

Sharemind Software Development Platform for VM based TEEs

Technical Report

Table of Contents

1. Introduction	3
1.1. Document Purpose	3
1.2. Status	3
1.3. Sharemind TSDP Purpose	3
1.3.1. Unique Value Proposition	3
1.3.2. Solution Level	3
1.4. Document Methodology	4
Glossary	5
2. System Stakeholders and Requirements	6
2.1. Stakeholders	6
3. Architecture Principles	8
3.1. Core Principles Guiding the Architecture	8
3.2. Sidecar Philosophy	8
4. Architectural Views	10
4.1. Context View	10
4.1.1. System Runtime	10
4.1.2. Solution Instance Initialization	12
4.2. Functional View	15
4.2.1. Authentication & Authorization	15
4.2.2. Dataflow Configuration - DFC	15
4.2.3. Startup APIs	17
4.2.4. Runtime APIs	18
4.3. Trust Building	26
4.3.1. Usage Sites of Sensitive Data	27
4.4. Operational View	27
4.4.1. TVM Assembly ^[GAP-303]	27
4.4.2. TVM Deployment	28
4.4.3. Monitoring	28
4.4.4. TSDP Shutdown ^{[GAP-203][GAP-303]}	28
4.4.5. TVM Upgrade	29
5. System Qualities	30

5.1. Performance	30
6. Gap Registry	31
6.1. Referencing Gaps	31
6.2. Gap Types	31
6.3. Current Gaps	31
6.3.1. Remote Attestation & Trust	32
6.3.2. Key Management & Cryptography	32
6.3.3. TVM Lifecycle & Persistence	33
6.3.4. Image Building & Initialization	34
6.3.5. Network & Communication	35
6.3.6. Authorization & Identity	36
Appendix A: Technical Details	38
A.1. Trusted Execution Environments	38
A.2. Architecture Alternative Overview	39
A.2.1. TEE Granularity	39
A.2.2. Connection of Application And Remote Attestation	40
A.3. Remote Attestation	41
A.3.1. Round-trip Schema	41
A.3.2. SIGMA-Protocol Authenticated Key Exchange	43
A.3.3. Quote Verification	46
A.3.4. Authentication via Identity Provider	47
A.4. Authorization via Open Policy Agent ^[GAP-502]	49
Appendix B: Intel TDX setup	51
B.1. Overview	51
B.2. Host setup	51
B.3. Guest setup	51

Date	Version	Description
2025-08-28	0.1	Draft for initial review
2025-09-01	1.2	Revised version for release
2026-07-10	1.3	Update of document
2026-09-30	2.0	Updated document after open source release

Abstract

Release architecture document of the Sharemind TEE Software Development Platform (TSDP). Its content reflects the current open sourced version and may change with future releases.

1. Introduction

1.1. Document Purpose

This document describes the architecture of Sharemind TEE Software Development Platform (TSDP), part of the Sharemind platform family. Sharemind TSDP is the current generation of the Sharemind platform that uses Trusted Execution Environments (TEEs) such as Intel® TDX or other, see [Section A.1](#). It succeeds Sharemind HI, the predecessor product which used hardware enclaves. Both products share the same Sharemind platform principles but differ in their underlying hardware trust model and deployment architecture. From this point onward, "TSDP" refers to Sharemind TSDP throughout this document.

1.2. Status

The document and the TSDP Framework is **Work-In-Progress**.

Throughout this document gap between vision and current status are documented and referenced. For more details see the [Gap Registry](#).

1.3. Sharemind TSDP Purpose

1.3.1. Unique Value Proposition

Build software where every step of processing sensitive data is protected against a single malicious stakeholder. I.e. an attacker needs to (1) compromise multiple stakeholders, (2) collude with another stakeholder, or (3) break the [TEE](#) protection to leak sensitive data.

1.3.2. Solution Level

The TSDP is a framework for building privacy preserving applications using [TEE](#) technologies. As a security product, its raison d'être is best described through its main [security goal](#), which aligns with the security goal of the Sharemind platform:

Provide an analytics platform where the uploaded data of data owners is protected by secure computing technologies at all points, up until the point where the derived analysis results reach the designated output consumers. Data is protected in transit, at rest and during processing. At no point should any sensitive data be at the mercy of a single human being if they do not own that data. The TSDP wants to make it easier for solution developers to create such solutions. It hides boilerplate, e.g. around access control and peculiarities of [TEE](#) technologies, and transparently handles the encryption and decryption of sensitive data for the analysis code.

1.4. Document Methodology

This document follows the methodology of the book: *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* by Nick Rozanski and Eoin Woods, Addison Wesley 2011, Second Edition (<https://www.viewpoints-and-perspectives.info/>)

Glossary

Application

Something executable which runs inside of the TVM and is provided by the solution developer, i.e. not by Cybernetica.

DFC

[Dataflow Configuration](#), which specifies authentication and authorization details.

End User Application

Software which was written by the solution developer, i.e. not by Cybernetica.

MPA

Multi Party Authorization. Some process requires actions of multiple legal entities before having an effect.

Peer

A Stakeholder, Dynamic End User or an Application (explained in [Section 4.1.1](#)).

Solution Instance

One particular TVM (cluster) through its own lifetime.

Also refer to the tables in [Section 2.1](#) and [Section 4.1](#).

Official standards with relevant terminology are at:

RATS

<https://www.rfc-editor.org/rfc/rfc9334>

2. System Stakeholders and Requirements

2.1. Stakeholders

These are the stakeholders and roles as they exist on the contract level. Note that a stakeholder could also have multiple roles, like the Coordinator and Result Owner being the same entity, or Cybernetica being the Solution Developer.

Technical stakeholder roles are described in [Section 4.1.1](#) and [Section 4.1.2](#) (mixed with technical components).

Stakeholder/Role	Explanation
Coordinator	A legal entity which is the primary driver (champion) behind the project, who brings the stakeholders together and sees the project from cradle to grave. <i>Motivation:</i> They want to gather new insights which is possible if data from different sources is brought together. Data Owners are not allowed to share their data under non-TEE circumstances due to either business secret (e.g. bank) or regulatory (e.g. gene database) reasons. The coordinator organizes the cooperation between the other stakeholders.
Cybernetica (not a role)	Provider of the TSDP and documentation, which is used by the Solution Developer to build the Solution, and by the Coordinator to explain the security of the Solution to the rest of the stakeholders. <i>Motivation:</i> Cybernetica provides support, tooling and projects/solutions around the challenges of employing secure computing technologies.
Data Owner	A legal entity which has data and is incentivised by the project prospects to share their sensitive data with the TEE Solution. They likely are some form of an End User . <i>Motivation:</i> They are contracted by the Coordinator (or Result Owner).
Result Owner	A legal entity which wants to get the results out of the analysis. They likely are some form of an End User . <i>Motivation:</i> They want to get new insights from a cross-company view upon data, and monetize on this. They pay the Coordinator.
Service Provider	A legal entity which hosts TSDP, and possibly some existing infrastructure into which the TSDP must fit into. They might rent TEE hardware from a <i>Cloud Service Provider</i> , and they employ an <i>Administrator</i> who does the hands-on configuration and operation of the TSDP solution. This could be the <i>Coordinator</i> or the <i>Result Owner</i> . <i>Motivation:</i> They are likely an employee of the Coordinator or Result Owner.
TEE Provider	Passive. It provides the hardware and the root certificate for the remote attestation collateral to identify the genuineness of the attested platform ("it's really a TEE"). List of TEE providers: Intel, AMD, Arm, and others. ^[1]

Stakeholder/Role	Explanation
Cloud Service Provider	The Solution is likely hosted in some cloud. The consideration of whom to choose is relevant, since standard TEE technologies are not secure against sophisticated physical attacks. A CSP with a good reputation should be used. <i>Motivation:</i> They make money by giving easy access to managed hardware.
Administrator / Orchestrator	Puts in place the configuration files and starts the technical components (TVM, Database, Untrusted Gateway) and monitors the operations. <i>Motivation:</i> They are likely an employee of (and thus paid by) the Coordinator or Result Owner.
Solution Developer	They develop the Solution, based on the TSDP and documentation (from Cybernetica). <i>Motivation:</i> They are likely an employee of (and thus paid by) the Coordinator or Result Owner.
Solution Auditor	Ideally independent legal entity which makes sure that the solution is built as proclaimed in the contract, i.e. the sensitive data will be processed as the stakeholders agreed upon. <i>Motivation:</i> Paid by the Coordinator.

3. Architecture Principles

3.1. Core Principles Guiding the Architecture

- The TSDP needs to fulfill the [Unique Value Propositions](#).
- The TSDP should hide cryptographic details (encryption^[GAP-104] details, remote attestation) from the applications as much as possible. Solution developers should be able to mostly focus on the business logic.
- Solution developers should be able to use any programming language they want. This means, application development should not require a special SDK, exceptions are made for necessary wrappers or integration tooling.
- The TSDP should allow integration into existing systems, e.g. as a [sidecar](#) behind gateways.

3.2. Sidecar Philosophy

TEE technologies are commonly advocated via lift-and-shift offerings, where you can take an existing app and just move it into the TEE. The app manages its own TLS certificates and end users use HTTPS to communicate with it. There are two shortcomings with it (with an overview in [the table below](#)):

1. Regarding the [UVP](#), the admin is not allowed to look around in the running system inside of the TEE, as they would otherwise be able to see the sensitive data. But if all of the system is inside of the TEE, they cannot do their regular maintenance work (e.g. debug new errors, optimize database indices, ...). A band-aid solution would be to provide lots of specialized monitoring capabilities which don't leak any sensitive data, but this is likely a huge amount of work.
2. We noticed that usually we don't just lift and shift applications, but implement a new module for an existing system where anyway new code needs to be written. The goal is to support such constructs where only the minimum amount of software is inside of the TEE, and the TEE does not necessarily be directly available from the internet.

Table 1. Comparison of the lift-and-shift approach (Full solution in the TEE) with the sidecar approach (Small solution component in TEE).

Approach	Pros	Cons
Lift-and-Shift (Full solution in the TEE)	• Easier to explain • Works with just about any legacy/containerized application • In principle easier dev & CI/CD, all goes into a single VM image.	• Either admin sees all the data OR locked-down, complicated-to-diagnose the system
Sidecar (Small solution component in TEE)	• Only a small system component needs to be locked-down (inaccessible for admins), can be as small as allowed by the requirements and cleverness. • Rest of the system fully manageable as usual by admins (and can also run in a TVMs just not fully locked-down), easily updatable without DFC upgrades.	• Dev & CI/CD more complicated as there is a system split.

Note: There are more specialised variants, which are left out for brevity of the document.

4. Architectural Views

4.1. Context View

4.1.1. System Runtime

In the presented architecture the TEE acts as a [sidecar](#) for a larger, existing system. The TEE is a [single TVM](#) where further internal isolation is done via standard Linux tools (concept of users, file permissions, ...).

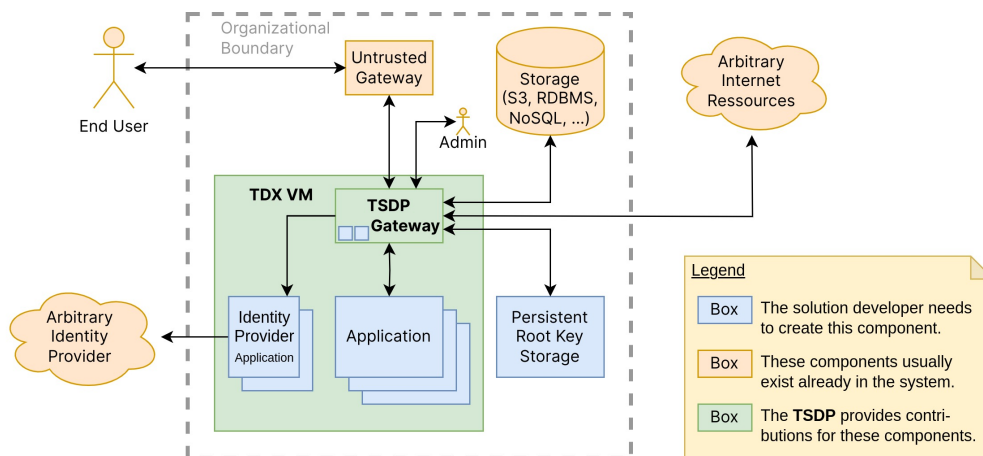


Table 2. Explanation of participants

Participant	Explanation
End User	Someone who is allowed to communicate with the TSDP Gateway to perform operations listed in Section 4.2.4.13 ^[2] . There are two types of End Users: <i>Stakeholders</i> ^[GAP-500] and <i>Dynamic End Users</i> . They use the TSDP Client Library to communicate with the TVM.
Stakeholder ^[GAP-500]	A user whose identity is hardcoded in the DFC. They can have a saying in operational lifecycle processes. Also, they could be authorized to perform more actions, in comparison to a <i>Dynamic End User</i> , as their identity is hardcoded in the DFC.
Dynamic End User	A user who can more easily be added or removed from the system as the authentication request is outsourced to a third-party trusted identity provider. These users have no saying in the operational lifecycle of a Solution Instance (like approving a DFC, or other).
Untrusted Gateway	The service provider likely has existing infrastructure which they manage in a common way. This includes a gateway, or proxy, which terminates TLS connections, processes incoming requests, etc. To make integration easier, the assumption is that this gateway is the only way to enter the system, i.e. if the end user wants to communicate with a component inside the TVM, its request needs to move through the untrusted gateway. Note: There does not have to be an untrusted gateway. The TSDP can interface directly with end users.

Participant	Explanation
Admin	The administrator of the system.
TVM	The TEE Virtual Machine isolated from the host infrastructure by hardware-enforced mechanisms to guarantee data confidentiality, data integrity, and code integrity for data in use. It hosts the confidential service. Our value proposition is that we provide tools to create this VM using the application(s), and the admin can deploy it.
TSDP Gateway	The TSDP Gateway is in front of the application(s). It manages remote attestation, authentication, authorization ^{[GAP-502][GAP-503]} , connects end users with applications and manages crypto keys, ideally performing transparent encryption (such that the application never sees the crypto keys). Especially for the transparent encryption, there is the idea that it can be enhanced with plugins, symbolised by the tiny rectangles. (In practice the TSDP Gateway is provided as a software library, a <i>Rust crate</i> , and the solution developer makes a binary out of that, compiling the crate and the "plugins" into one binary.)
Application	The applications contain the actual confidential analysis. They need to be written by the developer (i.e. are not directly part of the TSDP - the TSDP just enables easy development of applications). The goal is that they can be written in any language with an HTTP server and HTTP client, without being restricted from special SDK availability.
Identity Provider Application (IdP Application)	One or more separate applications which authenticate end users and provide means of authorization ^[GAP-503] for them. Since this is provided by the Solution Developer, it must be implementable in any language. This makes it easier to use existing SDKs. E.g. some identity provider may only provide a Java library, and another identity provider may only provide a Python library. In this case one could use two different identity provider applications, one in Java and one in Python. Note: This component is optional, if there only exists a handful of fixed end users which can be modeled as stakeholders.
Storage	For data persistence, external storage needs to be made available. TVM local storage is intended to be <i>fully ephemeral</i> ^[GAP-200] , i.e. lost after a shut-down.
Arbitrary Internet Resource	The applications may want to use other internet resources (e.g. another TVM).
Arbitrary Identity Provider	The IdP Applications may want to use external identity providers. Note that some identity providers have their own way of managing users (like GitHub), while others allow for a smaller domain which could be managed by some Stakeholder of the solution.

Table 3. Explanation of connections

Endpoint	Endpoint	Explanation
End User	Untrusted Gateway	This connection is the only entrance for the end user to the service. Thus this connection also tunnels messages meant for any component in the TVM.

Endpoint	Endpoint	Explanation
Untrusted Gateway	TSDP Gateway	The untrusted gateway forwards messages from end users, and sending own management requests through the same connection.
Admin	TSDP Gateway	Isolated connection(s) for monitoring, management, etc.
TSDP Gateway	Application	Applications don't open public ports themselves, but they get their messages from the TSDP Gateway. All the remote attestation, channel encryption and tunneling logic is hidden from the applications. The applications themselves just expose a regular HTTP service.
TSDP Gateway	IdP Application	For the authentication used during remote attestation by the TSDP Gateway, and extended authorization ^[GAP-503] demands within the Application.
TSDP Gateway	Storage	The TSDP Gateway may want to store encrypted keys or other data in some external data store.
TSDP Gateway	Arbitrary Internet Resources	The TSDP Gateway may act as a proxy for when the Applications want to communicate with external data sources.
IdP Application	Arbitrary Identity Provider	The IdP Application is likely (but not necessarily) connected to an existing identity provider. If all the relevant identity provider SDKs support proxies, then this connection could also be done through the TSDP Gateway.

4.1.2. Solution Instance Initialization

The first step of the solution instance initialization is the [DFC](#) setup. Since this is a communication between everyone and the Coordinator, this is not shown in the picture.

1. Coordinator and stakeholders agree upon a solution architecture.
2. Stakeholders^[GAP-500] create key pairs and send their public keys to the Coordinator.
3. The Coordinator puts together the DFC, except for the [VM identity](#).
4. Developers program all necessary components and create the TVM, a nixOS image with *qcow2 format*^[GAP-200]
5. Enforcers audit the source code and build process, *calculate the VM identity*^[GAP-001], sign it and send everything to the Coordinator. This includes:
 - Auditing the source code (maybe done by dedicated source code auditors),
 - Verifying that all the tasks as a whole, together with the role assignments, are correct,
 - Verifying out of band the correctness of the stakeholder public keys^[GAP-501],
 - Sending out of band their public keys to the other stakeholders (so they can configure these in their configuration)
 - Agreeing with the other Enforcers about the correct VM Identity,
 - Signing the DFC,

- Sending the **VM identity** and Signature to the Coordinator

6. Coordinator extends DFC by the **VM identity** and Enforcer signatures and sends out the finished DFC to all stakeholders and the Orchestrator.

Now the orchestrator can start the application and the rest happens. As the TVMs are currently considered *Fully Ephemeral*^[GAP-200] the case of persistence^[GAP-202] is not included.

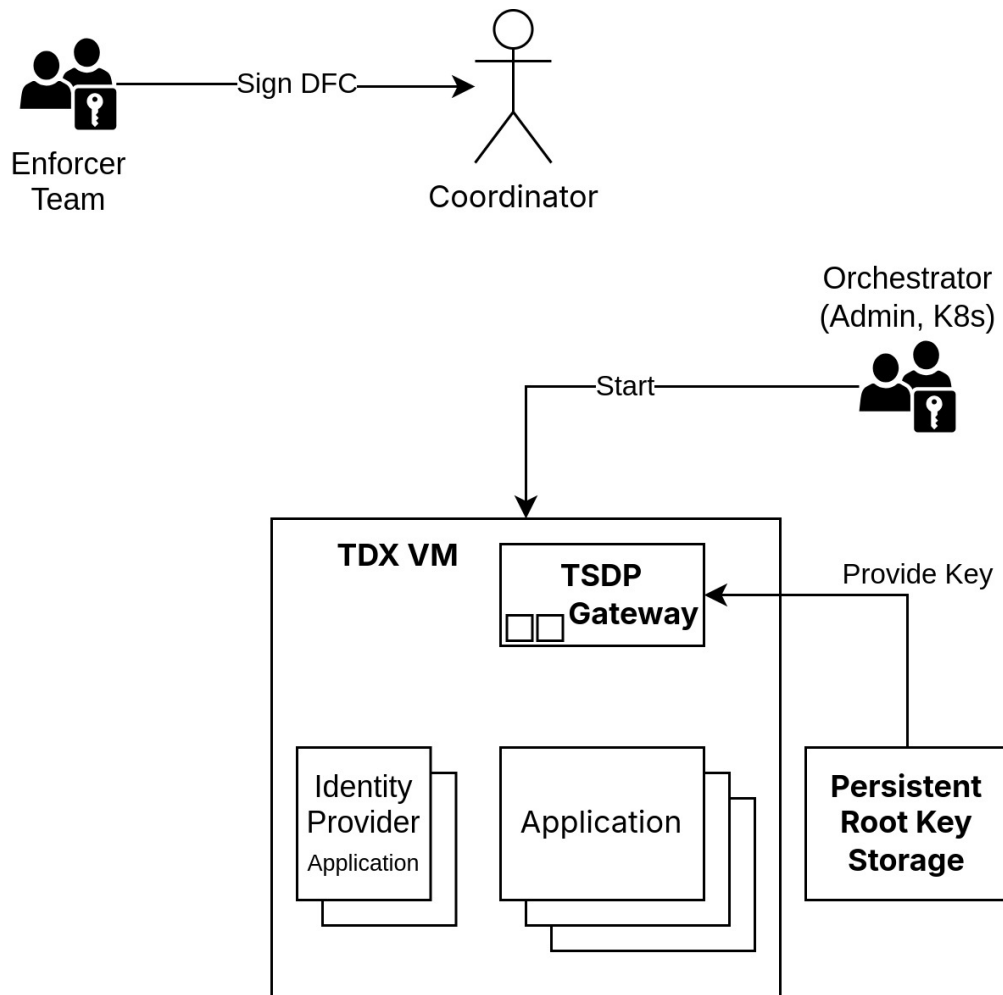


Figure 1. Solution Instance initialization steps after the setup of the DFC has been performed.

Table 4. Explanation of participants

Participant	Explanation
Coordinator	This is the legal entity which sets up the contract with all the other stakeholders. They organize the project from start to finish.
Enforcer Team	The enforcers need to make sure that the DFC is correct ^[GAP-001] . This means that correct permissions are given to the correct stakeholders and programs, according to the project contract. Enforcers may employ additional code auditors or similar to help verify the DFC .
Orchestrator	Administrators (likely with helper tools) which set up the infrastructure to run the Solution Instance, as well as monitor it. The TSDP Gateway can have a dedicated HTTP port for admin operations.

Table 5. Explanation of connections

Endpoint	Endpoint	Explanation
Enforcer	Coordinator	When an enforcers thinks that the DFC is correct ^[GAP-001] , they sign it. The DFC file, which contains the signature, is then sent to the Coordinator which will further forward it to the Orchestrator eventually.
Orchestrator	TVM	Starting the TVM and TSDP Gateway.

4.2. Functional View

4.2.1. Authentication & Authorization

Authentication

Associate a *Peer* with a unique identity. There are three types of peers, all having their own authentication principle:

Stakeholder

Identity is their name, and proof of identity is a private key. The name and public key are recorded in the [DFC](#), the TSDP Gateway recognizes them.

Dynamic End User

Identity is an identifier (name or some code) given by the Identity Provider Application. The TSDP Gateway forwards the authentication request to the Identity Provider Application. The proof of identity is out of control of the TSDP.

Application

Identity is the application name. It is part of the DFC and managed by the TSDP Gateway.

Authorization

Open Policy Agent (OPA)

All authorization [\[GAP-502\]](#)[\[GAP-503\]](#) questions are answered via [OPA](#).

4.2.2. Dataflow Configuration - DFC

The DFC describes means of authentication and authorization of end users and of the contents of the TVM. At the end of the following DFC example, each entries purpose is shown (see the numbers below).

```
[[stakeholders]] ①
name = "milten"
public_key = "hexencoded_pubkey"

[[stakeholders]] ①
name = "gorn"
public_key = "hexencoded_pubkey"

[identity_provider.smart_id] ②
path = ["/usr/bin/idP", "--some-arg"]

[identity_provider.another_one] ②
argv = ["/usr/bin/idP2"]

[applications.app1] ③
# The app can be called with command line arguments. This might help with
# reuse of VM images, where customization is controlled via the DFC.
# Working directory or the app is unspecified.
```

- ① Hardcode individual stakeholder identities
- ② Describe a [mechanism for authenticating](#) a dynamic set of end users.
- ③ Describe the identities of the applications (how they shall be started). These are the ones which the TSDP Gateway starts (and restarts when they die)
- ④ Describe the content of the TVM. For more details see [VM Identity](#) and [RA Quote Verification](#).
- ⑤ Currently, only Intel TDX and a mock TEE implementation are supported. The latter requires an empty `[mock_tee]` section instead of `[intel_tdx]` in the DFC.
- ⑥ Measurement registers that describe the TVM, some of these will change upon TVM rebuilds.
- ⑦ Quote Verification collateral can be collected from either a PCCS instance or from Intel's PCS.
- ⑧ For self-hosted PCCS instances, a self-signed HTTPS certificate must be added. If this value is not present, invalid certificates will be accepted during collateral collection.

- ⑨ Describe the [authorization policy](#) for all peers (stakeholders, dynamic end users, applications). This is based on the Open Policy Agent tool

4.2.3. Startup APIs

4.2.3.1. TVM

The TVM has multiple startup modes, depending on the chosen initialization mode. It really depends on the project and so only high level approaches are listed here:

Immediate Initialization

The TVM just needs to be started. The DFC and applications are already stored in the TVM. The [liveness endpoints](#) can be used to check whether the system is ready.

Lazy DFC Initialization

Like the Immediate Initialization, but then the [DFC needs to be sent](#) to the TSDP Gateway before any other endpoint can be queried.

Lazy TVM Content Initialization

A generic TSDP helper application starts which listens the DFC and other data (applications, configuration, etc). The LazyInit helper app verifies the application fingerprints^[GAP-300] according to the DFC file (this requires a different DFC content than is proposed here) and then continues as normal.

4.2.3.2. TSDP Gateway Library

The TSDP Gateway is a binary which is created on a project basis. Its interaction with the environment is thus *solution specific*.

The common functionality of the TSDP Gateway is wrapped in a reusable library (*crate* in Rust terms). This allows for a more flexible integration, trivial implementation of startup and shutdown logic, etc.

```

struct TsdGateway { .. /* everything private */ }

impl TsdGateway {
    fn new<A: ToSocketAddrs>(
        persistence: Box<dyn PersistencePlugin>,
        notify_stop: std::sync::Arc<tokio::sync::Notify>,
        root_key: RootKey, ①
        // Dfc type definition omitted for brevity
        dfc: Dfc, ②
        rego_policy: Path,
        port_for_users: A,
        port_for_admins: A,
    ) -> Result<TsdGateway, Box<dyn std::error::Error>>;

    // Blocks until `notify_stop` is activated.
    async fn run(&mut self) -> Result<(), Box<dyn std::error::Error>>;
}

struct RootKey([u8; 32]);

```

① Currently this root_key is created at runtime^[GAP-103].

② Currently DFC input is a path string^[GAP-301].

4.2.3.3. Application

An Application is started with the following environment variables:

- TSDP_GATEWAY_DOMAIN_NAME=localhost
- TSDP_GATEWAY_PORT=60001
- TSDP_MY_OWN_PORT=60002

The ports are used for communication. The application should listen on incoming HTTP/S traffic^[GAP-401] on port TSDP_MY_OWN_PORT.

4.2.3.4. IdP Application

The IdP Application is started with the following environment variables:

- TSDP_MY_OWN_PORT=60003

The port is used for communication. The IdP application should listen on incoming HTTP/S traffic^[GAP-401] on port TSDP_MY_OWN_PORT.

4.2.4. Runtime APIs

The following sections should illustrate the API patterns. They are not exhaustive.

4.2.4.1. Direct: End User → Untrusted Gateway

Protocol

Solution specific, likely HTTPS.

Transported Messages

Forwarding any message from the end user to the TVM.

Since this connection is solution specific, the TSDP client library **MUST** take in a callback to perform the IO. The callback is implemented by the solution developer who needs to transport the respective BLOB to the Untrusted Gateway. The message flow continues in [Section 4.2.4.3](#).



If the TSDP Gateway HTTP endpoint is exposed to the End User, then the End User could also talk directly to the TSDP Gateway using the [Untrusted Gateway → TSDP Gateway](#) messages.

4.2.4.2. Direct: End User ← Untrusted Gateway

The TSDP is not intended to send messages to the end user on its own behalf.

4.2.4.3. Direct: Untrusted Gateway → TSDP Gateway

Protocol

HTTP e.g. on Port 80, just a single URL endpoint. This way integration into the Untrusted Gateway should become easier.

Transported Messages

Request from End Users

A single JSON object as request and response whose forms are described in [Section 4.2.4.11](#) (JSON is available everywhere and we can change it if necessary). Note that no further information is transported via the URL, everything necessary is inside of the JSON object. This makes forwarding easier, i.e. integration into the Untrusted Gateway. Note that an error message might be present in the response header set which is meant for the administrator, but not for the end user.

Request

```
POST /endusermsg HTTP/1.1
Content-Type: application/json
Content-Length: 71

{
  "see": "Section 4.2.4.11",
  "just": "forward this request",
  "don't": "modify"
}
```

Response

```
HTTP/1.1 200 OK
Content-Length: 89
Content-Type: application/json
TSDP-Error-Message-For-Admin: some nice text

{
  "see": "Section 4.2.4.11",
  "just": "forward this response",
  "don't": "modify"
}
```



For administrative actions, another URL endpoint is provided to make sure that the Untrusted Gateway does not need to filter the "admin" messages out from the clients, see [Section 4.2.4.9](#).

4.2.4.4. Direct: Untrusted Gateway ← TSDP Gateway

None.

4.2.4.5. Direct: TSDP Gateway → Application

Protocol

HTTP on Port TSDP_MY_OWN_PORT. This means, applications don't require a special SDK, they just need a HTTP server library which should be available in the majority of serious languages.

Transported Messages

Solution specific, though with augmented headers, see below. As the TSDP Gateway just forwards the request from the user or another application, then the request schema is defined by the solution developer. See [Section 4.2.4.13](#).

No other messages are transported in this direction, i.e. applications don't need to implement some boilerplate API for management purposes.

Request message augmented headers

- shmtsdpeer-info from [OPA](#), hex encoded

4.2.4.6. Direct: TSDP Gateway ← Application

Note that these messages resemble the messages from [end users to the TSDP Gateway](#).

Protocol

HTTP on Port TSDP_GATEWAY_PORT

Transported Messages

POST /process/... [\[GAP-500\]](#)

Call the `process` function. Meant for performing cryptographic operations inside of the

TSDP Gateway, so the application does not need to deal with the keys.

POST /application/\$app_name/...
Send a message to a [peer application](#).

GET /authorize?idp=name_from_dfc&request=json_encoded_string^[GAP-502]
Forward an authorization request to the respective Identity Provider Application (idp parameter).

4.2.4.7. Direct: TSDP Gateway → Identity Provider Application

Protocol

HTTP on Port TSDP_MY_OWN_PORT.

Transported Messages

Authentication

This is used during the [remote attestation procedure](#) for Dynamic End Users. Peers should not require to call these functions themselves, so there is no mechanism available for that.

```
GET /authenticate?request=JSON_see_below HTTP/1.1
```

The response MUST be a JSON object.

```
type request = { ①  
  user_provided_identifier: string, ②  
};  
  
type response = object; ③
```

① In the success case this is any info which shall be associated with the dynamic end user, i.e. will be used as in the [OPA](#) input as `peer_info.dynamic`.

Authorization^[GAP-502]

This is forwarded [from the Application](#).

```
GET /authorize?from_application=appl&request=forwarded_from_the_application HTTP  
/1.1
```

The response is forwarded to the requesting application.

4.2.4.8. Direct: TSDP Gateway ← Identity Provider Application

No communication in this direction.

4.2.4.9. Direct: Admin → TSDP Gateway

The TVM has a dedicated port open for admin connections. This way it always knows

whether a request is coming from an end user or from an administrator. The surrounding infrastructure should ensure that end users don't have access to the admin port.

Protocol

HTTP on default port 8080.

Transported Messages

System Shutdown

See [Section 4.4.4](#).

VM initialization

The TVM might receive its DFC lazily, if no DFC can be found at some hard-coded location. In that case it waits for one incoming initialization request and only then continues with its startup, see [Section 4.2.3.1](#).

```
POST /init HTTP/1.1
Content-Type: application/json
Content-Length: allot

{
  "dfc": "...",
}
```

If the TVM was initialized this way, then the remote attestation quote covers the JSON object as part of its RTMR3 hash^[GAP-003].

Liveness Checks

These are designed after the Kubernetes [liveness, readiness and startup probes](#) (for HTTP they only support GET).

```
GET /health?probe=liveness&application=app1 HTTP/1.1
```

The probe query parameter can be any one of `startup` (can it start with processing requests?), `liveness` (is it still around to process requests?), `readiness` (is it currently busy and should temporarily not receive any new requests?).

If `application` is missing, then the TSDP Gateway answers these requests itself, otherwise the request is [forwarded](#) to the respective application at the endpoint `GET /tsdp/health?probe=liveness HTTP/1.1`.

4.2.4.10. Direct: Admin ← TSDP Gateway

Monitoring messages. Not specified yet.

4.2.4.11. Tunneled: End User → TSDP Gateway

This is tunneled through the Untrusted Gateway, see [Section 4.2.4.1](#) and [Section 4.2.4.3](#). The messages are JSON but MUST be treated as arbitrary BLOBs by the end user application and the Untrusted Gateway.

Note that some of these messages resemble the messages from [Applications to the TSDP Gateway](#).

Protocol

Custom JSON API.

Transported Messages

Remote Attestation Round 1 and 2

For details, refer to [remote attestation procedure](#).

```
// All binary fields are JSON arrays of decimal integers (0-255).

type Pem = {
  type_tag: string;           // e.g. "PUBLIC KEY"
  body: number[];
  headers: [string, string][];
};

type ra1_request = {
  function: "ra1";
  dh_public_key: Pem;         // Diffie-Hellman public key
  nonce: number[];           // 16 bytes
  info: number[];            // JSON bytes (see InfoA1)
};

type ra1_response = {
  function: "ra1";
  session_identifier: number; // u64
  dh_public_key: Pem;
  nonce: number[];           // 16 bytes
  info: number[];            // JSON bytes (see InfoB1)
} | public_error;

type ra2_request = {
  function: "ra2";
  session_identifier: number; // u64
  enc_payload: number[];     // encrypted attestation payload
  info: number[];            // JSON bytes (see InfoA2)
};

type ra2_response = {
  function: "ra2";
  session_identifier: number; // u64
  enc_payload: number[];     // encrypted attestation payload
  info: number[];            // JSON bytes (see InfoB2)
} | public_error;

// — info field contents (JSON-parse the byte array) —————

type InfoA1 = { A1: { key: Pem } };
```

```

type InfoB1 = { B1: { quote: number[] } };

type InfoA2 = {
  A2: {
    salt_ke1: number[];
    salt_km1: number[];
    peer: object;
  };
};

type InfoB2 = {
  B2: {
    salt_ke2: number[];
    salt_km2: number[];
    salt_ks: number[];
  };
};

```

Process ^[GAP-500]

This call is forwarded to `PersistencePlugin::process`, so this endpoint is typically only used if DEKs are involved somehow. For example the `process` function can create new DEKs, decrypt wrapped DEKs, encrypt something, decrypt something, whatever necessary.

```

type process_request = {
  function: "encmsg",
  // monotonically increasing, hence u64 sufficient
  session_identifier: number; // u64
  encrypted_payload: number[]; // AES-256-GCM encrypted JSON of EncmsgPayload
  payload_signature: number[]; // Ed25519 signature over encrypted_payload
};

// Presentation only - The response is raw opaque encrypted bytes (AES-256-GCM).
// The tsdp-client-lib decrypts it transparently.
// This shows the conceptual decrypted content:
type process_response_decrypted = {
  // Arbitrary content returned by the `process` function
  // (could be JSON, could be an HTTP response, etc.)
  ...any | encrypted_error,
} | public_error;

```

Notes on some of the fields:

info and enc_payload

These fields contain nested JSON/encrypted data. The `tsdp-client-lib` handles all serialization, encryption, and validation transparently. Implementers integrating with the library need only pass these through; implementers building a raw client should refer to the full protocol specification for cryptographic parameters.

session_identifier

u64. They are monotonically increasing (they should not leak any information). The TSDP Gateway creates this one during the first remote attestation round trip. This is used by the TSDP to look up the correct session key to decrypt the payload.

signature

An Ed25519 signature. In Encmsg requests, `payload_signature` covers the `encrypted_payload` field. Inside the encrypted RA2 `enc_payload`, the `SignatureAndHmac` struct contains an Ed25519 signature field.

enc_payload / encrypted_payload

Encrypted with keys derived from the Diffie-Hellman shared secret. In RA2, encrypted with round-specific keys (`enc_key_1/enc_key_2`). In Encmsg, encrypted with the session key.

public_error and encrypted_error

Depending on the availability of a working secure communication channel the error is encrypted or plaintext.

4.2.4.12. Tunneled: End User ← TSDP Gateway

None.

4.2.4.13. Tunneled: End User → Application

This is tunneled through the TSDP Gateway.

Protocol

Custom format, which ends up at the HTTP server of the Application. The client side uses an API which (remotely) resembles the [Fetch API](#).

Transported Messages

Custom messages, e.g. the end user ideally uses it just like any other HTTP service, and the application puts the respective handlers/routes in place.

```
// typescript - client side usage of the library.
const response = await session.query_application(
"$app_name", "/some/url/", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(data),
});
```

4.2.4.14. Tunneled: End User ← Application

None.

4.2.4.15. Tunneled: Admin → Application

This is tunneled through the TSDP Gateway, see [Section 4.2.4.9](#).

Protocol

HTTP

Transported Messages

Liveness Checks

Also see [Liveness Checks](#).

```
GET /tsdp/health?probe=liveness HTTP/1.1
```

The probe query parameter can be any one of `liveness`, `readiness` and `startup`.

4.2.4.16. Tunneled: Application → Application

Protocol

HTTP.

Transported Messages

Solution specific. Tunneling is done through the `/application/$app_name` endpoint of the [TSDP Gateway](#). Therefore, they don't communicate directly but through the TSDP Gateway.

4.2.4.17. Tunneled: Application → Identity Provider Application

Protocol

Custom HTTP requests.

Transported Messages

Custom JSON messages.

Authorization

[To the TSDP Gateway](#) and from there [to the IdP Application](#)^[GAP-503].

4.3. Trust Building

The TSDP is about the following security statement: *The sensitive data of producers is protected at all times*. Also see [Section 1.3](#) for a longer text. This section lists at what locations the sensitive data is used, and hence to what parts of the system the trust of the producer must extend.

4.3.1. Usage Sites of Sensitive Data

Life Cycle Step	Protection Measures for Sensitive Data
During Creation	At the data owner, so allowed to be under control of a single human being.
During Transit (Data Owner ↔ Application)	Encrypted via an ephemeral communication channel with the TSDP Gateway. The communication channel keys are generated during remote attestation and never leave the client device or TVM. The authenticity of the Application is enforced via remote attestation and the DFC.
During Processing	In the Application, hence protected by the TEE.
During Storage	Currently TSDP does not provide any long term storage for data.

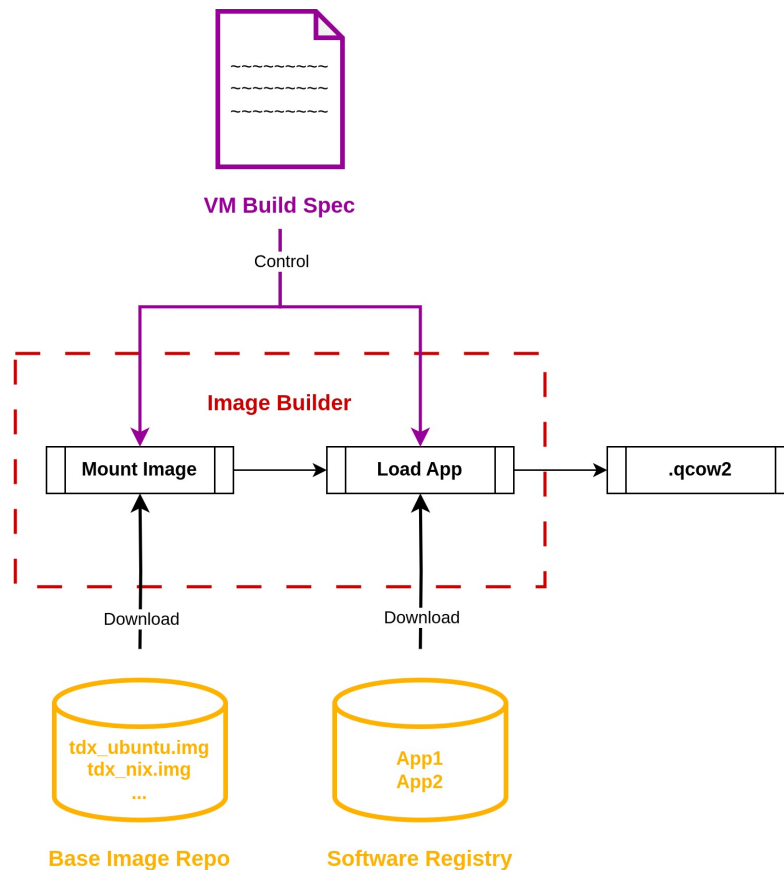
4.3.1.1. Usage Sites of the Root Key

Life Cycle Step	Protection Measures for the Root Key
During Creation	Created inside of the TVM at runtime ^[GAP-103] .
During Transit	Does not leave the TEE.
During Processing	Only used for encryption inside of the TVM.
During Storage (TVM)	Not stored long term.

4.4. Operational View

4.4.1. TVM Assembly^[GAP-303]

The TVM will be assembled with the *Image Builder* program. It assembles all input artifacts into a `.qcow2` file. Our example for the *TVM Build Spec* is to use nix configuration and build where the target is a nix TVM. Note: This approach suits all of the [startup modes](#). Its just a question of what Apps to put into the TVM.



4.4.2. TVM Deployment

This is solution specific. Simpler solutions might work with `systemd` service file which runs the `qemu` command directly, more complicated solutions want to use Kubernetes. However, confidential computing support for Kubernetes is still in the making, see for example: <https://github.com/cc-api/kubevirt-cvm>

4.4.3. Monitoring

The TSDP Gateway provides monitoring capabilities, and apps can publish monitoring endpoints themselves.

4.4.4. TSDP Shutdown ^{[GAP-203][GAP-303]}

Stopping the container is done with an HTTP message, as per the Kubernetes `preStop` [hook \(spec\)](#).

- TSDP Gateway has a "GET /admin" endpoint which accepts the arguments `?action=preStopHook`. Kubernetes only allows a HTTP Get here, so cannot send a JSON body (see the above spec reference).
- TSDP Gateway then sends special GET `/tsdp?action=preStopHook` and SIGTERM to all the applications.
- The operator (Kubernetes or whatever called `qemu`) kills the VM.

The TSDP Gateway application might want to itself determine that it should shut down. In that case the `TsdGatewayBuilder` could get a `with_graceful_shutdown` (like in `hyper`).

4.4.5. TVM Upgrade

Within this architecture, TDX VMs are immutable. Their user space applications cannot be updated. If an update is required, a new VM needs to be built and run, replacing the old running TDX VM. The replacement strategy is solution specific, as it may or may not need to be highly available.

5. System Qualities

5.1. Performance

Requirement	How Met
Performance	While testing and benchmarking TEE Integrations (https://cros.ec.europa.eu/book-page/joconde D5.1 - Demonstration Prototype, Test Cases and Test Results chapter 4.1.4) we have proven that TSDP framework operates with comparable speeds to Intel's own publications (link: https://www.intel.com/content/www/us/en/developer/articles/technical/trust-domain-extensions-on-4th-gen-xeon-processors.html)

6. Gap Registry

This section documents known divergences between the architectural vision described in this document and the current implementation of the Sharemind TSDP solution.

6.1. Referencing Gaps

Gap references are superscripts containing the relevant gap id/s and link/s to it.

Single gap reference example: ^[GAP-XXX]

Multiple gap references example: ^{[GAP-XXX][GAP-YYY]}

6.2. Gap Types

- **Stepping stone::** Transitional solution on the path to the vision
- **Known technical debt::** Intentional simplification with planned remediation
- **Alternative solution::** Vision replaced by a different approach and is under evaluation

6.3. Current Gaps

The gaps are organized by feature area below.

Organized gaps in these subgroups with ranged IDs based on related functionality:

- **0.Remote Attestation & Trust** - IDs in range {000:099}
- **1.Key Management & Cryptography** - IDs in range {100:199}
- **2.TVM Lifecycle & Persistence** - IDs in range {200:299}
- **3.Image Building & Initialization** - IDs in range {300:399}
- **4.Network & Communication** - IDs in range {400:499}
- **5.Authorization & Identity** - IDs in range {500:599}

6.3.1. Remote Attestation & Trust

Gap ID	GAP-001
Reference	TVM Identity hash calculation
Type	Alternative solution
Vision	Enforcers calculate/measure the VM Identity.
Current	The TVM itself reports the measurement values using industry standard remote attestation tools. There was no easy way to resolve measurement of the VM Identity, we decided to lean on the existing attestation components and simply report the measured values. Issue with this: adds additional requirement to Enforcers as it ties the measurement to TEE HARDWARE.
Remediation	TODO: Do better! Test nix's hash calculation. Match this with Intel TDX's quote generation and values can be calculated at build time.

Gap ID	GAP-003
Reference	Use of the RTMR3 register and supporting measurements
Type	Known technical debt
Vision	RTMR3 register is used to track any software addition between boot to RA and later extended by persistent use-cases via IMA/dm_verity measurements.
Current	RTMR3 measurement is not used. LazyInit does not calculate package hash and appends this to RTMR3 or to the reportdata. Persistence TVM use-cases are not in scope ATM.
Remediation	Feature addition is planned in the future.

6.3.2. Key Management & Cryptography

Gap ID	GAP-101
Reference	Encryption via TSDP Gateway
Type	Known technical debt
Vision	TSDP Gateway provides means to encrypt data, applications themselves should not encrypt data.
Current	In all currently existing TEE integrations, the in-TEE applications data has been encrypted by the application due to the use of containerization or other secure computing technologies (e.g., multi-party computation with secret sharing). Both of these could be considered outside of the intended vision.
Remediation	TODO: Understand how TSDP Gateway performs encryption on behalf of the in-TEE applications.

Gap ID	GAP-103
Reference	Root Key Creation
Type	Stepping stone
Vision	Root key is created preferably at boot and it is given(somehow, e.g: env var) to the TSDP Gateway.
Current	The TSDP Gateway itself creates the root key.
Remediation	Will improve this in the future as we gain more detailed understanding of root keys requirements.

Gap ID	GAP-104
Reference	Extension of unforeseen encryption schemes
Type	Known technical debt
Vision	A solution should be able to make use of encryption schemes which were not foreseen by the Sharemind team.
Current	Current solution uses systems cryptographic libraries like OpenSSH or libsodium to handle required and generic crypto. Stakeholders would require to modify these underlying components or source code of TSDP to introduce their own crypto. The current system is simpler and modular which allows for easy swap of used/integrated crypto library.
Remediation	Feature addition is planned in the future.

6.3.3. TVM Lifecycle & Persistence

Gap ID	GAP-200
Reference	Ephemeral TVM
Type	Known technical debt
Vision	TVM local storage is intended to be <i>fully ephemeral</i> , i.e. lost after a shut-down, unless the use case doesn't allow this (Terabytes of data).
Current	Current TVM solution does not enforce this intended <i>fully ephemeral</i> behavior, TVM Orchestration manages lifetime and cleanup of images, even if otherwise <i>ephemeral</i> VMs would be achievable, not constrained by RAM availability.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-202
Reference	Persistence for TVMs
Type	Known technical debt
Vision	TVMs provide means to persist their state, data in a secure recoverable manner.
Current	No persistence features are implemented for TVMs, they are treated as ephemeral.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-203
Reference	Stopping the TVM (notify_stop)
Type	Stepping stone
Vision	Vision details the stopping of the container via dedicated HTTP messages (notify_stop) in a manner where it is compatible with Kubernetes.
Current	While some WIP implementation of the <code>notify_stop</code> endpoint exist, it is not tested and used. In reality due to residual cached image file cleanups these steps are done via orchestration steps. In addition more analysis suggested that this area is a hot topic as termination and cleanup will be vulnerable to attacks by the untrusted host who could block or even cancel termination requests by intercepting them. Essentially untrusted host can keep a TVM alive. Need to research more into this topic.
Remediation	Feature addition is planned in the future.

6.3.4. Image Building & Initialization

Gap ID	GAP-300
Reference	LazyInit to verify initialised content (applications)
Type	Stepping stone
Vision	The LazyInit helper app verifies the application fingerprints according to the DFC file. This goal has some overlap with [GAP-003] !
Current	Currently it is expected that uploaded lazyinit data is auditable (Ok for open source scripts and configuration files). Proprietary data is not auditable. Application fingerprints are not verified by the LazyInit helper.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-301
Reference	DFC Input
Type	Stepping stone
Vision	The DFC argument to be used as serialized input.
Current	Currently DFC is used as a string path input only which requires caching the DFC file to drive. In hindsight if VM image caching is eliminated/fixed then what form DFC is shared won't matter.
Remediation	TODO: Fix VM image caching.

Gap ID	GAP-303
Reference	Image Builder and Distributor
Type	Stepping stone
Vision	Vision describes an application or service that takes the role of TVM Image building and distribution to supply orchestration of TVMs.
Current	In reality we are using build scripts that prepare inputs and delegate to or is resolved with nix build. Image distribution is achieved via infrastructure tools like Ansible.
Remediation	Feature addition is planned in the future.

6.3.5. Network & Communication

Gap ID	GAP-401
Reference	HTTP vs HTTPS traffic
Type	Stepping stone
Vision	In production use HTTPS traffic where possible for security.
Current	To aid development efforts initial integrations use HTTP traffic. Where required after further security assessments traffic is upgraded to HTTPS, at first with self signed certificates. For real production self signed certificates and their use have to be swapped out with real ones.
Remediation	Carry out assessments and transition to HTTPS where required.

Gap ID	GAP-402
Reference	Legacy ready VPN and RA
Type	Stepping stone
Vision	Remote Attestation sets up VPN tunnels as part of its process.
Current	Currently functionality is not integrated into TSDP. In the JOCONDE prototype Wireguard channels were hard coded in the setup due to lack of time.
Remediation	Feature addition is planned in the future.

6.3.6. Authorization & Identity

Gap ID	GAP-500
Reference	Stakeholder peers
Type	Known technical debt
Vision	Stakeholders are main peers of the system and are used throughout.
Current	With the addition of IdP functionality stakeholder peers became non functional. Every user is considered dynamic peer. Conflict between listing of stakeholders peers and dynamic peers led to the current status. This issue also blocks persistence ^[GAP-202] callback test and TSDP Gateway encryption ^[GAP-101] use cases as the callback was initially designed to be only used by stakeholders.
Remediation	Fix is planned in the future.

Gap ID	GAP-501
Reference	Stakeholder key verification
Type	Known technical debt
Vision	Stakeholder public keys are verified out of band for correctness.
Current	Currently stakeholder keys are not verified.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-502
Reference	Authorization through IdP and OPA
Type	Known technical debt
Vision	TSDP Gateway integrates authorization with the use of OPA and IdP mock.
Current	Current solution implements only an ALLOW ALL authorization policy via OPA.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-503
Reference	Authorization via IdP Application
Type	Known technical debt
Vision	Applications perform authorization via the IdP app when needed.
Current	The IdP mock does not implement authorization. Authorization use-case needs to be understood better.
Remediation	Feature addition is planned in the future.

Gap ID	GAP-504
Reference	Dynamic user authentication
Type	Known technical debt
Vision	Dynamic users are authenticated via IdP which successful result contains their public key.
Current	The IdP mocks successful authentication response that does not contain public key.
Remediation	Feature addition is planned in the future.

Appendix A: Technical Details

A.1. Trusted Execution Environments

This section contains a short primer on TEE technology. A significantly broader overview can be found in [D2.1 \(section 3\) of the JOCONDE project^{\[3\]}](#).

Sensitive data at rest and in transit can be protected with commonly available industry grade encryption. However, in traditional applications sensitive data in use—encryption keys, business secrets—is not encrypted. While the operating system can offer some level of protection, nothing protects from the components (hypervisor, OS) or participants (the administrator) being malicious or compromised. TEE technology provides an extra layer of protection for sensitive data while it is being processed. Therefore, the working memory of the software which is protected by the TEE technology is encrypted and tampering will be detected.

One important area for TEE technology is cloud computing, where organisations process sensitive data in the cloud. TEE technology protects the sensitive data from the cloud operator itself, thus providing a base for compliance with various data protection regulations. Remote users, like workload owners, can use *Remote Attestation* to cryptographically verify that the correct software is running on the cloud servers, without physical access to the cloud servers. As a result, a remote user can trust that the remote software runs the expected code in a secure environment. Additionally, *Remote Attestation* provides the user with a means to create a secure communication channel. Hence the user can then upload their sensitive data over the newly created secure communication channel for further processing.

There are two kinds of TEE technologies:

VM-based TEE technologies

A VM with all its components is protected from its environment. This means that regular applications can be protected with little or no changes needed. Examples are Intel® TDX, AMD SEV-SNP, and more. The TSDP architecture is primarily targeting Intel® TDX, but will also work with similar VM-based TEE technologies.

Process-based TEE technologies

A submodule of an application is protected from the environment. This means that applications need to be divided into a trusted and an untrusted part. The most important example here is Intel® SGX.

VM Identity

One important aspect is how a TEE technology represents the information about what software is running inside of the TEE. In Intel® SGX this is done by hashing the `.so` file which is loaded into the TEE, and this hash is presented during remote attestation as the so-called **MRENCLAVE** value. For VMs, the **VM Identity** (as it is called in this document) is more complicated. See details how VM Identity is defined in [remote attestation](#).

A.2. Architecture Alternative Overview

A.2.1. TEE Granularity

The TEE technology can be used in various ways to provide isolation, as shown in [Figure 2](#).

Keep in mind that the coarse grained, single TVM approach still (1) protects against spying intruders and (2) allows for remote attestation. It is just for application separation that one might ask whether TEE isolation is required for them or not, and given the trade-offs it is easier to just have a single TVM.

	Pro	Con
TEE Isolation (Fine grained TEE)	Arguably safer: Compromising one application is likely no help in compromising the next application.	Likely incompatible with orchestrators like Kubernetes. Higher host system resource consumption due to increased TVM count. Higher host system resource consumption (at least twice the price?).
Process Isolation (One TEE)	Easier to orchestrate (i.e. integrate), program and explain.	Less safe: Compromising one application means that other applications are not behind TEE boundaries anymore. More work is required for application sandboxing, if required.
Module Isolation (One TEE)	Easier to orchestrate (i.e. integrate) and explain. Stronger isolation through sandboxing.	WASM modules are similarly restricted like Intel® SGX enclaves, so one cannot use arbitrary languages and libraries for application development.

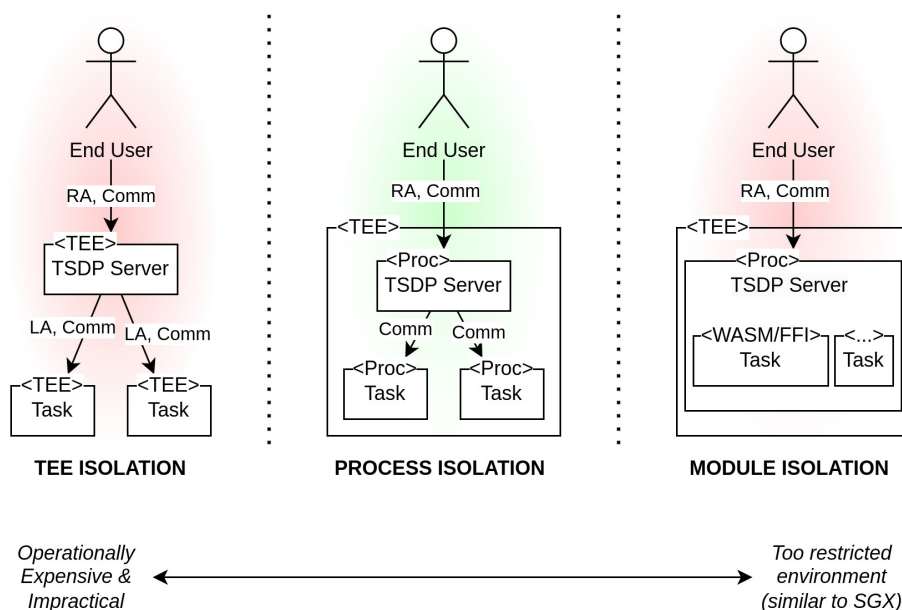


Figure 2. TEE Separation was used in Sharemind HI, where each application ("task") lived in a separate Intel® SGX enclave. In the TSDP the TEE technology is used to (1) shield against unwanted uses from outside and (2) allow for easier remote attestation (one quote to cover all software), but for the separation of applications from each other one can use tooling provided by the OS (processes, users, file permissions, ...).

A.2.2. Connection of Application And Remote Attestation

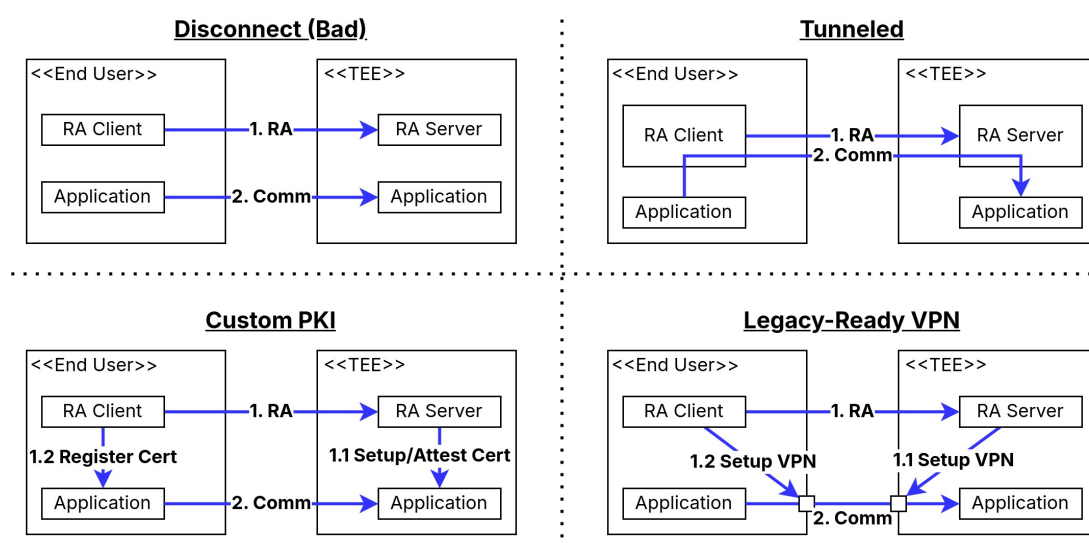


Figure 3. There are various ways how the application (a.k.a. *Business Logic*) is connected to the remote attestation logic. Note that the remote attestation components and the applications components could also live within the same process, i.e. the boxes do not represent isolation.

Type	Explanation
Disconnect Example: Keylime	The remote attestation components perform remote attestation. The applications communicate with each other. Problem is that the client side applications don't know that the TEE-side application is really in the TEE. Usually this is done in a way where the machines are just set up like legacy machines (humans/bash scripts do the manual part of connecting the RA confidence to the communication channel) and Keylime is used to monitor them. Pros: Simple, everything is standalone (allows legacy applications). Cons: There is no machine proof that the application is really running within the attested TEE.
Tunneled Example: Sharemind HI, TSDP	The client application fully relies on the remote attestation components to encrypt and forward their request to the TEE-side application. Cons: Requires that at least the client-side application is aware of the TEE context.
Custom PKI	Existing applications usually communicate via REST and handle encryption themselves via TLS. The client and server sides usually can be configured to use/accept special self-signed certificates. These certificates can be generated, exchanged and validated in advance during remote attestation. Pros: Works with legacy applications. NB: Similar to this approach is RA-TLS, but 2. Comm implicitly does 1. RA. Cons: At least the way how RA-TLS is implemented (2. Comm implicitly does 1. RA), this required respective support in the client side TLS library.
Legacy-Ready VPN Example: Sharemind MPC on TSDP	RA components use the derived key to set up a VPN^[GAP-402] for encryption with a <i>Pre-Shared Key</i>, e.g. Wireguard. The protection is fully transparent. The applications then just communicate as usual, they are oblivious to the surrounding TEE. Pros: Fully transparent, even works for applications which do not use TLS. Cons: Does not work for browser environments (there is no VPN), so does not work for TSDP (Only if the <i>RA Client</i> lives outside of the browser this might work, but requires additional setup / steps by the end user instead of just opening a web site).

A.3. Remote Attestation

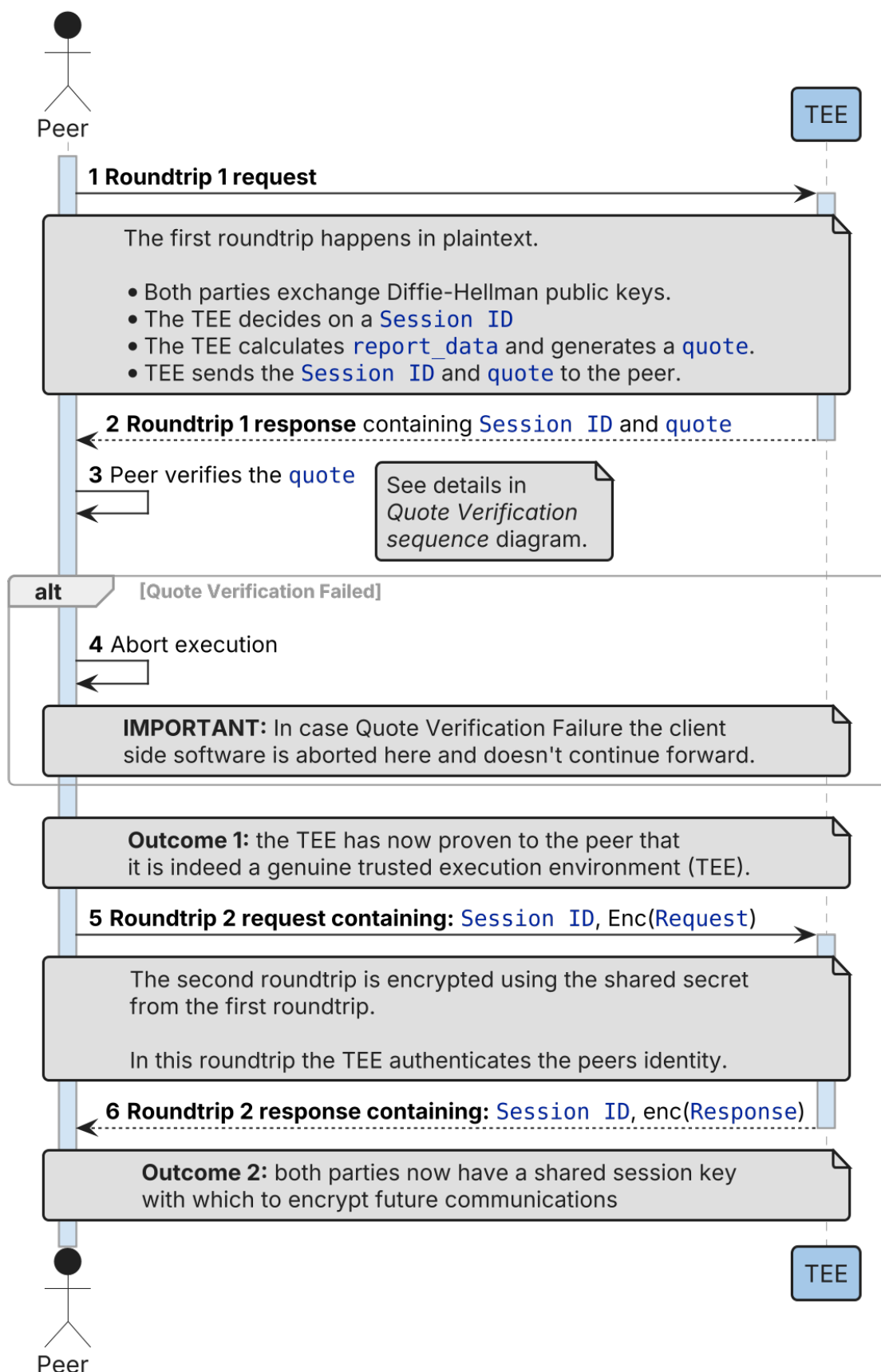
A.3.1. Round-trip Schema

The goal of the remote attestation process is for the confidential computing environment (the TEE) to demonstrate to a relying party (the human or other peers), that they are indeed a genuine, secure TEE running the expected configuration and code.

In TSDP, remote attestation takes place in two roundtrips:

- Roundtrip one (RA1) - Connection establishment with quote generation and verification.
- Roundtrip two (RA2) - Secure channel establishment and peer authentication.

Remote Attestation - Simplified Overview



SIGMA Authenticated Key Exchange with Quote Verification

A.3.2. SIGMA-Protocol Authenticated Key Exchange

The SIGMA key exchange protocol authenticates the TEE to the peer and establishes a shared session key. Once established, this key encrypts all communication for the remainder of the attestation flow, including the identity provider authentication step that follows. Following remote attestation, the key is commonly used in one of these ways:

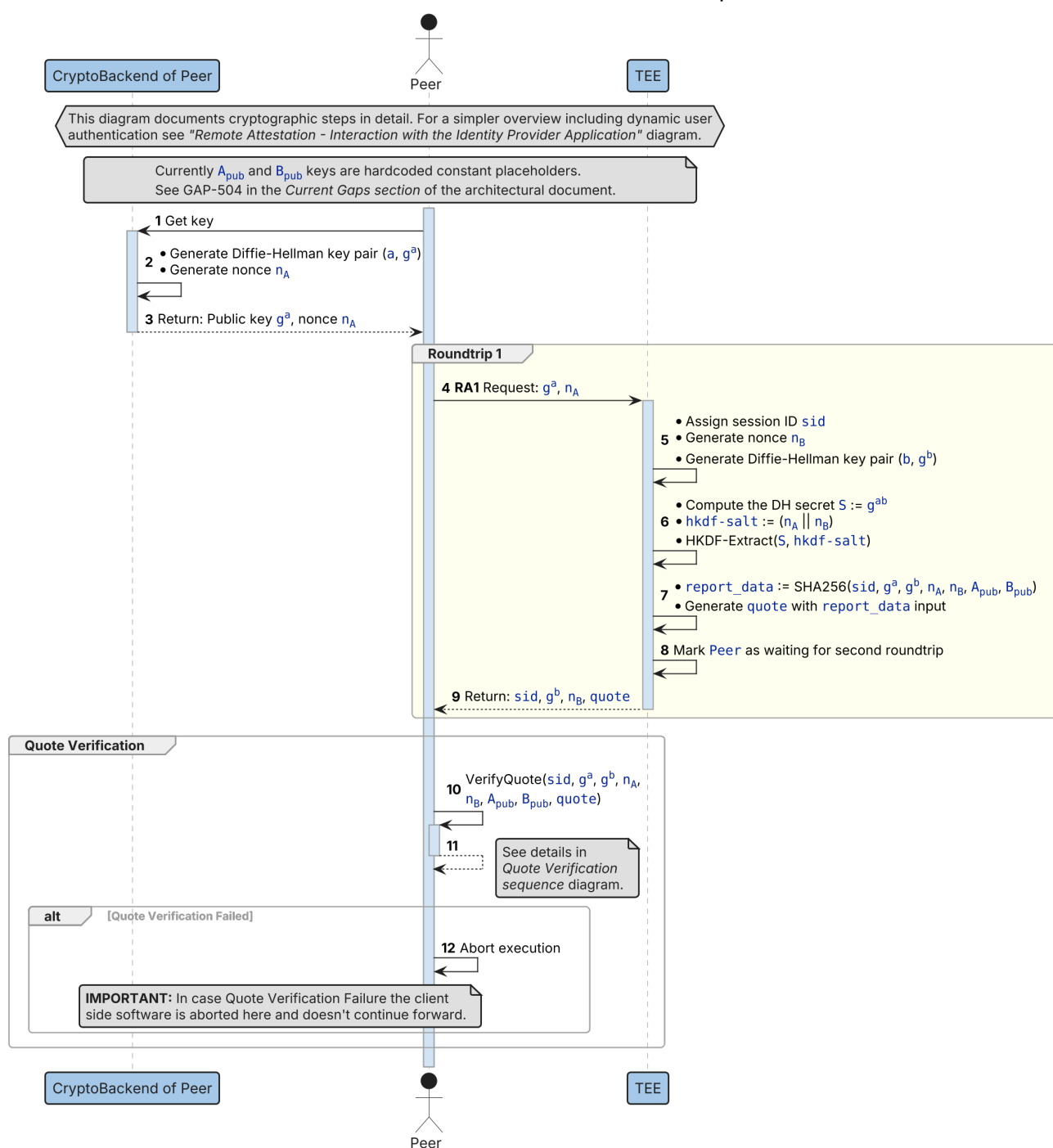
- for sending a single encrypted message
- for establishing an encryption channel which is used later on
- for setting up VPN tunnels^[GAP-402] with the peer (any communication can then flow through these hardware encrypted tunnels)

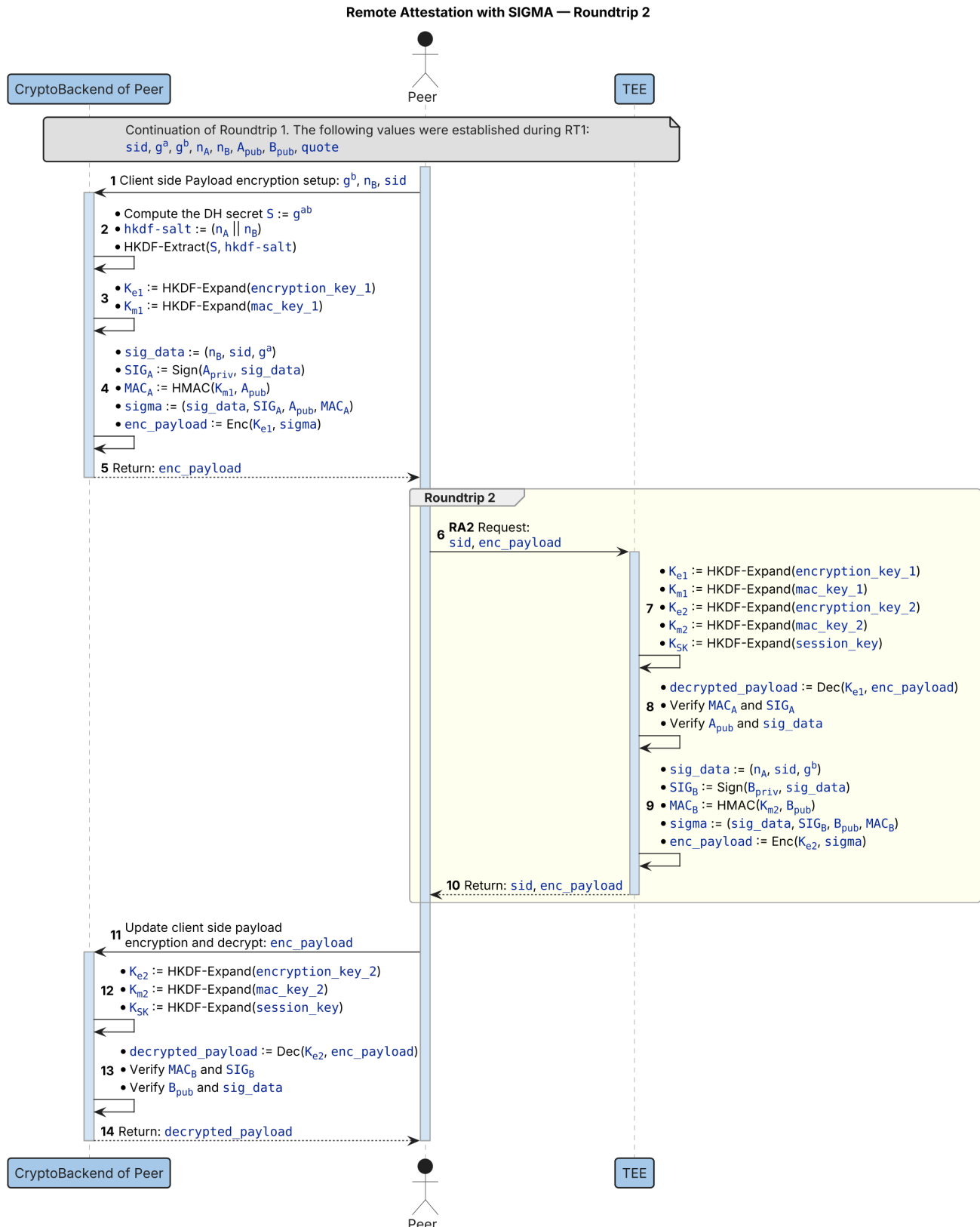
The key exchange protocol and its security properties (what kind of encryption to use, what do values sign and how, etc) are described in-depth in the [SIGn-and-MAC](#) (SIGMA for short) paper. Intel's standard [EPID remote attestation protocol](#)^[4] was based on section 5.1 of the paper—the basic version of the protocol. Some additional modifications are described [in this presentation](#).

For deriving new keys during the exchange, HKDF-Extract is called once on the Diffie-Hellman secret and HKDF-Expand is then used multiple times—see [this StackExchange answer](#) for a good brief explanation on how HKDF is meant to be used.

The protocol specifies that the encryption used must be safe against active attackers. For this, we use AES-256-GCM with all keys derived from HKDF-Expand.

Remote Attestation with SIGMA — Roundtrip 1

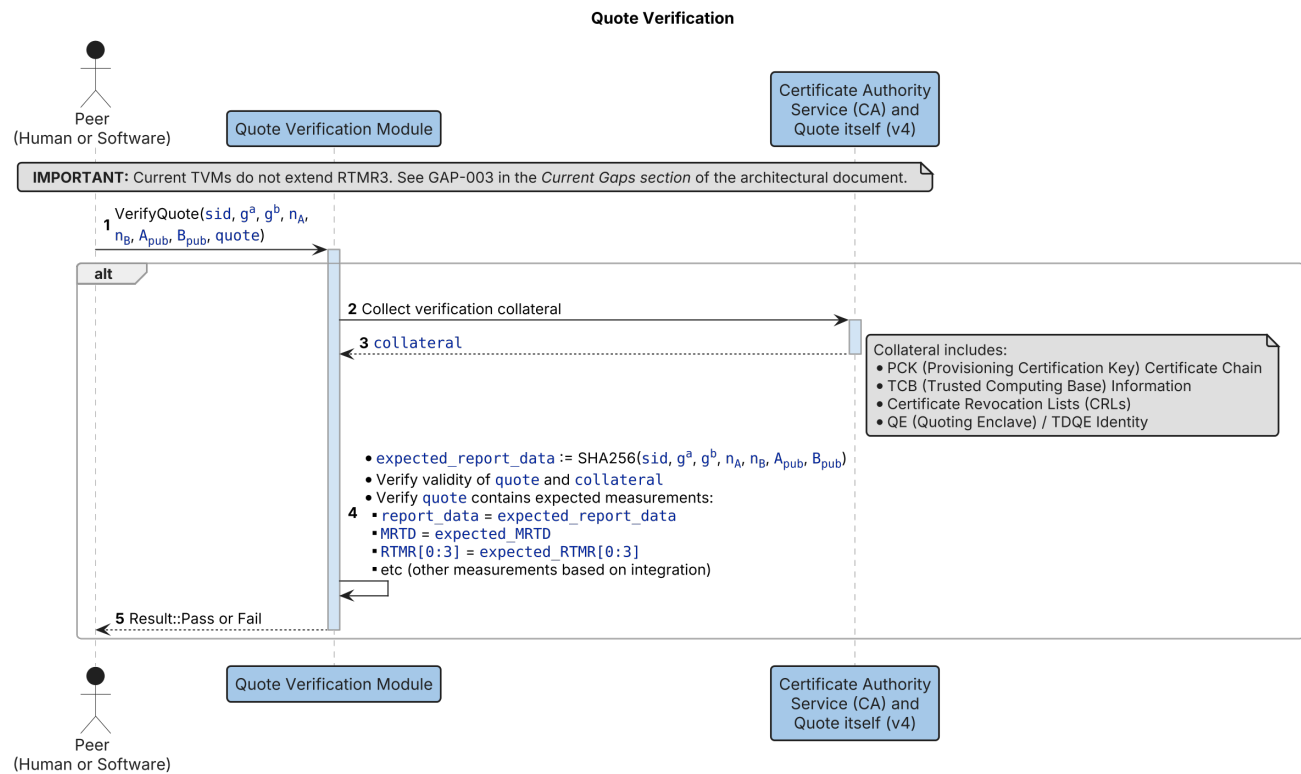




In other sequence diagrams, we abstract over the complexities of the process by using a similar simplified schema with two basic roundtrips, for the SIGMA rounds and quote verification.

A.3.3. Quote Verification

In the first roundtrip the TEE generates a quote that is shared with the peer. The report data field of the quote is set to a hash value that binds the quote to this session, peer, and specific TEE. During quote verification the peer verifies the quote itself and its containing measurement values.



Values the client expects to see are defined in DFC, with one section per TEE vendor:

```
[intel_tdx] ①

# Expected quote fields ②
mr_enclave = "E5A3A7B5D830C2953B98534C6C59A3A34FDC34E933F7F5898F0A85CF08846BCA"
mr_config_id = "0000000000000000000000000000000000000000000000000000000000000000..."
mr_owner = "0000000000000000000000000000000000000000000000000000000000000000..."
mr_signer = "DC9E2A7C6F948F17474E34A7FC43ED030F7C1563F1BABDDF6340C82E0E54A8C5"
mrtid = "4AA4768F20CD8D208E459E6BC90E42F545B8DA9017E22E1A5469E6370F225A43..."
rtmr0 = "CC0D249781E7A0B9297977C1C454193DA2B9CD29B1964572F23ADC2F3FDB7E75..."
rtmr1 = "4C6178002E3034BA6E8DD2BCC43F54730D610C4D3B4727386BA1B77B92FBDC9C..."
rtmr2 = "9288BB7924179E225A23F30D2700DF6158210FD7DDC2C8D537CAC39A9BBA7CBE..."
rtmr3 = "0000000000000000000000000000000000000000000000000000000000000000..."

collateral_url = "https://localhost:8081" ③
pccs_certificate = '' ④
-----BEGIN CERTIFICATE-----
MIIJIDCCB4igAwIBAgIQOTDpDP44ESNuCDL7Q1cVezANBgkqhkiG9w0BAQsFA...
-----END CERTIFICATE-----
'''
```

```
# Intel's root CA CRL
root_ca_crl = '''
-----BEGIN X509 CRL-----
MIIBIjCBYAIbATAKBggqhkhjOPQQDAjBoMRowGAYDVQQDDBFJbnRlbCBTR1ggU...
-----END X509 CRL-----
'''
```

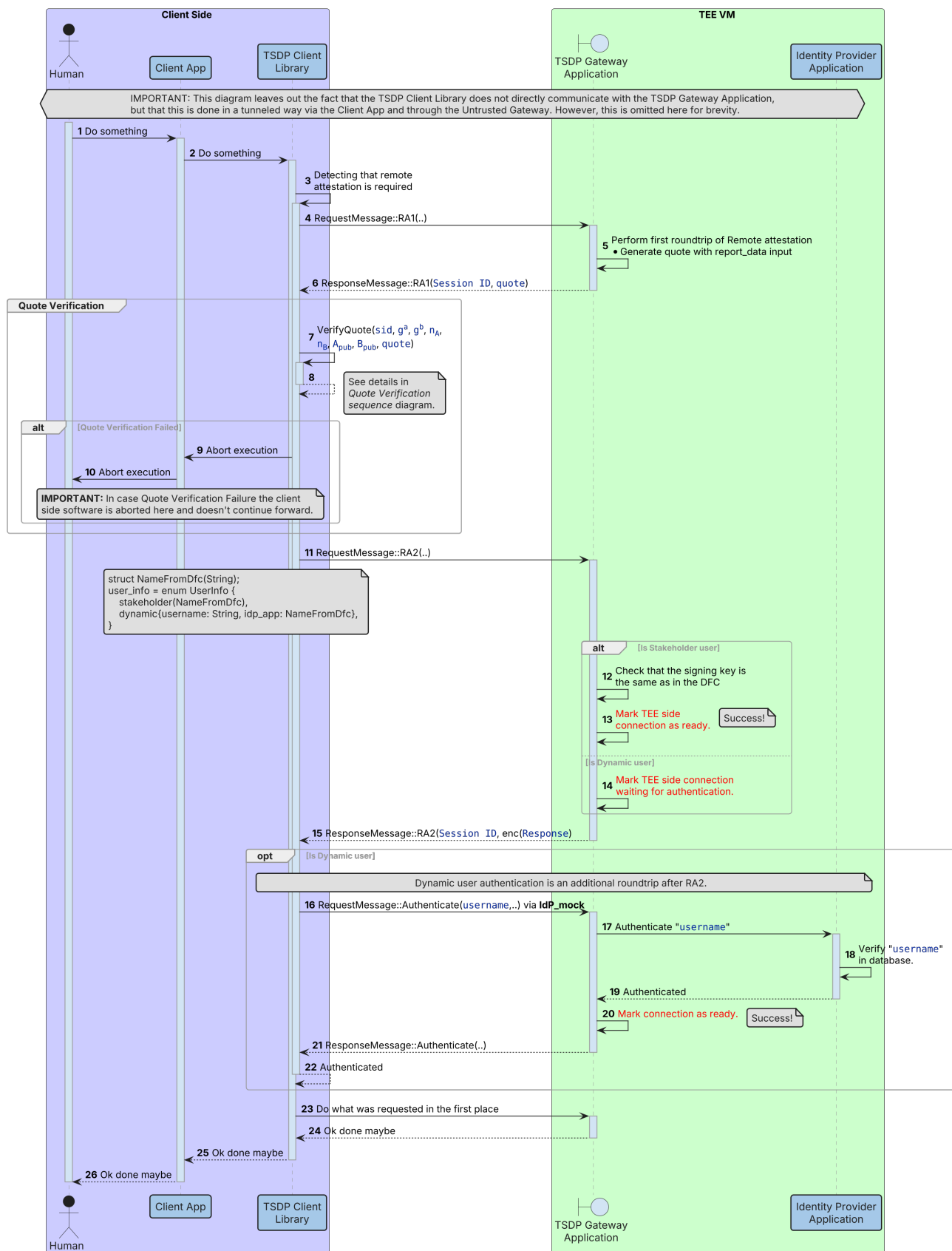
- ① Currently, only Intel TDX and a mock TEE implementation are supported. The latter doesn't require a section in the DFC
- ② Some of these will change upon VM rebuilds
- ③ Collateral can be collected from either a PCCS instance or from Intel's PCS
- ④ For self-hosted PCCS instances, a self-signed HTTPS certificate must be added. If this value is not present, invalid certificates will be accepted during collateral collection

A.3.4. Authentication via Identity Provider

While technically not a part of remote attestation, user authentication is tightly coupled with the process and relevant to the attestation flow. In this stage, the peer authenticates itself to the TEE, depending on their type:

- dynamic peers authenticate through an identity provider application
- for stakeholders, the TEE checks that their signing key^[GAP-501] is the same as in the DFC

Remote Attestation - Interaction with the Identity Provider Application



A.4. Authorization via Open Policy Agent^[GAP-502]

Open Policy Agent (OPA) is a tool which takes as input a JSON file and a program written in a domain specific language, and outputs a transformed JSON (mostly just a boolean `allow/not-allow` or a list of strings comprising error messages), tailored for the authorization use case.

In TSDP an OPA program is part of the DFC, and it is consulted for authorization purposes whenever a client [sends an encmsg query](#) to the TSDP Gateway (or the [Application to the TSDP Gateway](#) - if the Application bypasses the TSDP Gateway, then OPA is also sidestepped).

```
type input =
{
  request: request,
  peer_info: peer_info,
};

type peer_info =
{
  type: "stakeholder",
  stakeholder: {
    name: string,
  },
}
|
{
  type: "dynamic",
  dynamic: {
    user_provided_identifier: string, ①
    idp: {
      name: string, ②
    },
    attributes: object, ③
  }
}
|
{
  type: "application",
  application: {
    name: string,
  },
};

type request = ④
{
  type: "application",
  application: {
    application: string,
    url: string,
```

```
    method: string,
    headers: [[string, string]],
  },
|
{
  type: "process",
  process: {
    not_sure: "yet",
  },
}
;
```

- ① The user tells the TSDP who it thinks it is, and based on that the authentication process can be started.
- ② The name of the identity provider, as written in the DFC, which performs the authentication.
- ③ Whatever the identity provider spat out for this user. In the case of the PEDB, this must contain the permissions blob and the many admin signatures.
- ④ The request models all the tunneled End User → Application requests.

Connection between the Identity Provider and OPA: Right now there is no connection. The point is that the Identity Provider is just there for authentication. Access to resources should all be defined in the DFC. However, if this turns out to be too restrictive in practice, a future enhancement could allow the Identity Provider to pass authorization attributes to OPA during the authentication flow.

Appendix B: Intel TDX setup

The TDX ecosystem is quickly updating, which means there will likely be solutions that simplify the process of setting up everything needed to run TD-s.

B.1. Overview

An older version human-readable (still extremely laconic) and close-to-source document describing necessary steps to run TDs is <https://github.com/intel/tdx/wiki/TDX-KVM>. This older version highlighted 4 necessary components to run a VM with TDX guarantees:

- TDX KVM - Kernel-based virtual machine is a module in Linux kernel that enables kernel to act as a type 1 hypervisor. Intel provides a complete kernel with modified KVM such that it could be used to run TDs. <https://github.com/intel/tdx/tree/kvm> contains the source of the Intel modified kernel.
- TDX Qemu - Quick emulator(Qemu) is an emulation tool for running all sorts of operating systems and programs on the host hardware. It is often used together with KVM. As Intel modifies KVM, it also must modify Qemu to make the KVM-Qemu collaboration possible. TDX-qemu development is contained in <https://github.com/intel-staging/qemu-tdx/tree/tdx-qemu-upstream>
- TDX guest - A virtual machine that is run in TDX KVM and TDX Qemu under certain settings.
- TDVF - TDX guest virtual firmware, explained in depth in <https://cdrdv2-public.intel.com/733585/tdx-virtual-firmware-design-guide-rev-004-20231206.pdf>. Necessary host component for booting TDX guests.

B.2. Host setup

At the time of the writing, [canonical/tdx](https://github.com/canonical/tdx) github repository simplifies the setup process.

B.3. Guest setup

TDX guest is a regular TVM with a special kernel. The kernel is modified similarly to the server. It is likely that TVMs kernel config has lesser requirements, however it was not experimented with.

<https://github.com/canonical/tdx> provides ready made VM images with modified kernels. Canonical has fixed issues in upstream kernels and configuration that previously caused issues (<https://github.com/intel/tdx-linux/issues/1>).

[1] Currently integrated TEE providers consist of Intel and a mock_tee to allow development without hardware lock.

[2] In Sharemind HI there are *Producer*, *Consumer*, and *Runner*. However, with the much more flexible [authorization policy](#) based on [OPA](#) this distinction does not make much sense anymore.

[3] JOCONDE (Joint On-demand COMputation with No Data Exchange) is a project carried out by Eurostat in

collaboration with Cybernetica to foster the adoption of Privacy Enhancing Technologies (PET) in the European Statistical System, <https://cros.ec.europa.eu/joconde>

[4] Used for Intel SGX, deprecated as of April 2025