UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Markus Punnar

Experimental Integration of the Smart-ID Service Into Intel SGX Enclaves

Master's Thesis (30 ECTS)

Supervisor: Peeter Laud, PhD
Co-supervisor: Armin Daniel Kisand, MSc

Tartu 2023

Experimental Integration of the Smart-ID Service Into Intel SGX Enclaves

Abstract:

Privacy-preserving services are becoming increasingly important as they allow untrusted remote servers to process sensitive information while preserving the privacy of that information. To ensure the security and privacy of such services, strong authentication mechanisms based on public-key cryptography are required instead of password-based authentication. While there are several standardized authentication services available, such as Smart-ID and mobile-ID, they are not yet integrated with Sharemind HI, a development platform for privacy-preserving services.

This thesis aims to address this gap by developing a proof-of-concept service that runs in a trusted execution environment and authenticates users using the Smart-ID service provider. By leveraging the existing public-key infrastructure, the proposed service would allow for the development of privacy-preserving applications on a national scale where sensitive data remains secure from remote untrusted servers and administrators.

To achieve this goal, the prototype was developed on the Sharemind HI platform, which simplifies the development of privacy-preserving applications and is based on the Intel SGX platform. The prototype demonstrates the feasibility of securely communicating with the Smart-ID service provider from a trusted execution environment and integrating Smart-ID authentication into the Sharemind HI platform. However, further work is required to optimize the prototype in terms of time and space and to develop a scalable solution for integrating external authentication providers without adding unnecessary complexity to the core modules.

Keywords:

User authentication, trusted execution environments, privacy-preserving technologies.

CERCS: P170 Computer science, numerical analysis, systems, control.

Smart-ID Teenuse Eksperimentaalne Integreerimine Intel SGX Enklaavidesse

Lühikokkuvõte:

Privaatsust säilitavad teenused muutuvad üha olulisemaks, kuna need võimaldavad tundmatutel serveritel töödelda tundlikku teavet, säilitades samal ajal selle teabe privaatsuse. Nende teenuste turvalisuse ja privaatsuse tagamiseks on paroolipõhise autentimise asemel vaja tugevaid autentimismehhanisme, mis põhinevad avaliku võtme krüptograafial. Kuigi saadaval on mitmeid standardiseeritud autentimisteenuseid, näiteks Smart-ID ja mobiil-ID, ei ole nad veel integreeritud privaatsust säilitavate teenuste arendusplatvormiga Sharemind HI.

Käesoleva töö eesmärk on arendada välja autentimisteenuse prototüüp, mis töötab usaldusväärses täitmiskeskkonnas ja autentib kasutajaid Smart-ID teenusepakkuja abil. Olemasoleva avaliku võtme infrastruktuuri kasutamine võimaldaks arendada privaatsust säilitavaid rakendusi riiklikul tasandil, kus tundlikud andmed on kaitstud tundmatute serverite ja administraatorite eest.

Selle eesmärgi saavutamiseks loodi prototüüp Sharemind HI platvormil, mis lihtsustab privaatsust säilitavate rakenduste arendamist ja põhineb Intel SGX platvormil. Prototüüp suhtleb turvaliselt usaldusväärse täitmiskeskkonna kaudu Smart-ID teenusepakkujaga ning selle abil on võimalik integreerida Smart-ID autentimine Sharemind HI platvormiga. Edasiseks rakendamiseks on vajalik prototüübi optimeerimine ning skaleeritava lahenduse väljatöötamine väliste autentimisteenuse pakkujate integreerimiseks ilma põhimoodulitele tarbetut keerukust lisamata.

Võtmesõnad:

Kasutajate autentimine, usaldatavad täitmiskeskkonnad, privaatsust säilitavad tehnoloogiad.

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll.

Table of contents

Introduction

Authentication and authorization are essential in ensuring the security and integrity of computer systems and networks. Authentication verifies the identity of a user or system before allowing access to resources, while authorization determines what level of access the authenticated user or system has to those resources. Without proper authentication and authorization measures in place, unauthorized users could gain access to sensitive data or systems, potentially causing damage or harm. Therefore, these security measures are necessary to protect against data breaches, cyber attacks, and other malicious activities.

The most common authentication method involves an individual entering a username and a password, where the username is considered public knowledge, and the password is only known to the legitimate user. Despite its simplicity and widespread use, password authentication suffers from various weaknesses, including password reuse across multiple services, easily guessable passwords via brute force or dictionary attacks, or even social engineering tactics. Therefore, a growing number of services are adopting authentication systems based on public-key cryptography. These systems require users to decrypt a challenge sent by the system or digitally sign some data that can later be verified to validate the user's authenticity. Estonia and the Baltics have implemented various such authentication mechanisms, including ID-card, mobile-ID, and Smart-ID.

Trusted execution environments (TEE) refer to a group of technologies that enhance privacy by allowing secure code to run in unsecured computing environments. TEEs are particularly useful in implementing privacy-sensitive services that require a high level of security where the server host cannot be trusted. In some cases, such as with the Intel SGX platform [1], TEE implementations provide additional measures to ensure data privacy by encrypting working memory and implementing access controls on the processor chip level. This approach completely hides the data from the host machine,

thereby adding an extra layer of security to the TEE mechanism.

The objective of this thesis is to examine the feasibility of integrating a privacy-preserving service with Smart-ID, a commonly used authentication and digital signing solution in the Baltics. To achieve this goal, a prototype was developed to verify the identity of users within a trusted execution environment. The Sharemind HI platform was used to create the prototype, as it is a commercial product developed by Cybernetica AS with the primary objective of facilitating the development of privacy-preserving services. This platform utilizes Intel SGX as its underlying trusted execution environment.

The focus of this thesis was on the design and implementation of a prototype service for integrating a privacy-preserving solution with Smart-ID. The author did not enhance the Sharemind HI platform with additional functionalities in order to create the prototype. The author's work specifically involved the following activities:

- Designing two new server-side components that would facilitate the integration between Sharemind HI and the Smart-ID service.
- Implementing the prototype service to enable secure communication with the Smart-ID service provider and obtain user information.

This thesis is comprised of six chapters, in addition to the introduction. Chapter 1 presents an overview of the necessary background knowledge required to understand the design and implementation of the prototype service. Chapter 2 outlines the motivation for integrating privacy-preserving services with Smart-ID. Chapter 3 provides a detailed discussion of the architectural decisions made during the development process. Chapter 4 focuses on the implementation details of the prototype service. Chapter 5 briefly discusses the security implications of integrating Smart-ID to Sharemind HI as well as benchmark results for time and memory consumption are presented, and Chapter 6 concludes the thesis with a discussion of future work.

In the course of writing this thesis, ChatGPT was utilized by the author to refine and enhance the phrasing of selected sentences, where deemed appropriate.

Glossary

API Application Programming Interface

CA Certificate Authority

CPU Central Processing Unit

CSR Certificate Signing Request

DFC Dataflow Configuration File

DHKE Diffie-Hellman Key Exchange

EDL Enclave Definition Language

EPC Enclave Page Cache

EPID Enhanced Privacy ID

HSM Hardware Security Module

HTTP HyperText Transfer Protocol

IAS Intel Attestation Service

IETF Internet Engineering Task Force

JSON JavaScript Object Notation

MAC Message Authentication Code

MITM Man-In-The-Middle

PKI Public Key Infrastructure

PRF Pseudo-Random Function

PRM Processor Reserved Memory

QE Quoting Enclave

RAM Random Access Memory

RP Relying Party

SAML Security Assertion Markup Language

SDK Software Development Kit

SGX Software Guard Extensions

SSL Secure Socket Layer

SSO Single Sign-On

TCP Transmission Control Protocol

TCS Thread Control Structure

TEE Trusted Execution Environment

TLS Transport Layer Security

1 Preliminaries

This chapter provides an overview of the fundamental background knowledge that is necessary to understand the results presented in this thesis. First, chapter 1.1 offers an introduction to the concept of digital identity, encompassing key terminology and trust assumptions. Subsequently, chapter 1.2 discusses TLS, which serves as the underlying protocol for securing communications between parties over the internet. Chapter 1.3 introduces the concept of threshold cryptography, along with the SplitKey protocol that is implemented by the Smart-ID service and chapter 1.4 details the interface between the Smart-ID service and its users. Chapters 1.5 and 1.6 provide an in-depth overview of the underlying technologies employed in this thesis - Sharemind HI and Intel SGX.

1.1 Digital identity

In contemporary society, digital identity has become a critical aspect of daily life. It refers to the electronic portrayal of an individual's identity, personal information such as their name and national identity number. To establish trust in either the physical and digital domain, it is essential to have a reliable authority that verifies a person's identity. In a psychical world, this is generally accomplished by issuing an official document, such as a passport or an ID card, to the individual, which can be authenticated by other parties and is challenging to replicate. Such a document can serve as proof of a person's identity.

In the digital domain, physical identification documents are not viable for proving a person's identity to another individual. Instead, web services have traditionally relied on passphrases, which are expected to be known only to the legitimate user, for user authentication. However, this approach is associated with several limitations. Firstly, passphrases are susceptible to brute-force and dictionary attacks, as well as data leakages, which can compromise the security of the user's identity [2]. Secondly, the user-

name alone is insufficient to establish the real-world identity of the user, making it impractical for use cases that require the verification of the user's actual identity, such as digital signatures.

The concept of digital identity has undergone significant evolution over time, driven by the increasing need for enhanced security and privacy in online transactions. By leveraging asymmetric cryptographic primitives, it is possible to elevate authentication and digital transactions to a level equivalent to that of handwritten signatures, as per eIDAS regulations [3]. In this approach, an individual's personal information is bound to their public key through a public-key certificate, which is digitally signed by a trusted third party known as a certificate authority. This certificate authority is responsible for verifying that the public key corresponds to the intended individual or entity, and their certificates are publicly available for others to extract the associated identity and public key information. To authenticate, an individual must demonstrate their possession of the corresponding private key by either decrypting a challenge or digitally signing some data. Additionally, digital signatures can be verified by other parties using the public key information from the associated certificate [4].

1.2 TLS

1.2.1 Definition and history

TLS, formerly also known as SSL is a cryptographic protocol which provides communication security over a computer network [5]. TLS operates on top of the transport layer and and is agnostic to the underlying application protocol. It is used with many application level protocols, however, it is most widely known for its use in securing HTTP traffic.

The first version of SSL, called SSL 1.0 was developed by engineers and scientists at Netscape in 1995. However, since the protocol had serious security flaws, it was never

released to the public. Over the next few years, next revisions of the protocol called SSL 2.0 and SSL 3.0 were developed and released. SSL 2.0 was quickly discarded due to numerous security and usability issues. SSL 3.0 was a complete redesign of the protocol, addressing the issues present in its predecessors. It was the final revision of SSL and all newer TLS versions are based on the same concepts [6].

TLS 1.0 was an upgrade over SSL 3.0 published in 1999. The name change from SSL to TLS was mainly for differentiating the newer revisions as the closed-source development of SSL at Netscape transitioned to an open standardization process led by IETF [7]. Newer versions of TLS focus on extending the support for different use cases, deprecating the use of insecure hash functions, pseudo-random functions, message authentication codes and encryption schemes. Moreover, recent version of TLS aim to make the TLS handshake process more efficient and provide additional security properties such as perfect forward secrecy. Currently, there are two TLS versions being actively used. TLS 1.2 is the most widely used TLS version since TLS 1.1 was deprecated by all major browser vendors in 2020. However, TLS 1.3 was released in 2018, providing more secure cipher suites and faster handshake [8]. Today, TLS 1.3 is supported by all major browsers and is being widely adopted by most web servers as well as being supported by popular commercial and open-source libraries such as WolfSSL and OpenSSL.

1.2.2 TLS handshake

Since applications can communicate with or without TLS, the client needs to notify the server that it requests a TLS connection. This is most widely done by using a different port for TLS connections. For example, the default port for HTTP traffic is 80 and the default port for HTTPS (HTTP over TLS) traffic is 443 [9]. The main objective of a TLS handshake is for the parties to agree on a session key which can be used to securely exchange application data using a symmetric cipher. During a basic server-authenticated

TLS 1.2 handshake, the messages shown in Figure 1 are exchanged between the client and the server after the parties have established a TCP connection.

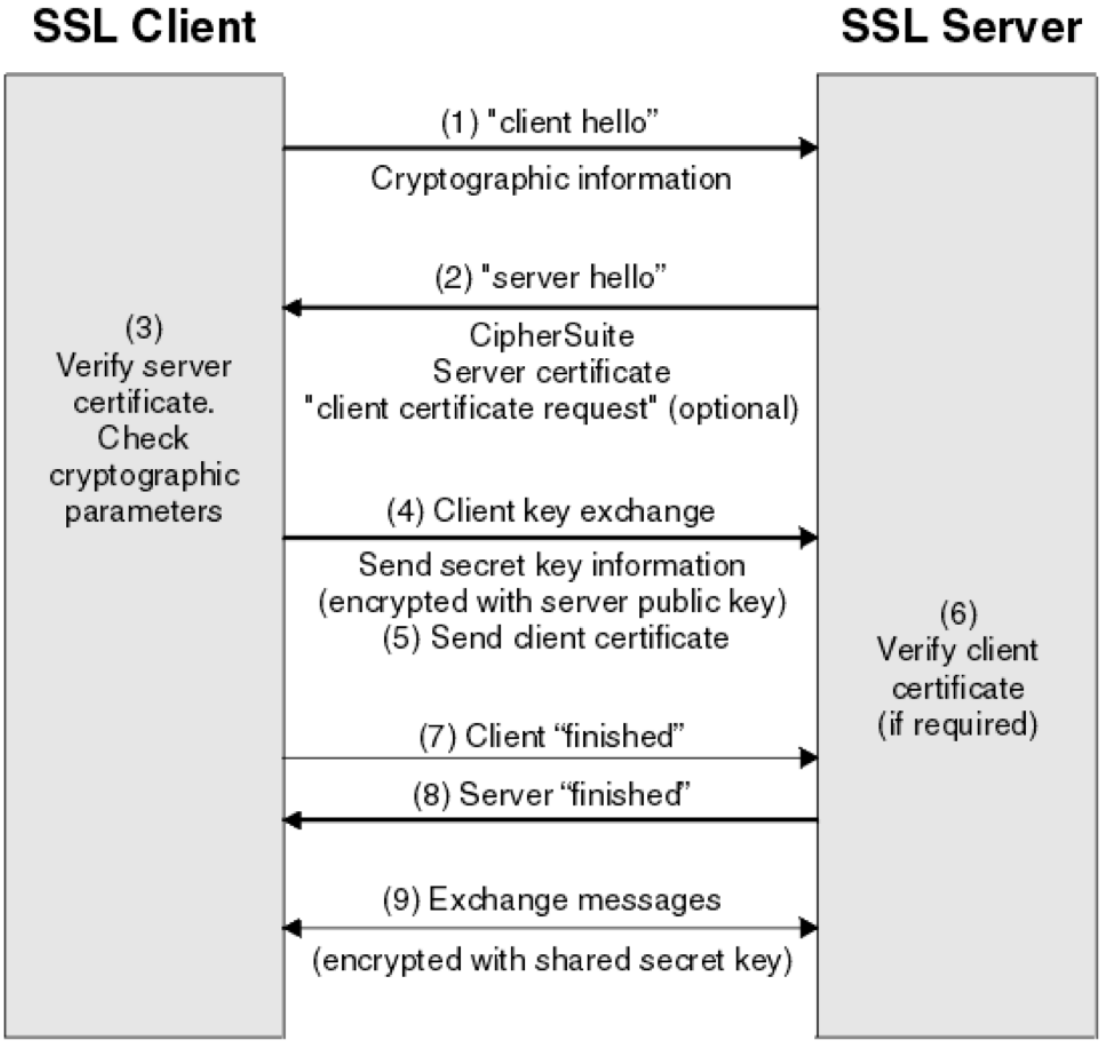

Figure 1. A standard TLS 1.2 handshake [10]

The main difference in the handshake in TLS 1.3 is that it attempts to complete the handshake in a single round-trip whereas a standard TLS 1.2 handshake is done in multiple round-trips as illustrated in Figure 1. This is possible mainly due to the fact that deprecating many insecure key exchange algorithms, hash functions, encryption functions and encryption modes, the number of supported cipher suites was reduced

from 37 to only 5 [8].

In the process of designing the TLS 1.3 protocol, it was identified that a number of TLS implementations had incorrectly implemented the protocol's version negotiation feature which allows both the client and the server to agree on a common TLS version. In addition to other faulty assumptions made during implementation, this forced the introduction of additional measures in TLS 1.3, such as relocating the version identifier to an extension and making TLS 1.3 appear similar to TLS 1.2 for various network middleboxes [11]. Consequently, TLS 1.3 is a more complex protocol than necessary due to the need to maintain backwards compatibility. This highlights how fragile the protocol is in practice due to decisions made during protocol design and implementation errors.

1.3 SplitKey protocol and the Smart-ID service

1.3.1 Overview of threshold cryptography

The field of threshold cryptography studies cryptosystems that enhance the protection of information by distributing the private key used for decryption and digital signatures among multiple participating parties. This implies that in order to carry out cryptographic operations involving the private key, several parties must cooperate. The number of required parties depends on the specific protocol and is called a threshold, resulting in these schemes being called (t, n)-threshold systems where n is total number of parties participating in the protocol and t is the threshold. In most protocols, t is strictly smaller than n, making the systems fault-tolerant when a single party is unavailable or refuses to cooperate.

The first cryptosystem with such threshold properties with a proof of security was published in 1994 and constructed a shareable variant of RSA [12]. Since threshold cryptosystems require a public and private key, these schemes can be built on top of

many existing asymmetric cryptosystem such as RSA, ECDSA and ElGamal. Historically only governmental agencies and certificate authorities with very important secrets used threshold cryptosystems to provide additional security. In recent years, however, threshold cryptosystems have gained traction in the general public as well, with NIST publishing a roadmap toward criteria for threshold schemes for cryptographic primitives in 2020 [13].

The main benefit that a threshold cryptosystem provides over a conventional asymmetric cryptosystem is the elimination of the single point of failure. In conventional system, the trust and responsibility is in the hands of a single individual, making him an obvious target for carrying out attacks against the system. In addition to becoming compromised, the individual can also become unavailable, introducing bottlenecks in the system. The computer itself can also become a point of failure in the case of a malfunction or a side-channel attack, where an adversary observes computer performing cryptographic operations to extract information about the private key or other secrets. Such attacks have been conducted in lab conditions since a novel attack by Kocher [14].

A threshold cryptosystem involves a group of independent entities who are responsible for generating their own unique private keys and calculating corresponding public key shares. These shares are then distributed and a common public key is computed. During the signing process, each party calculates its signature share, which is used to form a complete signature when combined with the shares of other parties. As long as the number of signature shares is equal to or greater than the threshold level, any party can combine these shares to create a complete signature, which can then be verified using the public key.

1.3.2 SplitKey scheme

Smart-ID is an tokenless authentication solution widely used by banks, e-commerce companies and government agencies to securely authenticate users. Before Smart-ID,

companies relied on possession based authentication means such as passwords and code cards which could be guessed or replicated. For some services, methods using hardware based security such as authentication using ID-cards or Mobile-ID were also available, but these methods require the client to either have a special purpose SIM-card or a ID-card reader to carry out cryptographic operations using the private key. Smart-ID, however, uses a $(2, 2)$-threshold cryptography scheme where the user's smartphone and the Smart-ID service provider are the parties participating in the authentication and digital signing processes, providing a more convenient user experience while not compromising the security provided by hardware based methods. Smart-ID authentication process to a standard web service consists of multiple distinct parties which can be seen in Figure 2. In this work the focus is on relying party integration. A relying party in Smart-ID is an organization or a service using the Smart-ID solution to authenticate its users and to sign documents.

1. A physical user who wants to authenticate to an online service
2. A user interface for the service - typically a browser
3. Smartphone belonging to the user
4. Relying party (RP) server
5. Smart-ID service provider

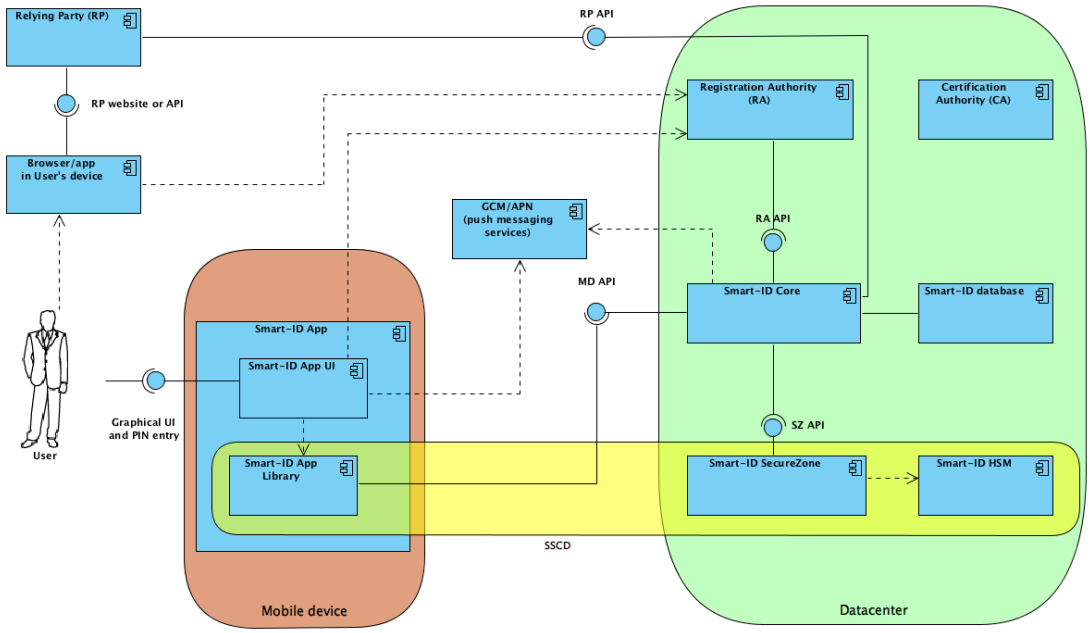

Figure 2. Overview of Smart-ID components [15]

The exact architecture of the Smart-ID service provider is out of the scope of this work. Next we will look at the details of Smart-ID authentication and digital signing processes. Note that this will not be a complete overview of these processes as additional measures are implemented against key cloning attacks, PIN brute force attacks, side-channel attacks and various transport level attacks.

Authentication process [15]

After the user navigates to the RP website, chooses the Smart-ID authentication method and enters the user identifier, such as national personal code, RP server sends authentication request to the Smart-ID service provider. After that

1. RP generates a random SHA-2 family hash to be signed during the authentication process and requests a new authentication session creation.

2. Smart-ID service provider creates a new authentication session and returns the

session ID to the RP.

3. RP computes the verification code of the authentication request and displays it to the user. The verification code is calculated from interpreting the two rightmost bytes of the SHA256 hash as a big-endian unsigned integer and extracting the last 4 digits in decimal.

4. Smart-ID service provider sends a push notification about the authentication request to the user's mobile device.

5. Smart-ID App connects to the Smart-ID service provider, requests the authentication request details and computes the verification code, asking the user for its PIN1.

6. After entering the PIN1, the Smart-ID App performs its signature generation steps with the authentication key pair and sends the result to the Smart-ID service provider. See the signature computation chapter for details.

7. Smart-ID service provider completes the composite signature computation steps and returns the authentication response to the RP.

8. RP verifies the validity of the authentication response and creates a new session for the authenticated user. RP must verify the validity of the end user's certificate and the signature.

Signature computation [16]

1. To sign a message m, first hash it using SHA-2 family hash function and pad it with PKCS #1 v1.5 algorithm.

2. Decrypt private key share $d_1^{'}$ with a key derived from the PIN and compute signature share $y = m^{d_1^{'}}$.

3. Send signature share y and message m to the server. For authentication, message m is a randomly generated hash.

To sign a hashed and padded message m, the Smart-ID app asks for a PIN from the user, decrypts its private key share $d_1^{'}$ using the key derived from the PIN and computes its signature share $y = m^{d_1^{'}}$. For authentication, the digest H is generated randomly by the RP and padded using the PKCS #1 v1.5 algorithm. Client then sends the signature share y and the message m to the Smart-ID service provider. The service provider then finishes the computation of the signature by

1. Compute the digest $H(m)$ and use the same aforementioned padding algorithm.
2. Calculate the client's signature $s_1 = y \cdot m^{d_1^{''}}$ and verify its validity by checking if $s_1^e = m \mod n_1$.
3. Calculate the server's signature share $s_2 = m^{d_2} \mod n_2$ and create the composite signature by using the Chinese Remainder Theorem $s = C_{n_1,n_2}(s_1, s_2)$ which satisfies $s \equiv s_i \mod n_i$ for $i \in \{1, 2\}$. This is done by finding constants α, β during key generation which satisfy $\alpha n_1 + \beta n_2 = 1$ via Extended Euclidean Algorithm.
4. Verify the validity of the composite signature using regular RSA signature verification methods.
5. Send the composite signature back to the client.

Note that in a production environment, additional security measures such as retry counters are included to increase security, which are omitted in this overview. However, using these base steps, an end user can authenticate to different service providers and create electronic signatures compliant with the eIDAS regulations using their smartphone as a party in a threshold cryptography scheme.

1.4 Smart-ID integration details

This chapter provides a comprehensive overview on the specifics of HTTP level requests and responses involved in Smart-ID authentication and digital signing sessions. These two sessions demonstrate analogous patterns involving two distinct HTTP services: one for initiating the session, and the other for polling the final outcome. Unless explicitly indicated, all information provided in this chapter is derived from the official Smart-ID integration guide [17].

1.4.1 Authentication session

The process of authenticating users in a relying party service begins with the initiation of an authentication session by the relying party, prompted by a user's request to log in. The Smart-ID service offers several means of identifying users, including a private identifier unique to the relying party, a document number, or a semantic identifier. This discussion, however, will concentrate on semantic identifiers, particularly the identity code.

To initiate the authentication process, the relying party submits an HTTP POST request to the Smart-ID service provider. In the case of semantic identifiers, the relevant endpoint is */authentication/etsi/:semantics-identifier*. The request body must include the relying party's UUID and name, which were acquired by registering the service with the Smart-ID service provider, along with a base64 encoded hash (generated using a SHA-2 family hash function) over a set of randomly generated bytes. Finally, the relying party must supply a list of interaction flows in order of preference, with only the simplest interaction flow that displays text and a PIN prompt on the user's smartphone being used for this purpose. An example payload for the session initialization request can be found in Listing 1.

```
{
   "relyingPartyUUID":"00000000-0000-0000-0000-00000000000",
   "relyingPartyName":"DEMO",
   "hashType":"SHA256",
   "hash":"mpLS4Slsv40OONo1w4JlnhhPXgqK12BVU0bY8ebVkAI=",
   "allowedInteractionsOrder":[
      {
         "displayText60":"Enter PIN1 to sign in",
         "type":"displayTextAndPIN"
      }
   ]
}
```

Listing 1. Example Smart-ID session initialization request body

When the Smart-ID service provider successfully finds a registered user with the provided semantic identifier, it generates a session identifier in response, which the relying party can use to later check the status of the session. An example of the session initialization response's payload is provided in Listing 2.

```
{
   "sessionID":"e65cca40-31e9-4907-94f7-c6c348b104e4"
}
```

Listing 2. Example Smart-ID session initialization response body

The HTTP roundtrip described above fulfills the first two steps of the authentication process outlined in chapter 1.3.2. Subsequently, the authentication flow proceeds according to the aforementioned chapter. During steps 3 to 6, the relying party server regularly polls the Smart-ID service provider to determine whether the Smart-ID service provider and the user's smartphone have completed the authentication steps. To accomplish this, the relying party server issues an HTTP GET request to the */session/:sessionID* endpoint, using the previously acquired session identifier. The Smart-ID service provider responds with either an intermediate response indicating that the authentication process is still ongoing, or a final poll response containing the user's signature and certificate details. Listings 3 and 4 provide possible successful responses from the service provider. The user's signature and certificate values are encoded in

base64 and are not included in the example response due to their size. The *endResult* field indicates whether the authentication flow was successfully completed, an error occurred, or the user cancelled the procedure.

```
{
   "state":"RUNNING"
}
```

Listing 3. Example intermediate Smart-ID session status response body

```
{
   "state":"COMPLETE",
   "result":{
      "endResult":"OK",
      "documentNumber":"PNOEE-39803022754"
   },
   "signature":{
      "value":"Omitted for brevity",
      "algorithm":"sha256WithRSAEncryption"
   },
   "cert":{
      "value":"Omitted for brevity",
      "certificateLevel":"QUALIFIED"
   },
   "interactionFlowUsed":"displayTextAndPIN"
}
```

Listing 4. Example final Smart-ID session status response body

Upon receiving a successful response from the Smart-ID service provider and verifying the received signature and certificate values' validity, the relying party can verify the user's identity and create an internal session using cookies, JWTs, or some other method, allowing the user to access content that would otherwise be restricted.

1.4.2 Digital signing session

The process of digitally signing data shares numerous similarities with the authentication flow discussed earlier. The key distinction lies in the initialization endpoint used to

initialize the session. In instances where the relying party requires advanced electronic signatures (AdES), the process entails two sessions: one for selecting a specific signing certificate in situations where a user possesses multiple certificates across various mobile devices, and another for actually signing the data. To trigger the certificate selection flow using the semantic identifier, the relying party submits an HTTP POST request to the */certificatechoice/etsi/:semantics-identifier* endpoint, providing solely the RP UUID and name. Upon obtaining the session identifier from the service provider, the relying party will utilize the identical endpoint for monitoring the session status as for authentication. Subsequently, the *documentNumber* field in the final poll response will indicate the document number selected by the client. Alternatively, the relying party may opt to use the document number obtained from a prior authentication poll response, thus bypassing the need for a separate certificate selection step.

Following the successful acquisition of the correct document number, the relying party can initialize the digital signing session via the */signature/etsi/:semantics-identifier* endpoint. The request payload for this endpoint mirrors the authentication initialization request as illustrated in Listing 3, with the exception of the *hash* field, which now contains the base64 encoded hash output of the data intended for signing. Once the request is initialized, the service provider and user's smartphone will jointly calculate a composite signature for the specified data. Upon completion, the resulting signature will be transmitted to the relying party via the same session status endpoint.

1.5 Intel SGX

Numerous software applications need to process sensitive information such as passwords, cryptographic keys, credit or health records. Only designated individuals should be allowed to access this data. The operating system enforces security policies that protect sensitive information from other users and applications by restricting access to other users' files and memory space. Applications themselves can protect data at rest

and in transit by conventional encryption methods. However, there exists little protection from malicious parties with administrative privileges over the operating system. Furthermore, during the processing of sensitive information it is accessible in memory without applied encryption.

Intel Software Guard Extensions (SGX) [1] is an extension of the instruction set available on Intel processors which enables developing secure applications when even the host operating system is not trusted. Intel SGX allows applications to create and operate with hardware-protected memory partitions called *enclaves*. Enclave provides a protected memory region with confidentiality and integrity guarantees, which hold even when a privileged adversary or malware is present in the system. By running sensitive information processing code in an enclave, it is possible to considerably reduce the attack surface of the application. The differences between the attack surface for regular and SGX applications can be seen in Figure 3. In addition to enclaves, Intel SGX relies on two more concepts to protect data - *attestation* and *data sealing*.

Figure 3 demonstrates the effectiveness of enclaves in significantly reducing the attack surface for an application. In the absence of enclaves, the entire application, as well as the underlying operating system, hypervisor, and hardware, must be trusted. Malware or malicious privileged users can exploit vulnerabilities in any of these components to launch an attack on the application. For instance, gaining privileged access at the operating system level may enable an attacker to extract the application's entire working memory, exposing sensitive data. This poses a security risk, as stakeholders are required to trust multiple third parties to ensure the application's security. However, with the use of enclaves, only the Intel SGX-capable hardware and correct implementation of the interface between the untrusted and trusted sections of the application need to be trusted. Further information regarding the interface description details can be found in Chapter 1.5.4.

All of the information regarding SGX in this chapter comes from Costan and De-

vadas' article Intel SGX Explained [18], unless explicitly cited otherwise. The article provides an excellent and thorough overview of the Intel SGX platform. The reader is heavily encouraged to refer to the article for any additional details on the topics presented in this chapter.

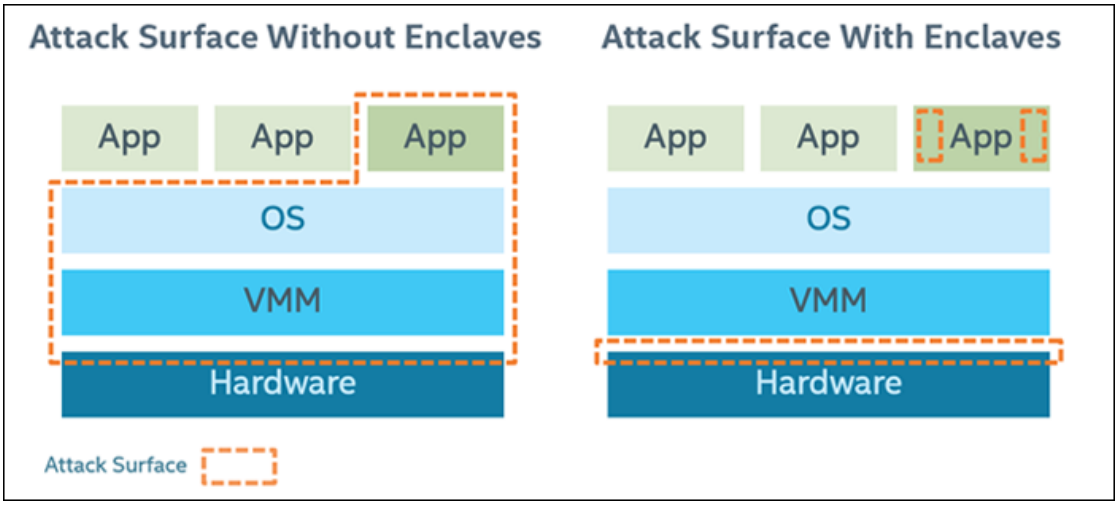

Figure 3. Attack surface with and without enclaves [19]

1.5.1 Enclaves

The enclave is a central component of SGX. It contains all of the code and additional data needed to perform operations on sensitive information. For example an enclave can decrypt multiple table files, aggregate the entries based on some business logic and encrypt the result again. The enclave's code and data is stored in Processor Reserved Memory (PRM) that cannot be directly accessed by other software, including system software. Direct memory access targeting the PRM is also rejected by the CPU. The PRM contains an Enclave Page Cache (EPC) which is split into 4kB pages that can be assigned to different enclaves, allowing the support of multiple enclaves in a system at the same time. This memory is managed by the same system software that manages the rest of the memory, using SGX instructions to allocate and deallocate EPC pages.

Since system software is untrusted by design, SGX processors check the correctness of allocation decisions. It is also possible for multiple logical processors to concurrently execute the same enclave's code via different threads using a Thread Control Structure (TCS).

1.5.2 Attestation

One of the main use cases of SGX is running trusted code on a remote untrusted host machine. Therefore a mechanism is needed to verify that the code and data have not been modified. This is achieved by using a cryptographic protocol called software attestation. This process also produces a shared key between two parties which can be used to establish a secure channel between the participants, allowing for exchange of secrets. Intel SGX offers two different types of attestation protocols - local attestation and remote attestation. Remote attestation requires that the parties trust Intel to verify the enclave and assure its authenticity.

Enclave measurement

When an enclave is built and initialized, SGX will generate a cryptographic log in order to build an enclave identity, which is a 32-byte hash digest of the log called the *MRENCLAVE* which can be thought of as the fingerprint of an enclave. The cryptographic log consists of the following:

- The enclave content - code, additional data, stack and heap
- Location of each memory page within the enclave
- Security flags

Local attestation

It may be necessary to create multiple enclaves in a system due to enclave size restrictions or a separation of concerns. Local attestation is a process used by two enclaves running on the same platform to prove to each other that they can be trusted. The result of a successful local attestation provides an assertion between two enclaves so they can exchange information securely. Enclaves use multiple special purpose CPU instructions for attestation, such as *EREPORT*, which generates a structure called REPORT, containing the *MRENCLAVE*, certificate-based identity, user data and a MAC tag over the contents. To derive the keys used for the local attestation process, enclaves use the *EGETKEY* instruction which can be used to derive sealing and report keys from key material such as *MRENCLAVE* and secrets embedded in the processor. Local attestation then performs the following steps assuring enclave A is attested by enclave B

1. B retrieves its *MRENCLAVE* value and sends it to A via an untrusted channel
2. A uses *EREPORT* to generate a REPORT using B's *MRENCLAVE* and sends it to B. This can also contain Diffie-Hellman key exchange (DHKE) data for trusted channel creation.
3. B calls *EGETKEY* to get a report key to verify the REPORT. If the report is successfully verified, then B assured that A runs on the same platform, because the report key is specific to the platform.
4. B uses A's *MRENCLAVE* from the REPORT to create a REPORT for A.
5. A verifies it similarly to Step 3.
6. If needed, then A and B can create a secure channel using DHKE using user data fields in the REPORT and exchange messages using the shared symmetric key.

Remote attestation

Another type of attestation supported by SGX is remote attestation. It is used to verify to a remote client that the enclave is running trusted code and to establish an authenticated communication channel between the enclave and the client. In order to accomplish this, a few additional components are introduced.

First, a special purpose enclave provided by Intel, called the Quoting Enclave (QE), is present on all SGX platforms. The objective of the Quoting Enclave is to verify local attestation reports from other enclaves, convert them into a quote and sign the quotes using a device-specific Intel Enhanced Privacy ID (EPID) key.

Furthermore, remote attestation requires a trusted third party, namely the Intel Attestation Service (IAS) to be present in order to verify the EPID signature over a quote.

The remote attestation process can be thought of as a slightly modified Sign-and-mac (Sigma) protocol between the application and the challenger [20]. The protocol needs to be modified due to the application enclave not being able to provide all the necessary information needed to participate in DHKE. The process therefore includes the enclave being attested, the Quoting Enclave, the untrusted application, the IAS and a challenger called a relying party. Remote attestation can be broken down into seven steps.

1. The attestation process is initiated by requesting to create an authenticated channel between the enclave and the challenger. The challenger sends the first protocol message, consisting of a challenge and a nonce.

2. The application, running in the untrusted operating system, requests an attestation report from the enclave and passes on the challenger's nonce.

3. The enclave generates the report, using the *EREPORT* instruction, and a manifest, consisting of the user data section from the report and optionally the nonce along

with a public key for the challenger to create the authenticated channel. Both the report and manifest are returned to the untrusted application.

4. The application forwards the report to the QE, which verifies the report using local attestation, converts it into a quote, and signs it with the EPID key.

5. The QE returns the signed quote to the application.

6. The application returns the quote from QE and the manifest from the enclave to the challenger.

7. The challenger sends the quote to IAS, who verifies the EPID signature over the quote and returns the result of the verification.

1.5.3 Data sealing

Sometimes it may be necessary to preserve enclave data or state following events during with an enclave is destroyed, which can happen when

- Application closes the enclave
- Application itself is closed
- Platform is hibernated or shut down

Normally, enclave data and state are lost when an enclave is closed. In order to preserve the data, it must be stored outside enclave boundaries. In order to protect this data, a mechanism called data sealing is in place in SGX, which allows the enclave to retrieve a key unique to an enclave based on the enclave measurement. This key can be retrieved using an *EGETKEY* instruction. It is also possible to retrieve a key based on the enclave author so that all enclaves of the author on the same platform can decrypt the data. It is important to note that this method goes against the goal of isolating the enclaves and minimizing attack surface. If any of the enclaves of the author are

compromised then all of the shared data is leaked. However, the compromised enclave will not gain access to any other data which limits the severity of the leak.

1.5.4 The Enclave-Definition Language

The Enclave-Definition Language (EDL) is a domain-specific language that is used in Intel SGX applications. It allows enclave developers to define enclave calls (ECALLs) and outside calls (OCALLs), which specify how data moves in and out of enclaves respectively [21]. The tool called *sgx_edger8r*, which is part of the Intel SGX software development kit (SDK), uses EDL as input. This tool produces C wrapper functions that implement multiple input and boundary checks during runtime for security and serve as the only access points to an enclave. Additionally, the wrapper functions copy input and output parameters from untrusted memory to trusted memory and vice versa. This behavior also applies by default to pointer-type arguments, in which case the entire array is copied. It is possible to provide modifiers to arguments, such as opting out of the copying functionality for pointers and leaving the pointer verification to the enclave developer. A comprehensive overview of the modifiers and how the EDL-generated code handles different argument types is available in the EDL whitepaper [21].

The EDL file consists of two sections - a trusted block and an untrusted block. As the names imply, ECALLs are defined in the trusted block and OCALLs in the untrusted block. Listing 5 illustrates the structure of an EDL file along with the contents of each block. Code generation from an EDL file takes place during enclave build time and produces two header files: *enclave_u.h* for OCALL wrappers and *enclave_t.h* for ECALL wrappers.

```
enclave {
   // Include files
   // Import other EDL files
   // Data structure declarations to be used as parameters of the function prototypes

   trusted {
      /* Define ECALLs here */
      // Included files will be inserted in the trusted header file (enclave_t.h)
      // Trusted function prototypes
      void test_ecall([user_check] int* ptr);
   };

   untrusted {
      /* Define OCALLs here */
      // Included files will be inserted in the untrusted header file (enclave_u.h)
      // Untrusted function prototypes
      void test_ocall([user_check] int* ptr);
   };
}
```

Listing 5. Structure of an EDL file [21]

1.6 Sharemind HI

1.6.1 Overview

Sharemind HI is a platform to develop data-driven services which require strong privacy, confidentiality and integrity guarantees throughout the entire lifecycle of the service. It achieves these guarantees by using modern cryptographic schemes and a trusted execution environment (TEE) technology. TEE technology is used to provide secure memory regions for confidential parts of the application which are isolated from the untrusted parts of the host machine using trusted hardware. Sharemind HI uses Intel SGX as its TEE technology to provide these memory regions in the form of enclaves [22]. Incoming data to a Sharemind HI instance is encrypted at the source by the data producer and then forwarded to the Sharemind HI service. The host of the service can only ac-

cess the data in encrypted form during all stages of processing, including during actual processing tasks.

Sharemind HI leverages the hardware-based security guarantees provided by Intel SGX and expands upon them by providing organizational measure and an access control framework. It aims to abstract away common concerns for privacy sensitive data processing applications such as key management and cryptographic operations. Often these applications involve multiple stakeholders, each with different roles and access rights. Sharemind HI allows for precise control over what data is accessible to whom, while preserving all security and privacy guarantees. This leaves the application developer only with the tasks to implement the data processing business logic and dataflow configuration.

1.6.2 Configuration

A Sharemind HI instance consists of multiple different components and stakeholders with different access permissions. There can exist any number of computational tasks with different inputs and outputs. A group of similar data is aggregated by a primitive called a data topic. These topics also serve as protection boundaries as a party or a task requires separate permission to read or write to each topic. All stakeholders, tasks, topics and access permissions are configured in a designated dataflow configuration file (DFC). This configuration file is written in YAML format and contains the following elements [23]:

- A listing of all stakeholders who are configured to access the system, including their names and references to their certificates.
- A listing of all tasks along with their enclave measurements and a list of stakeholders who are allowed to run a specific task.

- A listing of all topics, including their names and a list of stakeholders able to manipulate the data in the topic.

The stakeholders present in the system fall into three main roles. Note that a single stakeholder can act in multiple roles in a single instance

1. Data producers, who are responsible to uploading the encrypted input data to the data topics.
2. Data consumers, who download the output data from a data topic.
3. Task runners, who can trigger a task enclave to execute its computational task.

In addition to these main roles, Sharemind HI defines additional roles to further increase the deployment's security [23]:

- The auditor, who has access to the audit logs produced by the instance and is responsible for validating critical components and performing system audits.
- The coordinator, who is responsible to activities related to setup and deployment.
- The enforcer, who can approve the configuration and is responsible for checking if the configuration conforms to the security objectives that are agreed upon by the stakeholders.

Figure 4 illustrates a possible structure of a Sharemind HI instance defined by a simple DFC. There are three distinct stakeholders present in the system. Producers A and B both encrypt the input data for Task A and upload the encrypted data to topics A and B, respectively. Task A is configured to be able to download and decrypt the data from topics A and B to perform a computational task, encrypt the result and upload the result to topic C. The output data can then be downloaded by a third stakeholder, consumer A. In this example, consumer A also possesses a runner role, which means that he is able to trigger the task enclave to perform its computations. However, he does not have access to any raw input data provided by producers A and B.

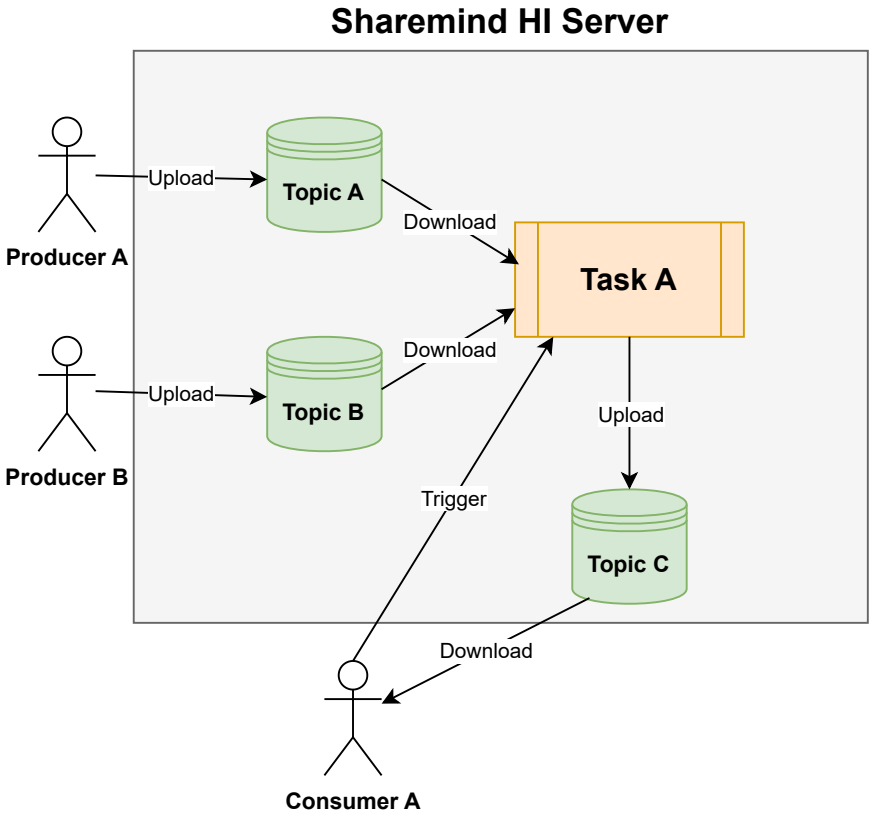

Figure 4. An example instance with a simple configuration

1.6.3 Architecture

Sharemind HI is implemented as a client-server application. The client includes the application specific code and leverages the Sharemind HI client libraries. The client can be implemented as a web application or native application as Sharemind HI provides an SDK for both TypeScript and C++ as well as a command-line interface. The client library provides common functionality such as communicating with the Sharemind HI server, performing encyption operations and remote attestation. The untrusted part of the Sharemind HI server is responsible for work coordination, file system access, network communication and forwarding messages between enclaves. Since this

part is untrusted, it does not run inside enclaves and therefore can not have access to any sensitive information in plaintext.

The trusted parts of the Sharemind HI instance consist of application-specific task enclaves - developed by the application developer - and three enclaves provided by Sharemind HI - key enclave, attestation enclave and core enclave. The server-side functionality is separated into multiple enclaves to keep the enclaves small and to separate concerns into separately auditable components and to mitigate risks when a single enclave becomes compromised. Their responsibilities in the service are as follows:

- The attestation enclave is responsible for performing remote attestation and setting up secure channels between the client and other enclaves.
- The key enclave is only responsible for storing and managing access to keys required to use any encrypted data.
- The core enclave manages the state of the instance, creates the audit log and coordinates the execution of tasks. It does not have access to any confidential data except shared secrets for communication channels.

1.6.4 Security

As Sharemind HI relies on Intel SGX for providing a trusted execution environment for sensitive data, most of its attack model is identical to Intel SGX. However, Sharemind HI adds additional abstraction layers which hide numerous details of data encryption and technical role enforcement, such that application developers can concentrate more on developing the business logic. A thorough attack model which lists all considered threats and countermeasures that are applied to mitigate those threats is provided in the Sharemind HI white paper [22].

As mentioned before, the main goal of Sharemind HI is to ensure that the data uploaded to the platform can only be accessed by stakeholders and task enclaves with

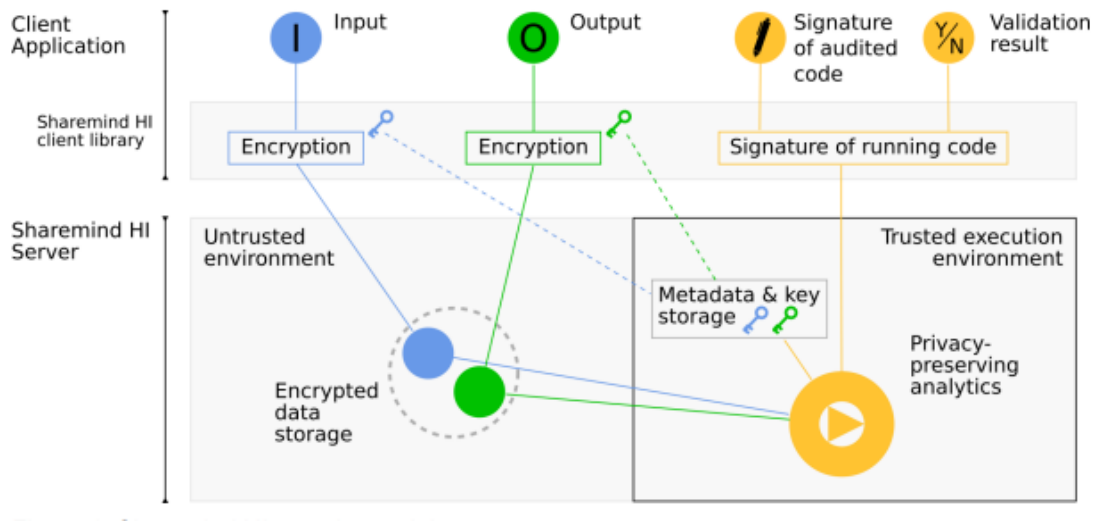

Figure 5. Overview of the security model of Sharemind HI [22]

sufficient permissions. Sharemind HI focuses in providing confidentiality and integrity of this data, whereas availability is left as a responsibility for the stakeholders. A visual overview of the security model is given in Figure 5.

To ensure the confidentiality and integrity of the data, the data owner encrypts the input data at the source and transfers the encryption keys to the key enclave via secure channels attained by remote attestation. The encrypted data is the stored in a topic in the Sharemind HI server. Similarly, after processing the input data, the output data is encrypted inside the task enclave and stored in a topic. When a stakeholder requests to download the output data, first his access rights are confirmed by the platform and then decryption keys are transferred from the key enclave to the authorized stakeholder.

Before running a Sharemind HI instance, a set of enforcers must verify that the task enclaves are configured with correct access permissions and give a cryptographically signed approval. All stakeholders must choose which enforcers they trust and not validate the configuration themselves. Stakeholders can then perform operations with input or output data only on a Sharemind HI server which has been approved by their trusted enforcers.

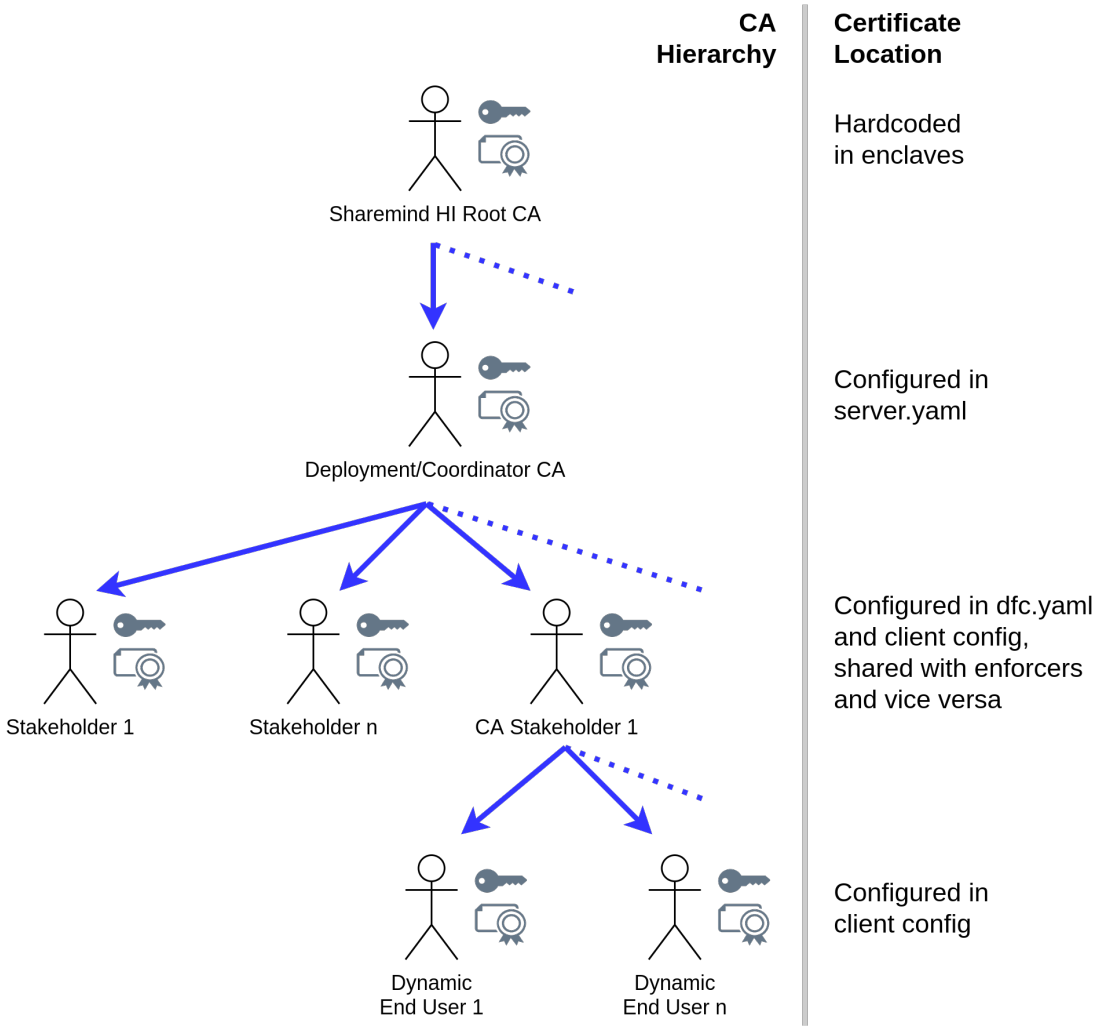

Figure 6. CA hierarchy in Sharemind HI

The coordinator has to generate an unique asymmetric key pair for each Sharemind HI instance. The certificate is signed by the Cybernetica Deployment Root CA for Sharemind HI, binding the identity of coordinator to the key pair. Similarly, each stakeholder who needs to communicate with the Sharemind HI server, generates an asymmetric key pair, and their certificate is signed with the coordinator's private key. The coordinator's signed public key certificate is loaded into the server during deployment and used to authenticate clients in remote attestation. Analogously, the Cybernetica

Deployment Root CA certificate is embedded into the server and verifies the validity of the coordinator's certificate. This ensures that only the parties explicitly added by the coordinator are allowed to access the deployment, and facilitates authenticating the stakeholders and enforcing access controls. An illustration about the certificate hierarchy in Sharemind HI is given in Figure 6.

Furthermore, prior to the instance deployment an auditor is required to validate the enclaves, ensuring they are secure and privacy-preserving, resulting in a cryptographic proof of the audited code. A client can, at any point after deployment, compare that proof against any of the deployed enclaves, ensuring the integrity of the server.

Sharemind HI is also configured to use all available mitigations provided by SGX against side-channel attacks. While these measures are not enabled by default due to slower execution speeds, Sharemind HI takes a conservative approach and uses all available measures, compromising some performance for security benefits [23].

2 Background on the underlying problem

This chapter provides a concise introduction to the problem under consideration. It begins by outlining the limitations of the present authentication mechanism utilized in Sharemind HI. Subsequently, the chapter describes the potential benefits of incorporating the Smart-ID solution into Sharemind HI. Additionally, the chapter explores several practical scenarios in which the proposed solution could be deployed, thereby underscoring the significance of investigating this area further.

2.1 Problem statement

As outlined in chapter 1.6, Sharemind HI's current authentication and authorization protocols are heavily reliant on the use of configuration files and X.509 certificates. The coordinator possesses full control over the system's access, with each stakeholder's certificate being signed with the coordinator's private key. The DFC must list every stakeholder, resulting in a secure yet restrictive environment that is difficult to scale. The integration of a new stakeholder requires the creation of a new asymmetric key pair, with the public key certificate being signed by the coordinator. The coordinator must then add the new stakeholder to the DFC and assign authorization for task enclaves and topics. Sharemind HI also supports dynamic end users, where a single stakeholder can act as an intermediate CA and sign certificates. This slightly improves bottlenecks and overall usability of the system, but possesses similar weaknesses.

Additionally, relying solely on an X.509 certificate-based authentication mechanism creates a cumbersome user experience for both end-users and applications integrating with Sharemind HI. Today's end-users expect to authenticate to a service using either a username and password or some form of PKI-based authentication mechanism, such as Smart-ID or Mobile-ID in the Baltics. However, Sharemind HI only supports a comparatively technical and inconvenient authentication method, limiting its usability.

There have been solutions developed with Sharemind HI that rely on a password-based authentication mechanism for specific projects, instead of the certificate-based authentication mechanism [24]. The registration process involves the generation of an asymmetric key pair, which is encrypted with the user's password using *scrypt* as a key derivation function. The resulting encrypted private key is then stored in a centralized storage system. An administrator signs the user's certificate, which contains their public key. During login, the certificate is retrieved from the storage system by username, and the private key is decrypted using the password. This approach can improve the user experience by removing the need for technical knowledge in setting up the authentication flow. However, it is susceptible to the same weaknesses as any other password-based scheme, as the security depends heavily on the strength of the password selected by the user. Additionally, the scheme requires the maintenance of an additional storage system by an administrator.

2.2 Motivation

Sharemind HI, in its current state, cannot be used in practice for applications with a large and constantly evolving user base, as each stakeholder must be listed separately in a DFC. This presents a challenge when building privacy-preserving services on a national level, as it would require all citizens to have a public key certificate listed in a DFC. However, multiple services, such as Smart-ID, are capable of authenticating users and providing their public key certificates to relying parties. As of March 2023, Smart-ID has over 3 million users in the Baltics who could use national privacy-preserving services using Smart-ID, much like how they authenticate to e-commerce or governmental services today [25]. As a result, integrating the Smart-ID authentication process with Sharemind HI and Intel SGX applications could enable the development of various privacy-preserving services at a national scale. There are currently numerous practical scenarios in which integrating Smart-ID authentication into Sharemind HI based

services could be advantageous. The process of implementing such an authentication mechanism could offer valuable insights into the necessary requirements and best practices for further successful applications in the future.

3 Design

This chapter presents an overview of the necessary requirements for the integration of Sharemind HI authentication and access control mechanisms with the Smart-ID service provider. Specifically, the chapter first discusses how to integrate existing services with the Smart-ID service provider and what the differences are between integrating a regular web service and an Intel SGX based platform. Next, a thorough architectural overview of the final solution will be provided, which satisfies all current Sharemind HI access control and auditability requirements. This integration enables the development of larger scale services combining the Sharemind HI platform with the user base of Smart-ID.

3.1 Integration mechanism for Smart-ID clients

The Smart-ID service components overview in Figure 2 demonstrates that the Smart-ID platform offers two external REST APIs for integration purposes. The first of these is the mobile device (MD) API, which facilitates communication between the Smart-ID application library and the Smart-ID core. While this API is utilized by an external party, an application developer seeking to integrate their service with the Smart-ID core is not required to interface with this API directly.

The second and primary API offered by the Smart-ID core is the relying party (RP) API. This API functions as the primary entry point for relying parties seeking to perform authentication and digital signing operations using Smart-ID. The RP API allows a service to create either an authentication, certificate selection, or digital signing session using a semantic identifier in a format specified by ETSI [26]. Once a Smart-ID session has been initiated, a relying party can poll the Smart-ID core to determine whether the operation has been completed successfully or whether the user is in the process of entering their PIN1 or PIN2, or selecting a certificate for signing. Chapter 1.4 details

the specific REST endpoints, request and response payloads, and status codes associated with Smart-ID authentication and digital signing sessions. Once a relying party has received a successful authentication session poll response from the Smart-ID core and has validated it appropriately, it obtains proof of the user's identity and can establish an internal session for the user as needed. Similarly, following a successful digital signing session poll response, the relying party receives a composite digital signature that is computed jointly by the mobile device and the Smart-ID core.

3.2 Intel SGX application integration specifics

As indicated in the preceding chapter, the RP API offered by the Smart-ID provider is a REST API, which utilizes HTTP as its underlying application-level protocol. To ensure the authenticity of the Smart-ID RP API endpoint, it is crucial to use HTTP over TLS in conjunction with a secure TLS cipher suite and HTTPS pinning. Neglecting to do so could enable an attacker to intercept the connection between the RP and the Smart-ID service provider and execute an active man-in-the-middle (MITM) attack. The attack scenario involves the following steps [17]:

1. The client requests Smart-ID authentication to log in. The RP server initiates a connection to the RP API authentication endpoint. However, the attacker intercepts the connection and responds instead. Consequently, the RP sends an authentication request with a random hash h_1 to the attacker.

2. The attacker establishes another connection to the RP server under the same identity as the original client and intercepts this connection as well. This time an authentication request with a random hash h_2 is sent to the attacker.

3. The attacker calculates the verification codes for both hashes utilizing the aforementioned method.

4. If the verification codes differ, the attacker drops its connection and repeatedly establishes a new one until the codes match. On average, it will take approximately 5000 connections until such a collision occurs.

5. The attacker sends the authentication request with the hash h_2 to the authentic RP API authentication endpoint.

6. The Smart-ID service provider dispatches a notification to the client's smartphone, prompting the user to verify the verification code. Since h_1 and h_2 produce a hash collision, the verification codes will match.

7. The client enters their PIN and completes the authentication process. The attacker receives the authentication response from the RP API and forwards it to the RP server, which verifies the signature's validity and establishes an authenticated session for the attacker under the client's identity.

In the context of Intel SGX applications, the aforementioned MITM attack would render it impossible for an untrusted component of the application to act as a client for RP API calls in the guise of a privileged user or malware. Such untrusted parties would have the capability to examine decrypted HTTP responses and interject genuine client calls to initiate a similar MITM attack, thereby nullifying the security of the system. This implies that HTTPS requests and responses need to be handled by enclaves. This requirement poses a number of complex architectural challenges. Firstly, initiating system native API calls from trusted application components to read from or write to sockets is not possible. Secondly, TLS is a complex protocol with vulnerabilities being discovered in several implementations in recent years. In addition, both open source and commercial TLS implementations may be susceptible to side-channel attacks, which have been the primary means of attacking Intel SGX systems [27] [28].

3.3 HTTPS enclave

3.3.1 Objective and requirements

As outlined in chapter 1.6.3, the Sharemind HI system comprises of three distinct enclaves, namely the core enclave, the attestation enclave, and the key enclave. These enclaves are designed to manage risks and enforce a clear separation of concerns. Consistent with this architectural framework, it is advisable to create an additional enclave responsible for initiating HTTPS traffic from the trusted segment of the application. This approach ensures that the HTTPS enclave has a well-defined role and does not pose a risk to the core enclave in the event of a vulnerability in the TLS protocol or a particular implementation.

The HTTPS enclave has been specifically designed to handle arbitrary HTTPS connections, on the condition that the server certificate used for the connection is signed by a trusted certificate authority. The enclave is unaware of any Smart-ID integration specific details, thus allowing it to be utilized for other use cases in the future. Its primary function is to receive plain HTTP requests from other enclaves via a secure channel created with the local attestation mechanism. It then establishes a TLS session with the server and forwards the encrypted TLS records to the untrusted portion of the application, where it can be written to a socket. Similarly, when a TLS record is received from the server, it is only decrypted within the enclave and then securely transferred to the originating enclave, where the required business logic is applied to the received response.

The HTTPS enclave must fulfill an additional requirement of being capable of handling multiple concurrent clients seeking to authenticate to a service via their Smart-ID accounts. As a result, the enclave is required to handle multiple TLS sessions and demand that clients provide a unique identifier for each HTTP request issued to enable multiplexing. Nonetheless, the responsibility of linking a response to the corresponding

request lies with the originating enclave, while the HTTPS enclave guarantees that the response comprises an identifier pointing to the appropriate request.

In order to ensure optimal resource utilization within an enclave, it is imperative to utilize asynchronous input/output (IO) operations, as the allowed thread count must be specified during the enclave's build process. A preconfigured thread count that is excessively high would lead to wastage of resources, while a thread count that is too low would result in incoming requests being stalled under load. Additionally, it is important to note that errors in multithreaded code pose a significant threat to SGX enclaves as they can be exploited by a malicious operating system [29]. Through the utilization of asynchronous IO operations, one can effectively process an arbitrary number of requests with a single thread until CPU usage reaches saturation. Several TLS implementations, including OpenSSL and WolfSSL, are capable of functioning in an asynchronous manner, making them well-suited for this type of application. If the TLS client is unable to read or write data at a particular time, the corresponding call will return with an appropriate status code, and the IO operation will be queued for later processing, thereby freeing up resources for other ECALLs to execute their respective tasks.

3.3.2 EDL interface

Next the interfaces used by the HTTPS enclave to communicate with untrusted parts and other enclaves are described. This procedure is done using the EDL notation. Specifically, for the HTTPS enclave, two distinct EDL interfaces are defined. The first interface concerns ECALLs and OCALLs implemented by the HTTPS enclave itself, while the second one requires the consumer of the HTTPS responses to implement it. Both interfaces offer lifecycle methods for enclave initialization and destruction. For instance, the initialization section performs the local attestation process and enclave communication setup. It is worth noting that the communication between different enclaves in the same machine passes through the untrusted segment of the application. Nevertheless, the

payload is encrypted with the session key that was negotiated during local attestation, making it impossible for the untrusted segment to observe the payload. The HTTPS enclave consumer interface comprises two methods: a single OCALL for performing an HTTPS request through the HTTPS enclave, and a single ECALL for receiving the HTTPS response once the entire response has been received. The method signatures for these methods are presented in Listing 6.

```
enclave {
   trusted {
      public void async_response_from_https_enclave_ecall(
         https_enclave_result_for_enclave_t result);
   };

   untrusted {
      void async_request_to_https_enclave_ocall(
            uint32_t fbs_discriminator,
            [user_check] const uint8_t * in_payload,
            size_t in_payload_size,
            https_enclave_peer_request_identifier identifier);
   };
};
```

Listing 6. EDL interface description for the HTTPS enclave consumer

The HTTPS request OCALL takes as input the encrypted payload containing the HTTP request, as well as the size of the payload. In addition, the caller provides the request identifier to the HTTPS enclave, enabling the response to be associated with the corresponding request. The *fbs_discriminator* argument is an implementation detail internal how Sharemind HI uses Flatbuffers, a serialization library employed for efficient message serialization between the client and various enclaves. The HTTPS response ECALL receives the encrypted HTTPS response object as an argument, containing a pointer to the response data along with its size and the request identifier.

```
enclave {
   trusted {
      public https_enclave_result_t https_from_enclave_ecall(
            uint32_t fbs_discriminator,
            [in, size=encrypted_fbs_size] const uint8_t* encrypted_fbs,
            size_t encrypted_fbs_size,
            https_enclave_peer_request_identifier identifier);

      public https_enclave_result_t https_from_socket_ecall(
            https_enclave_session_context_t session_context,
            int recv_return_value,
            int untrustedErrno,
            const uint8_t * socket_received_data,
            size_t socket_received_data_size);
   };

   untrusted {
      int convert_address_to_socketfd_ocall(const char* host, size_t host_size);
   };
};
```

Listing 7. EDL interface description for the HTTPS enclave

The HTTPS enclave itself implements a separate EDL interface, which consists of two ECALLs and an OCALL. The *https_from_enclave_ecall* ECALL is used to initiate a new HTTPS request from a different enclave. It is called by the consumer OCALL method and forwards the encrypted Flatbuffer message to the HTTPS enclave. Its arguments are therefore analogous to the OCALL defined in the consumer enclave. The *https_from_socket_ecall* is executed asynchronously after the HTTPS enclave returns a status indicating that the entire response has not yet been received. Since the ECALL operations for HTTPS are non-blocking, any subsequent IO operations performed by the untrusted are scheduled on a separate thread. After receiving additional data, the HTTPS enclave is notified via this ECALL. This loop can occur multiple times until the entire payload has been received. Along with the received socket data and size, the ECALL is passed a session identifier and potential error values as arguments. The inter-

face defines a single OCALL for generating a new socket when a client requests a new HTTPS session from the HTTPS enclave. Since the enclave is unable to execute system calls directly, it must delegate the creation to the untrusted portion of the application. The OCALL takes hostname details as arguments and returns the identifier of the generated file descriptor. The entire EDL interface for the HTTPS enclave is presented in Listing 7.

3.4 Sharemind HI action flow with Smart-ID

This chapter presents a theoretical overview of how the Sharemind HI authentication mechanism can be incorporated with the Smart-ID service provider. The HTTPS enclave is used to issue RP API client calls as explained in the previous chapter. Although Sharemind HI offers numerous request types for clients related to audit log downloads, configuration approvals, and recoveries, we will only examine the authentication and client payload signing flow as they differ with Smart-ID usage. Finally, an authorization solution for handling Smart-ID clients is provided.

3.4.1 Data upload

Figure 7 offers a detailed sequence diagram for the Sharemind HI authentication flow with specific Smart-ID integration steps highlighted in groups. The integration involves introducing two new enclaves, as well as additional round trips from the client to the Smart-ID enclave, first to establish an internal session and later to sign the payload for the Sharemind HI operation such as a data upload.

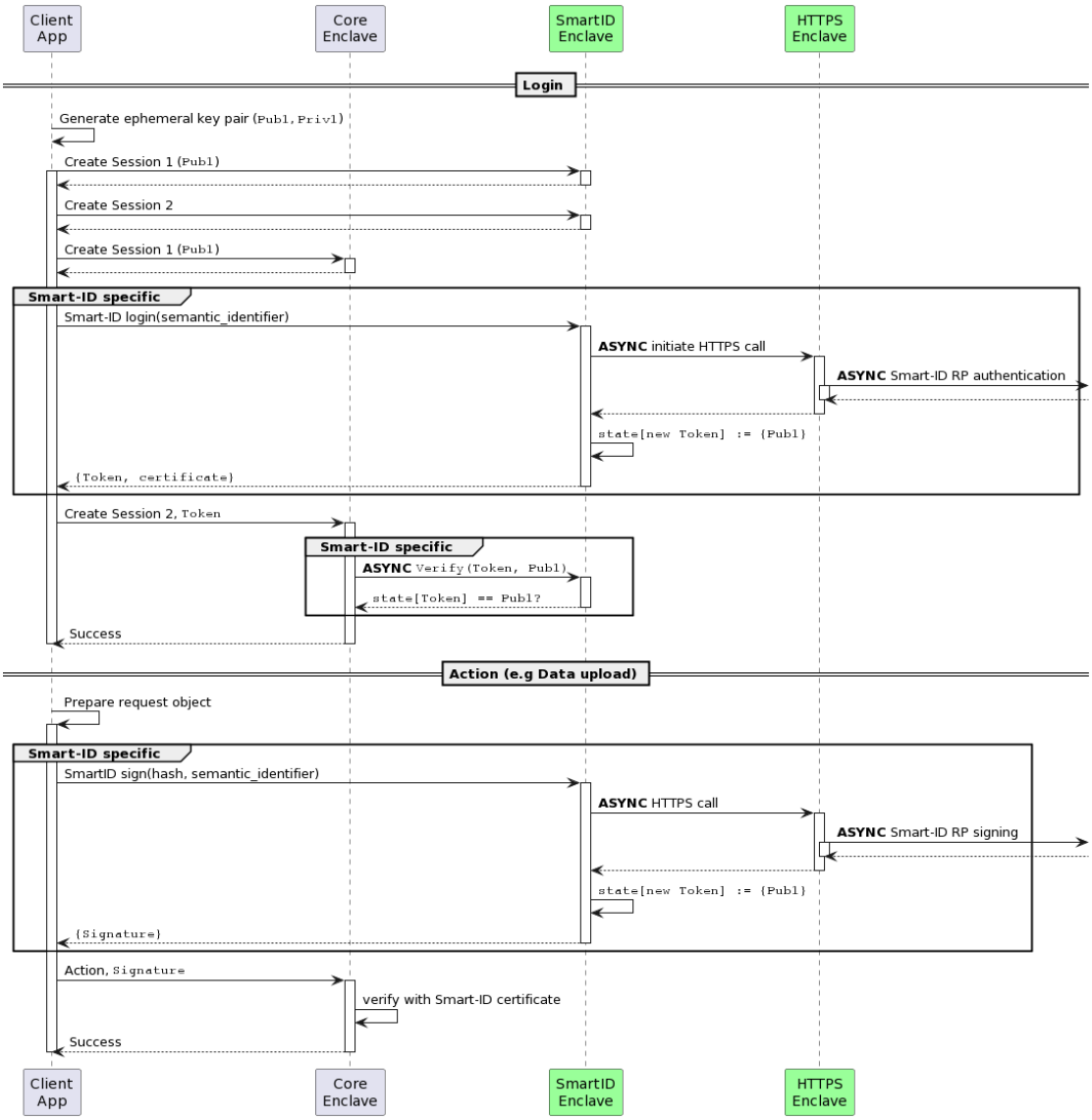

Figure 7. Sharemind HI action flow with Smart-ID integration

The authentication process starts by creating a secure session between the client and the Smart-ID enclave through remote attestation. Afterwards the client also generates an ephemeral asymmetric key pair to use during the Sharemind HI operation. In order to initiate the authentication procedure using the Smart-ID identity provider, the client must inform the Smart-ID enclave and provide its national identity code. The Smart-

ID enclave then establishes a new authentication session using the HTTPS enclave, generating a random token upon successful completion, which is stored locally along with the ephemeral public key. It is noteworthy that the HTTPS request is performed asynchronously without making any OCALLs. At the conclusion of the authentication process, the Smart-ID enclave returns the token and the Smart-ID certificate to the client, which can be used for authentication purposes with the core enclave. All error conditions from the Smart-ID service, as well as any response verification errors, are propagated to the client, and the authentication procedure will fail.

The Sharemind HI operation payload signing process follows a similar procedure to the authentication flow, with the added requirement for the client to provide the hash of the payload to be signed. The Smart-ID and HTTPS enclave operate in a similar manner, but with the use of a separate Smart-ID RP API endpoint for creating a signing session. Following the retrieval of the client's signature from the Smart-ID service provider, the core enclave verifies it using the client's Smart-ID certificate. The signing of the operation payload serves the purpose of ensuring non-repudiation in the audit logs.

The Smart-ID enclave performs an identity provider function similar to that of SAML authentication or OpenID Connect in the scheme described above. The general process involves the user agent (client application) requesting an identification proof from the identity provider before requesting a resource from the service provider (core enclave). Typically, these proofs have an expiration date, which cannot be verified within enclaves. To prevent the malicious use of tokens returned by the Smart-ID enclave at a later time, communication between the core enclave and the Smart-ID enclave has been established to validate the one-time token's legitimacy.

3.4.2 Authorization

The current dataflow configuration files explicitly list the permissions for each stakeholder. However, one of the main objectives of integrating Sharemind HI with the

Smart-ID service is to enable the development of larger-scale privacy-preserving services where listing every stakeholder is not feasible. Therefore, the same approach cannot be used with services that leverage the benefits of Smart-ID. As a result, every client authenticated using Smart-ID must currently have identical access permissions, which are listed in a new section defined in the dataflow configuration. Listing 8 provides a minimal configuration file with a new section for listing identity providers and permissions for users who are using the provider.

```
Stakeholders: [{Name: "Alice", CertificateFile: "client.crt"}]

Auditors: ["Alice"]
Enforcers: ["Alice"]

IdentityProvider:
  - Name: Smart-ID
    EnclaveFingerprint: "aa531..."
    SignerFingerprint: "a3535..."

Tasks:
  - Name: sample_task
    EnclaveFingerprint: "34e7b..."
    SignerFingerprint: "79010..."
    Runners: ["Alice"]

Topics:
  - Name: input
    Producers: ["Smart-ID"]
    Consumers: ["sample_task"]
  - Name: output
    Producers: ["sample_task"]
    Consumers: ["Smart-ID", "Alice"]
```

Listing 8. Example dataflow configuration file with Smart-ID stakeholder

The dataflow configuration file now has the ability to include a separate section for identity providers in addition to stakeholders, auditors, enforcers, and a variety of tasks

and topics. While currently restricted to Smart-ID, there is potential for other external identity providers, such as mobile-ID, to be accommodated in the future. Every identity provider is assigned a name, which is cited in the producer, consumer or runner lists. This naming convention authorizes each individual authenticated with the respective identity provider to execute the operations listed. In the example above, all users authenticated using Smart-ID are permitted to upload data to the input topic and download data from the output topic. However, they are not permitted to run the sample task which produces output data.

4 Implementation

This chapter provides a comprehensive overview of the implementation details of the Smart-ID integration prototype that is based on the Sharemind HI platform. The prototype comprises of an untrusted portion of the application in conjunction with two Intel SGX enclaves, namely the HTTPS enclave and the Smart-ID enclave. The HTTPS enclave, which is introduced in Chapter 3, is responsible for initiating HTTPS traffic to the Smart-ID service provider. The Smart-ID enclave hosts the Smart-ID specific business logic and communicates with the HTTPS enclave.

4.1 Development environment

Due to the specific processor requirements for running Intel SGX applications in hardware mode, a virtual machine with a compatible Intel processor and Intel SGX software development kit (SDK) preinstalled was established as a remote development environment. While it is possible to simulate Intel SGX applications without the need for compatible hardware, a remote machine was preferred to create an environment that closely resembles production. This approach allowed for more precise benchmarking, leading to more accurate evaluations of real-world use cases.

In addition, the prototype, albeit not yet fully integrated, was developed on the Sharemind HI platform due to the platform's useful abstractions for creating enclaves, performing local and remote attestation, and executing other Intel SGX-specific operations. Both the Sharemind HI platform and the prototype were programmed in C++17 which allowed the prototype to utilize an existing build system built with CMake, a popular C++ build tool. The build system involves the generation of C++ code from Flatbuffers and EDL definition files, as well as the signing of enclaves.

4.2 TLS implementation

When creating a HTTPS client, the selection of a suitable TLS implementation is a crucial factor to consider for establishing a secure channel with the server before exchanging application data. For UNIX systems, OpenSSL, an open-source toolkit for cryptography and secure communication, is a commonly used choice. However, for the prototype under consideration, WolfSSL was preferred as the TLS library for several reasons. Firstly, OpenSSL is a rather extensive library, whereas WolfSSL, due to its modular nature, has a considerably smaller build size and a runtime memory footprint [30]. Secondly, WolfSSL already offers basic integration with Intel SGX, along with an example enclave that serves as an excellent reference point for creating an enclave that links with a TLS library [31]. Implementation details for the HTTPS enclave are discussed in chapter 4.6. While WolfSSL is a commercial TLS implementation, it can be used free of charge for educational purposes. For future commercial use cases, it is rather straightforward to change TLS implementations as the WolfSSL API is compatible with the OpenSSL API.

4.3 Flatbuffers message definitions

The serialization of messages exchanged between different enclaves is accomplished through the use of the Flatbuffers library, as mentioned in chapter 3.3.2. Since the HTTPS enclave requires information on when to transmit data over an HTTPS connection or terminate the connection, the responsibility for providing this information lies with the caller enclave. Listing 9 offers a comprehensive definition of the request and response messages relevant to the transmission of data over an HTTPS connection.

```
table HttpsEnclaveSendRequest {
   new_or_continuation:HttpsEnclaveSendNewOrContinuation;
   path:string;
   method:string;
   headers:string;
   payload:[ubyte];
}

table HttpsEnclaveSendResult {
   result: HttpsEnclaveSendResultUnion;
}

table HttpsEnclaveSendResponse {
   http_status:uint;
   payload:[ubyte];
   session_context:TlsSessionContext;
}

table TlsSessionContext {
   session_identifier:[ubyte];
}

table HttpsEnclaveSendNew {
   url: string;
}

table HttpsEnclaveSendContinuation {
   session_context: TlsSessionContext;
}

union HttpsEnclaveSendNewOrContinuation {
   HttpsEnclaveSendNew, HttpsEnclaveSendContinuation
}

union HttpsEnclaveSendResultUnion { HttpsEnclaveSendResponse, EnclaveError }
```

Listing 9. Message definitions for sending data over an HTTPS connection

The two primary tables included in the definition are the *HttpsEnclaveSendRequest* and *HttpsEnclaveSendResult*. The *HttpsEnclaveSendRequest* message is sent by the

Smart-ID enclave when it has to initiate a Smart-ID RP API request. The message contains a resource path, an HTTP verb, along with the request headers and payload, which the HTTPS enclave can use to build a raw HTTP body before transmitting it to WolfSSL for wrapping it in TLS records. Moreover, the client provides a structure that specifies whether the HTTPS enclave should create a new TLS session or reuse an existing one. In the case of a new session, the HTTPS enclave must be given a server address to which it can connect. For session continuation, the client provides a session identifier, which is returned with every HTTPS enclave response, in addition to the HTTP status code and the response payload in the *HttpsEnclaveSendResult* message. In the case of an internal error, an *EnclaveError* is returned instead, containing an error string. When the client has received all necessary responses, it must send a separate *HttpsEnclaveCloseRequest* along with the corresponding session identifier.

4.4 Untrusted component

As previously stated, the prototype comprises of three distinct components - the untrusted component, the Smart-ID enclave, and the HTTPS enclave. The untrusted component serves as the primary entry point to the application. Moreover, the untrusted component has three specific objectives:

1. Upon startup, create and initialize both the Smart-ID and HTTPS enclaves.
2. Initiate the Smart-ID authentication flow by making an ECALL to the Smart-ID enclave.
3. Provide OCALL implementations for methods defined in the untrusted blocks in all EDL files.

Initialization

In every Intel SGX based application, the untrusted component is responsible for creating the necessary enclave upon startup. When using the Intel SGX SDK directly, the *sgx_create_enclave* is used to create an enclave. However, Sharemind HI provides numerous utility classes and mix-ins to ease the creation and initialization of an enclave.

Smart-ID authentication initiation

Similarly to regular Smart-ID authentication procedures, the user is required to provide their national identity code to the relying party during each authentication session. On the Sharemind HI platform, the identity code may be obtained through several means, such as via an user interface using a HTTP request, a configuration file or a command line, depending on the use case. Since enclaves are unable to carry out any IO operations directly, the identity code will be sent to the untrusted component. After receiving the identity code, the untrusted component will prompt an ECALL to the Smart-ID enclave in order to initiate the Smart-ID authentication session. Due to the fact that the input originates from an untrusted source, the Smart-ID enclave is obliged to conduct validity checks before proceeding with the flow.

OCALL implementations

In chapter 3.3.2, both EDL interfaces define an OCALL, which the untrusted component is required to implement. The first OCALL, *convert_address_to_socketfd_ocall*, acts as a wrapper around the C function *getaddrinfo*, which is used to translate domain names and hostnames into an *addrinfo* structure. By means of the address family, socket type, and protocol, this function generates a new socket that can connect to the socket address, and eventually returns the file descriptor for the newly created socket. This file descriptor is returned to the trusted component and later used to read and write application data. This OCALL is invoked by the HTTPS enclave when it needs to create a new

session with the Smart-ID RP API. Listing 10 provides an example on how to use the *getaddrinfo* function.

```
int address_to_sockfd(const char* address) {
   struct addrinfo* result;
   struct addrinfo* res;
   int error;

   error = getaddrinfo(address, NULL, NULL, &result);
   // Error handling omitted

   for (rp = result; rp != NULL; rp = rp->ai_next) {
      sockfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
      if (sockfd == -1) { continue; }
      if (connect(sockfd, rp->ai_addr, rp->ai_addrlen) != -1) {
         break;
      }
      close(sockfd);
   }

   freeaddrinfo(result);
   return sockfd;
}
```

Listing 10. Example of using a getaddrinfo function

The *async_request_to_https_enclave_ocall* is used to transmit the raw HTTP request data from the Smart-ID enclave to the HTTPS enclave. The purpose of this OCALL is to allow the Smart-ID enclave to issue an HTTPS request to the Smart-ID RP API either to establish a Smart-ID session or poll its status. This OCALL queues a task to initiate an *https_from_enclave* ECALL to the HTTPS enclave, which takes in the input arguments provided by the Smart-ID enclave. The execution of this task takes place asynchronously on a different thread. Upon receiving a response from the HTTPS enclave, the OCALL then queues a task to forward the response back to the Smart-ID enclave if the complete HTTP response was obtained. If the entire response was not received, the untrusted component performs additional socket read or write operations,

based on the input arguments from the HTTPS enclave, and queues a task to notify the HTTPS enclave of the newly read data. Further details regarding the return values of HTTPS enclave ECALLs can be found in chapter 4.6.

4.5 Smart-ID enclave

The Smart-ID enclave is responsible for performing successful Smart-ID authentication and digital signing operations. It initiates the necessary business logic for these processes. The enclave is required to implement both the Smart-ID and HTTPS consumer EDL interfaces, which consist of a singular ECALL and enclave lifecycle methods. During the initialization phase, the enclave must perform local attestation with the HTTPS enclave to establish a secure communication channel.

The *smart_auth_init* ECALL, defined in the Smart-ID EDL interface, is utilized by the untrusted component to initiate a Smart-ID session. As discussed in the preceding chapter, this ECALL receives the national identity code of the user, which must be validated by the Smart-ID enclave before use. Following this validation, the enclave generates 64 random bytes and computes their SHA-2 family hash. This resulting hash is signed during the authentication session. Subsequently, the hash is used to calculate the verification code, utilizing the algorithm described in chapter 1.3.2. The Smart-ID enclave then creates a *HttpsEnclaveSendRequest* to initialize the Smart-ID session. An example of the request body can be seen in Listing 3. Furthermore, to ensure that the request is accepted by the Smart-ID service provider, it is essential to correctly set the Host, Content-Type, and Content-Length headers. Prior to forwarding the request to the HTTPS enclave, the Smart-ID enclave establishes a new internal session to monitor the status of all ongoing Smart-ID sessions. The sessions are identified by a unique session identifier which is sent to the HTTPS enclave along with the HTTP request. Once the *HttpsEnclaveSendRequest* object is constructed, the Smart-ID enclave encrypts the payload utilizing the session key negotiated during local attestation. Finally, the enclave

invokes the *async_request_to_https_enclave_ocall* to notify the HTTPS enclave of the new request.

The *async_response_from_https_enclave* ECALL, defined in the HTTPS consumer EDL interface, is utilized by the untrusted component when the HTTPS enclave indicates that the complete HTTP response has been received from the Smart-ID service provider. It will first decrypt the message sent by the HTTPS enclave with the symmetric session key and find the correct internal session based on the session identifier included in the HTTPS enclave response message. Note that the raw HTTP response is already parsed by the HTTPS enclave and the Smart-ID enclave can access the status code, headers and the potential response payload directly. In the case of a successful HTTP response, indicated by a 200 status code, the JSON payload can be parsed by the enclave. When the request was unsuccessful, a descriptive error message depending on the exact status code in propagated to the user. After the response JSON has been successfully parsed, the following action depends on the current state of the internal session:

1. If the session is currently initializing, then the Smart-ID enclave received an session initialization response from the Smart-ID service provider, provided in Listing 4. The enclave extracts the *sessionID* value from the response, initiates the first status polling request in similar manner to the initialization request using the *sessionID* and updates the internal session state.

2. If the session is in polling state, then the Smart-ID enclave received either an intermediate or a final session status response from the Smart-ID service provider. Examples of these can be found in Listings 5 and 6. If the *state* value in the response is RUNNING, then another poll request is initiated after a small delay. If the state is COMPLETE, the enclave can extract the client signature and certificate values from the response. Since the Smart-ID session is completed, the internal session is removed from the memory as well.

After receiving the signature over the randomly generated hash and the public key certificate of the client, the Smart-ID enclave verifies the validity of the response. According to the Smart-ID RP integration guide [17], the relying party must verify the following:

1. The *endResult* field in the response has the value OK.
2. The received certificate is valid, meaning that it is a well-formed X.509 certificate and the current time is between the timestamps provided in the validity period fields in the certificate.
3. The received certificate is trusted, meaning that it is signed by the Smart-ID certificate authority. To do this, the Smart-ID enclave must have access to the certificate of the Smart-ID certificate authority. This certificate is published by the maintainers of Smart-ID and hardcoded into the Smart-ID enclave. If the certificate expires or is revoked, the Smart-ID enclave must be rebuilt and redistributed.
4. The received signature is a valid signature over the randomly generated hash. The signature can be verified using the public key from the certificate.

4.6 HTTPS enclave

The HTTPS enclave is responsible for initiating TLS sessions, converting application data transmitted from the Smart-ID enclave to TLS records, and extracting application data from TLS records received from the Smart-ID service provider. To create socket file descriptors for a particular server address and execute IO operations, it interacts with the untrusted component through the HTTPS enclave EDL file-defined interface. Furthermore, the HTTPS enclave communicates with the Smart-ID through the HTTPS enclave consumer interface. During enclave initialization, along with local attestation performed with the Smart-ID enclave, the HTTPS enclave also initializes the WolfSSL

context. The WolfSSL context is necessary for WolfSSL to establish TLS sessions when the Smart-ID enclave initiates an HTTP request. To perform TLS session verification, the WolfSSL library requires access to the certificate of the certificate authority responsible for issuing the certificate of the Smart-ID RP server. As a result, the HTTPS enclave must embed the DigiCert root CA certificate. However, since root CA certificates are typically valid for extended periods, this hardcoding approach poses no significant concerns.

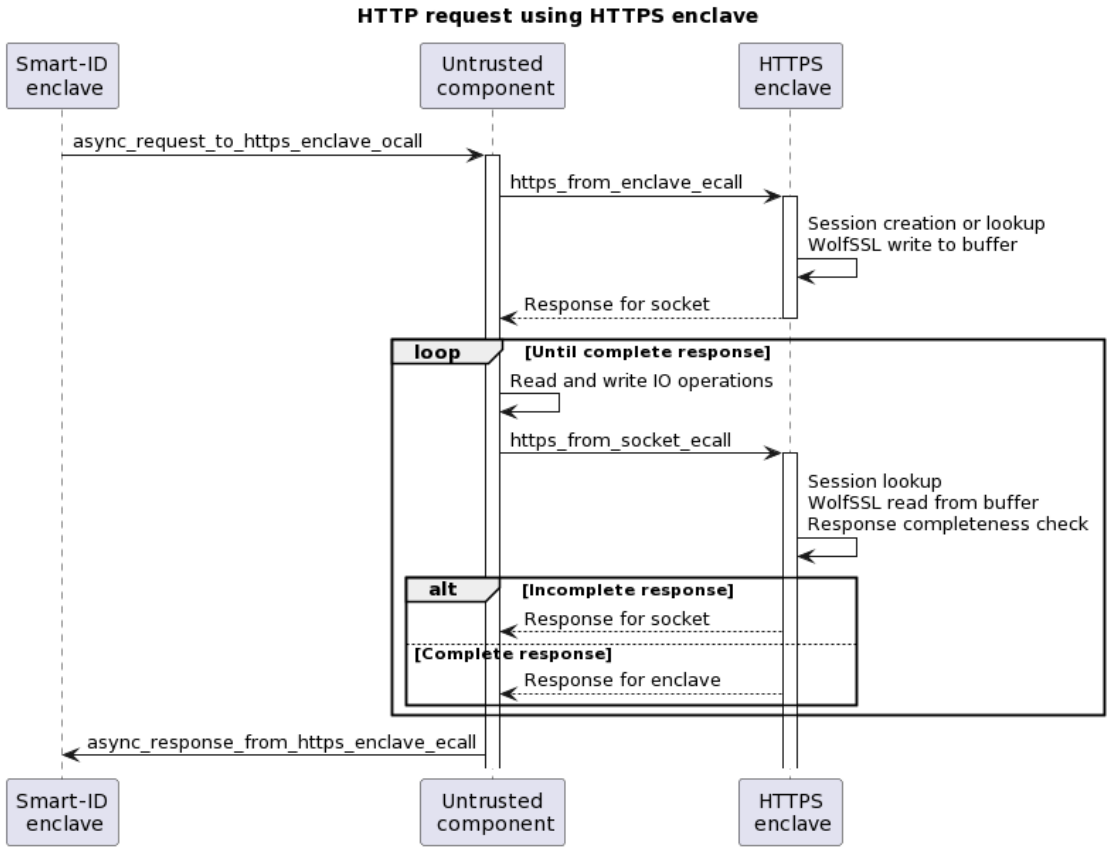

Figure 8. Complete HTTP request flow using HTTPS enclave

The *https_from_enclave_ecall* function serves as the primary access point to the HTTPS enclave. It is utilized by the untrusted component to relay the encrypted payload from the Smart-ID enclave to the HTTPS enclave. The function's behavior depends on

whether the request is a send or a close request, which is determined through the use of the Flatbuffers discriminator. If the request is a close request, the enclave will clear the internal session associated with the session identifier and close the socket file descriptor to release any allocated resources.

The majority of requests sent to the HTTPS enclave are typically send requests. The caller provides information on whether the HTTPS enclave should initiate a new TLS session or maintain an existing one. This decision is made using the *newOrContinuation* field within the request. If an existing session is requested, the enclave performs an internal session lookup based on the provided session identifier. If such a session does not exist, an error response is transmitted to the caller.

```
struct SessionMapData {
   std::unique_ptr<WOLFSSL> ssl;
   int sockFd;
   bool closeSessionAfterRequest;
   https_enclave_peer_request_identifier peerRequestIdentifier;

   std::vector<std::uint8_t> outCiphertextBuffer;
   std::vector<std::uint8_t> inCiphertextBuffer;

   std::string outPlaintextBuffer;

   std::string inPlaintextBuffer;
};
```

Listing 11. Definition of an internal HTTPS enclave session

In contrast, if a new session is requested, the HTTPS enclave initiates an OCALL to acquire a socket file descriptor based on the server URL supplied by the Smart-ID enclave. Currently this is a blocking operation which could be changed in the future as the OCALL can perform a rather expensive DNS lookup. Subsequently, the HTTPS enclave establishes a new WolfSSL session using the acquired socket file descriptor and an internal session that contains the session metadata and necessary plaintext and

ciphertext buffers for reading and writing data. Both the internal session and the generated session identifier are stored in memory, facilitating session data retrieval through the identifier and socket file descriptor lookups. A complete definition of the HTTPS enclave internal session is provided in Listing 11.

Once the appropriate session is obtained, either through creation or lookup, the enclave proceeds to populate the *outPlaintextBuffer* with a raw HTTP request using of the method, resource path, headers, and a body provided by the Smart-ID enclave. This buffer is then utilized by the WolfSSL write functions as input to encrypt raw application data and encapsulate them into TLS records. Notably, a key difference between the HTTPS enclave and standard HTTPS clients is that the WolfSSL write function typically writes directly to the socket using system calls. However, this approach is not possible within the enclave due to the inability to execute system calls. As a result, the HTTPS enclave must declare and implement custom *send* and *recv* functions that write and read bytes to and from session buffers, respectively, rather than socket file descriptors. Consequently, the *send* function writes data to be transmitted to the *outCiphertextBuffer*, while the *recv* method writes received data to the *inCiphertextBuffer*. However, when using a blocking socket, data is expected to be returned, although the actual write operation has not yet occurred. In such cases, WolfSSL will return a *WOLFSSL_ERROR_WANT_READ* error on the initial write, requiring separate handling. Ultimately, the ECALL returns a result indicating that the untrusted component must execute IO operations on a specified socket.

Upon receipt of the ECALL result, the untrusted component initially verifies whether the returned response is a complete response for the Smart-ID enclave or if additional IO operations are necessary. This is done by inspecting a discriminator provided in the response. If the response is deemed final, it is directly transmitted to the Smart-ID enclave via the *async_response_from_https_enclave* ECALL, and the HTTP request has completed successfully. However, if additional IO operations are required, a *send* sys-

tem call is first executed with the provided data and the socket file descriptor, followed by a *recv* call to block for the response. To prevent blocking the thread handling the ECALL response itself, these operations are queued on a separate thread. Subsequently, upon receiving the response bytes, a *https_from_socket* ECALL is made to the HTTPS enclave to forward the received data for further processing.

The *https_from_socket* ECALL begins by performing a lookup for the internal session. This step requires that the session has already been created, and if the lookup fails, the function cannot proceed. Subsequently, the incoming data from the *recv* call is placed in the *inCiphertextBuffer*, and any errors encountered during this process are returned to the caller. This step is performed before any WolfSSL methods are invoked so that it can be used as input to the *recv* method inside the enclave. Once the incoming ciphertext buffer is populated, it can be read by WolfSSL to retrieve the server response. If there is no data available in the incoming buffer for WolfSSL, a *WOLFSSL_ERROR_WANT_READ* error is raised, and a new write is requested from the untrusted component. This write can consist of a TLS handshake message or application data if the handshake has already been completed and the outgoing plaintext buffer is not empty. When there are response bytes available, they are read into a dynamically expanding *inPlaintextBuffer*. This buffer is then analyzed to determine whether the entire HTTP response has been received. First, it checks for the presence of two CLRF blocks, indicating the end of the HTTP headers. If these blocks are present, the length of the body can be determined by examining either the Content-Length value or a zero-length chunk in the case of chunked encoding. It is critical that the HTTPS enclave performs the HTTP response parsing to minimize the workload for consumers and keep security risks localized. Once the HTTPS enclave determines that the entire response has been received and successfully parsed, it creates a response indicating that the response is final and should be forwarded back to the caller through the *async_response_from_https_enclave* ECALL. The complete flow indicating how the

Smart-ID enclave, HTTPS enclave and the untrusted component interact during a HTTP request is provided in Figure 8.

4.7 Testing

The Smart-ID service provider offers a distinct environment for relying parties to evaluate their Smart-ID integrations. This environment provides a public service for both versions of the Smart-ID RP API, along with a unique relying party name and UUID for testing purposes. Additionally, the CA certificate is provided, which can be utilized to verify the client's signature in the final session status response. However, the testing environment does not have access to the actual Smart-ID accounts, and specific semantic identifiers issued by the service provider must be used for testing purposes. Each response scenario, including positive responses, user cancellations, verification code mismatches, and timeouts, has a separate semantic identifier. It is crucial to note that the testing environment does not transmit push notifications to a mobile device. Instead, it returns a final poll response after a random delay. A comprehensive explanation of the Smart-ID RP testing environment can be found in the Smart-ID documentation [32].

As the prototype has not been fully integrated into Sharemind HI, a small application was developed for testing purposes. This application utilized dummy semantic identifiers to trigger Smart-ID authentication sessions using the *smartid_auth_init* ECALL function. Subsequently, the testing application entered a wait state for a period of 10 seconds before terminating. This wait period allowed for completion of the Smart-ID authentication process by other concurrent threads.

5 Results

The prototype Smart-ID authentication service developed as a part of this thesis was deployed on a server with the following technical specifications:

- CPU: Intel Xeon CPU E3-1225 v5 @ 3.30GHz
- RAM: 4x8GiB DDR4, ECC UDIMM, 2133 Mbps
- Storage: Samsung SSD 850 PRO 1TB

5.1 Memory usage

In order to ensure effective utilization of memory resources within enclaves, it is crucial to gain a clear understanding of the memory allocation patterns for both HTTPS and Smart-ID enclaves during a single authentication request, as well as during subsequent and concurrent authentication requests. This is because the memory available for each enclave is restricted by a predefined metric set at the time of enclave construction. To monitor and track memory allocation for a C++ application, it is possible to incorporate a modified version of the *malloc* and *free* functions, which are capable of logging statistics about memory allocations. This approach enables efficient identification and analysis of memory allocation behaviors within the enclaves. Listing 12 provides an example for the modifications. The modified version delegates the actual memory allocation to the original function, but uses a *enclave_printf_log*, a specialized function to print data to standard output, to log every memory operation. Upon enabling memory tracking during authentication flows, the resulting log lines can be used as input for a dedicated program to generate visual representations of the memory usage.

```
extern "C" {
   void* dlmalloc(size_t);
   void dlfree(void*);

   void* malloc(size_t s) {
      auto ptr = dlmalloc(s);
      enclave_printf_log("HTTPS malloc %zu %p", s, ptr);
      return ptr;
   }

   void free(void* ptr) {
      enclave_printf_log("HTTPS free %p", ptr);
      return dlfree(ptr);
   }
}
```

Listing 12. Definitions of specialized functions for memory tracking

A Python script was developed utilizing the Matplotlib library for effective data visualization purposes. Figure 9 presents the memory usage for both enclaves during three subsequent Smart-ID authentication requests. As demonstrated in the graph, the memory usage for a single Smart-ID session peaks at approximately 50kB when reading the final session status response, as this involves a large JSON object containing the user's signature and certificate. The same peak can be observed in the Smart-ID enclave. Moreover, the HTTPS enclave memory overhead remains around 10kB between requests, while the Smart-ID enclave exhibits barely any overhead. Furthermore, the graph illustrates that the enclaves do not experience large memory leaks, and used resources are properly cleaned up. Figure 10 illustrates the same memory usage during three concurrent Smart-ID authentication sessions. Here the memory usage for the HTTPS enclave peaks at around 100kB since the enclave has to keep multiple internal sessions with independent buffers in memory.

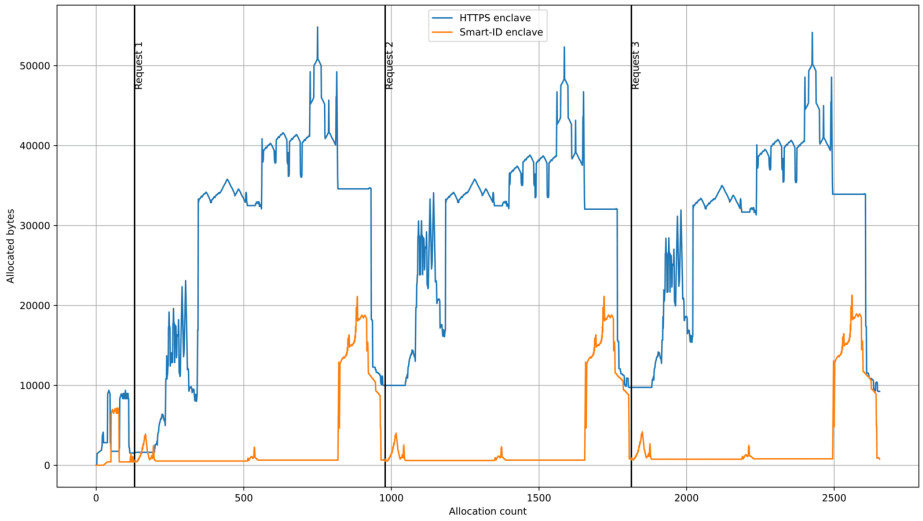

Figure 9. Memory usage during subsequent authentication requests

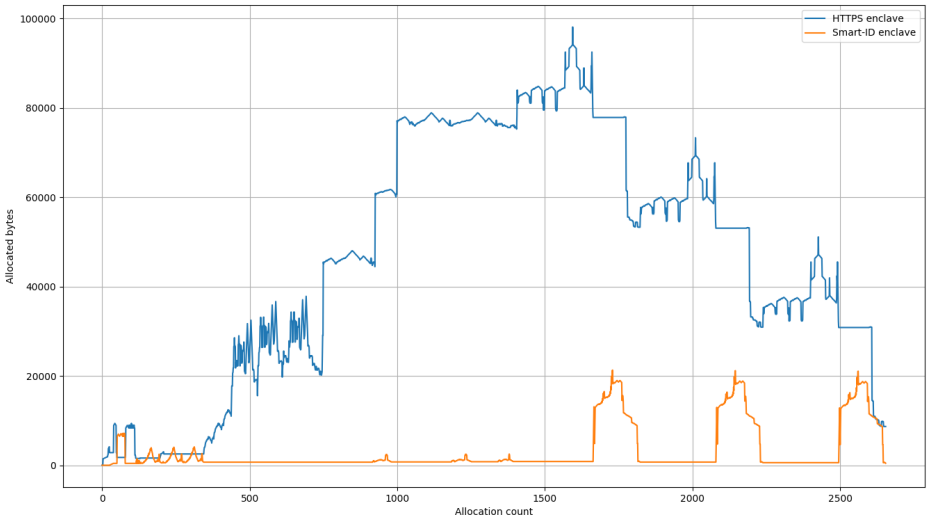

Figure 10. Memory usage during concurrent authentication requests

5.2 Performance

In addition to the memory usage, time complexity is another important metric to consider when designing an authentication service. It is a significant metric as it influences the length of the idle period that users must endure after inputting their PIN code, thereby directly affecting the user experience. In this context, Figure 11 illustrates timing results for a single authentication request, with error boundaries displayed across multiple requests. The elapsed time is plotted using a logarithmic scale as different ECALLs differ greatly in execution time. It is important to note that the figure only takes into account the ECALL overhead associated with the prototype service, and does not include the IO operations carried out by the untrusted component, which is a primary bottleneck in an Smart-ID authentication request. Furthermore, the ECALLs to both HTTPS and Smart-ID enclaves are asynchronous due to a queuing mechanism, which adds additional overhead.

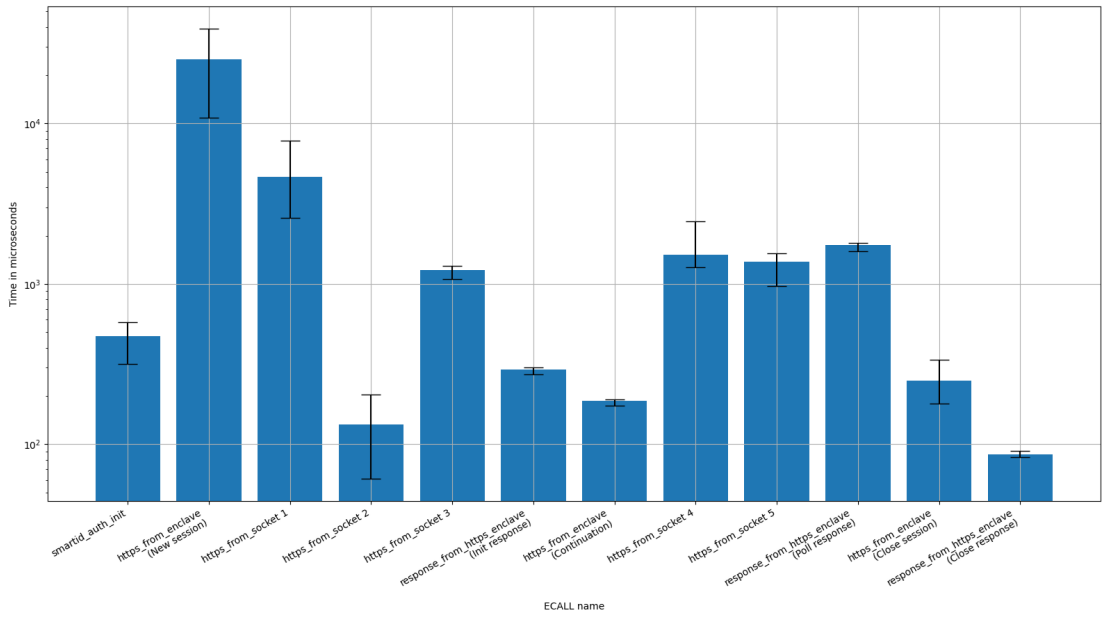

Figure 11. Elapsed time for ECALLs during an authentication request

The figure indicates that the majority of elapsed time is due to the initial HTTPS

request made by the Smart-ID enclave, and its corresponding socket ECALLs, which are responsible for negotiating a TLS session with the remote server and involve relatively slow asymmetric cryptographic operations. This could potentially be improved in the future by using connection pooling to reduce the number of required TLS handshakes over a longer period of time. During most authentication requests, it was found that three invocations of the *https_from_socket* ECALL were necessary to receive the session initialization response, and a further two to receive the final poll response. Occasionally, an additional roundtrip was observed, but the reasons for this are unclear and may be affected by factors on both the client and server sides, such as the TCP buffer state. However, this did not appear to cause noticeable delays in response time. The time taken by the Smart-ID enclave to process responses received from the HTTPS enclave was proportional to the size of the JSON payload received. The figure illustrates that the final poll response took considerably longer to process as it included a much larger payload and required cryptographic validation operations.

5.3 Security of the prototype

The incorporation of Smart-ID authentication into Sharemind HI has been executed without compromising its security guarantees. The conventional method of utilizing Smart-ID authentication and signing has been mapped one-to-one onto corresponding infrastructure components within Sharemind HI, with implementation details such as performing asynchronous HTTPS requests addressing limitations in the Sharemind HI code base. In a regular Smart-ID relying party service, the client communicates with the RP server via HTTPS, relying on TLS for security. Sharemind HI, however, uses *libsodium* secretstreams [33] to secure the communication between the client and Sharemind HI, which provides comparable security guarantees for the the communication channel between the user and the system. The separation of the external identity provider into a distinct enclave next to the core enclave has been implemented using

methods similar to established SSO protocols, with modifications made to accommodate Sharemind HI's deficiencies, such as the absence of time support.

The Smart-ID enclave has been designed to exclusively handle the task of informing the Core enclave about the outcome of Smart-ID authentication and the corresponding client identity. The Sharemind HI core workflows remain mostly unchanged. As in the previous setup, the DFC determines the access rights of the client, which is done in a similar fashion as with the existing CA stakeholders and dynamic end users within Sharemind HI.

In the previous setup, stakeholders were required to generate a long-term private key and certificate while configuring themselves to use Sharemind HI. However, with the introduction of Smart-ID authentication, a client generates an ephemeral key pair. The private key was previously used for creating signatures for the Sharemind HI audit log, but with the Smart-ID signing capabilities, a similar signature can be created. Therefore, the security of the system remains unaffected, as all the cryptographic primitives used earlier remain intact.

6 Discussion

This chapter provides a summary of the research carried out in this thesis, focusing on the authentication mechanism enhancements achieved for the Sharemind HI platform. Alongside the improvements, this chapter also enumerates potential directions for future research. These directions illustrate how the outcomes and techniques introduced in this thesis could be extended and improved.

6.1 Conclusion

The aim of this thesis was to create a proof-of-concept integration between the Smart-ID service provider and the Sharemind HI platform. By incorporating this integration, it would be possible to build privacy-preserving applications on the Sharemind HI platform, leveraging the widespread usage of the Smart-ID service in the Baltics. The research conducted in this thesis provides an overview of the underlying technologies and protocols, such as SplitKey, Intel SGX, and Sharemind HI, and discusses the prototype's design, implementation, and performance.

While examining the integration details for connecting an Intel SGX based application with the Smart-ID service provider, it was discovered that the security properties primarily depend on a secure HTTPS connection between the relying party and the service provider. The Smart-ID documentation demonstrates that a man-in-the-middle attack could be executed, which necessitates making the HTTPS requests from within the enclave, which Sharemind HI did not support.

The prototype solution presented in this thesis incorporated two new enclaves alongside the existing three enclaves in the Sharemind HI server. The first new enclave was dedicated to executing HTTPS requests to remote servers, while the second was utilized to encapsulate Smart-ID specific business logic and initiate Smart-ID RP API calls through the HTTPS enclave. By integrating these two new enclaves into the existing use

case flows of Sharemind HI, it was demonstrated that Smart-ID authentication could replace the current authentication mechanisms employed in Sharemind HI, thereby increasing flexibility.

The reference implementation provided in this thesis showcases the functioning of the integration and successfully receives an authentication session response from the Smart-ID service provider. This enables Sharemind HI to authenticate the stakeholder's identity and carry out its operations.

6.2 Future work

While the proof-of-concept implementation serves as a good baseline, it has multiple limitations which do not allow it to be used in Sharemind HI in its current form. First off, the WolfSSL TLS implementation used in the reference implementation due to its ease of integration with Intel SGX needs a commercial license to be used in products such as Sharemind HI. If this turns out to be an issue, then WolfSSL needs to be replaced. Therefore it will be necessary to either solve the licensing issues or migrate to an alternative implementation such as OpenSSL.

Additionally, to allow authentication to Sharemind HI instances using Smart-ID, it is necessary to develop the Sharemind HI SDK that can support Smart-ID specific calls from the client to the core enclave. As this thesis is solely concerned with the server-side implementation, any enhancements to the client-side must be carried out as part of future work.

As of the time of writing, Cybernetica AS is engaged in further developing the prototype with the aim of rendering it commercially viable. A considerable amount of work is still required before the service can be offered commercially. Furthermore, it is important to highlight that the current implementation suffers from certain limitations, as previously discussed. Additionally, the prototype has not been optimized in terms of time and space, and has not been subjected to thorough testing.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," 2013.

[2] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password strength: An empirical analysis," in *2010 Proceedings IEEE INFOCOM*, 2010, pp. 1–9.

[3] "Regulation (eu) no 910/2014 of the european parliament and of the council of 23 july 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec," https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32014R0910 (07.05.2023).

[4] J. Fruhlinger, "What is PKI? and how it secures just about everything online," https://www.csoonline.com/article/3400836/what-is-pki-and-how-it-secures-just-about-everything-online.html (07.05.2023).

[5] "The transport layer security (TLS) protocol version 1.2," https://www.rfc-editor.org/rfc/rfc5246 (07.05.2023).

[6] J. Schwenk, *A Short History of TLS*. Cham: Springer International Publishing, 2022, pp. 243–265.

[7] T. Dierks, "Security standards and name changes in the browser wars," https://tim.dierks.org/2014/05/security-standards-and-name-changes-in.html (07.05.2023).

[8] "The transport layer security (TLS) protocol version 1.3," https://www.rfc-editor.org/rfc/rfc8446 (07.05.2023).

[9] "Upgrading to TLS within HTTP/1.1," https://www.rfc-editor.org/rfc/rfc2817 (07.05.2023).

[10] A. Arampatzis, "Where is a TLS/SSL handshake most vulnerable?" https://venafi.com/blog/where-tlsssl-handshake-most-vulnerable/ (07.05.2023).

[11] N. Sullivan, "Why TLS 1.3 isn't in browsers yet," https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/ (07.05.2023).

[12] A. D. Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely," in *Symposium on the Theory of Computing*, 1994.

[13] L. T. A. N. Brandão, M. Davidson, and A. Vassilev, "NIST roadmap toward criteria for threshold schemes for cryptographic primitives," 2020.

[14] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[15] "Smart-ID technical overview," https://github.com/SK-EID/smart-id-documentation/wiki/Technical-overview (03.12.2022).

[16] A. Buldas, A. Kalu, P. Laud, and M. Oruaas, "Server-supported RSA signatures for mobile devices," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 315–333.

[17] "Smart-ID documentation," https://github.com/SK-EID/smart-id-documentation/blob/master/README.md (02.02.2023).

[18] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[19] "SGX 101," https://sgx101.gitbook.io/sgx101 (07.05.2023).

[20] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc' approach to authenticated diffie-hellman and its use in the IKE protocols," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 400–425.

[21] "Input types and boundary checking in enclave-definition language (EDL) files," https://www.intel.com/content/dam/develop/external/us/en/documents/input-types-and-boundary-checking-edl-737361.pdf (17.04.2023).

[22] "Sharemind HI white paper," https://cyber.ee/uploads/sharemind_hi_white_paper_ec24e8189a.pdf (21.10.2022), 2021.

[23] "Sharemind HI overview," Internal document.

[24] "CoNurse," https://cyber.ee/research/projects/conurse (07.05.2023).

[25] "Smart-ID," https://www.smart-id.com/ (26.04.2023).

[26] "Electronic signatures and infrastructures (ESI); certificate profiles; part 1: Overview and common data structures," https://www.etsi.org/deliver/etsi_en/319400_319499/31941201/01.04.04_60/en_31941201v010404p.pdf (13.04.2023), 2021.

[27] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 178–195. [Online]. Available: https://doi.org/10.1145/3243734.3243822

[28] S. Weiser, R. Spreitzer, and L. Bodner, "Single trace attack against RSA key generation in intel SGX SSL," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 575–586. [Online]. Available: https://doi.org/10.1145/3196494.3196524

[29] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham: Springer International Publishing, 2016, pp. 440–457.

[30] "WolfSSL versus OpenSSL," https://www.wolfssl.com/docs/wolfssl-openssl/ (13.02.2023).

[31] "WolfSSL linux enclave example," https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux (13.02.2023).

[32] "Environment technical parameters," https://github.com/SK-EID/smart-id-documentation/wiki/Environment-technical-parameters (26.01.2023).

[33] "Encrypted streams and file encryption," https://libsodium.gitbook.io/doc/secret-key_cryptography/secretstream (09.05.2023).

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Markus Punnar**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Experimental Integration of the Smart-ID Service Into Intel SGX Enclaves,
   *(*title of thesis)

   supervised by Peeter Laud and Armin Daniel Kisand.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Markus Punnar
09/05/2023