Advanced Secure Cloud Encrypted Platform for Internationally Orchestrated Solutions in Healthcare

Project Acronym: **ASCLEPIOS**

Project Contract Number: **826093**

Programme**: Health, demographic change and wellbeing**

Call: **Trusted digital solutions and Cybersecurity in Health and Care to protect privacy/data/infrastructures**

Call Identifier: **H2020-SC1-FA-DTS-2018-2020**

Focus Area: **Boosting the effectiveness of the Security Union**

Topic**: Toolkit for assessing and reducing cyber risks in hospitals and care centres**

Topic Identifier: **H2020-SC1-U-TDS-02-2018**

Funding Scheme: **Research and Innovation Action**

Start date of project: 01/12/2018 Duration: 36 months

Deliverable:

D4.3 Interoperability of ITEEs in the context of eHealth systems

Due date of deliverable: 31/05/2020 Actual submission date: 30/05/2020

WPL: Nicolae Paladi/RISE

Dissemination Level: Public

Version: 1.1

Table of Contents

Table of Contents 2
List of Figures and Tables 4
Status, Change History and Glossary 5
1 Introduction 8
1.1 Scope 8
1.2 Objectives 8
1.3 Relation to Other Work Packages and Deliverables 8
1.4 Outlook 8
2 Current TEE Landscape 10
2.1 Approaches to TEE implementation 10
2.1.1 Dynamic Root of Trust for Measurement 10
2.1.2 Intel Software Guard Extensions 10
2.1.3 ARM TrustZone 12
2.1.1 IBM Protected Execution Facility 12
2.1.2 AMD Secure Encrypted Virtualization 13
2.2 Comparison of SGX and SEV 13
2.2.1 Memory limits 13
2.2.2 Usability 13
2.2.3 Integrity protection 14
3 Applications development for TEEs 16
3.1 Symmetric Searchable Encryption (SSE) and TEE 16
3.1.1 SSE 16
3.1.2 SSE components to be deployed in TEE 17
3.1.3 Architecture and implementation 18
3.2 Implementation with Intel SGX 19
4 TEE Interoperability 23
4.1 Enarx 23
4.1.1 High-level description 23
4.1.2 Security Model 23
4.1.3 TEE Hardware and CPU Support 24
4.1.4 Runtime requirements 24
4.1.5 Architectural components 24
4.1.6 Relation to ASCLEPIOS 25
4.2 Asylo 25
4.2.1 High-level description 25

4.2.2 Security Model 26
4.2.3 Architectural components 27
4.2.4 Interoperability / Portability 29

4.3 Open Enclave SDK 29

4.3.1 High-level description 29
4.3.2 Security Model 30
4.3.3 TEE Hardware and CPU Support 30
4.3.4 Architectural components 30

5 Standardization work 33

5.1 IETF TEEP 33

5.2 IETF RATS 35

5.3 Global Platform 35

5.3.1 Global Platform TEE system architecture 36
5.3.2 TEE management framework 37
5.3.3 TEE APIs 37
5.3.4 TEE Initial configuration 38

6 Conclusion 39

List of Figures and Tables

Figures

Figure 2 Intel SGX execution mechanism 11
Figure 3 AMD execution mechanism 13
Figure 4 Architectural view of the SSE scheme 16
Figure 5 High level description of SSE scheme within TEE 19
Figure 6 A typical example of Intel SGX based application........20
Figure 7 Intel remote attestation flow [25]........21
Figure 8 Enarx process flow overivew........25
Figure 9 Asylo process flow........27
Figure 10: Open Enclave based application architecture 31
Figure 11: Open Enclave adaptation of SSE as an example 31
Figure 12: Enclave Definition Language (EDL) code example........32
Figure 13 TEEP architecture........34
Figure 14 RATS conceptual data flow. 35
Figure 15 Remote attestation with TEEP and RATS. 35
Figure 16 Example of GlobalPlatform remote procedure call (ARM TrustZone). 36

Tables

Table 1: Status Change History........5
Table 2: Deliverable Change History........5
Table 3: Glossary 7
Table 4 comparison of SGX and SEV 14
Table 5 Comparison of SGX and AMD application security 15
Table 6 Comparison of SGX and SEV vulnerabilities........15

Status, Change History and Glossary

Status:	Name:	Date:	Signature:
Draft:	**Arash Vahidi** **Nicolae Paladi**	27/02/2020	
Reviewed:	**Razvan Venter** **Christiaan Hillen**	20/05/2020	[illegible]
Approved:	**Tamas Kiss**	30/05/2020	Tamas Kiss

Table 1: Status Change History

Version	Date	Pages	Author	Modification
V0.1	18/03/2020	8	Nicolae Paladi	Create working document
V0.2	18/03/2020	13	Nicolae Paladi	Update TOC, add info about Enarx
V0.3	25/03/2020	15	Nicolae Paladi	Expand Introduction
V0.4	10/04/2020	18	Nicolae Paladi	Describe current TEE landscape
V0.5	04/05/2020	31	Amjad Ullah/Hai-Van Dang	Describe the Application development for TEEs
V0.6	09/05/2020	34	Amjad Ullah	Describe the Open Enclave SDK
V0.7	2020-05-11	34	Arash Vahidi	Updated sections about RATS, TEEP and GlobalPlatform
V0.8	2020-05-11	39	Nicolae Paladi	Update description of Enarx; update figures; write conclusion
V0.9	2020-05-12	41	Nicolae Paladi	Review document, prepare for review
V1.0	2020-05-25	41	Nicolae Paladi	Edit & address internal review comments
V1.1	2020-05-29	42	Nicolae Paladi	Add Section 1.4, extend conclusion

Table 2: Deliverable Change History

Glossary

ACPI	Advanced Configuration and Power Interface
AES	Advanced Encryption Standard
CPU	Central Processing Unit
CA	Certificate Authority
DMA	Direct Memory Access
DoS	Denial of Service
DRAM	Dynamic Read-Only Memory
EDMM	Enclave dynamic memory management
EDL	Enclave Definition Language
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
FIQ	Fast Interrupt Request
GDPR	General Data Protection Regulation
HECI	Host Embedded Controller Interface
HAIEE	Hardware-Assisted Isolated Execution Environments
IETF	Internet Engineering Task Force
IRQ	Interrupt Request
ITEE	Isolated Trusted Execution Environment
ME	Management Engine
MEE	Memory Encryption Engine
MMU	Memory Management Unit
MPU	Memory Protection Unit
NS	Non-Secure
OEM	Original Equipment Manufacturer
OS	Operating System
PCI	Peripheral Component Interconnect
PCR	Platform Configuration Registers
PRM	Processor Reserved Memory
REE	Rich Execution Environment
ROM	Read-Only Memory
RISE	Research Institutes of Sweden
RSM	Resume for System Management Mode
TA	Trusted Application
TAM	Trusted Application Manager

TEE	Trusted Execution Environment
TEEP	Trusted Execution Environment Platform
TOC	Time-of-Check
TOU	Time-of-Use
SEV	Secure Encrypted Virtualization
SENTER	Secure ENTER
SCR	Secure Configuration Register
SGX	Software Guard Extensions
SKINIT	Secure Init and Jump with Attestation
SoC	System-on-Chip
SMC	Secure Monitor Call
SMI	System Management Interrupt
SMM	System Management Mode
SMU	System Management Unit
SMRAM	System Management RAM
SSE	Symmetric Searchable Encryption
WASI	Web Assembly System Interface
WASM	Web Assembly

Table 3: Glossary

1 Introduction

1.1 Scope

The scope of this document is a review of the state of the art in the interoperability of TEEs and portability of applications for TEEs. We further consider the interoperability aspects of TEE applications for e-health security, in the context of project ASCLEPIOS. Beyond a review of the state of the art, the document includes reviewing practical aspects of developing applications for some types of TEEs, a review of TEE application development and deployment frameworks and on-going standardization work, conducted both within project ASCLEPIOS and outside of it.

1.2 Objectives

The specific objectives of this document are as follows:

- Review the current landscape of TEE implementations;
- Describe practical aspects of developing applications for common TEEs;
- Review existing projects for TEE application development and deployment;
- Review of the standardization work towards TEE interoperability.

1.3 Relation to Other Work Packages and Deliverables

This document constitutes Deliverable D4.3 within Work Package 4. While it contains practical aspects of developing applications for TEEs (Specifically in Section 3), in this deliverable we look at the upcoming and on-going projects that aim to enable interoperability between TEEs. In many cases, such projects are not yet functional or have very limited functionality. Therefore, the deliverable focuses on the *documented* functionality rather than a first-hand experience. A notable exception, as noted above, is Section 3, where we describe application development for TEEs using the *Open Enclave* project[1].

This deliverable complements deliverables D4.1 and D4.2. In D4.1 we describe the key, firmware and workload management in several common TEE architectures. In D4.2 we describe the approaches to workload attestation and its use in project ASCLEPIOS. When it comes to other work packages, the contents of this deliverable may serve as a guidance in selecting a suitable TEE for software components supporting the ASCLEPIOS framework.

1.4 Outlook

Task 4.3 was initially designed with the goal of developing a framework facilitating the development of applications portable across Trusted Execution Environment (TEE) architectures. However, throughout the course of the task, its scope was adjusted considering the following three points identified in the course of WP4:

[1] Open Enclave SDK https://openenclave.io/

1. deployment, attestation and management of applications within the same TEE architecture remains underspecified, thus undermining the foundations of any application portability effort;
2. relevant standardization efforts aim to create a common deployment and attestation architecture for applications in TEEs, with only one partial open-source implementation available;
3. portability of applications *across* TEE architectures is an increasingly difficult problem due to the diverging architectural choices made by vendors for emergent features. Despite significant efforts so far, portability projects only have limited support for *one* TEE architecture, despite the declared goal of supporting several most relevant ones.

The scope of this task was defined to address, in order, the factors defined above. As a result, work in Task 4.3 focused on the following: **detailing** in Section 3 the practical choices, trade-offs and decisions for developing, attesting and deploying applications in a TEE (in particular, in Intel SGX using the OpenEnclave SDK), thus addressing point 1; **contributing** to the specification of - and implementing a prototype of - the Trusted Execution Environment Platform Architecture defined by the Internet Engineering Task Force (addressing point 2, described in Section 5), and **evaluating** in Section 4 the existing open-source projects that work towards application portability (addressing point 3).

This document provides a comprehensive overview of the on-going efforts in terms of inter-operability; it supports the **implementation** work on the **Trusted Execution Environment Platform Deployer (TEEPD)** within project ASCLEPIOS. TEEPD implements the Trusted Execution Environment Platform architecture defined in [4] and described in Section 5.1. **Implementation, integration and evaluation of TEEPD** will be continued throughout WP 5 and WP 6 within project ASCLEPIOS.

2 Current TEE Landscape

In this section, we review the landscape of user-programmable Trusted Execution Environments relevant in the context of medical data protection. The review includes both existing an upcoming (or *announced*) architectural approaches for Trusted Execution Environment.

2.1 Approaches to TEE implementation

2.1.1 Dynamic Root of Trust for Measurement

The Trusted Computing Group (TCG) introduced Dynamic Root of Trust for Measurement (DRTM), also called "late launch", in the TPM v1.2 specification in 2005. It is an alternative to the Static Root of Trust for Measurement (SRTM). Unlike SRTM which operates at boot time, DRTM allows the root of trust for measurement to be initialized at any point [1]. To implement this technology, Intel developed Trusted eXecution Technology (TXT), providing a trusted way to load and execute system software (e.g., OS or VMM). Its primary purpose is to detect the potential presence of certain types of attacks, notify system owners about the detected attacks and prevent the creation of an Measured Launch Environment in the event of a compromise [29]. This is done by combining the SRTM and DRTM capabilities, along with additional support in software and in the instruction set architecture (ISA). At power-on, SRTM is used to establish and extend a chain of trust from the Intel processor (and chipset) to and including the BIOS. Once booted, the operating system or an application executing on the operating system can initiate a measured launch sequence by invoking the GETSEC(SENTER) instruction, which triggers the loading of the Measured Launch Initialization (SINIT). Intel TXT makes no assumptions about the system state and provides a dynamic root of trust for late launch. Thus, TXT can be viewed as a hardware-assisted trusted execution environment capable of running security sensitive tasks, at the cost of a significant overhead on the late launch operation [1].

2.1.2 Intel Software Guard Extensions

Intel SGX provides a TEE in recent processors since generation Skylake. Applications create secure *enclaves* to protect the integrity and confidentiality of the code being executed and its associated data [2]. Such enclaves rely for their security on a trusted computing base of code and data loaded at initialization creation time, processor firmware and processor hardware. Program execution in an enclave is transparent to both the underlying OS and other enclaves. Many mutually distrusting enclaves can operate on the platform. Intel SGX was applied widely adopted and used in several application domains, including cloud and network security [33-35]. The SGX mechanism is illustrated in Figure 2.

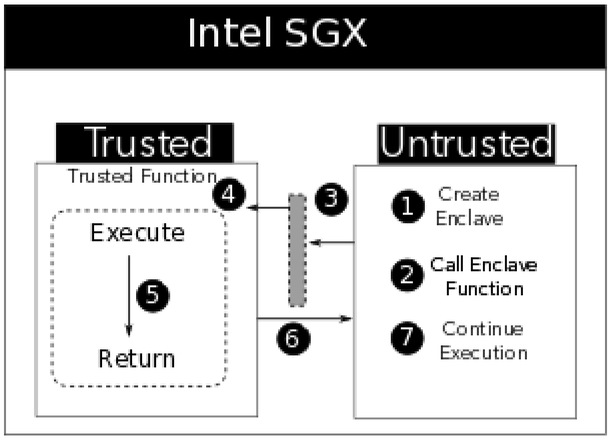


Figure 1 Intel SGX execution mechanism

The life cycle of an SGX enclave starts with a creation stage, when the ECREATE instruction invoked by the system software allocates a memory page for the SGX Enclave control structure and populates it with data about the memory size and layout of the enclave, made available by the system software. Once the enclave is created, system software uses the EADD instruction to load code and data into the enclave using the EEXTEND instruction to update the measurement of the enclave. Finally, the system software obtains an initialization token (EINITTOKEN) from a dedicated *Launch Enclave* and initializes the enclave (using the EINIT instruction). Once the enclave is initialized, the application deployed to the enclave can execute [30]. A *remote attestation protocol* (not shown in Figure 2) allows an enclave to provide guarantees of its contents and that it runs on a genuine Intel processor with SGX enabled. An application using enclaves must ship a signed, plaintext shared library that can be inspected, (including by malicious attackers). The *enclave page cache* (EPC) is a 128 MiB area of memory predefined at boot, dedicated to storing enclave code and data. At most 93.5 MiB can be used by an application; the remaining area is used to maintain SGX metadata. Any access to an enclave page outside the EPC triggers a page fault. The SGX driver interacts with the CPU and decides which pages to evict. Traffic between the CPU and the system memory is kept confidential by the *memory encryption engine* (MEE) [2], also in charge of tamper resistance and replay protection. If a cache miss hits a protected region, the MEE encrypts or decrypts data before sending to, respectively fetching from, the system memory and performs integrity checks. Data can also be persisted on stable storage, protected by a seal key. This allows storing certificates and waives the need of a new remote attestation every time an enclave application restarts [3].

The execution flow of a program using SGX enclaves is as follows. First, an enclave is created (see Figure 2, step 1). When a program needs to execute a trusted function (2), it invokes the SGX *ECALL* primitive (3). The program goes through the SGX call gate to bring the execution flow inside the enclave (4). After the trusted function is executed by one of the enclave's threads (5), its result is encrypted and sent back (6) prior to returning control to the main processing thread (7) that continues the execution. Since its introduction, Intel SGX was and remains under intense scrutiny from the security research community. This resulted in exposing numerous security vulnerabilities [10-19]. A range of improvement have been proposed [20-22] and the SGX specification was updated on several occasions.

2.1.3 ARM TrustZone

ARM TrustZone is a hardware feature to create isolated execution environments. It provides two environments (or “worlds”): the “secure world”, i.e. the Trusted Execution Environment (TEE), and the “normal world”, i.e. the Rich Execution Environment (REE). To ensure complete isolation between the two environments, TrustZone provides security extensions for hardware components including CPU, memory, and peripherals [1]. The two environments correspond to the security modes of the TrustZone enabled ARM CPU. Each processor mode has its own memory access region and privilege. Code running in the normal world cannot access the memory in the secure world, while code running in the secure world can access the memory in normal world. The secure and normal worlds can be identified by reading the NS bit in the Secure Configuration Register (SCR), modifiable in the secure world. TrustZone uses Monitor mode that only runs in the secure world to serve as a gatekeeper managing the switches between the two worlds. The normal world can call a special instruction called the Secure Monitor Call (SMC) to enter the Monitor mode and modify the NS bit to switch into the secure world [1]. From a user perspective, ARM TrustZone offers only limited programmability, since applications deployed in ARM TrustZone must be signed by the hardware vendor. In practice, this limits the number of TrustZone application providers. Another notable limitation of this approach is that there is no isolation among the applications running in TrustZone.

2.1.1 IBM Protected Execution Facility

IBM has announced in 2018 the *Protected Execution Facility (PEF)* technology [36]. PEF leverages a combination of the TPM and additional processor instructions to create a virtualization environment with enhanced security guarantees. PEF introduces Secure Virtual Machines (SVMs) and allows to protect SVM (including code and data) against attacks from outside SVM components. PEF allows secrets to be embedded in SVM at creation, and supports conversion of existing VMs into SVMs. PEF does not limit amount of protected memory, allowing existing application code to run in an SVM.

To enable PEF support, a new processor mode is added – the *Ultravisor* mode, that is higher privileged than the hypervisor mode. Architecturally, the Ultravisor is a shim layer beneath the hypervisor. The Ultravisor controls the memory space where the Secure VMs run, such that the hypervisor and normal VMs cannot reference the memory used by SVMs. Hypervisors must do an *ultracall* (a new type of syscall) to access secure memory or utravisor privileged resources; moreover, hypervisors can only see secure memory in *encrypted* form. PEF relies on a root of trust, implemented using TPMs available in OpenPOWER systems. The Ultravisor uses a secure channel to the TPM to get access to the symmetric key protecting the SVM [36]. To the best of our knowledge, no hardware supporting PEF is available at the moment.

2.1.2 AMD Secure Encrypted Virtualization

AMD secure encrypted virtualization (SEV) provides transparent encryption of the memory used by virtual machines. This requires the AMD secure memory encryption (SME) extension to be available and supported by the underlying hardware. The architecture relies on an embedded hardware AES engine, located on the core's memory controller. SME creates one key that is used to encrypt the entire memory. This is not the case for SEV, where multiple keys are being generated. The overhead of the AES engine is minimal [3].

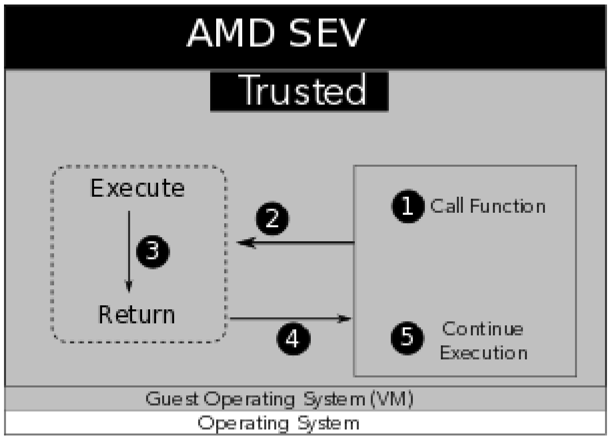


Figure 2 AMD execution mechanism

SEV delegates the creation of *ephemeral* encryption keys to the AMD secure processor (SP), an ARM TrustZone-enabled system-on-chip (SoC) embedded on-die [3]. These keys are used to encrypt the memory pages belonging to distinct virtual machines, by creating one key per VM. Similarly, there is one different key per hypervisor. These keys are never exposed to software executed by the CPU. AMD SEV allows to attest encrypted states by using an internal challenge mechanism, so a program can receive proof that a page is correctly encrypted [3]. From the programmer perspective SEV is transparent and the execution flow of a program using it is the same as a regular program. Notably different from Intel SGX, all the code runs inside a trusted environment, without a fine-grained separation of the "trusted" and "non-trusted" part of the code. The execution flow is illustrated in Figure 3. First, a program needs to call a function (Figure 3, step 1). The kernel schedules a thread to execute that function (2) before executing it (3). The execution returns to the main execution thread (6) until the next execution is scheduled (5) [3].

2.2 Comparison of SGX and SEV

We briefly highlight the differences between these two technologies along three different criteria, summarized in Table 4, Table 5 and Table 6 below.

2.2.1 Memory limits

The EPC area used by SGX is limited to 128 MiB, of which 93.5 MiB are usable in practice by applications. The size of the EPC can be controlled (reduced) by changing settings in the UEFI setup utility from the BIOS of the machine. There is no such limit for SEV: applications running inside an encrypted VM can use all its allocated memory [3].

2.2.2 Usability

To use SGX enclaves, a program must be modified—requiring a re-compilation or a relink—*e.g.*, using the official Intel SGX SDK. It is the responsibility of developers to decide which sections of the programs will run inside and outside the enclave. Several semi-automatic tools have been introduced to facilitate this process [31], [32]. As mentioned above, no changes need to be made to programs when using SEV [3].

2.2.3 Integrity protection

Intel SGX has data-integrity protection mechanisms built-in. Memory pages read from EPC memory by an enclave are decrypted by the CPU, and then cached within the processor. In the reverse data flow, data that is being written to the EPC by an enclave is encrypted inside the CPU before leaving its boundaries. Data integrity is safeguarded by associating integrity protected metadata. The metadata is stored in a Merkle tree structure, the root of which is stored in SRAM, inside the processor. These integrity mechanisms incur an overhead that has been previously evaluated and shown to be acceptable for sequential read/write operations, but up to 10× for random read/write operations [3]. Conversely, to the best of our knowledge, the current version of AMD SEV (or SME) does not provide any integrity protection mechanism. This can be exploited to break the security guarantees of SEV [7-9, 24]. We expect that limitation to be addressed in future revisions.

The main advantages of SEV in comparison to its main competitor - Intel SGX - are *(1)* memory size, *(2)* efficiency and *(3)* No SDK or code refactoring are required. SGX allocates only 128MB of memory for software and applications and thus, making it a good candidate for microtransactions and login services. However, SEV's memory is up to the available RAM and hence, making it a perfect fit for securing complex applications. Moreover, in situations where many calls are required, like in the case of a multi-client cloud service, SEV is known to be much faster and efficient than SGX. Table 4, Table 5 and Table 6provide a collective comparison with the main features offered by SGX and SEV [3].

Table 4 comparison of SGX and SEV

TEE	Access Level	Memory Size	SDK	Attestation	Protection
SGX	Ring3	Up to 128MB	Provided	Through Intel Remote Attestation Protocol	Confidentiality and Integrity of the Code and Data in the Enclave
SEV	Ring0	Up to Available System Memory	Not Required	Through AMD Secure Processor	Confidentiality of the Code and Data

Table 5 Comparison of SGX and AMD application security

SGX	SEV
Initial design targeted Microservices and small workload	Confidentiality and Integrity of the Code and Data in the Enclave
Requires major software changes and code refactoring	Does not require software changes and code refactoring
SGX works with ring 3 and is NOT suitable for many system calls	SEV works with ring 0 and is suitable for broader range of workload
SGX is suitable for small but sensitive workload	SEV is suitable for securing large enterprise level applications.

Table 6 Comparison of SGX and SEV vulnerabilities

SGX	SEV
Provides Memory Integrity	Does NOT Provide Memory Integrity
Vulnerable to Side Channels	Vulnerable to Side Channels
Vulnerable to DoS Attacks	Vulnerable to DoS Attacks
Vulnerable to Speculative Attacks	NOT Vulnerable to Speculative Attacks

3 Applications development for TEEs

3.1 Symmetric Searchable Encryption (SSE) and TEE

This section provides an overview to the practical aspects of application developments for TEEs in the prospects of the SSE scheme proposed within the scope of ASCLEPIOS. The sub-section 3.1 introduce SSE scheme, followed by a justification in subsection 3.2 why it is necessary to run some components of the SSE scheme within the TEEs. Lastly, subsection 3.3 provides an overview to the implementation aspects of SSE components in combination with two potential candidates of TEEs

3.1.1 SSE

The SSE is one of the core security components, currently under development, within the scope of ASCLEPIOS. It is an encryption technique that enables the search on the outsourced encrypted data while preserving the privacy of both data and search queries. Figure 3 presents the high-level architecture of the SSE scheme. It mainly consists of three core components: a Trusted Authority (TA), SSE Server, and a client application.

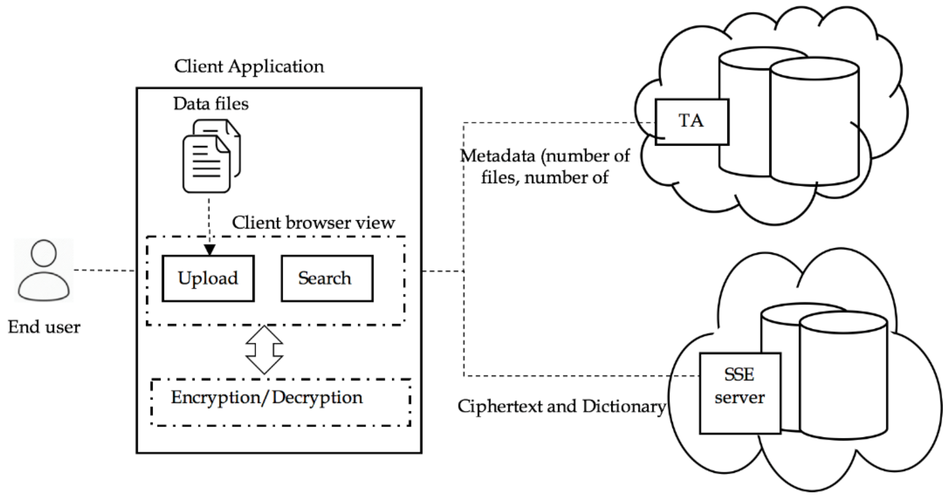


Figure 3 Architectural view of the SSE scheme

The Trusted Authority (TA) stores metadata which consists of the following two dictionaries: one counts the number of files containing each keyword and the other counts the number of previous searches on each keyword. These will be used to assist the client application to search over the encrypted data. The number of files of corresponding keywords get updated when a new file is added, while the number of searches changes after each search. Such changes over number of files and searches results in generating search token differently over time, even for a same keyword. This prevents the SSE Server from learning the search pattern.

The SSE Server represents the cloud service provider that is responsible for data storage. The data sent for storage is encrypted with a symmetric encryption key (K_1) and therefore, the

Server cannot decrypt the stored data. Furthermore, the SSE Server keeps a dictionary which maps extracted keywords to data file identifiers. The extracted keywords in the dictionary are not stored in plaintext; instead they are computed over some hash function with the keyword, number of files containing the keyword, number of previous searches on the keyword, and a symmetric key (K_2) as input. Similarly, the way of computation on the extracted keywords ensures the computed values get updated after each search and SSE server learns nothing about the search queries. The key K_2 is shared between the client application and TA, which will use it to compute and provide the SSE server with a verification proof for search query.

A *client application*, as the consumer of the SSE scheme, encrypts data with a symmetric encryption key, and creates a dictionary at the end-user side before sending them to the SSE Server for storage. Additionally, the application sends metadata to the TA such as number of files and number of searches of extracted keywords along with their hashed value. The hash computations prevent TA from learning the keywords content.

When end-users wish to search over encrypted data, they provide the client application with searched keywords. Using the keywords and with metadata retrieved from TA, the client application creates search tokens and sends them to the SSE Server to retrieve the specific encrypted data from cloud storage. Upon receiving the search tokens, the SSE Server requests a proof from TA, which is computed with the shared key K_2 and metadata of the keyword. Upon reception, the SSE Server verifies the proof. If the verification passes successfully, the SSE Server filters the stored ciphertext and returns the ones that match the query to client application. It further updates the stored dictionary with new values in the search tokens. Amongst the components of SSE, the TA and SSE Server components must be deployed and run within the trusted execution environment. Section 3.1.2 motivates this design choice.

3.1.2 SSE components to be deployed in TEE

One of the functional challenges in the domain of symmetric searchable encryption is to provide multi-client settings that enable multiple clients to perform searches over their outsourced encrypted data. Such a functionality required synchronization among many clients. In the context of the SSE scheme, the metadata is used to generate search tokens, and it gets updated after each search. The metadata storage at clients can easily leads to inconsistencies at client ends, hence leading to failure in terms of generating valid search tokens. Therefore, it is required that the metadata is synchronized amongst all clients. The complexity of synchronization can be easily overcome with the use of an external trusted component (Trusted Authority). In order to build a Trusted Authority, we rely on TEE to secure executions to generate verification proof for SSE server and the used symmetric key (K_2). Additionally, TEE will also ensure the integrity of the executions, failure of which can fail search operation of users. In addition to TA, the SSE server also needs to run in TEE. With the support of TEE, the integrity of executions at SSE server side can be verified before the client provides tokens to add or search data. Thus, the provided tokens will be protected inside TEE, and cannot be utilized by malicious host server to query for information, for e.g. verification proof, from TA.

3.1.3 Architecture and implementation

The initial implementation of SSE scheme is independent from any TEEs related aspects. Currently, we are in the process of transforming the SSE implementation such that it can only be deployed within the TEE and can be used only by following secure computational guidelines, e.g. the SSE components in itself will guarantee that the underlying execution environment is TEE, the components will remotely attest themselves to the remote party, and any secrets can be provisioned securely after the attestation process. In the following paragraphs, we provide the practical aspects in terms of SSE implementation related to TEE environment.

Figure 4, an adaptation of Figure 3, presents the high-level interactions of SSE components amongst each other, when the TA and SSE Server runs in TEEs. These are additional interactions to the basic functional interactions described in Figure 3. Since the Client component is the consumer of both the TA and SSE Server, both these components must remotely attest themselves to the Client. Remote attestation is the process of proving that the service has been running in a secure hardware environment. Analogous to the Client, the SSE server also uses some functions from the TA. Hence, the TA must prove itself to the SSE Server by remotely attesting itself.

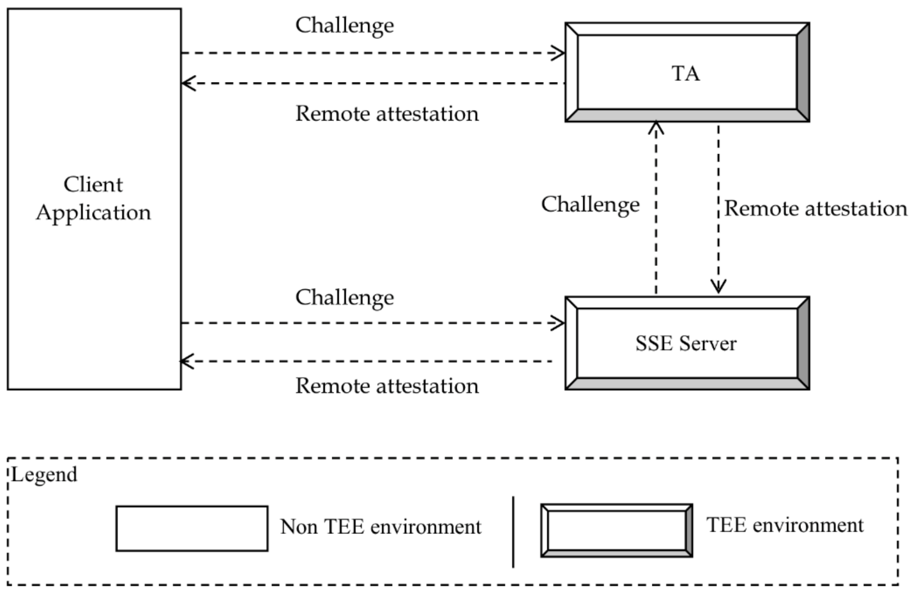


Figure 4 High level description of SSE scheme within TEE

3.2 Implementation with Intel SGX

The implementation of the required aspects to make the TA and SSE Server components TEE compliant depends on the underlying TEE technology. Currently, we are investigating Intel SGX as one potential candidate to be used as TEEs. In the following paragraphs, we provide a practical overview of how Intel SGX can be adapted in the context of SSE.

Intel SGX is a set of processor extensions for establishing a protected execution environment within an application. Intel SGX guarantees the integrity and confidentiality of security-sensitive computation performed on a computer where all the privileged software (kernel, hypervisor, etc.) can be potentially malicious. The Intel SGX technology allows part of the application to run in secure containers called enclaves. Such enclaves have dedicated memory regions that are secured with on-chip memory encryption. The enclave has its own dedicated code and private data to process. The data inside enclave cannot be accessed from outside.

Figure 5 illustrates a typical example of an Intel SGX based application. An SGX application consists of two parts; untrusted and trusted. The trusted part of the application run in an enclave, guaranteeing the integrity and confidentiality of the computation. The untrusted part of such an application is responsible, along with any other non-secure computation, is to create and initiate the necessary enclaves. On the other hand, the code running in the enclave is responsible for the required secure computation over any confidential data that shall restricted to the enclave.

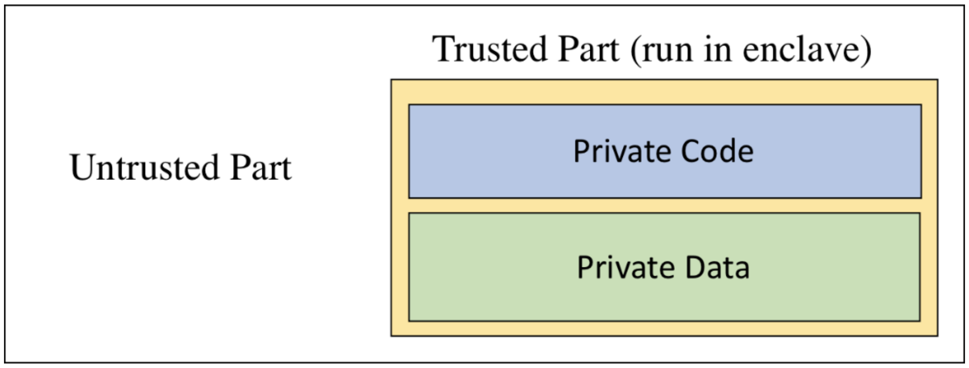


Figure 5 A typical example of Intel SGX based application

In the context of SSE, the TA and SSE Server components must run in such a TEE environment. Both these components are REST based services that expose their key functions through REST based interfaces. In light of the above-mentioned SGX based application example, both the TA and SSE Server components are envisioned as follows:

1. The untrusted part of the component will provide REST interfaces to the key functions of the component. In addition, each component will also provide a /challenge REST interface that will be used by the remote party to demand the remote attestation from the target component, the TA and SSE Server.
2. The key functions of each component will be implemented within the enclave or respective component.
3. The rest interfaces for each function, as described in step 1, will make the ECALLs to the respective key function of the enclave. The ECALLs are the entry to the enclave and lets the computation move from the untrusted space to the trusted space.

Both components on the receipt of challenge REST call will initiate the remote attestation process in order to attest itself to the remote party (or challenger). The entity that has to attest itself is called the *Verifier*, whereas the entity that demand remote attestation is called remote party (or challenger). In the case of SSE, as can be seen from Figure 3, the remote attestation process can be carried out at the following three different occasions: (1) when the TA attests itself to the client, (2) when the SSE Server attests itself to the client, and (3) when TA attests itself to the SSE server. In the first two cases, the Client is the *Challenger* and TA and SSE Server are the *Verifiers*, whereas in the last case, the SSE Server is the *Challenger* and the TA is the *Verifier*. Irrespective of which component is the *Challenger* and which component is the *Verifier*, Figure 6 presents the Intel SGX remote attestation flow.

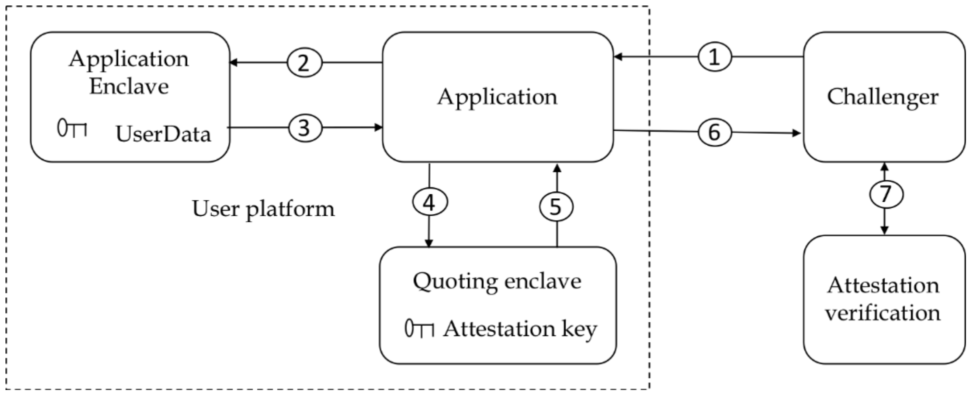


Figure 6 Intel remote attestation flow [25]

The brief description of each message appears in the above-mentioned Intel remote attestation flow are as follows:

1. The off-platform *Challenger* requests the application running in intel SGX based TEE to attest itself.
2. The *Application* requests its enclave to produce an attestation.
3. The enclave returns a local-attestation report.
4. The application forwards the local attestation report to the platform-oriented *Quoting enclave* that verifies the local attestation report.
5. The Quoting enclave further converts the local attestation report to a remote attestation report and sends back to *Application*.
6. The *Application* returns the remote attestation report to the off-platform *Challenger*.
7. The *Challenger* verifies the remote attestation report using the Intel attestation service. Based on the verification, the *Challenger* makes decision and provision any secret data, if required.

The implementation of TA and SSE Server components in light of the above-mentioned details will require the use of following additional tools:

1. Intel SGX related software[2]: The following three tools/SDKs from the Intel SGX software stack will be required for the development of both TA and SSE Server components.
   a. Intel SGX driver,
   b. Intel SGX SDK,
   c. Intel SGX platform software (PSW) SDK,

[2] Intel(R) Software Guard Extensions for Linux OS https://github.com/intel/linux-sgx

2. Microsoft cpprestsdk[3]: The Microsoft C++ REST SDK or any other alternative SDK of same nature will be required to implement the REST based untrusted part of both components.

Further development, testing and deployment of SSE will require the availability of Intel SGX based TEE. Microsoft Azure provides various virtual compute services that facilitate leveraging Intel SGX to set up TEEs. More specifically, the current offering from Microsoft Azure, under the theme of confidential computing, provides DCsv2-series virtual machines of various ranges that are built on top of the latest generation of Intel Xeon processors capable of Intel SGX technology. Currently, to the best of our knowledge, Microsoft Azure and Google Cloud Platform are the only public cloud providers that facilitate Intel SGX capable virtual machines.

[3] The C++ REST SDK https://github.com/microsoft/cpprestsdk

4 TEE Interoperability

We next review current development efforts towards enabling TEE interoperability and portability. We review three active projects that aim to enable such interoperability. While they have similar goals, the three projects build on vastly different assumptions, and adopt different approaches in their architecture, design and implementation.

4.1 Enarx

4.1.1 High-level description

Enarx is an application deployment system that enables applications to run in Trusted Execution Environments (TEEs) without being rewritten for particular platforms or SDKs. Enarx handles attestation and delivery into a run-time "*keep*" based on WebAssembly, offering developers a wide range of language choices for implementation. Enarx is CPU-architecture independent, enabling the same application code to be deployed across multiple targets, abstracting issues such as cross-compilation and differing attestation mechanisms between hardware vendors [28].

4.1.2 Security Model

The security model of Enarx aims for a minimal trusted computing base and relies on the WebAssemly standard for its runtime and interface. We next describe the Enarx security model in detail.

- **Trusted CPU:** like most other TEE architectures, Enarx assumes a trusted CPU. This is a cornerstone assumption trustworthy computation.
- **Trusted Microkernel:** this component is provided by Enarx and is trusted to perform standard kernel operations. An explicit goal of the project is to maintain the microkernel footprint as small as possible and open source.
- **Trusted WebAssembly runtime (WASM)** - provided by Enarx, and is trusted to provide the runtime for the application within the Enarx Keep, and includes silicon architecture specific JIT (Just In Time) compilation for performance optimization.
- **Trusted WebAssembly System Interface (WASI)** - provided by Enarx and is an interface for WebAssembly applications running on server-type systems (rather than in browsers, for instance). It is focused on security and portability and is trusted.
- **Trusted Application** - The application layer is the workload provided by the client to run within the Enarx Keep. It is not provided by Enarx but is considered trusted by the client as it was provided by them.

Along with the list of trusted components listed above, the Enarx Security Model explicitly notes two *untrusted* components. Both the Operating system kernel and the hypervisor are *explicitly* not trusted.

4.1.3 TEE Hardware and CPU Support

We next review the hardware and CPU support for execution in TEEs. While this list is currently relevant, it will likely be become outdated soon, as vendors release new generations with TEE support and the Enarx project evolves to support further platforms.

4.1.3.1 AMD SEV

AMD SEV is targeted at secure VMs. Developer applications attest to a signature by AMD, which includes a hash of *firmware,* which in this context is code injected into the VM. The firmware allows host to have some code within the TEE: that code will form an Enarx Keep. Enarx runs as "firmware" which is injected into the VM[4]. AMD provides a signature from a key burned into the CPU over a hash of the firmware to be loaded.

4.1.3.2 SGX

Enarx assumes for its functionality the presence of SGX 2 with Enclave dynamic memory management (EDMM) support. Attestation is done only involving attester and verifier, using the Data Center Attestation Primitives. In terms of CPU support, Intel 10th Gen Core CPUs are primarily targeted for implementation. 9th Gen Core CPUs could work but are likely to be harder to set up.

4.1.4 Runtime requirements

In terms of runtime requirements, the call-out API is implemented through the Web Assembly System Interface (WASI). Furthermore, it requires the *JIT Wasmtime*, a standalone WASM JIT.

4.1.5 Architectural components

We next discuss the architectural components of Enarx.

4.1.5.1 Attestation

In order to run in an Enarx Keep, an application needs to attest two things:

1. The hardware TEE (Trusted Execution Environment) providing Keeps.
2. A measurement of the Enarx runtime. This means that trusted third party may need to launch a service to abstract attestation. The way that this works is that the client requests attestation from Enarx. Enarx supplies a blob. The client forwards this to attestation service. The attestation service will then complete attestation of the hardware environment and translate the measurements of Enarx into a something which allows the developer to identify the specific version of Enarx.

[4] Information on Enarx with SEV https://github.com/enarx/enarx/wiki/SEV-architectural

From the client's point of view, the attestation steps of Enarx end up with the following two cryptographically validated assertions:

1. *The TEE type and version*;
2. *The Enarx version and integrity*. The attestation processes associated with the various hardware architectures are very different (as noted above under Section 4.1.3. Providing a common mechanism to abstract this is expected to be a major part of the work associated with project Enarx.

A high level overview of the Enarx process flow is illustrated in Figure 7.

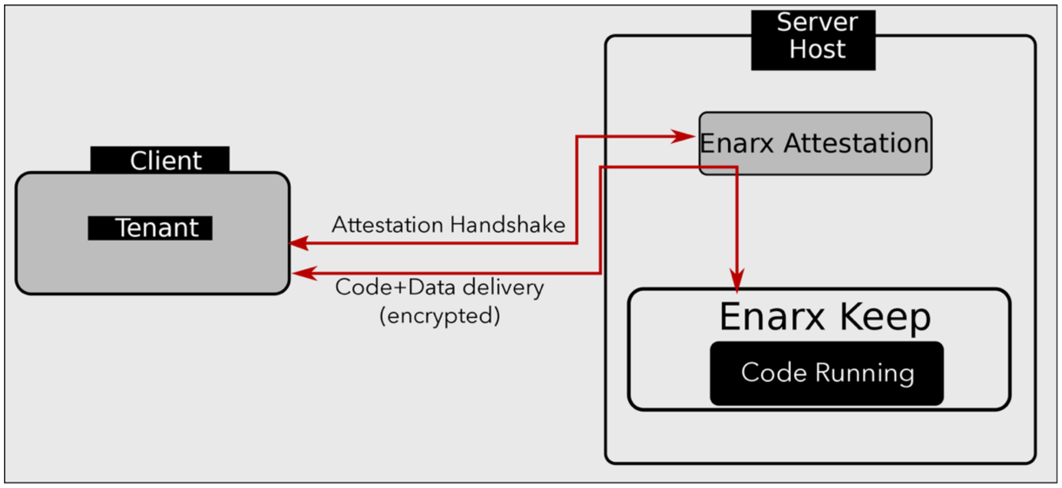


Figure 7 Enarx process flow overivew

4.1.5.2 Enarx API & core

The Enarx project defines the WASI APIs and manages the attestation for all of the TEEs that leverages the Enarx runtime.

4.1.6 Relation to ASCLEPIOS

Throughout project ASCLEPIOS, we continue monitor the development of project Enarx. Whenever possible, we will re-use the best practices and lessons learned to implement attestation and workload management for both AMD and Intel based platforms.

4.2 Asylo

4.2.1 High-level description

Asylo is an open source framework that enables applications to run in trusted execution environments (TEEs) without requiring changes to the code [27]. Developers can choose any enclave backend for their applications and use those enclaves to perform sensitive calculations or store data in a secure manner. Currently, Asylo offers a docker image for Intel

Software Guard Extensions (SGX) but there are plans to support other TEEs like AMD Secure Encryption Virtualization technology.

An enclave runs isolated from the rest of the system including the operating system kernel. Usually applications implicitly trust the operating system but this comes with certain drawbacks and risks. An application that runs encryption/decryption functions might be at risk if the operating system is compromised since. Sensitive data like private keys may end up in memory that will be accessible by the operating system, eliminating the security that the application tries to offer. Such issues can escalate to cloud infrastructures. If a cloud service provider wants to perform malicious actions then nothing can stop them since memory that different VMs use, is accessible and readable.

By using an enclave, an additional protection layer is added, ensuring that sensitive information will not be accessible by the operating system but only from specific enclaves. There are mechanisms used to ensure that the enclave is isolated from the rest of the system. This includes sealing, local and remote attestation and hardware-specific data structures that provide integrity and trust to the enclave.

So far, developers have had a tough time utilizing such capabilities since running enclaves is not a simple task. Enclaves need to be created and configured before they can run the application's code, adding a new level of complexity to development. Asylo provides a solution as it works like an application wrapper. A docker image that offers all necessary runtime configuration for the enclaves to run is currently provided from the development team. Using an API, calls can be made to and from the enclaves, transferring execution between the trusted part of the application that runs in the enclave, and the untrusted part that is accessible by the operating system kernel.

4.2.2 Security Model

Asylo aims to offer support for both software and hardware backends. Depending on an application's requirements, a choice can be made between isolation provided by hardware virtualization or an implementation of a proprietary CPU manufacturer such as Intel's SGX or ARM TrustZone.

Integrating Asylo in an application, security guarantees are established to sensitive workloads. Code and data that are protected by an enclave are secured against any vulnerability that is caused by a malicious Guest virtual machine (VM), user or host operating system. In addition, enclaves provide confidentiality and integrity guarantees for the communication between the untrusted environment and the enclaves. Finally, local or remote attestation mechanisms ensure the integrity of an enclave that executes the sensitive workloads.

4.2.3 Architectural components

An application integrated with Asylo is divided into the *Untrusted* execution environment and the *Trusted* execution environment. As illustrated in Figure 8, between those two, a Manager component is responsible to coordinate all communication between the two environments using secure communication channels. The trusted execution environment will contain the implementation of the sensitive functions and public methods that will provide access to the sensitive functions through the manager component. The untrusted execution environment can invoke the sensitive functions by using the interface provided by the manager component. During any interaction, certain data structures that are called *Protocol Buffers* are used. Protocol buffers is a method to serialize structured data providing an easy way to transfer and handle such data.

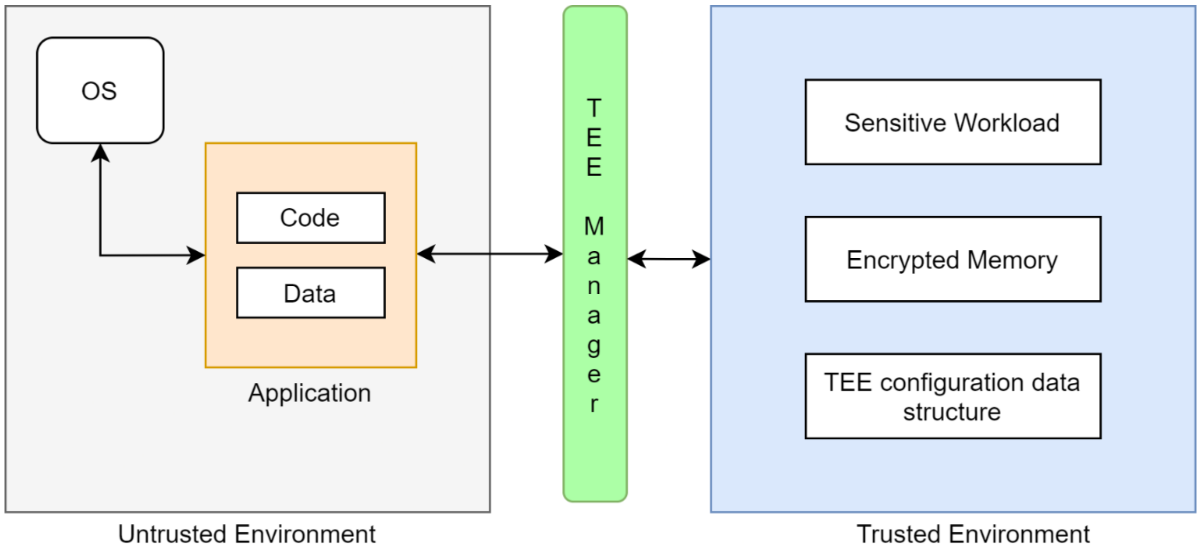
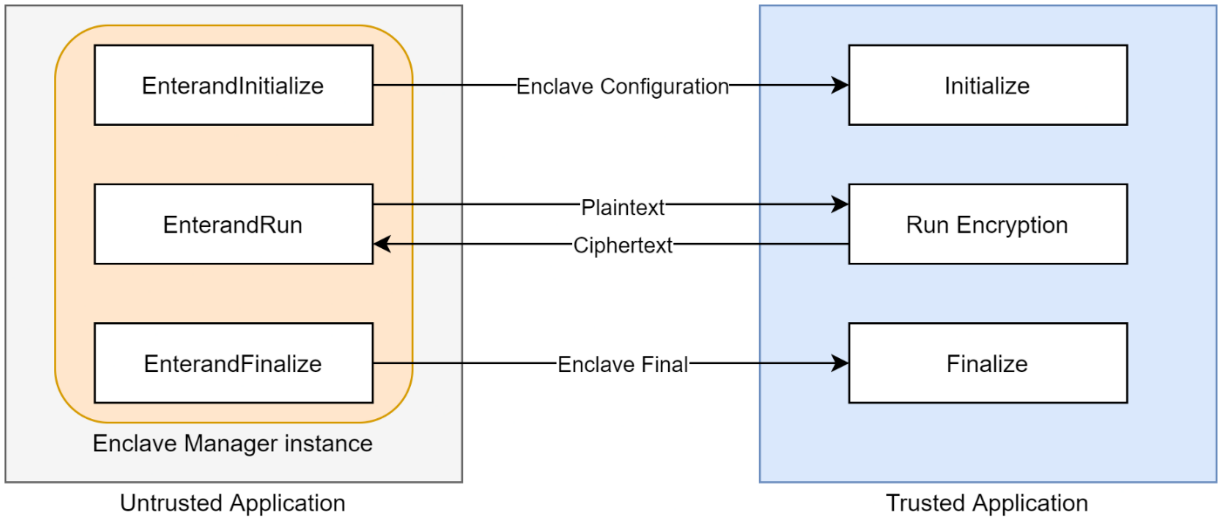


Figure 8 Asylo process flow

To get a better understanding of how Asylo works and how it can be used to run an application in a TEE (i.e. containing a trusted and untrusted part), we will describe the procedure of how an application that can be ported to Asylo. To this end, we will show how an application that performs AES encryption and needs to use sensitive/private information (i.e. encryption/decryption key(s)) can be integrated to Asylo and have this function run in an enclave using Intel's SGX technology. We will divide the application into two parts, the Trusted part which will be the code running in the enclave and the Untrusted part which will be running outside the enclave and can be accessed by anyone.

The trusted part will provide *three entry points* that will be accessible from the untrusted application through the manager component. These entry points are used to initialize and finalize the enclave and run the encryption function (i.e. use the symmetric key in a trusted mode in order to encrypt some data). The trusted application will also contain the implementation of the encryption function.

The untrusted application will create an instance of the enclave manager and use it to communicate with the enclave. First, the initialization phase takes place. This begins with the call of the *EnterandInitialize* entry point. A *protocol buffer message* that contains configuration settings is passed in order to set up the enclave. This step is essential as it is not possible to run any code in the trusted environment without the proper initialization. Then the application is ready to run the encrypt function by using the *EnterandRun* entry point. Both input and output protocol buffers are passed to the entry point. In our case, a plain text buffer will be passed as input along with an empty output buffer. The manager component will transfer execution to the enclave, running the *EnterandRun* function. The enclave will read the input, encrypt and populate the output buffer with the result ciphertext, which will be returned to the untrusted application. Once the execution is done, the *EnterandFinalize* entry point is used in order to destroy the enclave. This flow is illustrated in Figure 8.



To make our application more complete, we must also implement a decryption function. To do so, we need to divide the *EnterandRun* entry point to *EnterandEncrypt/EnterandDecrypt* or use the *EnterandRun* entry point in such a way where a flag that directs the application to the function needs to run will be provided.

Asylo provides base code implementations for the TrustedApplication and EnclaveManager classes for developers to use or modify. Also, a Bazel BUILD file is provided that defines the enclave's logic stating which TEE backend to use.

4.2.4 Interoperability / Portability

Interoperability is another challenge that developers must face when creating applications that are using TEEs. Currently, choosing a TEE technology implies that vendor specific changes will be made to the application to satisfy the requirements of the underlying TEE. Each TEE has its own runtime configurations and ways to manage the isolates entities. Also, as research develops and new vulnerabilities arise, a certain TEE technology might not be suitable anymore and a move to a different vendor might be needed. This means that choosing which TEE technology to use is a difficult task that needs to be examined thoroughly.

Asylo tackles portability as one of its main focuses. Once the application is adapted to the Asylo API, the use of a different backend means simply re-compiling and re-packaging the application. Changes to the code are *not* needed as the Asylo API was created with main aim to work with various TEE technologies.

In ASCLEPIOS, we plan to use Asylo mainly for porting an ABE library into SGX. This task is considered as rather demanding and it is expected to face certain difficulties due to the long list of dependencies.

4.3 Open Enclave SDK

4.3.1 High-level description

The Open Enclave, an open source initiative from Microsoft, is a library for the development of Trusted Execution Environment (TEEs) based applications in C and C++. The Open Enclave SDK aims to provide a single unified enclave abstraction for developers to develop TEE based applications independent from the underlying TEEs, hence enabling TEE agnostic secure applications that can be utilize on any hardware-based TEEs.

The key design principle of Open Enclave is to facilitate generalization, thus enabling developers to build enclave application model to minimize hardware and software specific concepts. The Open Enclave SDK supports the following key functionalities:

1. *Enclave creation and management*: The Open Enclave provides all the necessary function calls that are required for the management of the lifecycle of an enclave within an application.
2. *Enclave measurement and identity*: Open Enclave provides the expressions of enclave measurement and identity.
3. *Communication*: Open Enclave includes mechanisms for describing interfaces to define Enclave in and out calls and also handles the data marshalling associated with the in/out calls.

4. *Data sealing*: Open Enclave provide functions to facilitate the sealing of runtime Enclave data/secrets.
5. *Attestation*: Open Enclave provide mechanisms that facilitate both kind of attestation procedures, i.e. local and remote.
6. *Runtime and cryptographic libraries*: Open Enclave provide pluggable cryptographic libraries and runtime to facilitate the required cryptographic support inside enclaves.

The Open Enclave can be used directly on the hardware of proprietary CPU manufacturer such as Intel's SGX or ARM TrustZone, and can also be used in the cloud virtualized environment. E.g. currently, Microsoft Azure provides specialized VMs that support Intel SGX based TEEs where Open Enclave based applications can be deployed.

4.3.2 Security Model

The integration of Open Enclave SDK in an application enable the developers to builds security sensitive programs, where sensitive data can be protected by an enclave against malicious access. The enclaves created through the Open Enclave SDK guarantees the confidentiality and integrity of the data under processing inside the enclave as well as all the communication between the enclave and the untrusted part of the application. The Open Enclave SDK, as mentioned earlier, also facilitates both kind of attestation features to confirm the integrity of the enclave to a challenger. In addition, the functionalities like data sealing and support of cryptographic libraries/runtimes within enclave greatly enhance its suitability as a potential candidate to build security sensitive applications that required to be executed in TEEs.

4.3.3 TEE Hardware and CPU Support

The Open Enclave SDK aims to generalize the development of the enclave applications across TEEs from different hardware vendors. However, the current support is only available for Intel SGX and ARM TrustZone. In terms of operating systems, the Open Enclave SDK support is available for both Linux and Windows platforms.

4.3.4 Architectural components

Figure 9 depicts the high-level architecture of applications developed using Open Enclave. The *Node* in the figure represents the TEE based machine that host Open Enclave based application/s, e.g. *Application 1* and *2* in this case. An application developed using Open Enclave is structured in two parts, the untrusted part of the application and the trusted part. The trusted part of the application must run within an Enclave. In the following figure, this can be seen as the boxes labelled as *Application Enclave*. The enclaves of an application run in TEE, where the untrusted part runs outside TEE. Each enclave, as can be seen from Figure 9, has its own private code and data. The enclave data is restricted only to the enclave and it cannot be accessed from the untrusted part of the application and also not by other enclaves of the same application. The untrusted part of the application is usually responsible to provide interface to the external world in addition to any other untrusted computation. Hence any external interaction, with the enclaves of the application, must be mediated through the untrusted part of the application, represented as Host Application 1 and 2 in this case.

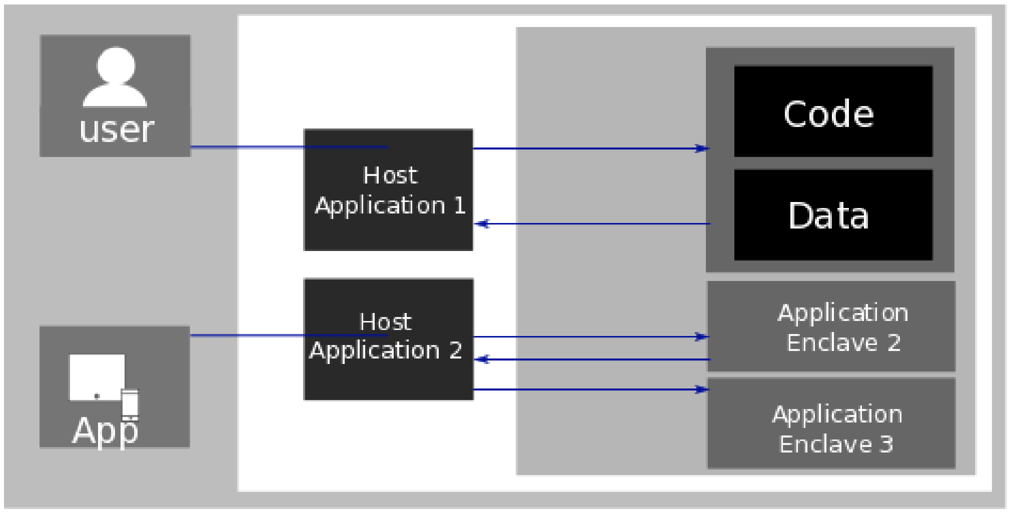


Figure 9: Open Enclave based application architecture

In order to better understand the operating procedure of the Open Enclave SDK and to put it into the context of Asclepios project, let us consider the example of the interactions between the two components, i.e. *Client application* and Trusted Authority (*TA)* from the SSE scheme. The detailed description of SSE is earlier described in Section 3.1. The *TA* component in the SSE scheme is responsible for handling the meta data required to facilitate searching over the encrypted data. The *TA* is a REST service and it provides various functions related to the management of meta data. For this example, we only consider the interaction of *Client Application* with the *TA* for the two core functions, *get* and *update* metadata. The following figure presents all the necessary interactions in light of the Open Enclave architecture earlier explained and presented in the previous figure.

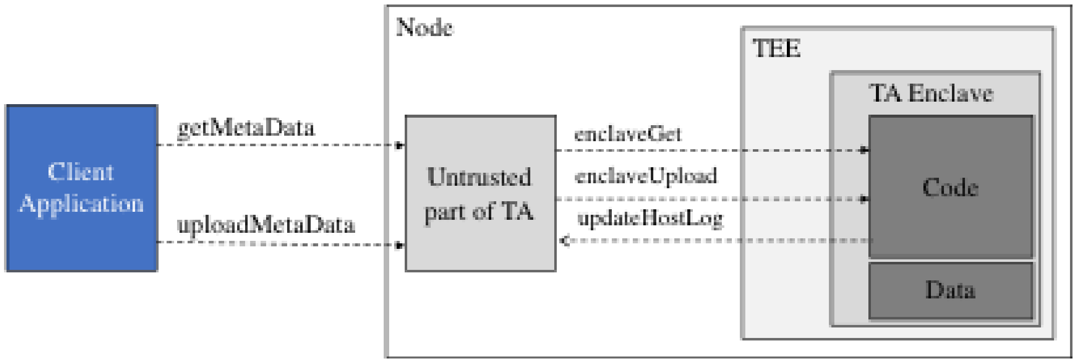
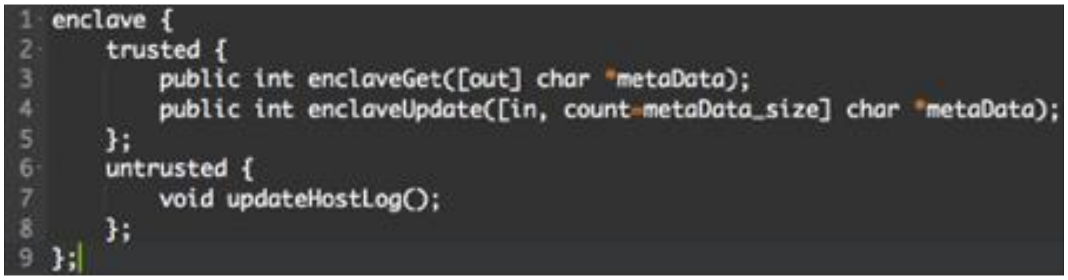


Figure 10: Open Enclave adaptation of SSE as an example

The TA component will be divided into the following two parts: (1) the Untrusted part, which will be running outside Enclave and will consist of interfaces to enable interactions from external components with *TA*, and (2) the Trusted part, which will be running inside the enclave and provide implementation to the core functionality of *TA*. The enclave must provide internal interfaces to the core functions that will be allowed to triggered from the untrusted part of the *TA*.

The Open Enclave architecture relies on the Enclave Definition Language (EDL) for the definition of the interfaces between the trusted and untrusted part. In the current example, the Untrusted part, on the receipt of the requests for *getMetaData* and *uploadMetaData,* triggers the corresponding enclave *get* and *upload* functions. These are secure enclave functions and are required to run inside TEE. Therefore, calling these functions from the Untrusted part pass the execution control from non-secured computation to secured computation that happens in TEE. The code inside enclave is responsible to only perform the secure computation and therefore if there is the need to execute some code from the Untrusted part, the enclave code must trigger the corresponding interface of the Untrusted part, such as calling the *updateHostLog* function in this case.

The following code snippet represents the corresponding EDL code that describe the definitions of the above-mentioned three interfaces (entry points). As it can be seen from the following code snippet that the *enclaveGet* and *enclaveUpdate* are the secure functions of the enclave, whereas the *updateHostLog* is the interface to a function in the Untrusted part of TA. It is important to note that the in/out parameters in the following code snippet are the simplified version of the actual in/out parameters of corresponding *TA* functions. Furthermore, it is important to note that the data communicated between the *Client application* and the *TA Enclave* must be encrypted and the Untrusted part of the TA is also unable to decrypt it. Only the Client Application and the TA Enclave will be able to decrypt the communicating data.



```
enclave {
    trusted {
        public int enclaveGet([out] char *metaData);
        public int enclaveUpdate([in, count=metaData_size] char *metaData);
    };
    untrusted {
        void updateHostLog();
    };
};
```

Figure 11: Enclave Definition Language (EDL) code example

5 Standardization work

In this section we review the most relevant standardization work aimed at realising the interoperability of TEEs.

5.1 IETF TEEP

As described above, TEEs use hardware enforcement combined with software protection to secure trusted applications (TAs) and their data. Moreover, TEEs typically offer a more limited set of services to TAs than is normally available to Untrusted Applications. Considering diversity of TEE implementation (briefly overviewed in Section 3), TEEs offer different security properties, different features, and different control mechanisms to operate the TAs. Some vendors may themselves market multiple different TEEs with different properties attuned to different markets. A device vendor may integrate one or more TEEs into their devices depending on market needs. This highlights the need for an interoperable protocol for managing TAs running in different TEEs of various devices. Furthermore, in this TEE ecosystem, there often arises a need for an external trusted party to verify the identity, claims, and rights of TA developers, devices, and their TEEs. This trusted third party is the Trusted Application Manager (TAM) [4]. The focus of the IETF TEEP Work group is to create such an interoperable protocol and ancillary technical details.

The Trusted Execution Environment Provisioning (TEEP) protocol addresses the following problems [4]:

- An installer of an Untrusted Application that depends on a given TA wants to request installation of that TA in the device's TEE so that the Untrusted Application can complete, but the TEE needs to verify whether such a TA is authorized to run in the TEE and consume potentially scarce TEE resources.
- A TA developer providing a TA whose code itself is considered confidential wants to determine security-relevant information of a device before allowing their TA to be provisioned to the TEE within the device. An example is the verification of the type of TEE included in a device and that it can provide the required security protections.
- A TEE in a device intends to determine whether an entity that wants to manage a TA in the device is authorized to manage TAs in the TEE, and what TAs the entity is permitted to manage.
- A TAM (e.g., operated by a device administrator) wants to determine if a TA exists (is installed) on a device (in the TEE), and if not, install the TA in the TEE.
- A TAM wants to check whether a TA in a device's TEE is the most up-to-date version, and if not, update the TA in the TEE.

- A TA developer wants to remove a confidential TA from a device's TEE if the TA developer is no longer offering such TAs or the TAs are being revoked from a user (or device). For example, if a subscription or contract for a service expired, or a payment by the user has not been completed or has been rescinded.
- A TA developer wants to define the relationship between cooperating TAs under the TA developer's control and specify whether the TAs can communicate and share data and key material.

The TEEP notional architecture (illustrated in Figure 12) considers one or several actors (such as a Trusted Application Developer, or Device Administrator) the deploy TAs over one or several TAMs. Whenever a TA is to be installed on a device carrying a TEE, the first step is to install a support application. Next, the support applications invoke the TEEP Broker (part of the software support for the TEE available on the platform) to request the installation of (or updates to) TAs in the TEE. The TEEP broker, deployed on devices with TEEs, contacts the TAM in order to poll for TAs (or updates) and transfers the received data to the TEEP Agent that manages the installation and patching of TAs *inside* the TEE.

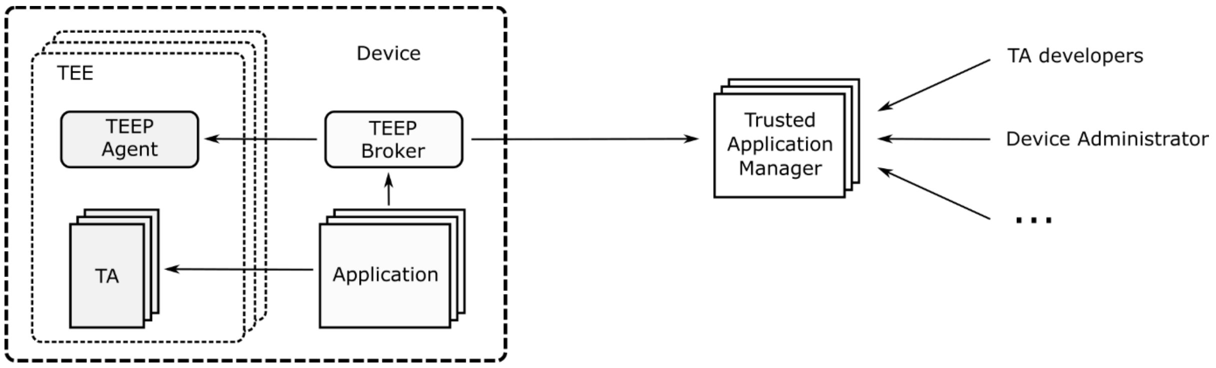


Figure 12 TEEP architecture

Note that the IETF TEEP Working Group is active at the moment of writing and the TEEP Architecture document [4] remains work in progress. Therefore, deviations from the problem statement and illustration above can be expected. Within project ASCLEPIOS, RISE has contributed to the formulation of the Trusted Execution Environment Platform (TEEP) architecture. Furthermore, RISE leads an implementation of the TEEP Architecture, realized within project ASCLEPIOS as the *TEEP Deployer* (TEEPD) component.

5.2 IETF RATS

Remote attestation, discussed above, is a cornerstone for establishing the trustworthiness of workload executing in a TEE. The IETF RATS working groups focuses on establishing a standardized attestation procedure with cross-vendor TEE support [5].

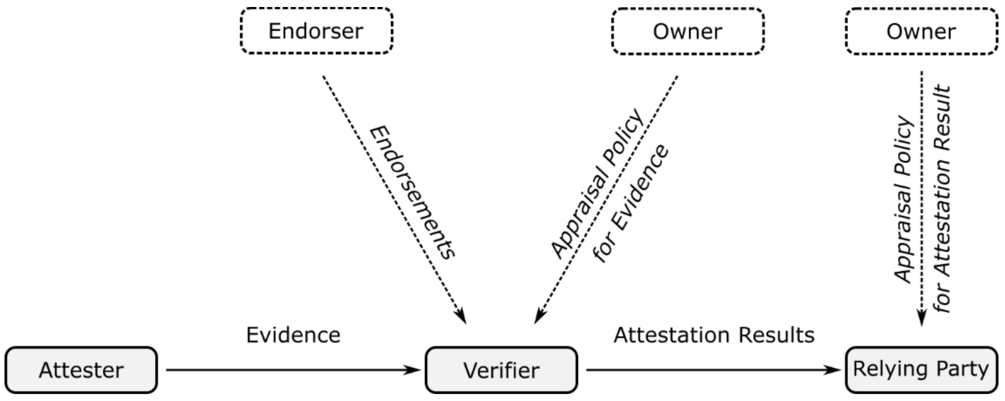


Figure 13 RATS conceptual data flow.

RATS defines a universal set of terms that can be mapped to various existing and emerging Remote Attestation Procedures, under the simplified model seen in Figure 13. For remote attestation RATS maps different attestation components to their TEEP counterpart as seen in Figure 14.

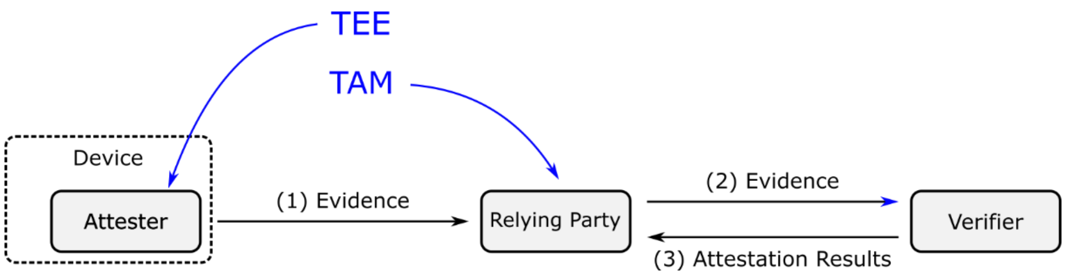


Figure 14 Remote attestation with TEEP and RATS.

This standard is discussed in detail in ASCLEPIOS D4.2, to which we refer readers for more information.

5.3 Global Platform

Global Platform works on the standardization and interoperability of application management within a TEE to deliver flexible security that answers the unique requirements of a range of different markets and use cases. The benefits of this work are as follows:

- Device manufacturers can embed a standardized and certified TEE that meets the needs of service providers for the protection of digital services from fraud and attack.

- Service providers are free to focus on enhancing their offerings by using a secure component to solve security challenges. They can also develop their service just once and deploy it universally across any device with a certified TEE, with the assurance that security levels will be consistent across devices.
- Digital service users benefit from greater simplicity, convenience, security and privacy for their digital services and personal data.

GlobalPlatform defines a mechanism for secure communication between normal and trusted applications. The aim is to allow procedure calls that can carry small (e.g. a number) or large parameters (e.g. a large memory area) from normal applications in rich OS into trusted applications and back in a secure manner. For this the standard defines a very narrow API where up to 4 parameters are allowed in each call. The underlying Trusted OS is responsible for handling call routing and transferring parameters between the security domains in a transparent but secure manner.

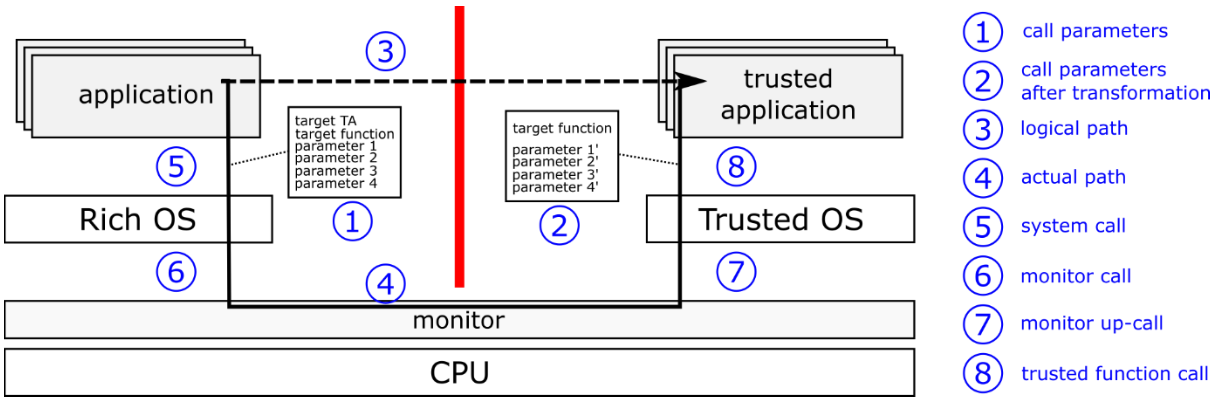


Figure 15 Example of GlobalPlatform remote procedure call (ARM TrustZone).

Figure 15 illustrates steps necessary for routing a procedure call with GlobalPlatform on platforms that require full isolation between secure an and unsecure applications and OS:es.

The key TEE standardization artifacts produced by GlobalPlatform are the *TEE system architecture*; *TEE management framework, TEE initial configuration and TEE APIs* , as described in [37].

5.3.1 Global Platform TEE system architecture

The TEE system architecture document [38] explains the hardware and software architectures behind the TEE. It introduces TEE management and explains concepts relevant to TEE functional availability in a device [37]. The document describes the general device architecture associated with the TEE and provides a high-level overview of the security requirements of a TEE [38], without mandating an implementation architecture. The document outlines different hardware and software architectures to answer to the TEE security and functional requirements. Computing devices offer a Rich Execution Environment (REE), providing a hugely extensible and versatile operating environment, introducing both new capabilities and vulnerabilities to software threats. The TEE System Architecture outlines the high-level functional and security features of a TEE operating alongside the REE and providing a safe area of the device to protect assets and execute trusted code [37].

5.3.2 TEE management framework

The Global Platform TEE Management Framework [39] describes the security model for the administration of Trusted Execution Environments (TEE) and of Trusted Applications (TA), as well as of the corresponding Security Domains (SD). It describes the roles and responsibilities of the stakeholders involved in the administration of a TEE and TA, the life cycle of administrated entities, mechanisms involved in administration operations, and the protocols used to perform such operations [39]. In particular, the framework defines methods for remotely and dynamically managing TEEs, including data and key provisioning, security domain management, trusted application (TA) management, audit, and overall TEE management [37]. The framework further presents the roles and responsibilities of the different stakeholders involved in the administration of TEEs and TAs, the life cycle of administrated entities, mechanisms involved in administration operations, and the protocols used to perform these operations [37]. The framework enables this by defining protocols and interfaces that can be accessed either through the GlobalPlatform TEE Client API [40] or via extensions to the recently released TEE Internal Core API [41].

5.3.3 TEE APIs

The Global Platform TEE API specifications include the TEE Client API Specification [40] and the TEE Core API Specification [41]. The TEE Client specification defines a communications API for connecting Client Applications executing in an REE with security related Trusted Applications running inside a TEE [40]. Global Platform TEE Client API specification considers a TEE that is a trusted environment within the main device system-on-a-chip. Additionally, it might complement traditional security environments such as a UICC SIM card. The document contains implementation guidelines for Client Applications running within the rich operating environment and which use Trusted Applications, Trusted Applications running inside the TEE which need to expose an externally visible interface to Client Applications and the TEE and the communications infrastructure required to access it [40].

The Global Platform Internal Core API Specification [41] defines a set of C APIs for the development of Trusted Applications running in TEEs, reachable through the GlobalPlatform TEE Client API, considered specifically protected against malicious attacks and only running code trusted in integrity and authenticity. The APIs defined in the Internal Core API specification are defined for the C programming language and provide a set of functionalities to TA developers, including basic OS-like functionalities (memory management, timer, and access to configuration properties), communication means with client Applications running in the REE; as well as facilities for trusted storage, cryptographic operations and peripheral interface and Event handling [41].

5.3.4 TEE Initial configuration

The GlobalPlatform Device TEE Initial Configuration document [42] describes common implementation requirements of core features of the GlobalPlatform Device Specification [37]. It defines configurations logically grouping together certain specifications to provide a coherent and consistent package. Furthermore, the document specifies configuration requirements for implementing the TEE initial configuration of GlobalPlatform TEE Specifications [42]. It refines the features of the internal core [40] and the client specification [41]. It is primarily targeted towards TEE vendors and application developers and is the basis for the development of a test suite for use in the compliance program. The TEE initial configuration combines the TEE Client API [40] and Internal Core [41] Specifications, updated in response to the latest feedback from the TEE testing and compliance ecosystem's live implementations. The configuration, along with the functional and security test suites, aims to enhance TEE interoperability and security and facilitate TEE vendors to ensure compliance with GlobalPlatform's Device Specification [37].

6 Conclusion

In this document we reviewed the interoperability of Trusted Execution Environments (TEEs). We started with a brief review of the approaches to implementing Trusted Execution Environments. We further focused on two TEE architectures (Intel SGX and AMD SEV) implemented by the most popular commodity server platform vendors (Intel and AMD respectively). Next, we provided an account of the implementation of a Trusted Application designed to run in a Trusted Execution Environment within project ASCLEPIOS. Further, we reviewed the major efforts towards interoperability between TEE architectures (Enarx, Asylo and OpenEnclave). Finally, we reviewed the major standardization efforts towards TEE interoperability. In particular, we discussed IETF TEEP, which was the target of contributions from project ASCLEPIOS; IETF RATS, which aims to describe a cross-platform TEE attestation protocol; and Global Platform, which focuses primarily on the standardization and interoperability of application management within TEEs.

This document describes a comprehensive view of TEE interoperability and may serve as a guide in choosing the suitable target TEE architecture for components within the ASCLEPIOS framework. This document complements the earlier deliverables D4.1 and D4.2, which focus on application management within TEEs and on remote attestation procedures for various TEE architectures. The review of interoperability efforts contained in this document contributes to the implementation of the Trusted Execution Environment Deployer (TEEPD) component. Implementation work on TEEPD will continue beyond WP4: in particular, TEEPD will be integrated with the ASCLEPIOS platform in WP5 and will be evaluated as part of the ASCLEPIOS demonstrator in WP6.

Bibliography

[1] Zhang, Fengwei, and Hongwei Zhang. "SoK: A study of using hardware-assisted isolated execution environments for security." *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016. 1-8.

[2] McKeen, Frank, et al. "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave." *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016. 1-9.

[3] Göttel, Christian, et al. "Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms." *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2018.

[4] Mingliang Pei, et al. "Trusted Execution Environment Provisioning (TEEP) Architecture". Internet Engineering Task Force (IETF). draft-ietf-teep-architecture-08 (work in progress), 2020.

[5] Henk Birkholz, et al. "Remote Attestation Procedures Architecture". Internet Engineering Task Force (IETF), draft-ietf-rats-architecture-02 (work in progress), 2020.

[6] Morbitzer, Mathias, et al. "Severed: Subverting amd's virtual machine encryption." *Proceedings of the 11th European Workshop on Systems Security*. 2018.

[7] R. Buhren, C. Werling och J.-P. Seifert, ”Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation,” i *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[8] Wilke, Luca, et al. "SEVurity: No Security Without Integrity--Breaking Integrity-Free Memory Encryption with Minimal Assumptions." *arXiv preprint arXiv:2004.11071* (2020).

[9] GlobalPlatform Technology, TEE Internal Core API Specification Version 1.2.1
Public Release, May 2019, Document Reference: GPD_SPE_010

[10] M. Bishop, ”Race Conditions, Files, and Security Flaws;or the Tortoise and the Hare Redux,” TechnicalReport CSE-95-8, University of California at Davis, 1995.

[11] Y. Swami, ”SGX Remote Attestation is not Sufficient,” i *BlackHat USA*, 2017.

[12] P. Kocher, J. Horn, A. Fogh, a. D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz och Y. Yarom, ”Spectre Attacks: Exploiting Speculative Execution,” i *40th IEEE Symposium on Security and Privacy*, 2019.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom och M. Hamburg, ”Meltdown: Reading Kernel Memory from User Space,” i *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[14] J. Szefer, ”Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses,” *J. Hardware and Systems Security,* vol. 3, pp. 219-234, 2019.

[15] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun och A.-R. Sadeghi, *Software Grand Exposure: SGX Cache Attacks Are Practical,* 2017.

[16] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang och C. A. Gunter, *Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX,* 2017.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin och T. H. Lai, *SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution,* 2018.

[18] W. He, W. Zhang, S. Das och Y. Liu, ”SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution,” i *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

[19] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas och C. W. Fletcher, ”MicroScope: Enabling Microarchitectural Replay Attacks,” i *Proceedings of the 46th International Symposium on Computer Architecture*, New York, NY, USA, 2019.

[20] V. Costan, I. Lebedev och S. Devadas, ”Sanctum: Minimal Hardware Extensions for Strong Software Isolation,” i *25th USENIX Security Symposium (USENIX Security 16)*, Austin, 2016.

[21] S. Hosseinzadeh, H. Liljestrand, V. Leppänen och A. Paverd, ”Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization,” i *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, New York, NY, USA, 2018.

[22] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen och A.-R. Sadeghi, ”DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization,” i *Proceedings of the 35th Annual Computer Security Applications Conference*, New York, NY, USA, 2019.

[23] D. Lee, D. Kohlbrenner, S. Shinde, D. Song och K. Asanović, *Keystone: An Open Framework for Architecting TEEs,* 2019.

[24] M. Morbitzer, M. Huber, J. Horsch och S. Wessel, ”SEVered: Subverting AMD's Virtual Machine Encryption,” i *European Workshop on Systems Security (EuroSec'18)*, 2018.

[25] Johnson, Simon, et al. "Intel® software guard extensions: Epid provisioning and attestation services." *White Paper* 1 (2016): 1-10.

[26] Open Enclave Software Development Kit https://openenclave.io/sdk

[27] Asylo project https://asylo.dev/

[28] Enarx project documentation https://github.com/enarx/enarx/wiki

[29] W. Arthur and D. Challener, A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security. Berkely, CA, USA: Apress, 2015.

[30] Paladi, Nicolae: *Trust but Verify-Trust Establishment Mechanisms in Infrastructure Clouds*. PhD Dissertation No. 104. Lund University, 2017.

[31] Tsai, Chia-Che, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library {OS} for Unmodified Applications on {SGX}." *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017.

[32] Wang, Huibo, et al. "Running Language Interpreters Inside SGX: A Lightweight, Legacy-Compatible Script Code Hardening Approach." *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 2019.

[33] Paladi, Nicolae, and Christian Gehrmann. "TruSDN: Bootstrapping trust in cloud network infrastructure." *International Conference on Security and Privacy in Communication Systems*. Springer, Cham, 2016.

[34] Paladi, Nicolae, Linus Karlsson, and Khalid Elbashir. "Trust anchors in software defined networks." *European Symposium on Research in Computer Security*. Springer, Cham, 2018.

[35] Medina, Jorge, Nicolae Paladi, and Patrik Arlos. "Protecting OpenFlow using Intel SGX." *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019.

[36] Protected Execution Facility On Power - Guerney Hunt, Ram Pai & Michael Anderson, IBM. At the 2019 OpenPOWER Summit, August 19-20 2019, San Diego, USA https://sched.co/SfRU

[37] Global Platform: Introduction to Trusted Execution Environments. May 2018. URL: https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf

[38] GlobalPlatform Technology TEE System Architecture Version 1.2. November 2018
Document Reference: GPD_SPE_009.

[39] GlobalPlatform Device Technology TEE Management Framework, Version 1.0. November 2016
Document Reference: GPD_SPE_120

[40] GlobalPlatform Device Technology TEE Client API Specification, Version 1.0. July 2010
Document Reference: GPD_SPE_007

[41] GlobalPlatform Technology TEE Internal Core API Specification Version 1.2.1. May 2019
Document Reference: GPD_SPE_010

[42] GlobalPlatform Device TEE Initial Configuration. Version 1.1, November 2016
Document Reference: GPD_GUI_069