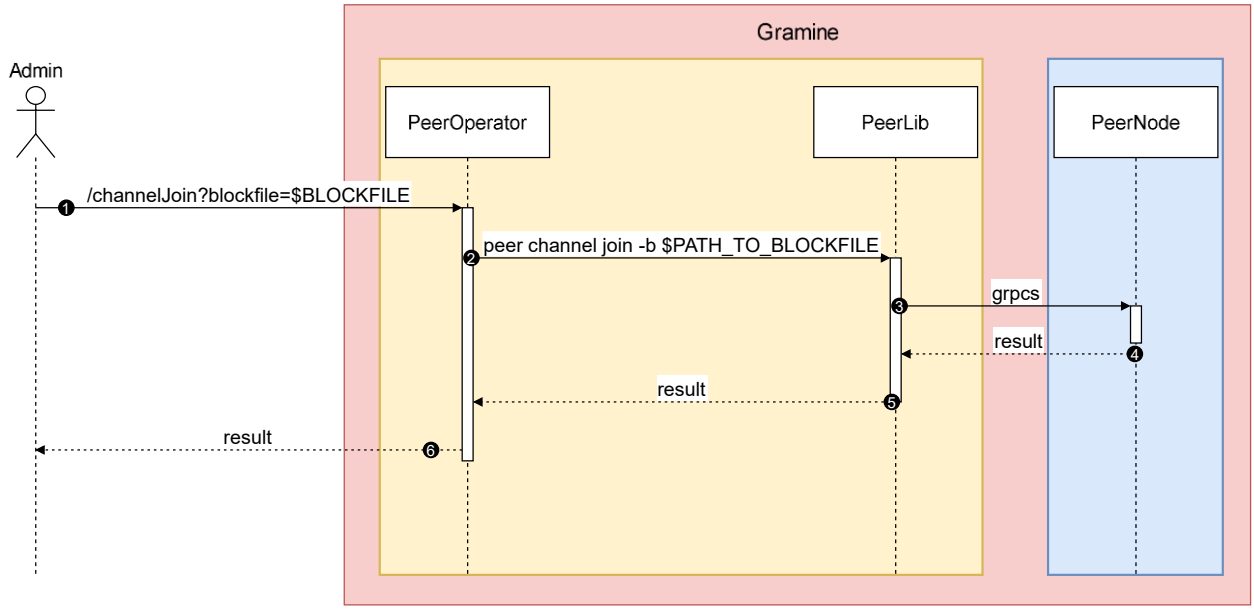

Fig. 18 Peer Operator channel join example sequence diagram

   file is already uploaded within the PO container. Since it contains only public information, it can be transferred in plaintext. In the current implementation, it is copied from the host into the container using Docker CLI.
2. The PO translates the request into the corresponding Peer CLI command and invokes the modified Peer CLI as library (denoted as PeerLib in the figure). In this case, the command is:

   ```
   peer channel join -b $PATH_TO_BLOCKFILE
   ```

   allowing PeerLib to access the specified block file.
3. Internally, PeerLib communicates with the Peer node (denoted as PeerNode in the figure) via gRPC, forwarding the request. The Peer node performs the Channel join procedure and returns the result to the PO.

Overall, the HTTPS server exposed by the PO supports a set of endpoints that map directly to Peer CLI commands, ensuring consistency with standard HLF operations while enabling programmatic access via HTTPS. These endpoints enable remote management of Peer operations while maintaining compatibility with enclave-based execution.

A summary of the supported endpoints is provided below. Notice that some blocks, transactions or results are left in plaintext, since they contain public information:

URL	Method	Parameters	Description

/channelFetchConfig	GET	`ordererAddress`: address of the Orderer `channel`: the Channel name	Retrieves the latest configuration block for a given Channel (`channel`) from the specified Orderer (`ordererAddress`) using TLS configuration (`TLS_CONF` (see the Peer CLI syntax)). The result is stored in an the `OUTPUT` file, which must be exported outside the container.
	Respective Peer CLI syntax		
	`peer channel fetch config $OUTPUT -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress -c $channel $TLS_CONF`		
/channelFetch0	GET	`ordererAddress`: address of the Orderer. `channel`: the Channel name. `blockfile`: the output file.	Retrieves the genesis block for a given Channel (`channel`) from the Orderer (`ordererAddress`) using TLS configuration (`TLS_CONF`) and stores it in the specified `blockfile`, which must be copied outside the Peer's container.
	Respective Peer CLI syntax		
	`peer channel fetch 0 $blockfile -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress -c $channel $TLS_CONF`		
/channelJoin	GET	`blockfile`: the path to the genesis block.	Joins the Peer to the `blockfile`'s Channel. The remaining blocks are subsequently retrieved via the gossip protocol.
	Respective Peer CLI syntax		
	`peer channel join -b $blockfile`		
/channelSignConfigTx	GET	`configtxfile`: path to the file containing a transaction to be signed.	Signs the configuration transaction file (`configtxfile`) using the Peer administrator's identity.
	Respective Peer CLI syntax		
	`peer channel signconfigtx -f $configtxfile`		
/channelUpdate	GET	`ordererAddress`: address of the Orderer. `channel`: the Channel name. `envelope`: the configuration transaction file path.	Signs and submits a configuration transaction (located at `envelope`) referred to the Channel `channel` to the specified Orderer (`ordererAddress`) using TLS configuration (`TLS_CONF`).
	Respective Peer CLI syntax		

	`peer channel update -f $envelope -c $channel -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress $TLS_CONF`		
/channelCreate	GET	`ordererAddress`: address of the Orderer. `channel`: the Channel name. `blockfile`: generated genesis file.	Creates a new Channel using a configuration transaction file `channel.tx` communicating with the specified Orderer (`ordererAddress`) and using TLS configuration (`TLS_CONF`). It also generates the corresponding genesis block located at `blockfile`.
	Respective Peer CLI syntax		
	`peer channel create -outputBlock $blockfile -f $channel.tx -c $channel -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress $TLS_CONF`		
/ccQueryInstalled	GET	No parameters	Retrieves the list of installed Chaincodes in JSON format and stores the result in `/tmp/output`. `CORE_PEER_ADDRESS` and `CORE_PEER_TLS_ROOTCERT_FILE` come from the Peer's environment
	Respective Peer CLI syntax		
	`peer lifecycle chaincode queryinstalled --output json --peerAddress $CORE_PEER_ADDRESS --tlsRootCertFiles $CORE_PEER_TLS_ROOTCERT_FILE`		
/ccInstall	GET	`ccName`: name of the Chaincode being installed. `ccVersion`: version of the Chaincode being installed.	Packages and installs a Chaincode (identified by `ccName` and `ccVersion`) using the CCAAS deployment mode.
	Respective Peer CLI syntax		

	`# steps to package Chaincode in CCAAS, storing`
	`# the package at`
	`# etc/hyperledger/fabric/cc/$ccName.tar.gz`
	`# then`
	
	`peer lifecycle chaincode calculatedpackageid etc/hyperledger/fabric/cc/$ccName.tar.gz > /tmp/output`
	
	`peer lifecycle chaincode install /etc/hyperledger/fabric/cc/$ccName.tar.gz --peerAddress $CORE_PEER_ADDRESS --tlsRootCertFiles $CORE_PEER_TLS_ROOTCERT_FILE`		

/ccApproveForMyOrg	POST	`ordererAddress`: address of the Orderer.	Approves the Chaincode `ccName` definition in the Channel `channelName` for the Peer's

<table>
<tr><td></td><td></td><td>channelName: the Channel name.
ccName: the Chaincode name.
ccVersion: the Chaincode version.
pkgId: the Chaincode's package ID.
sequence: the Chaincode sequence.
initRequired: a flag for stating whether the Chaincode must be initialized before being used.
ccEndPolicy: the Chaincode's endorsement policy.
ccCollConfig: path to the JSON PC configuration.</td><td>Organization, including parameters such as version (ccVersion), package ID (pkgId), sequence number (sequence), endorsement policy (ccEndPolicy), and PC configuration (located at ccCollConfig).
It is done by communicating with the specified Orderer (ordererAddress) and using TLS configuration (TLS_CONF).
Notice that if initRequired is not set, the respective flag is omitted in the CLI command.</td></tr>
<tr><td></td><td colspan="3">Respective Peer CLI syntax
<pre>peer lifecycle chaincode approveformyorg -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress $TLS_CONF --channelID $channelName -peerAddress $CORE_PEER_ADDRESS --tlsRootCertFiles $CORE_PEER_TLS_ROOTCERT_FILE -name $ccName --version $ccVersion --package-id $pkgId --sequence $sequence --init-required --signature-policy $ccEndPolicy --collections-config $ccCollConfig</pre></td></tr>
<tr><td>/ccCommit</td><td>POST</td><td>The same body as /ccApproveForMyOrg.
peerEndpoints: all Peer endpoints that have approved the Chaincode definition.</td><td>Commits a previously approved Chaincode definition to the channel channelName. It also aggregates Peer endpoints (peerEndpoints) and their corresponding root certificates in PEER_ADDRESSES_WITH_ROOCERT.
It is done by communicating with the specified Orderer (ordererAddress) and using TLS configuration (TLS_CONF).
Notice that if initRequired is not set, the respective flag is omitted in the CLI command.</td></tr>
</table>

	Respective Peer CLI syntax `peer lifecycle chaincode commit -o $ordererAddress:7050 --ordererTLSHostnameOverride $ordererAddress $TLS_CONF --channelID $channelName --name $ccName --version $ccVersion $PEER_ADDRESSES_WITH_ROOCERT --sequence $sequence --initRequired --signature-policy $ccEndPolicy --collections-config $ccCollConfig`		
/ccQueryCommitted	GET	`channelName`: the Channel name. `ccName`: the Chaincode name.	Checks whether a Chaincode `ccName` has been committed to the Channel `channelName`. The result is stored into `/tmp/output` in plain-text format. Notice that `CORE_PEER_ADDRESS` and `CORE_PEER_TLS_ROOTCERT_FILE` come from the Peer's environment.
	Respective Peer CLI syntax `peer lifecycle chaincode querycommitted --channelID $channelName --name $ccName --output plain-text --peerAddress $CORE_PEER_ADDRESS --tlsRootCertFiles $CORE_PEER_TLS_ROOTCERT_FILE`		
/ccQueryApproved	GET	`channelName`: the Channel name. `ccName`: the Chaincode name.	Checks whether a Chaincode `ccName` has been approved by the Peer's Organization. into the Channel `channelName`. The result is stored into `/tmp/output` in plain-text format. Notice that `CORE_PEER_ADDRESS` and `CORE_PEER_TLS_ROOTCERT_FILE` come from the peer's environment.
	Respective Peer CLI syntax `peer lifecycle chaincode queryapproved --channelID $channelName --name $ccName --peerAddress $CORE_PEER_ADDRESS --tlsRootCertFiles $CORE_PEER_TLS_ROOTCERT_FILE`		
/registerEnroll	GET	No parameters	Generates cryptographic material for the Peer, if not already present.

Table 17 Peer Operator REST interface

Since the PO executes within the same enclave as the Peer, it inherits the same Gramine configuration. This ensures that all operations, including command execution and cryptographic material handling, benefit from the same confidentiality and integrity guarantees provided by the enclave.

3.2.8. RA-REST

As previously introduced, this component is implemented in Go and acts as an RA-TLS server that enforces RA for every incoming request. It leverages Gramine's Secure Channel Attestation abstraction level.

3.2.8.1. Remote Attestation library

To improve flexibility and reusability, the server functionality has been encapsulated within a dedicated Go library. Through the use of cgo, the library interfaces with Gramine's Attestation primitives, which are implemented in C/C++. The library also provides partial support for Python (client-side only). The EGo module [47] was not adopted, as EGo and Gramine environments cannot be nested.

The library is divided into two main components: **client** and **server**.

3.2.8.1.1. Client-side functionality

The client component provides the following methods:

```
func DoMutualRemoteAttestation(serverEndpoint, mrenclave, mrsigner, isvProdId, isvSvn string,
queryParameters map[string]string) (string, error)
// …
func DoRemoteAttestation(serverEndpoint string, queryParameters map[string]string) (string, error)
// …
```

- `DoMutualRemoteAttestation()` performs mutual RA with the server. In this mode, both client and server exchange and verify RA-TLS certificates. The client must provide the expected measurements for server verification. Upon successful Attestation, query parameters are sent to the server in the form `{"key": "[VALUE]"}` (this interface may be extended in future work).
- `DoRemoteAttestation()` performs one-way Attestation, where only the client proves its genuineness to the server, without verifying the server instead. Upon successful Attestation, the same parameters as the mutual RA are sent to server.

Both methods return the server response as a string or an error in case of failure. Internally, the Go implementation invokes C code to use the Gramine's Secure Channel, through which the Attestation and communication are performed. In all cases, the client transmits its Quote to the server. Currently, the Python client supports only `DoRemoteAttestation()`.

3.2.8.1.2. Server-side functionality

The server provides:

```
func DoMutualRemoteAttestation(serverEndpoint, mrenclave, mrsigner, isvProdId, isvSvn string, callback
func(string) string) bool
// ...
func DoRemoteAttestation(serverEndpoint, mrenclave, mrsigner, isvProdId, isvSvn string, callback
func(string) string) bool
// ...
```

- `DoMutualRemoteAttestation()` initializes a server listening on `serverEndpoint`. Upon receiving a connection, mutual Attestation is performed. The server validates the client's measurements and, upon success, invokes the callback function `callback func(string) string`. This function processes the URL-encoded query parameters received from the client and returns a stringified response.
- `DoRemoteAttestation()` initializes a server listening on `serverEndpoint` that performs one-way Attestation, verifying only the client's measurements without requiring server verification by the client.

As with the client, the server-side Go implementation relies on underlying C code to execute Attestation via Gramine's Secure Channel. A detailed flowchart of both client and server operations is provided in the Appendix.

It is important to note that mutual Attestation must be consistently enforced on both sides: if one party performs mutual Attestation, the other must do so as well.

3.2.8.2. RA-REST behavior

The RA-REST server operates as illustrated in Fig. 19 and follows the workflow below:

1. It loads the set of accepted measurements from environment variables,
2. It invokes `DoRemoteAttestation()`, and waits for incoming client connections,
3. Upon connection, the client presents its RA-TLS certificate. The server verifies the certificate, the embedded Quote, and the associated measurements,
4. If any verification step fails, an error is returned to the client,
5. If verification succeeds, the client is considered trusted and submits a request containing a container identifier (`cntId` in the figure),
6. The server connects to the HLF network. User credentials are generated and sealed prior to RA-REST server startup, using the same mechanism described for the graminized application (Section 3.2.6),
7. The server queries all events associated with the specified `cntId`,
8. The HLF network returns the corresponding events (or an empty array if none are found),
9. The server disconnects from the HLF Channel. It is worth noting that the repeated connection and disconnection to the HLF channel introduces negligible overhead in terms of system performance,
10. The result is returned to the client.

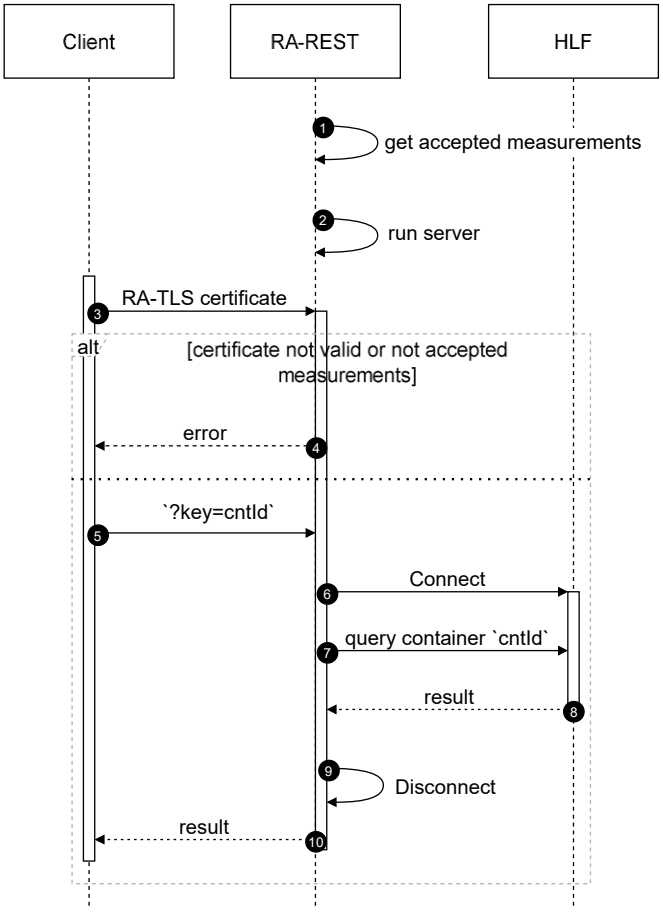

Fig. 19 RA-REST sequence diagram

Trust is established not only by verifying that the client is executing within a genuine Intel SGX enclave, but also by validating specific enclave identity attributes. In particular, the values of MRENCLAVE, MRSIGNER, ISVPRODID, and ISVSVN—provided within the client's Quote (through RA-TLS)—must match those defined in the server's configuration.

Each HLF Organization deploys its own RA-REST server within a Docker container using GSC. A specialized container image is required, as the Intel SGX Quote generation library is not included in standard (or patched) Gramine images and must be explicitly added.

3.2.8.3. Client examples

Two RA-REST client implementations were developed in Python: one trusted and one untrusted. Both execute outside Docker containers as standard processes, but within Gramine enclaves.

The **trusted client** is associated with a specific MRENCLAVE value recognized by the RA-REST server. Upon successful Attestation, it retrieves events corresponding to a given `cntId` and processes them using a pre-trained Qwen3-0.6B model [63], via the Transformers[7] library. Since execution occurs within an SGX enclave:

1. The RA-TLS certificate's private key,
2. The retrieved event data, and
3. The model's computation results

remain protected in encrypted memory, preventing data leakage even at the hardware level. This approach introduces significant performance overhead (approximately 10–15x slower), primarily due to the absence of GPU acceleration. However, this setup serves as a proof of concept demonstrating secure in-enclave data processing.

Despite these protections, a malicious client could still misuse legitimately accessed data—for example, by persisting it in plaintext on disk. This behavior is demonstrated by the **untrusted client**, which is associated with a different MRENCLAVE value. As a result, the RA-REST server rejects its Attestation request, preventing access to the data and returning an error.

These two clients, representing different TPSs, together with the RA-REST component, ensure the privacy enforcement policy required in my work.

3.2.8.4. Security analysis

This section provides an informal security argument for the RA-REST component: the message sequence, the adversary model at the message level, the binding and freshness claims, and the residual limitations. No mechanized formal analysis is provided; the argument is structured but informal.

Message sequence. (1) The client enclave generates an ephemeral key pair and a self-signed RA-TLS X.509 certificate whose `report_data` field binds the SGX Quote to the hash of the certificate's public key. (2) The client opens a TLS connection to the RA-REST server, presenting the RA-TLS certificate during the handshake. (3) The server verifies, in order: that the handshake proves possession of the private key matching the certificate; that the Quote is valid and issued by genuine SGX hardware (DCAP); that `report_data` equals the hash of the certificate's public

[7] https://huggingface.co/docs/hub/transformers

key; and that the reported measurements (MRENCLAVE, MRSIGNER, ISVPRODID, ISVSVN) belong to the accepted set. (4) On success, the client sends its request `(?key=cntId)` over the established channel. (5) The server queries HLF for the events of cntId, returns the result over the same channel, and closes the connection. Attestation is enforced per request, on a fresh connection.

Adversary model (message level). Consistent with the system threat model, the adversary controls the network and the untrusted host: it can observe, drop, reorder, inject, and replay messages, and can run its own enclaves with arbitrary (non-accepted) measurements. It cannot violate SGX guarantees nor extract an enclave's private key.

Binding. Because the Quote's `report_data` binds the attestation to the certificate's public key, and the TLS handshake proves possession of the corresponding private key, the attested measurement is bound to the specific TLS session. A "quote-then-swap" attack — presenting a valid Quote for one enclave and then conducting the session with a different key or enclave — is prevented, since the key that establishes the channel is the one that is attested. Data is thus delivered over a channel cryptographically bound to the attested enclave.

Freshness. A replayed RA-TLS certificate is useless to an adversary, as completing the handshake requires the private key, which never leaves the genuine enclave. Combined with the per-connection ephemeral key and the per-request enforcement of attestation, this bounds the staleness of the attested identity: each request re-establishes a freshly attested session, and the TLS handshake randomness prevents replay of a previous session.

Residual limitations. First, attestation certifies the client's code and identity at connection time; it does not constrain what a correctly attested client does with the data afterwards. A genuine, accepted enclave that is malicious by design or is compromised after attestation could still exfiltrate the received data — a time-of-check-to-time-of-use limitation on the data (not on the channel), already acknowledged as a residual risk in Table 16 and inherent to attestation-based data sharing. Second, in one-way mode (`DoRemoteAttestation`) the server verifies the client but not vice versa; mutual attestation (`DoMutualRemoteAttestation`) is required when the client must also verify the server, and mutual attestation must be enforced consistently on both sides. Third, the freshness argument relies on the standard RA-TLS binding of the Quote to the public key rather than on a server-supplied nonce embedded in the Quote; introducing an explicit challenge–response nonce would further harden freshness and is a candidate refinement.

3.2.9. Management of trusted measurements

Enclave measurements are used in two distinct ways in the proposed system. For sealing (Section 3.2.1), an MRSIGNER-based policy is adopted, so a change in the enclave binary does not invalidate sealed data. For attestation, instead, the RA-REST server validates the client's MRENCLAVE, MRSIGNER, ISVPRODID, and ISVSVN against a set of accepted values (Section 3.2.8). This second case requires an explicit operational policy: in the current implementation, each verifying party — the per-Organization RA-REST server — holds its own accepted-

measurement set, provided through configuration at startup. When, for example, a new version of a software component is released, its MRENCLAVE changes, and the corresponding measurement in the accepted set can be updated. In this case the old MRENCLAVE is revoked, but, also, in case of compromised platforms, the revocation is handled by the Intel attestation infrastructure [25, 37] through the CRLs, orthogonally to the application-level policy.

3.3. Experimental results

3.3.1. Port of Genoa Use Case Layout

The GP has been validated in the context of the Port of Genoa scenario. The experimental network consists of multiple HLF Organizations, including the Port Authority (serving as the administrative Organization), a Terminal Operator, a Shipping Agent, and a Transport Company (even if the tests were conducted without the Shipping Agent), with the possibility of extending the network to additional logistics-related entities. The network is configured with a single CA shared across all Organizations and three Orderer nodes operated by the administrative Organization. Within this setting, the Shipper acts as a SO, while the Terminal Operator is considered as a SH.

A TPS may require access to selected shipment-related events—for example, to support machine learning-based analytics—even though it does not belong to the HLF channel. To address this trust gap, the RA mechanism is enforced on a per-request basis. To evaluate these privacy-enforcement properties of GP, two RA-REST client implementations were developed, as already discussed in Section 3.2.8.3.

3.3.2. Experimental Setup

The experimental evaluation was conducted on an SGX-enabled system whose hardware configuration is reported in Table 18.

Parameter	Value
Hardware	Intel Xeon Silver 4509Y, 16 cores, 256 GB RAM, 436 GB storage
Operating system	Ubuntu 24.04 LTS
SGX generation	Intel SGX (Gramine v1.8 modified)
HLF version	Hyperledger Fabric v2.5.11 (modified)
Organizations	3 (Port Authority, Shipper, Terminal Operator) + Ordering Service
Load testing tool	Grafana k6 (Docker v28.1.1 and k6 v1.5.0)
Test duration	20 min per concurrency level
Concurrency levels (virtual users)	1, 4, 7, 10
Configurations compared	Graminized HLF (SGX) vs. standard HLF deployment (Base)

Table 18 Experimental setup.

3.3.3. Performance

Performance benchmarks were conducted on the REST application interacting with the Chaincode, both within Gramine and in a traditional (non-confidential) environment. The

evaluation was performed using the Grafana k6 tool, deployed as a Docker container on the same machine. The experimental evaluation is indicative rather than exhaustive, and several methodological limitations should be noted. First, each configuration and concurrency level was measured in a single run over a fixed time window (Table 19), without repetitions; consequently, no variance, confidence intervals, or statistical-significance testing are reported, and the p95 figures should be read as point estimates. Second, the evaluation was conducted on a single hardware and software configuration and on a network of three Organizations with a single shared CA (Section 3.3.1), so absolute figures may not transfer to larger networks, or different processors. Third, only the REST-application-to-Chaincode path was exercised, and throughput was not evaluated independently of latency. These limitations do not affect the qualitative conclusion — that the graminization overhead is moderate, which is the main focus of the evaluation — but they bound the strength of the quantitative claims.

A custom test script was developed to execute the following sequence of operations:

1. Create a new shipment S for a randomly generated container C, handled by the Terminal, via the Shipper REST endpoint,
2. Retrieve S via the Shipper REST endpoint,
3. Generate 10 events associated with container C via the Terminal REST endpoint,
4. Retrieve all events related to C via the Shipper REST endpoint,
5. Retrieve the most recent event as a string via the Shipper REST endpoint,
6. Verify the retrieved event via the Terminal REST endpoint.

Tests were conducted over 20-minute intervals at four concurrency levels (1, 4, 7, and 10 virtual users), representing a range from minimal to moderate concurrent load. For each configuration, two deployments were compared: a fully graminized HLF deployment in which all components execute within SGX enclaves, and a standard HLF deployment without enclave protection (Base). Performance is evaluated in terms of the 95th percentile (p95) request execution time, which provides a more robust characterization of tail latency than average values. The p95 latency results, together with aggregate request counts and timeout rates, are reported in Table 19 or all concurrency levels and both configurations.

Operation	SGX 1VU	Base 1VU	SGX 4VU	Base 4VU	SGX 7VU	Base 7VU	SGX 10VU	Base 10VU
Create shipment	55.16	43.26	46.06	38.11	45.30	40.57	52.81	39.53
Get event	19.98	16.39	14.68	14.74	14.95	15.06	18.12	13.77
Get shipment	29.03	23.53	26.24	22.15	33.56	33.24	30.43	23.78
Query event (rich query)	30.78	28.06	32.06	35.60	29.48	34.42	43.37	34.11
Send event	46.21	32.64	41.24	32.68	48.64	43.36	50.09	44.56
Verify event	23.67	8.72	15.26	6.98	11.83	7.99	11.05	8.00
Avg. p95 overhead (×)	1.49	–	1.29	–	1.10	–	1.28	–
Total requests	1905	1935	7680	7740	12,550	13,470	18,003	19,059

Table 19 p95 latency (ms) per operation across different concurrency levels. SGX denotes the graminized deployment, while Base refers to the standard HLF deployment.

The results indicate that the overhead introduced by graminization is moderate and consistent with the expected cost of SGX memory encryption and Gramine-based enclave execution. At a concurrency level of 1 virtual user (VU), the average p95 overhead is approximately 1.49×. As concurrency increases to 7 VU, the overhead decreases to approximately 1.10×, suggesting that enclave-related costs become less dominant under higher load, where network and I/O factors play a larger role. At 10 VU, the overhead increases again to approximately 1.28×, consistent with increased contention within enclave execution. Write-intensive and cryptographically expensive operations exhibit the highest absolute overhead, whereas read operations remain comparatively less affected, with overhead ratios typically ranging between 1.0× and 1.3×. Overall, the mean p95 overhead across all workloads is approximately 1.3×, which is deemed acceptable for the targeted logistics scenario.

3.3.4. Memory Footprint

Memory consumption was evaluated by observing container-level RAM usage via docker stats. The corresponding results are reported in Table 20, together with a comparison with Base deployment.

Organization	RAM (GB, SGX)	RAM (GB, Base)	Principal Containers Contributing to Footprint (SGX)	Principal Containers Contributing to Footprint (Base)
Shipper	34.79	1.5	Peer (11.62), REST client (8.90), RA-REST (4.59), CCAAS (8.60)	Peer (0.1), REST client (0.5), Chaincode (0.5)
Terminal Operator	33.24	1.5	Peer (11.11), REST client (8.21), RA-REST (4.60), CCAAS (8.25)	Peer (0.1), REST client (0.5), Chaincode (0.5)
Port Authority	21.20	0.6	Peer (12.67), CCAAS (8.44)	Peer (0.1), Chaincode (0.5)
Ordering service (3 orderers)	23.78	0.08	Orderer 0–2 (~7.3–8.2 each)	Orderer 0–2 (~0.04-0.02 each)
Total	113.0	3.68	All containers combined	All containers combined

Table 20 Observed memory usage.

The total observed memory consumption across all containers is approximately 113 GB (w.r.t. 3.68 GB of Base), with each Organization (Shipper and Terminal Operator) consuming on the order of 33–35 GB (w.r.t. 1.5 GB of Base). This footprint is significantly higher than that of a standard HLF deployment and constitutes the primary operational limitation of the proposed architecture. The increased memory usage is primarily attributable to Gramine, as even lightweight processes require several gigabytes of memory within the enclave environment. Memory consumption scales approximately linearly with the number of processes. This overhead must therefore be carefully considered in production scenarios. Potential mitigation

strategies include a more detailed optimization of Gramine's memory configuration or the adoption of alternative Confidential Computing technologies such as Intel TDX, which operates at virtual machine granularity and may offer a more efficient memory profile for multi-process workloads.

4. Conclusions

In this work, an existing HLF-based solution for the logistics domain has been extended to operate within a CC environment, with the objective of strengthening privacy and confidentiality beyond what can be achieved through application-level mechanisms alone. The proposed architecture targets a threat model that native HLF deployments cannot address: a privileged or compromised hosting platform capable of accessing data both in memory and on the filesystem, regardless of application-layer access control policies.

Among the available CC technologies, Intel SGX has been selected in combination with the Gramine framework. SGX is preferred over alternatives such as Intel TDX and AMD SEV-SNP due to its finer-grained isolation model, broader availability for development and testing, and smaller TCB. Gramine was adopted because it enables the execution within SGX enclaves of unmodified applications written in arbitrary programming languages, automates the graminization of existing Docker images via the GSC tool, provides a configuration-driven sealing mechanism applicable to selected filesystem paths without requiring code modifications, and exposes multiple abstraction levels for remote attestation, including RA-TLS support.

The main contribution of my work is a Gramine-based deployment strategy that enables all HLF components—Peer nodes, Orderers, Chaincodes deployed via CCAAS, and REST-based client applications—to execute within SGX enclaves. This has been achieved through two novel architectural components: the PO, a RESTful service that executes administrative Peer operations inside the enclave, and the RegisterEnroll component, which generates and seals the cryptographic materials required by each HLF component during startup. A further novel component, RA-REST, enforces per-request RA for external TPS access to private data. The deployment required targeted modifications to both Gramine (directory rename support) and HLF (including Peer library refactoring, refactoring of CCAAS deployment scripts, and removal of loopback-address detection logic).

The combined use of PCs, RA, Sealing, and enclave-based execution establishes a layered privacy guarantee. PCs restrict data visibility at the application layer among HLF channel participants. RA enables a verifiable trust relationship with external TPSs on a per-request basis, ensuring that sensitive data is disclosed only to entities whose integrity and identity have been cryptographically validated. Sealing protects data stored on the filesystem—cryptographic material and Blockchain state—using CPU-derived keys that remain inaccessible outside the enclave. Enclave-based execution ensures protection of data in use

at the hardware level, preventing exposure even to privileged system software. Together, these mechanisms extend HLF's privacy guarantees from the application level to the hardware level, mitigating the risks associated with a compromised hosting environment.

It is worth clarifying the respective roles of the HLF layer and the CC layer, since RA could suggest that a consensus mechanism is no longer necessary. The two layers are, in fact, complementary rather than interchangeable. HLF contributes precisely these properties: ordering and consensus, an immutable and tamper-evident shared history, per-identity non-repudiation, and decentralized governance in which no single Organization can unilaterally alter the data or the rules. The CC layer (Intel SGX and Gramine) protects data in use and data stored into filesystem (sealing) against a privileged host. RA provides code-integrity assurance about a component at the moment of attestation, and — most importantly for this work — extends trust to a TPS outside the network; but it does not, by itself, provide agreement on a single ordered history among independent Organizations, nor immutability and non-repudiation of that history over time. Indeed, rollback and replay on sealed data are explicitly out of scope (Table 16). The division of responsibilities between the two layers is summarized in Table 21.

Security property	HLF	CC
Data-in-use confidentiality	✗	✓
Sealing	✗	✓
Application-level access control	✓ (PCs + dACL)	✓ (RA-TLS)
Ordered, immutable, tamper-evident, non-repudiable history	✓	✗
Multi-party consensus / decentralized governance	✓	✗
Code-integrity / authenticity of running node	✗	✓
Trust extension to external, unknown parties	✗	✓ (RA-TLS)

Table 21 Security properties provided by each layer of the proposed architecture ("tick mark" = provided, "cross" = not provided).

My work has been applied to the Genoa port use case, where public entities such as Shippers and Terminals require strict data confidentiality. In this context, data sharing with external entities is conditioned on the ability to verify their trustworthiness through RA.

Moreover, during my doctoral research, I also applied CC to additional use cases. One such application involves the design of a confidential Key Management System (KMS) for an HLF-based REST application — through an ad-hoc ERC-1155[8] implementation — in the context of sustainable mobility with reward mechanisms. By default, HLF stores cryptographic keys in

[8] https://eips.ethereum.org/EIPS/eip-1155

plaintext on the filesystem; however, it supports delegation of cryptographic operations to a Hardware Security Module (HSM) [64]. While effective, HSM-based solutions may be costly and inflexible, making CC a viable alternative [3]. End users interact with the network through a REST application that custodies their key pairs rather than requiring users to manage keys themselves. If this REST were deployed in the cloud, the custodied keys would be exposed to a privileged host. Thus, the REST is executed inside an SGX enclave similarly to the graminized application and RegisterEnroll of Section 3.2.6, and all user key pairs — together with the REST's own credentials — are sealed under the MRSIGNER policy. What is new with respect to GP is the sealing scope, which broadens from a single component identity to a custodial, multi-user key store. Client-to-server trust currently relies on trusting the server; but, enforcing it through RA with the library of Section 3.2.8.1 is a straightforward extension. Moreover, this approach is conceptually similar to Ren et al. [4]; however, their implementation relies on the Intel SGX SDK without higher-level frameworks, preventing the use of the official Fabric SDK/Gateway. In contrast, the use of Gramine enables both secure key sealing and usage of the official Fabric SDK/Gateway.

Furthermore, I also applied CC to a Federated Learning use case for satellite anomaly detection, in which multiple Collaborators train local models and a central Aggregator combines their weights. Each component — Collaborators, their Generators, and the Aggregator — is executed unmodified inside an SGX enclave via Gramine, so that model data remains protected in use. The remainder of the design differs markedly from GP: there is no blockchain, so none of the HLF-specific components apply; secure, disruption-tolerant communication between enclaves is delegated to an external platform, which provides end-to-end encryption and store-and-forward persistence; and the machine-learning training and inference is the workload to be protected. This use case therefore shows that the enclave-execution core of the approach generalizes to non-blockchain settings.

4.1. Limitations and future work

Several extensions are identified as future work, as reported in Table 13 and Table 16. First, RA should be extended to the administrative plane (PO–client interaction), enabling the PO to verify client identity prior to executing administrative Peer operations. Second, mutual RA between each Peer and its corresponding CCAAS Chaincode container should be introduced, extending trust guarantees to the Peer–Chaincode interaction. Third, graminization of CouchDB remains an open limitation of the current architecture. Fourth, the adoption of Intel TDX as an alternative Confidential Computing technology operating at VM granularity may substantially reduce memory overhead compared to the current SGX-based design. Finally, the introduction of a Secret Provisioner could provide a more flexible and scalable alternative to direct filesystem sealing for the management of cryptographic material.

Additionally, migrating from Docker to Kubernetes could improve scalability and deployment flexibility.

Acronyms

AA	Attestation Agent
AK	Attestation Key
AS	Attestation Service
BiTAS	Blockchain in Transport Alliance Standard
CA	Certification Authority
CC	Confidential Computing
CCA	Confidential Compute Architecture
CCC	Confidential Computing Consortium
CCAAS	Chaincode As An external Service
CDH	Confidential Data Hub
CNCF	Cloud Native Computing Foundation
CoCo	Confidential Containers
CRL	Certificate Revocation List
CVM	Confidential Virtual Machine
dACL	distributed Access Control List
EDL	Enclave Definition Language
ELRANGE	Enclave Linear Address Range
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
ER	Edge Routine
FPC	Fabric Private Chaincode
GP	Graminized Platform
GSC	Gramine Shielded Containers
HLF	Hyperledger Fabric
HSM	Hardware Security Module
ISVPRODID	Identifier of the software module signed by the enclave author
ISVSVN	Security version number associated with the software module
KBS	Key Broker Service
KMS	Key Management System
LibOS	Library OS
NSSA	Number of SSA
MAC	Message Authentication Code
MRENCLAVE	Enclave Measurement
MRSIGNER	Enclave Signer Measurement
MSP	Membership Service Provider
PA	Port Authority
PAL	Platform Adaptation Layer
PC	Private Collection
PCCS	Provisioning Certificate Caching Service
PCK	Provisioning Certification Key
PCS	Provisioning Certification Service
PeerLib	HLF's peer node as a library
PO	Peer Operator
PRM	Processor Reserved Memory
RA	Remote Attestation
RA-REST	RA-based REST
RegisterEnroll	Component generating and sealing HLF cryptographic materials

RSK	Root Sealing Key
RPK	Root Provisioning Key
RVPS	Reference Value Provider Service
PCE	Provision Certificate Enclave
PKI	Public Key Infrastructure
QE	Quoting Enclave
SECS	SGX Enclave Control Structure
SEV-SNP	Secure Encrypted Virtualization – Secure Nested Paging
SGX	Software Guard Extensions
SH	Shipment Handler
SIGSTRUCT	Signature Structure
SME	Secure Memory Encryption
SO	Shipment Owner
SSA	State Save Area
TCB	Trusted Computing Base
TCS	Thread Control Structure
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TP	Trusted Part of ERs
TPS	Third-Party Software
UP	Untrusted Part of ERs
VMM	Virtual Machine Monitor
VU	Virtual User

Appendix A – HLF transaction flows (standard and private data)

Before introducing private data management, it is useful to describe the standard transaction flow in HLF. Figure 3 (taken from [65]) illustrates the sequence of steps that leads from the invocation of a Chaincode method by a client application to the creation and commitment of a block in the Blockchain:

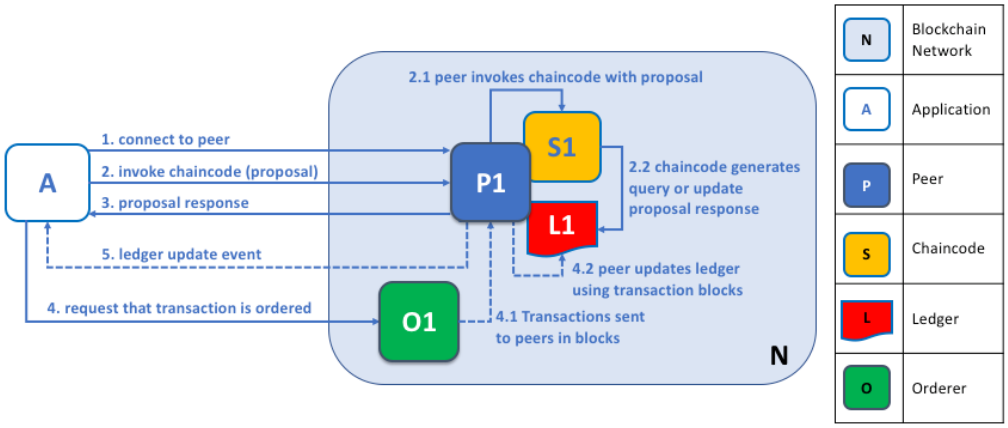

Fig. 20 HLF transaction flow

a. After connecting to a Channel through a Peer (denoted as P1 in the figure), the client application (A in the figure) invokes a Chaincode method—i.e., a transaction—by sending a request to P1,
b. Peer P1 forwards the request to the Chaincode. The Chaincode simulates the transaction and does not immediately apply the resulting state changes, as consensus from other Peers is required,
c. P1 returns a transaction proposal response. If multiple Endorsing Peers are involved, A waits to receive proposal responses from all required Peers,
d. Once the proposal responses have been received and verified to be consistent, A submits them to the Ordering Service,
e. The Ordering Service groups them into a block and distributes it to the Leading Peers of the participating Organizations, which subsequently propagate the block to the other Peers in their Organization,
f. After receiving the block, Peers verify the transactions and update their local Ledger (this phase is called "commit phase"). Once the Ledger has been updated, they emit a Ledger update event to A, indicating that the transaction has been successfully committed.

The process described above corresponds to "submit transactions", i.e., transactions that modify the Ledger state. In contrast, "query transactions" only retrieve information. In this case, once the Chaincode executes the query, P1 simply returns the result to A, and the process terminates without involving the Ordering Service.

HLF allows private data to be accessible only to a subset of Organizations. For all other Organizations, as well as for Orderer nodes and the Blockchain itself, only the hash of the private data is visible. This functionality is implemented through the PC mechanism. Within a PC, each authorized Organization maintains both the plaintext version of the private data and

its corresponding hash in its CouchDB database. In contrast, unauthorized Organizations store only the hashed representation. HLF supports two types of PCs:

- **Explicit PCs**. These collections must be explicitly defined in a configuration file. The configuration specifies parameters such as the collection name, the private data distribution policy, the collection-level endorsement policy, and the `blockToLive` parameter, etc. The latter determines the number of blocks after which a private data entry is automatically deleted if it has not been updated. This configuration is provided during Chaincode deployment and must be approved according to the policies defined for the Channel.
- **Implicit PCs**. Each Organization automatically possesses an implicit PC that can be accessed exclusively by that Organization. Access to this collection does not require authorization from other Organizations.

Because of their confidential nature, private data transactions follow a flow that differs from the standard transaction one. The private data transaction flow is illustrated in Fig. 21 (taken from [66]) and can be summarized as follows:

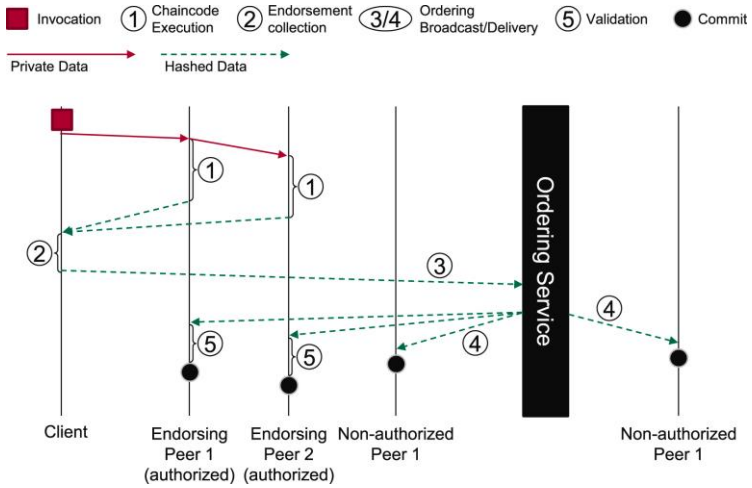

Fig. 21 HLF private data transaction flow

a. Like in the standard case, the client application connects to the network and invokes a Chaincode method containing private data, sending the request to an authorized peer,
b. The Peer distributes the request to other authorized Peers according to the private data distribution policy,
c. Each Peer returns a transaction proposal response that contains only the hash of the private data rather than the plaintext values,
d. The client application forwards the endorsed transaction to the Ordering Service, which packages the transactions into blocks and distributes them to all Peers, including those that are not authorized to access the private data,
e. Unauthorized Peers verify only the hash values contained in the block. Authorized Peers, on the other hand, compare the received hash with their local plaintext data, which has been temporarily stored during the process in a dedicated structure known as the "transient data store",
f. If validation is successful, the commit phase begins: Peers append the block to the Blockchain and send a Ledger update event to the client application. Authorized Peers

also clear the contents of the transient data store once the transaction has been committed.

In both the standard and private data transaction flows, if an error occurs or if the required policies are not satisfied during any stage of the process, the transaction flow is immediately aborted and an appropriate error message is returned to the client application.

Appendix B – Intel SGX SDK enclave example

In this section, an example (adapted from the official SGX SDK sample code [67]) of a simple application developed using the Intel SGX SDK to implement an enclave application is presented. It is assumed that the underlying OS is Linux.

Typically, a generic enclave application project is structured as follows:

```
sgx-example
├── app
│   └── app.cpp
├── enclave
│   ├── Enclave.config.xml
│   ├── Enclave.cpp
│   ├── Enclave.edl
│   └── Enclave.h
└── Makefile
```

In this example, two directories are defined, namely `app` and `enclave`, which are referred to as UP and TP, respectively. `Makefile` is used to compile the project, generate the ERs for both UP and TP, and produce the enclave shared object.

The `enclave` directory contains files for both configuration and code definition of the enclave `Enclave`. The file `Enclave.config.xml` is an XML configuration file specifying SGX configurations such as PRODID and the number of threads (`TCSNum`), etc., as reported below:

```
<EnclaveConfiguration>
  <ProdID>0</ProdID>
  <ISVSVN>0</ISVSVN>
  <StackMaxSize>0x40000</StackMaxSize>
  <HeapMaxSize>0x100000</HeapMaxSize>
  <TCSNum>1</TCSNum>
  <TCSPolicy>1</TCSPolicy>
  <DisableDebug>0</DisableDebug>
</EnclaveConfiguration>
```

The file `Enclave.edl` defines the enclave interface using the Enclave Definition Language (EDL), specifying ECALLs and OCALLs. An example is shown below:

```
enclave {

  trusted {
    public void an_enclave_function(int i, [out, size=1024] char* output);
    public void another_enclave_function([in, string] const char *str);
  };

  untrusted {
    void an_outside_function([in, string] const char *str);
  };
};

```

This code snippet defines:

- the ECALL `an_enclave_function()`, which has no return value and takes two parameters: an integer and a character pointer used to return data,
- The ECALL `another_enclave_function()`, which has no return value and accepts a string parameter,
- The OCALL `an_outside_function()`, which has no return value and accepts a string parameter.

Pointer parameters in EDL follow a specific syntax with dedicated handling [68]. The `out` attribute specifies that, when an application performs an ECALL, the ER allocates a zero-initialized buffer in the TP. Upon completion of the ECALL, the ER copies the buffer content from TP to UP, even if the ECALL terminates prematurely. Conversely, the `in` attribute specifies that the ER copies memory content from UP into a buffer in TP before invoking the ECALL function. The same semantics apply to OCALLs `out` and `in` attributes. A parameter may simultaneously have both attributes. Another attribute, `user_check`, disables automatic checks by the ER, delegating responsibility to the developer. Due to the memory-copying mechanism just described, the ER must always know the size of arrays in ECALL/OCALL definitions. An exception is represented by parameters annotated with the `string` attribute: in this case, the length is computed in UP, and after copying, the ER ensures null termination in TP. This attribute can be combined with `out` and `in`, like `public void my_ecall([in, out, string] const char* s)`.

In addition to these attributes, modifiers can be specified. The most relevant ones, applicable to both `out` and `in`, are:

Modifier	Description	Example

count	It specifies the number of elements of the pointed type to be copied, expressed as a multiple of `sizeof(pointed_type)`; this value can be a constant or a function parameter;	`// copies `10*sizeof(int)` bytes`
		`public void my_ecall([in, count=10] int* p);`
		`// copies `c*sizeof(int)` bytes`
		`public void my_ecall([in, count=c] int* p, unsigned int c);`

size	It specifies the number of bytes to be copied , also defined as a constant or parameter. It can combined with `count`, in which case the ER copies a number of bytes equal to the product of `count` and `size`.	`// copies `10` bytes`
		`public void my_ecall([in, size=10] char* p);`
		`// copies `s` bytes`
		`public void my_ecall([in, size=s] char* p, unsigned int s);`

Table 22 Modifiers for pointers and array in ECALL and OCALL definition in EDL

In the example code, the `output` buffer in `an_enclave_function()` has a size of `1024` bytes (intentionally not defined as a string), while `another_enclave_function()` and `an_outside_function()` operate on `string` parameters.

The file `Enclave.h` contains the definition of the enclave interface. In this example:

```
 1. #ifndef _ENCLAVE_H_
 2. #define _ENCLAVE_H_
 3.
 4. #if defined(__cplusplus)
 5. extern "C" {
 6. #endif
 7.
 8. void an_enclave_function(int i, char *output);
 9. void another_enclave_function(const char *str);
10. void another_enclave_function_internal(const char *str, ...);
11.
12. #if defined(__cplusplus)
13. }
```

```
#endif

#endif /* !_ENCLAVE_H_ */

```

Notably, `another_enclave_function_internal()` is neither an ECALL nor an OCALL; it is an internal enclave function. This design is necessary because variadic functions cannot be defined in EDL.

The file `Enclave.cpp` contains the implementation of these functions:

```
#include "Enclave.h"
#include "Enclave_t.h" /* an_outside_function */
#include <stdarg.h>
#include <stdio.h> /* vsnprintf */
#include <string.h>

void an_enclave_function(int i, char *output)
{
    int c = i;
    for (int j = 0; j < i; j++) {
        int c = (c % 95) + 32;
    }
    output[0] = (char) c;
    output[1] = '\0';
}

void another_enclave_function(const char *str) {
    another_enclave_function_internal(str);
}
void another_enclave_function_internal(const char *str, ...)
{
    char buf[BUFSIZ] = {'\0'};
    va_list ap;
    va_start(ap, str);
    vsnprintf(buf, BUFSIZ, str, ap);
    va_end(ap);
    an_outside_function(buf);
}

```

In this example:

- `an_enclave_function()` performs a computation which cannot be observed at runtime outside the enclave and stores the result in the `output` buffer,
- `another_enclave_function()` acts as a wrapper for `another_enclave_function_internal()`,
- `another_enclave_function_internal()` prepares `str` to be passed to `an_outside_function()`, which is implemented outside the enclave (in this example in `app.cpp`).

Interaction with the enclave is demonstrated in `app.cpp`:

```
#include <iostream>
#include <cstring>
#include <unistd.h>

#include "Enclave_u.h"
#include <sgx_urts.h>

/* OCall function */
void an_outside_function(const char *str)
{
```

```
11.     std::cout << str << std::endl;
12. }
13.
14. int main(int argc, char **argv) {
15.     sgx_status_t ret = SGX_ERROR_UNEXPECTED;
16.     sgx_launch_token_t token = {0};
17.     sgx_enclave_id_t global_eid = 0;
18.     int updated = 0;
19.
20.     ret = sgx_create_enclave("enclave.signed.so", 0, &token, &updated, &global_eid, nullptr);
21.     if (ret != SGX_SUCCESS) {
22.         std::cerr << "sgx_create_enclave failed: " << std::hex << ret << std::dec << "\n";
23.         return -1;
24.     }
25.
26.     char output[512] = {};
27.     sgx_status_t ecall_ret = an_enclave_function(global_eid, 42, output);
28.     if (ecall_ret != SGX_SUCCESS) {
29.         std::cerr << "enclave failed: " << std::hex << ecall_ret << std::dec << "\n";
30.         return -1;
31.     }
32.
33.     std::cout << "result: " << output << std::endl;
34.     fflush(stdout);
35.
36.     sgx_status_t ocall_ret = another_enclave_function(global_eid, "abc");
37.     if (ocall_ret != SGX_SUCCESS) {
38.         std::cerr << "enclave failed: " << std::hex << ocall_ret << std::dec << "\n";
39.         return -1;
40.     }
41.
42.     sgx_destroy_enclave(global_eid);
43.     return 0;
44. }
45.
```

It implements:

- the OCALL `an_outside_function()`, which prints the input string to standard output.
- The `main()` function, which performs the following step:
  1. `sgx_create_enclave("enclave.signed.so", 0, &token, &updated, &global_eid, nullptr)` creates the enclave from the signed shared object `"enclave.signed.so"` with configuration parameters (e.g., `0` disables debug mode). This initializes the enclave identifier `global_eid`, which is used in subsequent calls to that enclave, and returns a status code indicating success or failure).
  2. `an_enclave_function(global_eid, 123, output)` invokes the corresponding ECALL by specifying the enclave id (`global_eid`) and its input (`123`). Although the ECALL itself does not return a value, the wrapper function returns a status code, like `sgx_create_enclave()`. `output` is used to retrieve the result.
  3. `another_enclave_function(global_eid, "abc")` invokes the second ECALL, which in turn performs an OCALL to `an_outside_function()`. Its return value follows the same convention as the previous ECALL.
  4. `sgx_destroy_enclave(global_eid)`, as the name suggests, destroys the enclave identified by `global_eid`.

Using the `Makefile`, the application and enclave shared object can be compiled and built, and the latter can be signed using the enclave author's key (e.g., generated via the `openssl` command). Prior to compilation, the ER for both UP and TP must be generated using the `sgx_edger8r` tool provided by Intel SGX. This tool takes the EDL file as input and produces the corresponding ERs, namely `Enclave_u.{h,c}` (for UP) and `Enclave_t.{h,c}` (for TP). Notice that they are named `Enclave` because in this example the enclave name is `Enclave`; in fact, it comes from the `Enclave.edl` file in the `enclave` directory and `sgx_edger8r` executable.

When executing the application described above, the expected output is as follows:

```
result: *
abc
```

Appendix C – EGo sealing example

This section presents a simple example demonstrating sealing using EGo. The project follows a standard Go application structure:

```
example
├── enclave.json
├── go.mod
├── go.sum
├── main.go
└── Makefile
```

`go.mod` and `go.sum` define dependencies, and `main.go` contains the application logic:

```
 1. package main
 2.
 3. import (
 4.     "log"
 5.     "os"
 6.     "path"
 7.
 8.     "github.com/edgelesssys/ego/ecrypto"
 9. )
10.
11. var filePath string = path.Join("/", "tmp", "afile")
12.
13. func seal() {
14.     sealed, err := ecrypto.SealWithProductKey([]byte("Hello World"), nil)
15.     if err != nil { log.Fatal(err) }
16.     if err := os.WriteFile(filePath, sealed, 0644); err != nil {
17.         log.Fatal(err)
18.     }
19. }
20.
21. func unseal() string {
22.     content, err := os.ReadFile(filePath)
23.     if err != nil { log.Fatal(err) }
24.     clear, err := ecrypto.Unseal(content, nil)
25.     if err != nil { log.Fatal(err) }
26.     return string(clear)
27. }
28.
29. func main()  {
30.     seal()
31.     clear := unseal()
32.
33.     log.Println(string(clear))
34. }
35.
```

The application seals a string (`"Hello World"`) and stores the encrypted data in the file `/tmp/afile`. The sealing operation uses a key derived from MRSIGNER (for MRENCLAVE-based sealing, the function `SealWithUniqueKey()` can be used). The program then reopens the file, unseals the stored content, and prints the resulting plaintext.

`Enclave.json` is the configuration file (see Section 2.6.2).

To execute the Go application within an SGX enclave using EGo, a sequence of commands, defined in this example into `Makefile`:

```
1. ego-go build
2. ego sign main
3. ego run main
```

Where, respectively:

1. Compile the application and generate the `main` executable, embedding EGo-specific metadata to enable execution both inside and outside the enclave.
2. Sign the executable using the key specified in `enclave.json`.
3. Execute the signed binary within the enclave.

The required key pair is automatically generated by the `ego sign` command if not already existing or not specified by `enclave.json`.

Appendix D – Server and client flowcharts in Remote Attestation Library

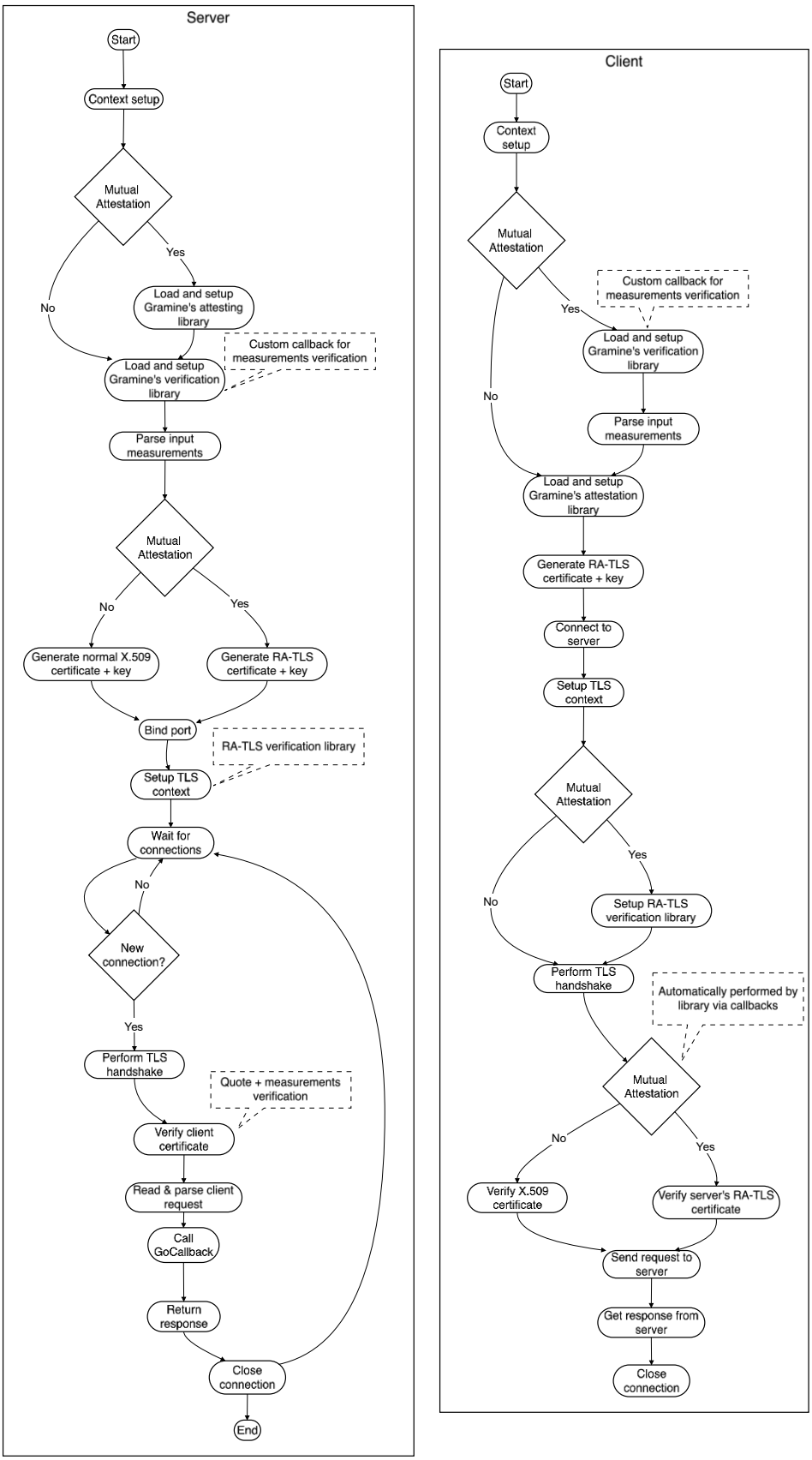

Fig. 22 Server and client flowcharts in Remote Attestation library

References

[1] S. Avola, P. Baglietto, M. Maresca and A. Parodi, "Preserving data privacy and confidentiality in a distributed ledger : a use case in logistics," in *2024 6th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2024.

[2] S. Avola, *Design of an Access Control System for Data Protection Enforcement in Permissioned Blockchains,* M.S. thesis, Genova: University of Genova, 2021.

[3] Edgeless Systems, "The 3 levels of confidential computing," [Online]. Available: https://www.edgeless.systems/wiki/what-is-confidential-computing/the-3-levels-of-confidential-computing. [Accessed 4 November 2025].

[4] Z. Ren, H. Li, G. He, R. Yu, Y. Tong, S. Xu and L. Deng, "An SGX-based key protection scheme for Hyperledger Fabric," *Cluster Computing,* vol. 28, no. 16, p. 1042, 2025.

[5] Hyperledger Fabric, "Ledger," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/ledger/ledger.html. [Accessed 4 November 2025].

[6] Hyperledger Fabric, "Glossary," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/glossary.html. [Accessed 4 November 2025].

[7] Hyperledger Fabric, "The Ordering Service," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html. [Accessed 4 November 2025].

[8] ConfidentialComputingConsortium, "Confidential Computing: Hardware-Based Trusted Execution for Applications and Data," November 2022. [Online]. Available: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC_outreach_whitepaper_updated_November_2022.pdf.

[9] ConfidentialComputingConsortium, "A Technical Analysis of Confidential Computing," November 2022. [Online]. Available: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf.

[10] Microsoft, "Trusted computing base," [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/trusted-compute-base. [Accessed 28 January 2026].

[11] ConfidentialComputingConsortium, "Common Terminology for Confidential Computing," December 2022. [Online]. Available: https://confidentialcomputing.io/wp-

content/uploads/sites/10/2023/03/Common-Terminology-for-Confidential-Computing.pdf.

[12] Intel, "Intel® Trust Domain Extensions White Paper," [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/690419. [Accessed 5 November 2025].

[13] G. D'Onghia, A. Lioy, I. Pedone and S. Sisinni, "Use of SGX to protect network nodes," 2023. [Online]. Available: https://webthesis.biblio.polito.it/26851/.

[14] AMD, "AMD Secure Encrypted Virtualization (SEV)," [Online]. Available: https://www.amd.com/en/developer/sev.html. [Accessed 5 November 2025].

[15] "Confidential Computing: A Security Overview and Future Research Directions," Politecnico di Milano, 2024. [Online]. Available: https://re.public.polimi.it/handle/11311/1272466.

[16] "AWS Nitro Enclaves Documentation," Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/enclaves/latest/user/what-is-nitro-enclaves.html.

[17] V. Costan and S. Devadas, *Intel SGX Explained,* Cryptology ePrint Archive, 2016.

[18] "SGX 101," [Online]. Available: https://sgx101.gitbook.io/sgx101/sgx-bootstrap. [Accessed 11 September 2025].

[19] Y. Jang, J. Lee, S. Lee and T. Kim, *SGX-Bomb: Locking Down the Processor via Rowhammer Attack,* Association for Computing Machinery, 2017.

[20] A. Nilsson, P. N. Bideh and J. Brorsson, *A Survey of Published Attacks on Intel SGX,* arXiv, 2020.

[21] Enclaive, "Confidential Computing 101 - Intel SGX - Local and Remote Attestation," [Online]. Available: https://docs.enclaive.cloud/confidential-cloud/technology-in-depth/intel-sgx/technology/concepts/local-and-remote-attestation. [Accessed 22 October 2025].

[22] Sharemind, "Intel SGX® Remote Attestation," [Online]. Available: https://docs.sharemind.cyber.ee/sharemind-hi/4/intel-sgx/remote-attestation.html?utm_source=chatgpt.com#_footnotedef_4:. [Accessed 22 October 2025].

[23] Intel®, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4," [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf. [Accessed 22 October 2025].

[24] Gramine, "Gramine Attestation and Secret Provisioning," [Online]. Available: https://gramine.readthedocs.io/en/v1.8/attestation.html. [Accessed 17 October 2025].

[25] Intel, "Design Guide for Intel® SGX Provisioning Certificate Caching Service (Intel® SGX PCCS)," [Online]. Available: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/SGX_DCAP_Caching_Service_Design_Guide.pdf. [Accessed 23 October 2025].

[26] Intel®, "Intel® Software Guard Extensions (Intel® SGX) - Developer Guide," [Online]. Available: https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Guide.pdf. [Accessed 27 October 2025].

[27] S. Kumar, A. Panda and S. R. Sarangi, *A Comprehensive Benchmark Suite for Intel SGX*.

[28] C. Zhao, D. Saifuding, H. Tian, Y. Zhang and C. Xing, *On the Performance of Intel SGX*.

[29] D. Porter, C.-C. Tsai and M. Vij, "Gramine: Enhanced Protection of Unmodified Linux Applications with Confidential Computing," 17 February 2022. [Online]. Available: https://www.youtube.com/watch?v=EtNYXu-1xes. [Accessed 10 October 2025].

[30] gramineproject, "Gramine's x86_64 libos_table.c," 6 August 2024. [Online]. Available: https://github.com/gramineproject/gramine/blob/v1.8/libos/src/arch/x86_64/libos_table.c. [Accessed 13 October 2025].

[31] gramineproject, "Gramine's PAL," 26 September 2024. [Online]. Available: https://github.com/gramineproject/gramine/blob/v1.8/pal/include/pal/pal.h. [Accessed 13 October 2025].

[32] gramineproject, "Gramine's PAL SGX OCalls," 6 June 2024. [Online]. Available: https://github.com/gramineproject/gramine/blob/v1.8/pal/src/host/linux-sgx/enclave_ocalls.h. [Accessed 13 October 2025].

[33] Gramine, "Gramine Manifest syntax," [Online]. Available: https://gramine.readthedocs.io/en/v1.8/manifest-syntax.html. [Accessed 15 October 2025].

[34] Gramine, "Gramine Glossary," [Online]. Available: https://gramine.readthedocs.io/en/v1.8/sgx-intro.html. [Accessed 16 October 2025].

[35] gramineproject, "Gramine's CI-Example HelloWorld Manifest," 2 May 2024. [Online]. Available: https://github.com/gramineproject/gramine/blob/v1.8/CI-Examples/helloworld/helloworld.manifest.template. [Accessed 16 October 2025].

[36] Mbed-TLS, "mbedtls," [Online]. Available: https://github.com/Mbed-TLS/mbedtls. [Accessed 20 October 2025].

[37] Intel ®, "Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation - Revision: 6.0," [Online]. Available: https://www.intel.com/content/dam/develop/public/us/en/documents/sgx-attestation-api-spec.pdf. [Accessed 19 November 2025].

[38] gramineproject, "Gramine's CI-Examples secret_prov," [Online]. Available: https://github.com/gramineproject/gramine/tree/v1.8/CI-Examples/ra-tls-secret-prov/secret_prov. [Accessed 21 October 2025].

[39] Gramine, "Encrypted Files in Gramine," [Online]. Available: https://gramine.readthedocs.io/en/stable/devel/encfiles.html. [Accessed 5 November 2025].

[40] Gramine, "Gramine Shielded Containers," [Online]. Available: https://gramine.readthedocs.io/projects/gsc/en/latest/#. [Accessed 21 October 2025].

[41] Occlum, "Occlum Documentation," [Online]. Available: https://occlum.readthedocs.io/en/stable/. [Accessed 30 October 2025].

[42] Occlum, "Occlum dcap_lib," [Online]. Available: https://github.com/occlum/occlum/tree/master/tools/toolchains/dcap_lib. [Accessed 30 October 2025].

[43] Occlum, "Occlum mod.rs," [Online]. Available: https://github.com/occlum/occlum/blob/master/src/libos/src/syscall/mod.rs. [Accessed 13 November 2025].

[44] Occlum, "Occlum Copy Bom," [Online]. Available: https://occlum.readthedocs.io/en/stable/tools/copy_bom.html. [Accessed 30 October 2025].

[45] Occlum, "Boot Flow Overview," [Online]. Available: https://occlum.readthedocs.io/en/stable/boot_flow.html. [Accessed 30 October 2025].

[46] Edgeless Systems, "EGo," [Online]. Available: https://docs.edgeless.systems/ego. [Accessed 31 October 2025].

[47] Go, "ego module," [Online]. Available: https://pkg.go.dev/github.com/edgelesssys/ego. [Accessed 31 October 2025].

[48] EGo, "EGo's attestation," [Online]. Available: https://github.com/edgelesssys/ego/blob/v1.8.0/internal/attestation/attestation.go#L41. [Accessed 31 October 2025].

[49] Confidential Containers, "Confidential Containers," [Online]. Available: https://confidentialcontainers.org. [Accessed 12 November 2025].

[50] Kata Containers, "Kata Containers," [Online]. Available: https://katacontainers.io. [Accessed 11 November 2025].

[51] Confidential Containers, "Architecture," [Online]. Available: https://github.com/confidential-containers/confidential-containers/blob/main/architecture.md#process-based-tee. [Accessed 15 December 2025].

[52] M. Brandenburger, C. Cachin, R. Kapitza and A. Sorniotti, *Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric,* 2018.

[53] Hyperledger, "Fabric Private Chaincode shim.h," [Online]. Available: https://github.com/hyperledger/fabric-private-chaincode/blob/main/ecc_enclave/enclave/shim.h. [Accessed 13 November 2025].

[54] R. Zappoli, *Go Language Support in Hyperledger Fabric Private Chaincode,* Université de Fribourg.

[55] BiTA, "Blockchain Standards," [Online]. Available: https://bita.studio/standards. [Accessed 4 November 2025].

[56] Hyperledger Fabric, "Introduction," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html. [Accessed 2 April 2026].

[57] "Gramine Features - Networking Sockets," [Online]. Available: https://gramine.readthedocs.io/en/stable/devel/features.html#networking-sockets. [Accessed 1 October 2025].

[58] "Hyperledger Fabric core.yaml," [Online]. Available: https://github.com/hyperledger/fabric/blob/main/sampleconfig/core.yaml#L46. [Accessed 1 October 2025].

[59] "Gramine - Issues - Rename syscall not working," [Online]. Available: https://github.com/gramineproject/gramine/issues/2110. [Accessed 1 October 2025].

[60] "Gramine Documentation - Encrypted Files in Gramine - Security properties of encrypted files," [Online]. Available: https://gramine.readthedocs.io/en/stable/devel/encfiles.html#security-properties-of-encrypted-files. [Accessed 1 October 2025].

[61] Gramine, "Performance tuning and analysis," [Online]. Available: https://gramine.readthedocs.io/en/v1.8/performance.html#multi-process-workloads. [Accessed 9 January 2025].

[62] "Go Wiki: cgo," [Online]. Available: https://go.dev/wiki/cgo. [Accessed 1 October 2025].

[63] Hugging Face, "Qwen/Qwen3-0.6B," [Online]. Available: https://huggingface.co/Qwen/Qwen3-0.6B. [Accessed 4 November 2025].

[64] Hyperledger Fabric, "Using a Hardware Security Module (HSM)," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/hsm.html. [Accessed 4 November 2025].

[65] Hyperledger Fabric, "Peers," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/peers/peers.html. [Accessed 4 November 2025].

[66] T. Guggenberger, J. Sedlmeir, G. Fridgen and A. Luckow, "An In-Depth Investigation of the Performance Characteristics of Hyperledger Fabric," 2021 (revised 2023).

[67] Intel, "Intel - confidential-computing.sgx - Sample Code - Sample Enclave," [Online]. Available: https://github.com/intel/confidential-computing.sgx/tree/main/SampleCode/SampleEnclave. [Accessed 24 March 2026].

[68] Intel®, "Input Types and Boundary Checking in Enclave-Definition Language (EDL) Files," [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwidxqfu3cSQAxXZ9bsIHZ77OtgQFnoECBgQAQ&url=https%3A%2F%2Fcdrdv2-public.intel.com%2F671446%2Finput-types-and-boundary-checking-edl.pdf&usg=AOvVaw1owscbNd1kXMtWNzPlDlt8&opi=89978449. [Accessed 28 October 2025].