DATA ESCROW WITH TRUSTED COMPUTING AND BLOCKCHAIN

DU XUE

(MComp, NUS)

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF COMPUTING, INFOCOMM SECURITY SPECIALISATION

DEPARTMENT OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2019

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

DU Xue

7 November 2019

Table of Content

Summary 4

List of Tables 5

List of Figures 5

1 Introduction 6

1.1 Motivation for Trusted Data Escrow 6

1.2 Hardware Based Root of Trust. 7

1.2.1 Existing Solution with TPM 7

1.2.2 Issues on TPM based Data Escrow Solution 7

1.3 Blockchain as the Root of Trust 8

1.3.1 Existing Solution 8

1.3.2 Issues with Blockchain based Data Escrow 8

1.4 Our Work 9

1.4.1 The Design Goals and Requirements 9

2. Background 10

2.1 IntelSGX as Trusted Execution Environment 10

2.1.1 Code and Data Isolation 10

2.1.2 Sealing and Unsealing 11

2.1.3 Attestation 11

2.2 Initial Design on Data Escrow Solution 11

2.2.1 Workflow for the Initial Design 12

2.2.2 Design Evaluation 13

2.3 Issues in Initial Designed and Our Contributions 13

2.3.1 Secure Processing Implementation 13

2.3.2 Against Man-in-the-Middle (MitM) Attack 14

2.3.3 Trusted Verification on Block Status 15

3 Enhanced Design 15

3.1 Threat Model 16

3.2 Trusted Computing Base (TCB) 16

3.3 Architecture Diagram 17

3.3.1 Smart Contract on Blockchain 19

3.3.2 Escrow Server Component 20

3.3.3 Authority App 23
3.3.4 Requester Console 24
3.4 Enhanced Escrow Workflow 24
3.4.1 Wallet and Enclave Initialization 25
3.4.2 Data Registration 26
3.4.3 Data Access Request 29
4 Implementation 32
4.1 Data Escrow DApp and Ethereum Test Network 32
4.2 Proxy App 41
4.3 Enclave App 43
4.3.1 Wallet and Enclave Initialization 43
4.3.2 Data Registration 43
4.3.3 Data Access Request 44
5 Limitations and Future Work 45
5.1 Data Availability Concern 45
5.2 Requester Side Data Protection 45
5.3 Lack of Key Rotation for Escrowed Data 46
6 Conclusion 46
7 Bibliography 47
8 Appendices 52
Appendix 1 Initial Design Flowchart 52

Summary

Data escrow solution provides a great method for managing the owner's data accessibility when requested, and confidentiality at rest, in a trusted third-party environment. Traditionally, the trust in the third-party escrow solution is based on legal agreement and organization reputation, but the trust can be easily breached when the system is compromised, or the company goes bankrupt. Some researchers proposed TPM-based escrowed data implementation to transfer the data owner's trust to the hardware based Trusted Platform Module (TPM) of the escrow server. However, the TPM-based solutions suffer two major drawbacks: 1) TPM itself is not totally secure under power interruption attack and 2) TPM serves as the single point of trust which can be compromised under a collusion scenario.

In this work, we design and implement a secure data escrow solution by combining blockchain and IntelSGX to be the root of trust. The data escrow solution provides data confidentiality by generating the encryption key in the hardware-protected isolated memory space, and protecting data using Dual Secret Key Encryption where the private key is split into 2 parts and stored separately: with an authority and within the IntelSGX.

In addition, the designed solution provides access log integrity and the data access request's authenticity by leveraging on smart contracts that are programs executed business logic in a transparent and distributed manner. The smart contract execution ensures the authenticity of the sender, and the consensus protocol ensures the integrity of the audit log block is verified by majority of nodes in Ethereum networks. Therefore, the data is tamper-proof, and the audit log supports public verification.

In section 2, we briefly introduced the IntelSGX functions and the initial design from pervious students. In section 3, we explained the threat model and the enhanced architecture design as well as the security properties achieved. In section 4, we described the workflow for data escrow process with technical details. We further dived into details in the system implementation and the challenges encountered in section 5. Last but not least,

in section 6, we discussed the limitations of the work and proposed the potential areas for future work.

List of Tables

Table 1: Function and Key Denotation 19

List of Figures

Figure 1: Architecture Diagram 18

Figure 2: Wallet and Enclave Initialization workflow 25

Figure 3: Data Registration Workflow 26

Figure 4: Data Access Request Workflow 29

1 Introduction

1.1 Motivation for Trusted Data Escrow

Data is becoming one of the most valuable assets for data driven business functions and for digital transformation of the strategic industries such as healthcare, public sector, e-commerce and financial services. However, that massive data about every aspect of our lives collected poses potential risks to individual privacy. There has always been debate on balancing public security functionality and privacy [4, 8, 17]. Escrowed Data is proposed as an approach that is capable of providing an appropriate balance - data collected is held in escrow for a certain period; during that period, the data may be accessed by an authority and it is expected that only a minority of the data needs to be accessed, e.g., for the purposes of crime investigation; after the escrow period, the data can be destroyed [1].

The escrowed data technique may be useful in a variety of use cases, for instance:

- Hospitals may use escrowed data technique to store and protect patients' personal health records [11]. Only with patients' authorization or emergency situation, doctors or laboratory with pre-defined privilege can access to their personal health records [12].
- Government may use escrowed data technique to ensure only under authorized scenarios, a legitimate requester can access to an individual's criminal records such as background check during employment.
- Internet companies may use escrowed data technique to backup their competitive advantage source code to ensure their business continuity and ensure only authorized stakeholders can access to the source code [7, 14].
- Regulators may use escrowed data technique to securely handle company or personal financial records for investigation or audit purposes. With the pre-authorization from the of data owners,

regulators can access the financial records when the predefined conditions are met.

The key for the implementation of escrowed data is to build the trust of the data owners to the party that holds the data. To establish the trust, it is essential for the data holding party to provide the assurance that only when the predefined conditions are met and only to the authorized persons, the data is released. The assurance level is low if the trust is relying on the data holding party as an authority to log the data access history activities and report the access log correctly and honestly.

1.2 Hardware Based Root of Trust.

1.2.1 Existing Solution with TPM

The Trusted Platform Module (TPM) provides hardware-based security and cryptographic functions with a secure hardware repository to facilitate key management, data encryption, and secure execution of trusted application. To improve the level of trust, researchers have built escrowed data mechanism leveraging on TPM to provide the root of trust [41]. With TPM, the data owner does not need to trust the data holding party as the TPM provides the technical evidence showing whether the data has been accessed at the end of the escrow period.

1.2.2 Issues on TPM based Data Escrow Solution

However, the TPM does not provide sufficient level of assurance to data owner due to the following weaknesses:

- TPM vulnerabilities allow attackers to reset the TPM and create a fake boot-up chain of trust by abusing power interrupts [3]. This reduced the level of trust that data owner can put in build to the data escrow with TPM because the trusted components can be tempered but not recorded with this attack.
- TPM is the single point of trust to enforce the data's access control. If the TPM state can be tempered or the TPM colludes with the data holding party, data access may be falsified deleted [13].

- There is no access control to ensure if the access requester meets the data access criteria.

1.3 Blockchain as the Root of Trust

Blockchain technology (e.g., bitcoin [15] and Ethereum [6]) provides high integrity and authenticity to each transaction (e.g., data) published in the distributed leger. Each transaction is digitally signed with the transaction owner's private key to ensure the authenticity of the transaction. And to add a block that contains the transactions to the chain, a majority of the nodes within the blockchain network must verify that the block and the transactions are valid and come to a consensus on the chain. Once the transaction in a block is added to the chain, it is difficult to change it. This enables the blockchain to be the root of trust - the data access can be logged in the blockchain as transactions. In this way, the TPM rebooting attack and the requester collusion with the data holding parity can be reduced since blockchain is tolerant to a number (minority) of failure nodes. In addition, access control policies can be enforced in blockchain, e.g., written as smart contracts in Ethereum, so that specific access control can be defined.

1.3.1 Existing Solution

In the literature, there have been escrow services using blockchain e.g. [5, 16], as blockchain provides better trust - instead of trusting one authority like banks, blockchain applications trust that the majority of nodes in the blockchain network behave honestly. They are mainly financial escrow i.e., the escrowed data is digital currencies. There is no escrowed data technique; and the existing escrow service cannot be directly adopted as data is not easy to be monetized. There has also been access control via blockchain, e.g., in health care systems [2, 9, 10].

1.3.2 Issues with Blockchain based Data Escrow

However, most of the systems are designed for storing/sharing data. That is, how the data is kept private is out of the scope. In fact, synthesising access control with the data protection is non-trivial. One may argue that data

protection can be achieved by encryption, e.g., asymmetric key encryption. However, how to store the private key to decrypt the ciphered data poses a technique challenge, as data in the blockchain is public, i.e., private keys stored in the transactions or smart contracts are trivially revealed to attackers.

1.4 Our Work

In our designed data escrow solution, we leverage hardware enforced trusted execution product - Intel's Software Guard Extensions (IntelSGX) to generate the keys required. We store the private key securely in the IntelSGX isolated memory space called enclave and distribute the key to authority via a secured channel. IntelSGX provides proof of secured code execution for trusted program that allow public verification via remote attestation and also capable of preventing privileged attack at operation system (OS) level. In addition, we aim to bother the data owner as less as possible. Thus, we do not require the data owner to be responsible for key storage and distribution. With smart contract and IntelSGX deployed as the root of trust, the key lifecycle and access control verification can be handled in a secured manner.

1.4.1 The Design Goals and Requirements

Our goal is to design and build an escrowed data solution based on blockchain and IntelSGX that provides the technical assurance that data can only be decrypted based on 2 criteria: 1) the access control rules are fulfilled; and 2) the access event is successfully logged on blockchain. We will design and implement our solution to ensure the following requirements are fulfilled:

- The solution maintains a minimal trusted computing base (TCB) where the integrity and correctness of program execution within the TCB can be verified publicly.
- The solution guarantees the confidentiality and integrity of the escrowed data both during transit among the system components, and when stored in escrow vault.
- The solution strictly enforces the escrowed data access control rules. Only when all the rules are verified in the TCB, the decryption key can be provided for data access.

- The solution guarantees the integrity and authenticity of the access request log. The log should be tamper proof and bind to the requester identity.

2. Background

2.1 IntelSGX as Trusted Execution Environment

IntelSGX is a set of hardware and CPU instructions that provides application with hardware enforced confidentiality and integrity protections. IntelSGX enables secured process execution in a protected memory address, referred to as enclave, that can only be accessed by a set of predefined trusted code. IntelSGX provides both hardware, privileged OS level attack and software attacks. The hardware-enforced checks prevent enclave memory from being accessed by non-enclave code in CPU SRAM. The hardware based SGX Memory Encryption Engine (MEE) provides encryption on enclave memory before writing it to the DRAM system memory. The following 3 capabilities of IntelSGX which we used extensively in our design: code and data isolation, sealing, and attestation.

2.1.1 Code and Data Isolation

The code and data inside enclave are protected from even privileged processes and can only be accessed by the code residing within the enclave memory space [25]. The communication between enclave and untrusted app can only happen via special instructions defined in the Enclave Definition Language (EDL) files. The EDL file contains the trusted libraries and untrusted libraries. The enclave functions which are exposed to the untrusted app are defined in the trusted EDL part. These endpoints can only be invoked using enclave call (ecall) function. When enclave invokes functions outside the enclave to use OS capabilities such as system calls, I/O operations, and so on, enclave makes outside enclave call (ocall). The ocall functions have to be defined in the untrusted EDL part. IntelSGX SDK provides libraries and APIs to help developers implement ecall and ocall functions.

2.1.2 Sealing and Unsealing

IntelSGX provides data encryption capability with a specific Seal Key which is generated dynamically based on enclave identity and the Root Seal Key embedded in the CPU during manufacturing process. The Seal Key can be recovered by the same enclave on the same platform. Sealing function encrypted data with Seal Key using AES-GCM algorithm and an Additional Authentication Data (AAD) to ensure both the confidentiality and authenticity. The sealed data can be stored in the non-volatile memory.

2.1.3 Attestation

Attestation process provides proof on the platform's identity and the program executable has been instantiated on the platform correctly. IntelSGX provides 2 attestation mechanisms: local attestation and remote attestation.

In local attestation, two enclaves running on the same platform can build a secure session provide assertion on their identity and using Intel Elliptical Curve Diffie-Hellman (DCDH) key exchange library.

The remote attestation process allows a remote party to gain confidence on a program that is securely running in the IntelSGX enclave. To enable remote verification, IntelSGX uses Intel Enhanced Privacy Identifier (EPID) key that is bound to the processor firmware to sign the enclave attributes and measurement, and the generated proof is referred as quote. Quote can be verified remotely using the Intel Attestation Service over TLS [23]. EPID group signature scheme relies on bilinear pairings on elliptic curves to achieve the anonymous of the signing process [24]

2.2 Initial Design on Data Escrow Solution

Our work is built on top of the initial design of the data escrow solution which builds the root of trust based on blockchain and hardware based trusted computing [18, 20]. The initial design comprises of 3 major components to achieve the level of trust:

- Smart Contract: ensures the integrity and authenticity of access logs for the escrowed data. In blockchain, each block is verified by majority of the nodes in the network, and all nodes maintain a copy of that data which makes the access log tamper proof.
- Dual Secret Key Encryption: ensures the confidentiality of the encrypted data. To reconstruct the secret key for data decryption, requester will need to obtain 1 share of key from authority and obtain the other share of key from the proxy by submitting the access log on the blockchain. Therefore, no single entity is able to decrypt the data escrowed.
- Trusted Execution Environment (TEE): provides a secure execution environment for key generation and performs trusted verification on the access log block status.

2.2.1 Workflow for the Initial Design

Referring to the initial design workflow diagram in Appendix 1 from the previous work [20], the workflow is demonstrated in following 2 processes:

- Sensitive Data Submission

To submit a data escrow request, the data owner submits a request form and instructs the proxy to generate a public key (PK), secret key 1(SK_1) and secret key 2 (SK_2). The proxy distributes SK_1 to authority and PK to the data owner for data encryption. Then the data owner sends the encrypted data to authority. The authority initiates a transaction to the smart contract to register the data owner's access control list (ACL) on the blockchain.

- Request for Access to Sensitive Data

Requester request for the encrypted data and the SK_1 from authority by authenticating the requester identity and privileges. The requester sends the access log to smart contract, and the blockchain nodes verify the request against the ACL stored in blockchain. The node running in the TEE of the proxy will verify the final block consensus status and issues SK_2 to requester for data decryption.

2.2.2 Design Evaluation

In the previous work, the security properties of the initial design are verified with formal verification using Process Analysis Toolkit (PAT). The results from the modelling of the proposed design in PAT asserts following 3 security properties are achieved:

- The data requester could access to the sensitive data at all time if authorized.
- The ACL for data access is transparent.
- Data owner's sensitive data can only be accessed by authorized requester with unbiased verification process by multiple independent parties.

With the components and techniques described in section 2.2.1, there are additional 2 security objective achieved:

- Using TEE and dual secret key encryption, ensures the user sensitive data remains secure from unauthorized access.
- Using smart contract ensures the requester's access records is secured and not subject to modification or deletion.

2.3 Issues in Initial Designed and Our Contributions

2.3.1 Secure Processing Implementation

The design of the TEE related implementation is an overview which is lack of practical considerations of TEE limitations that may affect the implementation of the proposed design. Following limitations pose the key challenges in the system design and implementation:

- TEE has limited memory size for secured code execution.
- Hosting an Ethereum node within TEE greatly increases the complexity and the size of TCB.
- TEE is lack of network connectivity as a memory space.
- Controls to ensure messages going out of TEE cannot be modified or impersonated by the proxy component.

In our work, we implement another type of TEE - IntelSGX with better API and SDK support. We define the minimally required functionalities to run in the IntelSGX enclave as explained in section 3.2.3. We further ensure the message delivery from enclave to external word to be secured with remote attestation function and TLS connection directly from the enclave. In our design, the size of TCB is much smaller comparing to the initial design and also ensures the required security properties can be fulfilled.

2.3.2 Against Man-in-the-Middle (MitM) Attack

In existing work, the proxy service is considered to be part of the authority and the TEE is located within the proxy server. The proxy plays a role as data relay service between TEE and other components such as blockchain, authority app, data owner, requester. The proxy is assumed to be trusted in this case to ensure the security properties. However, the assumption requires the proxy to be part of the root of trust for data owner which is not practical.

In our work, we removed this assumption and considered proxy as an untrusted component. The proxy is capable of launching a MitM attack by intercepting and modifying the messages in between the TEE and other components. The proxy is not trusted. Even when the proxy is compromised at the OS level, we ensure the sensitive data is still securely stored and the message integrity is verified for communication between IntelSGX enclave and other components. We leverage on following features:

- Secure function proof: leverage on the remote attestation process of IntelSGX to attest the code is not tampered.
- Secure key issuance process: when we generate the public and private key pair for the data owner, we use IntelSGX maintained private key (SK_{ENC}) to sign the generated data owner public key (PK_D) to allow data owner verification that the key is not modified by proxy.
- Secure key distribution process: the private key (SK_D) generated for data owner will be split into two private keys for distribution, named SK_{D1} and SK_{D2}. During key distribution process, we establish HTTPS connection from enclave to the contacted party directly to ensure the

confidentiality and integrity of the SK even if the traffic is relayed by the proxy. We leverage on the nature of HTTPS, which is a layer 7 security protocol, so that we can establish TLS between enclave and the third party directly at application layer but using the network capability of proxy host for TCP connections [19].

- Final verification attestation: the enclave verifies the block signature to ensure the block is not generated by the proxy and sign the verification status with SK_{ENC} to ensure Proxy is not modifying the status.

2.3.3 Trusted Verification on Block Status

In existing design, in order to verify the access log block status in blockchain within the TEE, it is proposed to host the blockchain nodes inside TEE. Booting a node in the enclave is not practical, considering the size of core Ethereum implementation is about 50K lines of C++ code and enclave uses expensive memory space for secure processing. This will increase the size of TCB and prone to errors and vulnerabilities [19]. The transaction actions require wallet setup and libraries which are not considered as as trusted enclave functions.

To enhance the existing work, we leverage on Ethereum transaction protocol which by default verifies the signature of each transaction added as a block. All transactions have to be initialized from a wallet account. Instead of booting a node in the enclave, we initialize the wallet account public key and private key in the enclave and ensures that the private key can only be accessed inside the enclave. Therefore, the block signature can only be signed within the enclave. This ensures the block verification transaction is initiated from the enclave through the authenticated channel [27].

3 Enhanced Design

Building on top of existing designs, we propose an enhanced design to address the issues discussed in section 2.3. We add additional security controls in the designed functionalities and workflow to address the security concerns during implementation.

3.1 Threat Model

The enhanced design aims to provide security properties like confidentiality, integrity, non-repudiation and authenticity. The system design considers following threats are present:

- **Spoofing:** an unauthenticated personal (insider or outsider) claims to be legitimate requests and request for data access.
- **Tampering:** an adversary intends to modify the access log or the program verifying the access rules.
- **Repudiation:** a malicious requester may deny his past access or authority deny past approval on a requester's access.
- **Information disclosure:** an adversary is trying to steal data in transit, at rest and in use.

3.2 Trusted Computing Base (TCB)

We limited the TCB to the minimal size to reduce program complexity and vulnerabilities due to human error. The TCB includes 2 main components: the smart contract on Ethereum, and the IntelSGX enclave application. The other components are considered not trusted in general and could be compromised to perform malicious activities described in section 3.1.

Data Escrow Smart Contract is deployed on all the nodes in the blockchain which is used to maintain a tamper proof audit trail for data access activities. The contract cannot be modified once published on blockchain and the program code is publicly verifiable. The security properties for contract is discussed in section 1.3, hence we consider this component is trusted.

Decentralized Application (DApp) is a web application used to interact with smart contracts deployed on the blockchain. Clients access this application with their wallet application to initialize contracts. The interaction between DApp and the contract is considered trusted because all transactions are initiated from an authenticated wallet account with signature. However, the communication between the DApp and Relay Application is considered not

trusted as attacker is able to perform a MitM attack to capture and modify the message sent from enclave to contract.

Relay Application is used as a backend relay function to facilitate the communication between the Enclave Application and the DApp. It is not trusted and is susceptible to MitM and spoofing attack by generating fake block to trick the enclave to provide SK for unauthorized data decryption.

Enclave Application is used to perform security critical functionalities. As discussed in section 1.4, we consider enclave functions are trusted.

As part of the private key (SK_{D2}) used for data decryption will be distributed to the Authority, who will distribute the key only after user authentication and escrow audit log verification. We assume the authority performs requester identity verification honestly as they are the identity provider (IDP) for requester. However, the authority is not trusted by the data owner, they should not be able to access to the data in plaintext.

3.3 Architecture Diagram

In Figure 1, we demonstrate the architecture design of the data escrow solution leveraging on IntelSGX and blockchain for access log maintenance. The TCB components mentioned in section 3.1 are highlighted in green boxes, while the untrusted components are highlighted in black boxes. We analysis how each component contributes to the design goals and security properties in following sections.

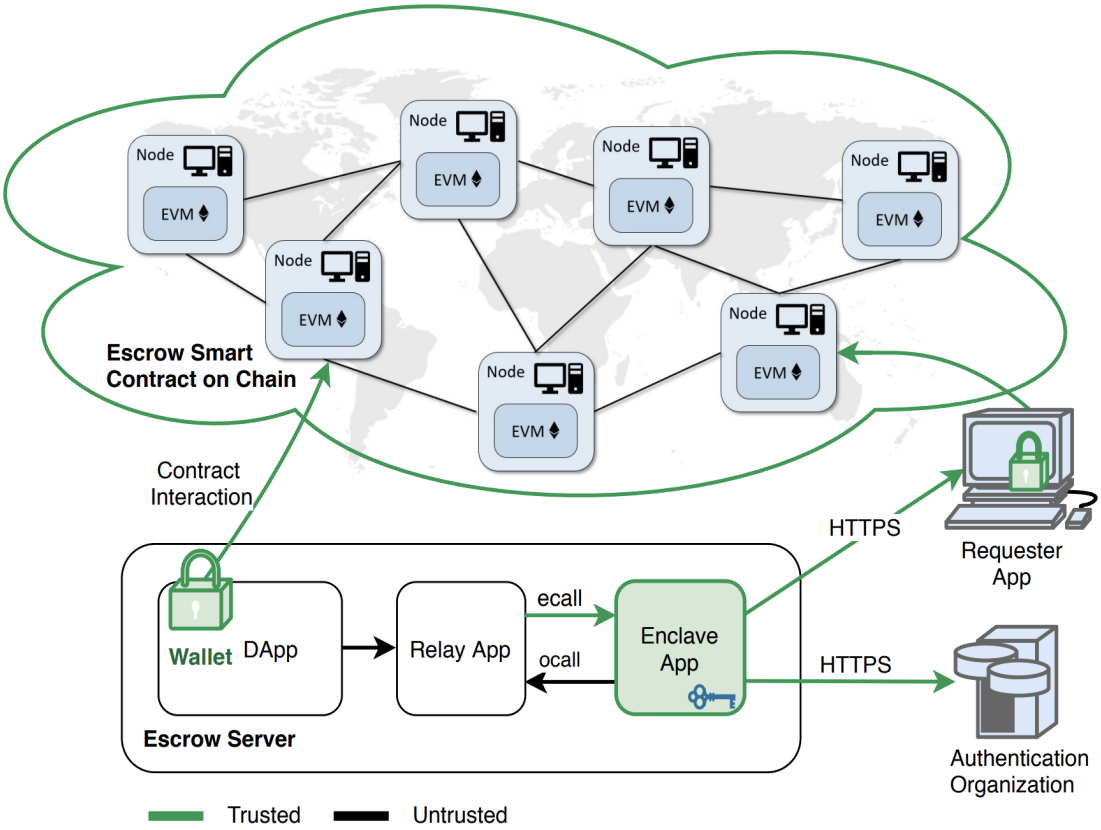


Figure 1: Architecture Diagram

We use the denotation defined in Table 1 for the keys and functions mentioned in the rest of the sections for simplicity.

Denotation	**Description**
$Qoute_{RA}$	Quote generated from enclave remote attestation process
$Sign_{SK}$	Signature generated with secret key SK on the rest of the payload.
$E(Data)_{PK}$	Encryption function that uses public key PK to encrypt data
$D[E(Data)]_{SK}$	Decryption function that uses secret key SK to decrypt encrypted data.
PK_{enc} , SK_{enc}	The public and private key of enclave's Ethereum wallet account.

SK_{seal}	The symmetric secret key for seal function in enclave
PK_{EPID} , SK_{EPID}	The public and private key of IntelSGX EPID group key scheme used for remote attestation
$PK_{RWallet}$, $SK_{RWallet}$	The public and private key of data access requester's Ethereum wallet account.
PK_D , SK_D	The public and private key for each data submitted for escrow.
PK_{Auth}, SK_{Auth}	The public and private key of authority app who is the IDP for requester.
DID	Escrowed data identifier
EID	Enclave identifier generated during enclave initialization
RID	Data access requester identifier

Table 1: Function and Key Denotation

3.3.1 Smart Contract on Blockchain

Smart contract is a program that contains any number of functions which is stored and distributed in all the nodes in Ethereum once deployed. Smart contract takes inputs, from another contract or from an externally owned account managed by a wallet app, and then executes the program in the EVM of all the nodes to ensure the execution trustworthiness [18]. Comparing to other blockchains that only ensure the trustworthiness of the storage of cryptocurrencies, smart contract makes Ethereum distinct from others [21]. The transaction in Ethereum is hashed with SHA3 and the digest is signed with Elliptic Curve Digital Signature Algorithm (ECDSA) at client-side to ensure the authenticity and integrity of the communication outside the blockchain [22]. Therefore, we consider the interaction between a DApp and a contract is secure.

The Smart Contract on Blockchain component performs the following functions:

- Record the data escrow request

A new data entry request is sent as a transaction and added as a new block in Ethereum. The transaction contains metadata of the escrowed data, the access control list (ACL) and access rules defined by the data owner. This ensures the integrity of the ACL and rules submission and storage.

- Record the data access request:

Data requester initiates a new transaction with its own wallet account to add a new block on chain with the data access request information including requester identity, data ID to be accessed, and the timestamp. The block signature can be validated by anyone (including the enclave app) to ensure non-repudiation, integrity and authenticity.

- Verify data access request against the ACL and the rules

This smart contract function will be executed by all nodes with trustworthy result.

3.3.2 Escrow Server Component

Escrow server is maintained by the data escrow service provider, be it an independent entity or a part of the authentication organization. We consider the server component is vulnerable to attacks at network level, OS level and application level, hence it is not trusted. The only trusted component in the server is the IntelSGX enclave which is able to provide remote attestation to prove the program running inside the enclave is not tampered [19].

3.3.2.1 DApp

DApp is a term used in Ethereum, which stands for decentralized app. It provides the user interface (UI) for an account to interact with the smart contract on blockchain. At the client side, DApp is able to interact with the client's wallet application to initiate transactions [18].

The Data Escrow DApp provides a web UI for data owner to submit escrow request and view their data access log. DApp is not trusted but the interaction communication with smart contract on chain is trusted.

3.3.2.2 Proxy App

Proxy app relays messages between the DApp and enclave app. It provides following functions:

- Data Relay

Proxy app relays data between the DApp and Enclave via ecall and ocalls defined in enclave provided edl functions. It translates the data received from DApp as the inputs to different enclave functions for processing and translate the enclave output to the smart contract transaction required information. We consider that an adversary is able to intercept and modify the data at this component, hence no sensitive information such as private key should be handled directly by Proxy app and the data source (DApp and Enclave) should send data with security measures to ensure the authenticity and integrity of the data such as a signature of the hash of the message.

- Provide OS service and Network Capability

Proxy app provides OS service, network capability and system time service for enclave app. In order to establish a secure connection with the requester app and authentication organization, the enclave app has to rely on Proxy host TCP network service, which can be potentially compromised. We split the HTTPS implementation in to 2 parts: the TLS handshake with the target server inside the enclave, and the lower layer TCP network service in proxy.

3.3.2.3 Enclave App

Enclave program is a set of hardware protected instructions which is executed in the protected memory address space and access to OS and system software only via controlled entry points (ocall and ecall). Following functions are realized in enclave application:

- Remote Attestation for Trusted Program Integrity

IntelSGX attestation function generates a hardware-based assertion for the enclave's identity, as well as the data and code running inside. The remote attestation function allows public verification on the enclave integrity.

- Key Generation for Data Encryption and Decryption

Enclave key generation function generates a private key and public key pair for each data escrow request, based on Elliptic curve cryptosystems (ECCs) with a key size of 256 bit [25]. The public key is sent as a transaction to the smart contract for the data owner to retrieve and use for data encryption. The private key is split into SK_{D1} and SK_{D2} with Shamir's Secret Sharing algorithm which requires a minimum of 2 shares to reconstruct the private key in order to decrypt the data [26]. Because only the SK_{D2} is distributed to the authority, the encrypted data remains confidential even if the organization is compromised. SK_{D1} is securely sealed by enclave and stored in Proxy storage.

- Secure Key Storage

As the secret generated in enclave is only stored in volatile memory and it will be removed once the enclave is destroyed, the secret has to be sealed and stored outside the boundary of enclave. The SK_{D1} will be sealed with the seal key uses AES-GCM algorithm. Seal key is not stored and will be regenerated every time a seal/unseal operation is initiated in the same enclave on the same platform. It is derived from the silicon and the enclave's SIGNER measurement register hence is unique to each enclave and platform. Seal key lifecycle is completely managed within the enclave boundary [25].

- Establish Secure Session with Remote Party

This secure session functions include the authentication and key exchange part. Intel SGX key exchange libraries performs key exchange using ECDH algorithm and address the MitM attacks in ECDH with additional authentication step. The enclave's identity is authenticated with EPID group key and the remote party can be authenticated with PKI [25]. The shared secret key is calculated based on Elliptic Curve variant of the standard Diffie Hellman algorithm. This function ensures the message can be delivered from

enclave to the remote party under a secured session. Relay is capable of eavesdropping and intercept the message but no cleartext can be obtained from this secure session.

- Access Request Block Verification

As discussed in section 2.3.3, we will initialize the wallet key pair from the enclave to send the block verification transaction with the escrow wallet whose private key is created and sealed by the enclave. Leveraging on the Ethereum feature that all blocks are self-authenticated using Merkle trees [6], the escrow transaction verifies the block data sent by the data requester. Once successfully verified, the escrow enclave confirms that the access log is published on the blockchain. Enclave unseals the SK_{D1}, establishes secure session with the callback URL, and sends over the SK_{D1}. However, an attacker is able to create a fake block copying other block's data and adding his callback URL to retrieve the SK_{D1} from enclave. To prevent this replay attack, following controls are added to the verification process:

- Requester identity is represented by his wallet account and registered with the authentication organization as discussed in section 3.3.3.
- The enclave verifies if the identity attested by authentication organization matches the access request block signature wallet account as they are the same public key bind to the requester wallet account.

In this case, even if the attacker replays previous valid block data with his own wallet account and callback URL, the enclave would still reject the block when observing the verified identity public key is different from the wallet account. To add an additional layer of protection, the escrow implements message level encryption to encrypt the SK_{D1} with requester's public key retrieved from the valid block and sends to the callback URL via secured channels.

3.3.3 Authority App

The authority app is the identity provider for the requester and in charge of requester authentication. We assume the authority app performs requester

authentication honestly as they are the IDP. However, the authentication status can be intercepted and tampered during data transmission or reused by a malicious requester or attacker. We address this issue with Json Web Token (JWT) signed by the authority app. Using JWT ensures the information transmitted in the payload of the token is not tampered and token is generated from the authentication organization [28, 29].

The JWT will be sent as part of transaction block of access request log by the requester to assert his/her identity. The PK_{Auth} will be hardcoded in the DApp to validate the JWT and send to smart contract to perform ACL check on the identity of requester against the Data Owner defined ACL and rules. The transaction will be successful added on chain only when the checks are fulfilled.

3.3.4 Requester Console

Requester console is used by the requester with his own wallet account to interact with the smart contract. The console's IP address is used as the callback URL for the enclave to establish secure session and transmit the SK_{D2}.

3.4 Enhanced Escrow Workflow

In this section, we illustrate the data escrow solution implementation steps performed by different app components in the flowcharts of 3 different phases. In the wallet and enclave initialization phase (Figure 2), the enclave is created at the application startup. The escrow wallet key pair is generated to create transactions within enclave app. In the data registration phase (Figure 3), the data owner submits the escrowed data information to escrow application and obtain the data key for data encryption. In the data access request phase (Figure 4), the requester sends the access log as a new block on Ethereum and obtains 2 shares of secret key from the authority app and the escrow enclave.

3.4.1 Wallet and Enclave Initialization

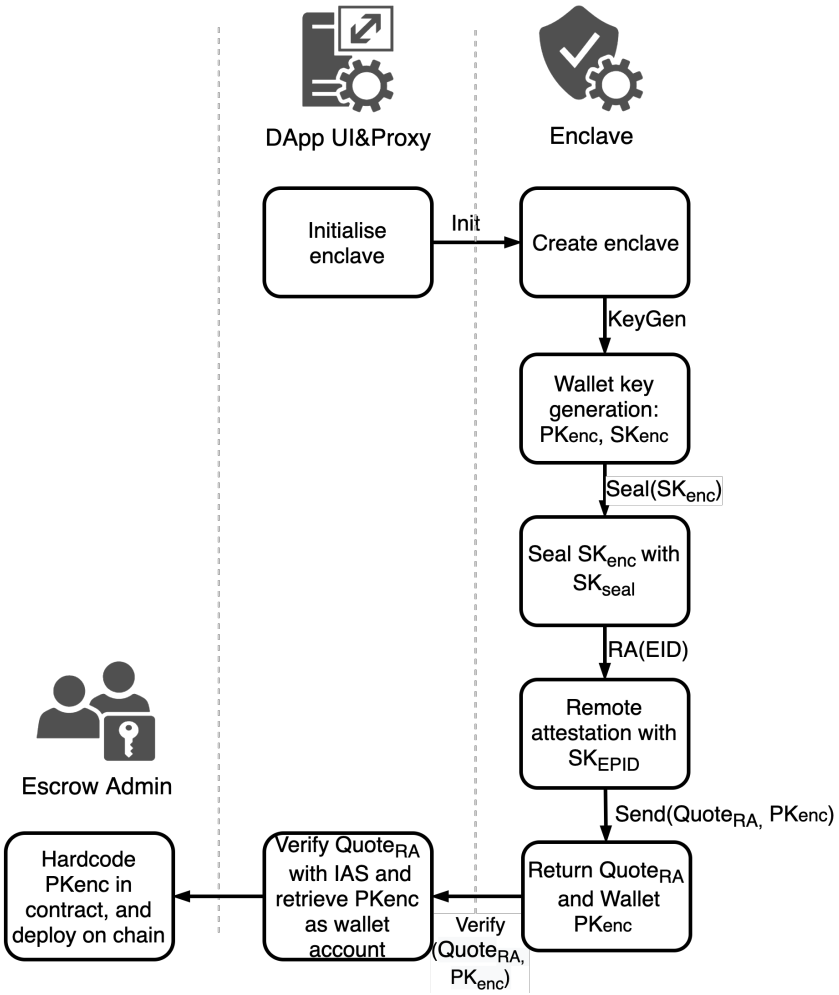


Figure 2: Wallet and Enclave Initialization workflow

When the data escrow application starts, the enclave is created and executes the trusted enclave functions. Enclave creates the escrow wallet's ECDSA key pair (PK_{enc} and SK_{enc}) in order to send transaction from enclave directly and leverage on the Ethereum protocol's security feature for authenticity and integrity. The SK_{enc} will be sealed by the enclave bond SK_{seal} which can only be unsealed inside the enclave to ensure the security of escrow wallet. The PK_{enc} will be used as the wallet account address. Enclave generates the EREPORT to attest on the program code, data, stack, heap contents, security attributes, enclave capabilities and the wallet public key. The EREPORT is sent to quote enclave which contains the SK_{EPID} to generate the $Quote_{RA}$ for remote attestation. The enclave returns the PK_{enc} and the $Quote_{RA}$ to the proxy app. Escrow admin verifies the $Quote_{RA}$ with IAS verification service and

hardcode the PK_{enc} into the smart contract and migrate the contract on Ethereum.

3.4.2 Data Registration

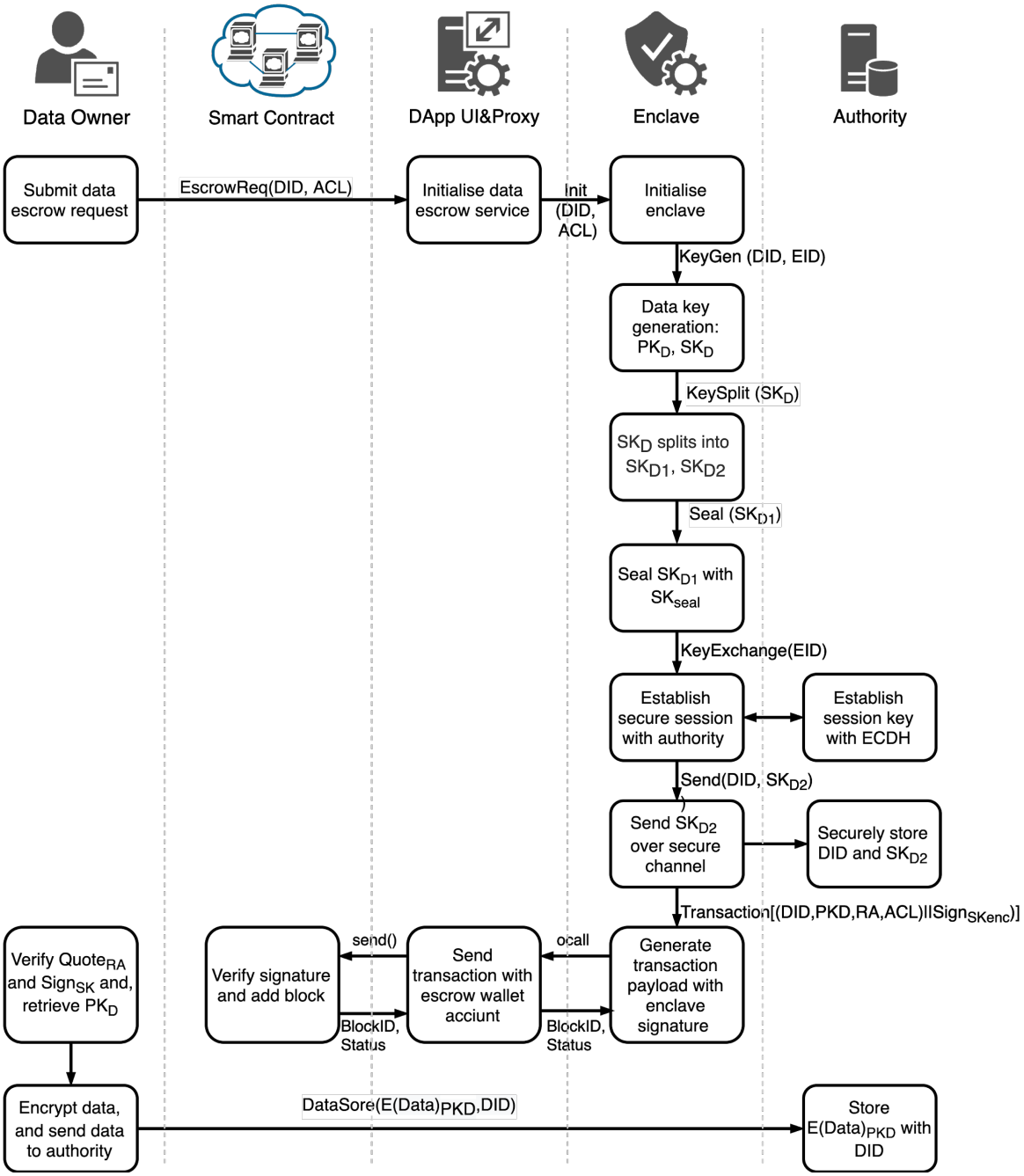


Figure 3: Data Registration Workflow

Data escrow service provides a user interface for data owner to submit data escrow request by providing DID which is the hash of the escrowed data, and authorized ACL containing a list of approved requesters and rules to grant

access. The proxy handles the data owner's request and sends to the enclave to prepare keys used to protect the data.

Enclave app performs a list of operations:

- Data key generation: enclave generates PK_D, SK_D using ECCs with a key size of 256 bit for the submitted DID.
- SK_D splits into 2 shares: the private key is split into SK_{D1} and SK_{D2}.
- Seal SK_{D1} with SK_{seal}: SK_{D1} is stored securely in the static disk space together with the DID. SK_{D1} is unsealed when a requester passed the ACL and rule check and enclave verified the access log has been added on the blockchain.
- Establish secure session with authority: SK_{D2} is distributed to authority app for storage with DID. SK_{D2} is provided to requester when the requester passed authentication and enclaved verified the access log has been added on the blockchain. In order to distribute secret securely, the enclave establishes a direct secure channel with authority using ECDH as discussed in section 3.3.2.3 to ensure the confidentiality, integrity, and authenticity.

The authority app securely stores DID and SK_{D2}. The SK_{D2} should be protected at rest with a key vault or HSM to prevent data leakage. The authority's application is out of scope of this discussion. As an untrusted component in the design, using Shamir's Secret Sharing ensures the data cannot be decrypted even if the authority is compromised.

The enclave app generates transaction payload with unsealed escrow wallet SK_{enc}. The payload should contain DID, PK_D, $Quote_{RA}$, ACL, and $Sign_{SKenc}$ of the payload. The transaction block is then sent to proxy app to initiate the smart contract transaction. In this step, if the proxy app modifies the message, the signature will be invalid hence not accepted by other nodes. Furthermore, proxy app cannot replay the payload and signature with his own SK, because the PK_{enc} is hardcoded on the contract and smart contract code is publicly verifiable and tamper proof.

When smart contract receives the transaction, it validates the $Sign_{SKenc}$ with the hard coded PK_{enc}. All nodes need to verify the block signature for validity check before accepting the block on the blockchain.

When the block for the DID registration is published on chain, data owner may perform offline check on the $Quote_{RA}$ and $Sign_{SKenc}$ validity. Upon successfully validation, data owner retrieves the PK_D for $E(Data)_{PKD}$ function and send DID and $E(Data)_{PKD}$ to authority for data storage.

3.4.3 Data Access Request

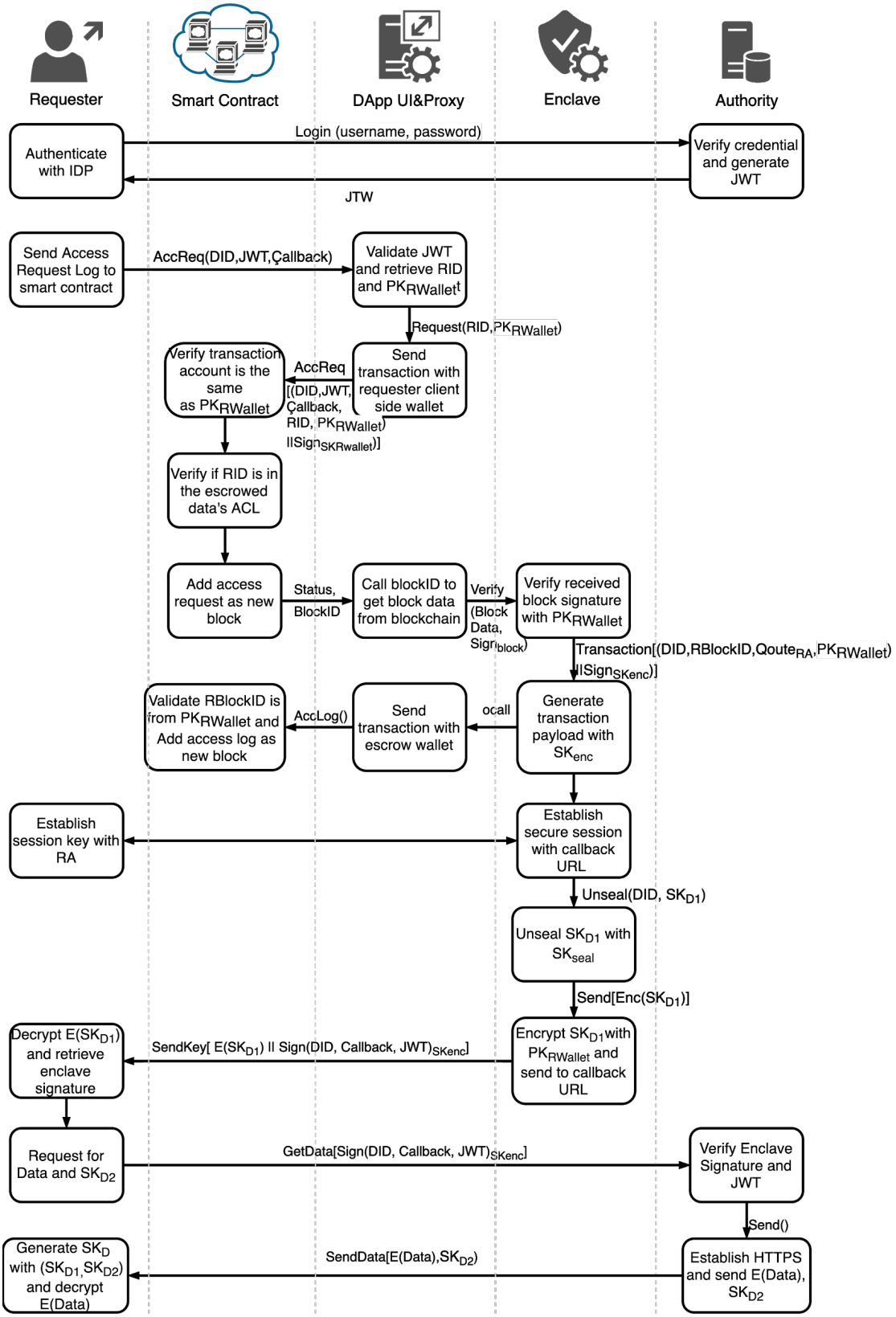


Figure 4: Data Access Request Workflow

A requester is an authenticated user from the authority organization with a defined role. The requester is only authorized to access to the escrowed data when he is authenticated with organization and fulfills the ACL and rules defined by the data owner in the smart contract. The data access request has two parts of identity and role checks: authentication with authority and publishing an access log through the escrow smart contract.

In the authentication with authority, requester login to the authority app with valid username and credentials. Upon successful authentication, the authority issues a JWT containing 3 components:

- The JWT header defines the signing algorithm used to ensure the authenticity of the token. We use the ECDSA signing algorithm.
- The JWT payload includes the RID, $PK_{RWallet}$, Token expiration time, and the authorized role.
- The JWT signature is generated using ECDSA signing algorithm with SK_{Auth} on the encoded header and payload data.

JWT allows external parties to validate the requester's identity assertion with PK_{Auth} before the token defined expiration time.

In the publishing an access log through the escrow smart contract phase, requester sends an access request to smart contract on the DApp UI with DID, JWT for identity assertion, and the callback URL. DApp performs the JWT verification and retrieves the RID and $PK_{RWallet}$ from JWT payload. The DApp leverages on the requesters wallet app extension on the browser to send the transaction to the escrow smart contract with $SK_{RWallet}$. The wallet app notifies the requester on the transaction and obtain requester's approval to pay for the transaction. When the smart contract receives the transaction, following checks are performed:

- It verifies the transaction sender wallet account is the same as $PK_{RWallet}$ to ensure the authenticity of the transaction.
- It then retrieves the block information identified by DID and obtains the ACL defined for this DID in order to verify if RID is in the defined ACL.

- After the successful verification, the escrow contract adds access request as a new block on the blockchain and return the success status and the new block ID.

The DApp passes the blockID and block data to proxy app. The proxy app relays the information into enclave for final status assertion. Enclave retrieves the block data ((DID, JWT, Callback, RID, $PK_{RWallet}$) ||$Sign_{SKRwallet}$) and performs following checks:

- Enclave verifies the block signature with the $PK_{RWallet}$ in the block data. This step cannot guarantee the block is authentic from the published block, as proxy app is able to forge a data on the same structure but not on chain. This is only a first step check to reduce the following expensive enclave computation when the data is not valid.
- If the block structure is valid, enclave generates the transaction payload (DID, BlockID, $Quote_{RA}$, $PK_{RWallet}$) and sign the payload with escrow wallet SK_{enc}. The transaction is passed to proxy app via ocall and relayed to the escrow smart contract.

The Ethereum transaction protocol validates the block received from enclave wallet is valid. The smart contract retrieves the BlockID provided in the enclave transaction to check if the BlockID is initiated from $PK_{RWallet}$. This guarantees the enclave received block data from proxy is not forged. If the check fails, the transaction is rejected. The access log verification block is added on the blockchain and allow data owner to verify in the future through DApp UI.

When the enclave receives the success status of the transaction, it performs following steps to release SK_{D1} to requester:

- Enclave establishes secure session with requester's callback URL as discussed in section 3.3.2.3.
- Enclave retrieves SK_{seal} to unseal SK_{D1} for DID as discussed in 3.3.2.3.
- Enclave encrypts $SK_{D1}$with $PK_{RWallet}$ and prepares attestation payload for requester to send to authority. The payload ($E(SK_{D1})$ || Sign(DID,

Callback, JWT)$_{SKenc}$) is sent to the requester callback URL via the secure channel.

Requester forwards the enclave's signature to the authority to retrieve the SK_{D1} and encrypted data. Authority validate the enclave's signature, and the validity of JWT token. Upon successful verification, authority app returns E(Data), and SK_{D2} via the secure channel to requester.

Requester reconstruct the SK_D using Shamir's Secret Sharing algorithm and decrypt the encrypted data.

4 Implementation

We partially implemented the proposed data escrow solution on the app components described in section 3 with various technology stacks.

4.1 Data Escrow DApp and Ethereum Test Network

Ethereum is the type of blockchain we choose to implement our data escrow solution on chain parts. The smart contract is executed in the EVM and Ethereum maintains the state of the EVM on the blockchain. Following subcomponents are required to perform the required functions:

- Smart contract

The smart contract is developed using Solidity language for the functions discussed in section 3.3.1. We declared the contract variables with hard coded PK_{enc} and data structure for data registration and access log.

```
pragma solidity >=0.4.22 <0.6.0;

contract DataEscrow {
    //define constant: enclave wallet public key hardcoded
    bytes32 constant enclaveWalletPK = "d439730b8e59401b9adb42a79f93bd52";

    // data registration data struct
    bytes32 dataID;
    struct EscrowData {
        bytes32 ownerID;
        bytes32 pkData;
        bytes32 quoteRA;
        bytes32[] authorizedList;
    }
    mapping(bytes32 =>EscrowData) dataEntry;

    //access request log data struct
    uint reqBlockID;
    struct AccessRequest {
        bytes32 requesterID;
        bytes32 dataID;
        bytes32 JWT;
        bytes32 callbackURL;
        bytes32 reqWalletPK;
        uint accessTime;
        bool isSuccess;
    }
    mapping(uint=> AccessRequest) accessLog;

   //SGX access log assertion data struct
   uint assertionBlockID;
   struct AccessAssertion {
       bytes32 dataID;
       uint reqBlockID;
       bytes32 quoteRA;
       bytes32 reqBlockHash;
   }
   mapping(uint=> AccessAssertion) enclaveVerified;
```

In solidity, event is an inheritable member of a contract. We use events to ensure the transaction logs are recorded on the blockchain. Events also works well with web3 framework to retrieve the return value of a smart contract function. We defined following events for transactions:

```
    event AddData(bytes32 indexed _dataID, bool success);
    event AddAccess (uint _reqBlockID, bool _isSuccess);
    event AttestAccessLog(uint _assertionBlockID, bool _isSuccess);
```

For the data registration function, we defined in smart contract as below:

```
function addDataEntry (bytes32 _dataID, bytes32 _ownerID, bytes32 _pkData, bytes32 _quoteRA,
bytes32[] memory _authorizedList) public returns (bool success) {
    EscrowData memory owner = EscrowData (_ownerID, _pkData, _quoteRA, _authorizedList);
    dataEntry[_dataID] = owner;
    emit AddData(_dataID, true);
    return true;
}
```

To add a data entry with test data, we added a new block on the blockchain.

```
[vm] from:0xca3...a733c
to:DataEscrow.addDataEntry(bytes32,bytes32,bytes32,bytes32,bytes32[]) 0x8c1...401f5 value:0 wei
data:0x363...00000 logs:1 hash:0xc1b...5b0e6

status                          0x1 Transaction mined and execution succeed
transaction hash                0xc1b8f320b3c5eb02f9d867a6be857a09eba0e90a52d85e1016503ca83f45b0e6
from                            0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to                              DataEscrow.addDataEntry(bytes32,bytes32,bytes32,bytes32,bytes32[]) 0x8c1ed7e19ab
                                aa9f23c476da86dc1577f1ef401f5
gas                             3000000 gas
transaction cost                174967 gas
execution cost                  143327 gas
hash                            0xc1b8f320b3c5eb02f9d867a6be857a09eba0e90a52d85e1016503ca83f45b0e6
input                           0x363...00000
decoded input                   {
                                        "bytes32 _dataID": "0xb17e770e9c51cb0bf15c686b9f34c74d58b38cb5d389c8b569
                                814f64a93a29be",
                                        "bytes32 _ownerID": "0xf87dd5abef28a45cdbc1eff60412a6fa76c6441d3a492387e
                                7c9ef3629483169",
                                        "bytes32 _pkData": "0x74690eabd1561b2ffa0e32adeba1a136cf3999265c1dd3e085
                                a186eb8a15eff4",
                                        "bytes32 _quoteRA": "0x5bd518e435e2cf89471a889f7b171867c2398394ccec6b413
                                03baa36cee7f8bb",
                                        "bytes32[] _authorizedList": [
                                                "0x63616e6469646174653100000000000000000000000000000000000000000
                                000",
                                                "0x6332000000000000000000000000000000000000000000000000000000000
                                000",
                                                "0x6333000000000000000000000000000000000000000000000000000000000
                                000"
                                        ]
                                }
decoded output                  {
                                        "0": "bool: success true"
                                }
logs                            [
                                        {
                                                "from": "0x8c1ed7e19abaa9f23c476da86dc1577f1ef401f5",
                                                "topic": "0xb17e770e9c51cb0bf15c686b9f34c74d58b38cb5d389c8b56981
                                4f64a93a29be",
                                                "event": "AddData",
                                                "args": {
                                                        "0": "0xb17e770e9c51cb0bf15c686b9f34c74d58b38cb5d389c8b5
                                69814f64a93a29be",
                                                        "1": true,
                                                        "_dataID": "0xb17e770e9c51cb0bf15c686b9f34c74d58b38cb5d3
                                89c8b569814f64a93a29be",
                                                        "success": true,
                                                        "length": 2
                                                }
                                        }
                                ]
value                           0 wei
```

When the requester submits an access request, the addAccessLog() function will be triggered in smart contract.

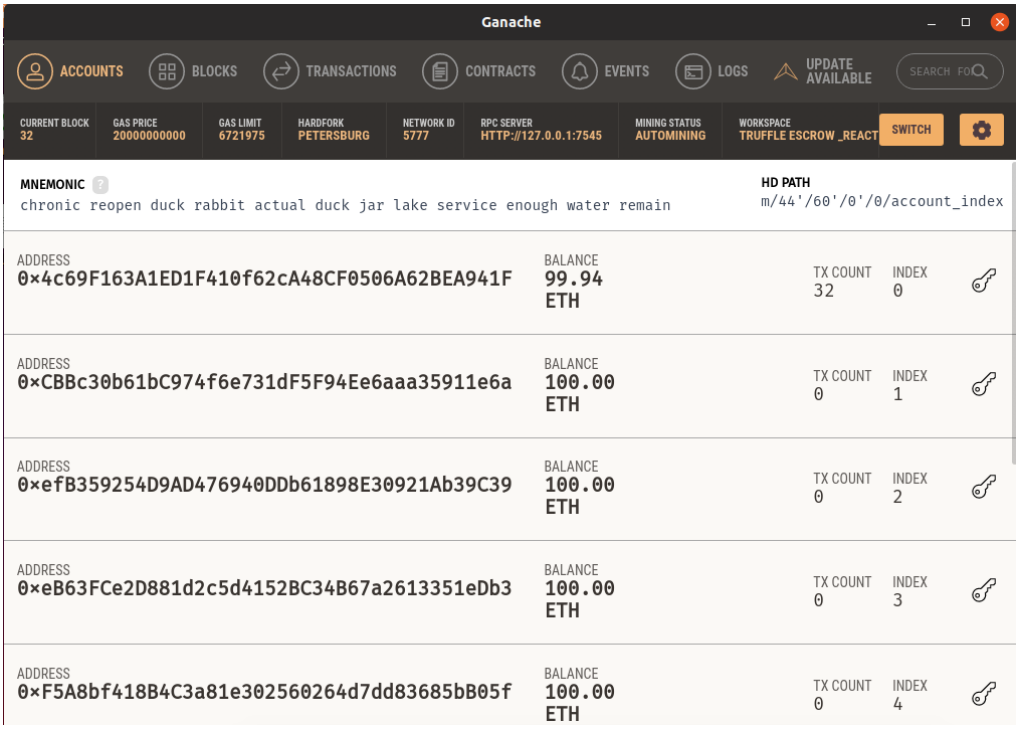
```
function addAccessRequest (bytes32 _requesterID, bytes32 _dataID, bytes32 _JWT, bytes32 _callbackURL,
    bytes32 _reqWalletPK) public returns (uint _reqBlockID, bool _isSuccess) {
    //check if the dataID is valid
    require(dataEntry[_dataID].ownerID!=0, "dataID does not exists");
    bool accessAllow = false;
    //verify requester is in ACL
    if (verifyAuthList(_dataID, _requesterID) == true){
        // add accessAllow success if the requester is authorized
        accessAllow = true;
    }
    AccessRequest memory access = AccessRequest(_requesterID, _dataID, _JWT, _callbackURL,
        _reqWalletPK, now, accessAllow);
    reqBlockID = block.number;
    accessLog[reqBlockID] = access;
    emit AddAccess(reqBlockID,accessAllow);
    return (reqBlockID,accessAllow);
}

function verifyAuthList(bytes32 _dataID, bytes32 _requesterID) public view returns (bool auth) {
 // getAuthList
    bytes32[] memory authList = dataEntry[_dataID].authorizedList;
 // verify if request is in authList
    for (uint i = 0; i < authList.length; i++) {
        bytes32 authID = authList[i];
        if (authID == _requesterID){
        return true; //mathed
         }
    }
    return false; //no match
 }
```

The enclave verifies the access request integrity and add a new assertion block on the blockchain to attest the access log is verified.

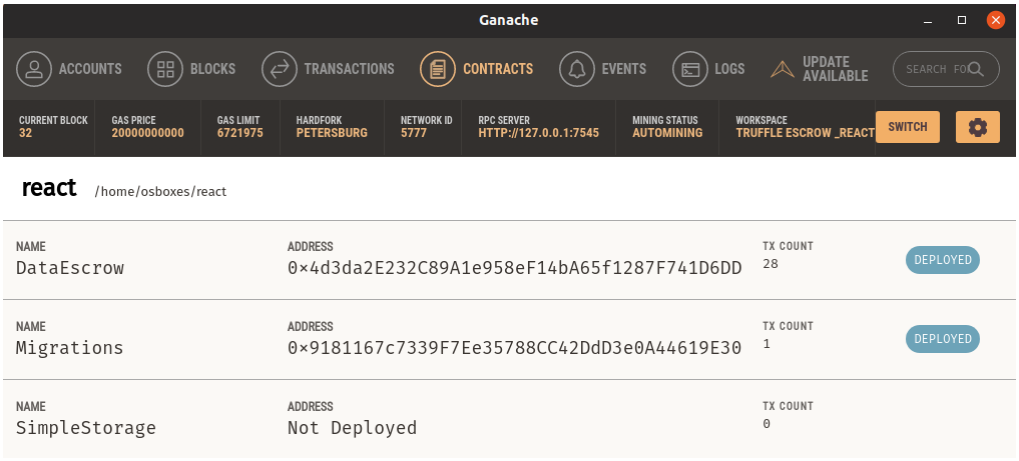
```
function accessLogAssertion (bytes32 _dataID, uint _reqBlockID, bytes32 _quoteRA,
    bytes32 _reqBlockHash) public returns (uint _assertionBlockID, bool _isSuccess) {
    //obtain blockhash with block ID and compare the hash sent by proxy
    require(blockhash(_reqBlockID)==_reqBlockHash, "Access request is not valid!");
    //Add attestation
    AccessAssertion memory assertion = AccessAssertion (_dataID, _reqBlockID, _quoteRA,
        _reqBlockHash);
    assertionBlockID = block.number;
    enclaveVerified[assertionBlockID] = assertion;
    emit AttestAccessLog (assertionBlockID,true);
    return (reqBlockID,true);
}
```

- Local test networks

The smart contract needs to be deployed in a test network for development and test before pushing into the Ethereum main network which cost ether for each transaction and once deployed the contract is permanent. For easy deployment of local test networks, we leverage on Ganache (previous TestRPC) personal blockchain for test accounts creation and an EVM environment for contract execution [30, 31]. We deployed Ganache in a virtual machine with 64 bit Ubuntu v19.04 OS. We created a workspace for data escrow project with 10 test nodes running virtually with a network ID of 5777 and RPC server HTTP://127.0.0.1:7545.



It also has a GUI for developers to view the contract deployment status, transaction history, and blocks in the test network.



- DApp

We build our DApp with Node.js for the web application UI. In order to interact with blockchain, we installed Web3.js libraries to add the infrastructure layer for the DApp to interact with the decentralized protocol stack. The contract function and input parameters need to be compiled into Application Binary Interface (ABI) for the DApp to import and make calls to the functions defined in ABI document in json format. Following code snapshot, shows the interaction with Web3 libraries to get test network

instance metadata, user's wallet account and creates the contract instance for contract interaction.

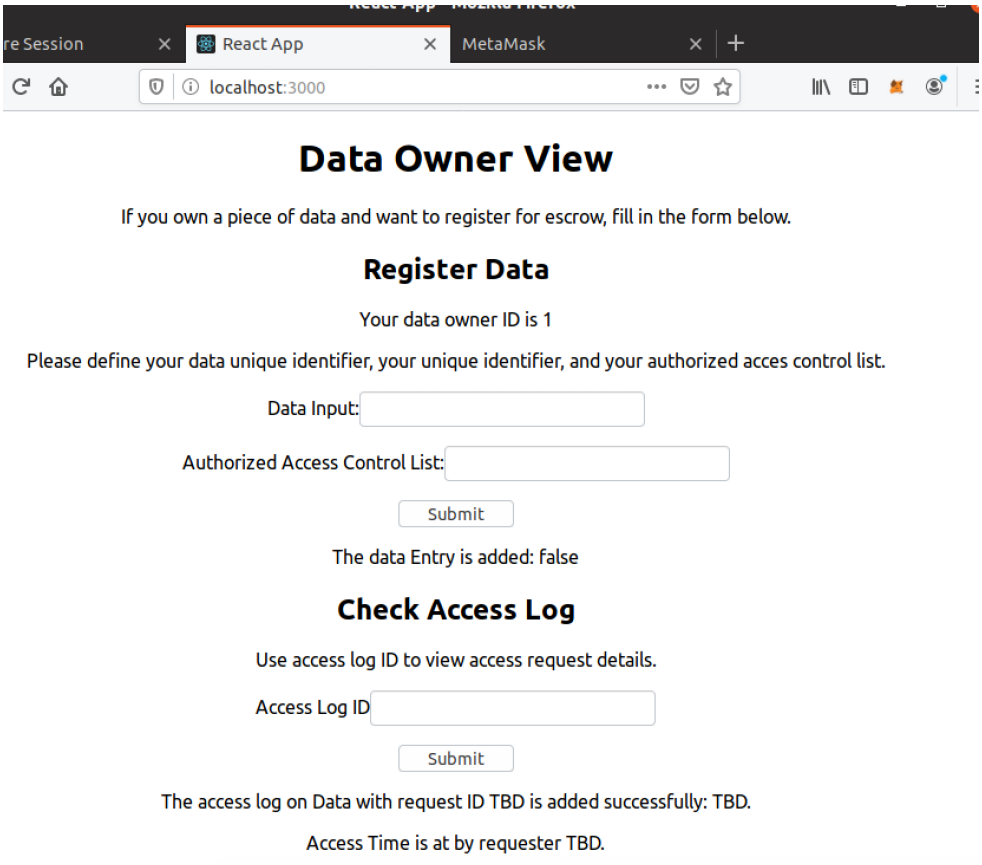
```
import React, { Component } from "react";
import ReactDOM from 'react-dom';
import DataEscrowContract from "./contracts/DataEscrow.json";
import getWeb3 from "./utils/getWeb3";
import DataEntryForm from "./DataEntryForm";
import CheckAccessLog from "./CheckAccessLog";
import AccessRequest from "./AccessRequest";
```

```
componentDidMount = async () => {
  try {
    // Get network provider and web3 instance.
    const web3 = await getWeb3();

    // Use web3 to get the user's accounts.
    const accounts = await web3.eth.getAccounts();

    // Get the contract instance.
    const networkId = await web3.eth.net.getId();
    const deployedNetwork = DataEscrowContract.networks[networkId];

    //console.log(DataEscrowContract.networks[networkId].address);

    const instance = new web3.eth.Contract(
      DataEscrowContract.abi,
      deployedNetwork && deployedNetwork.address,
    );

    // Set web3, accounts, and contract to the state, and then proceed with an
    // example of interacting with the contract's methods.
    this.setState({ web3, accounts, contract: instance }, this.runExample);
  } catch (error) {
    // Catch any errors for any of the above operations.
    alert(
      `Failed to load web3, accounts, or contract. Check console for details.`,
    );
    console.error(error);
  }
};
```

The handle request form function in the code snapshot below, shows the contract interaction is realized using a "await" keyword for the asynchronous function when invoking the ABI endpoint. This is to ensure the application wait until the promise returns the result. The transaction invocation is expecting delays in a function call as the mining process normally takes seconds to complete.

```
handleRequestForm = async (formDataID) => {
  const { accounts, contract } = this.state;
  var newAccessRequestID;
  var addAccessResponse;
  // add Data Access Request (requester ID, dataID)
  await contract.methods.addAccessRequest(this.state.requester_id, formDataID)
  .send({ from: accounts[0] }).on('receipt', function (receipt) {
    console.log(receipt.events.AddAccess.returnValues._requestID);
    newAccessRequestID = receipt.events.AddAccess.returnValues._requestID;
    addAccessResponse = receipt.events.AddAccess.returnValues._isSuccess;
  }).on('error', console.error);
  this.setState({ accessLogSuccess: addAccessResponse });
  this.setState({ newAccessRequest: newAccessRequestID });
}
```

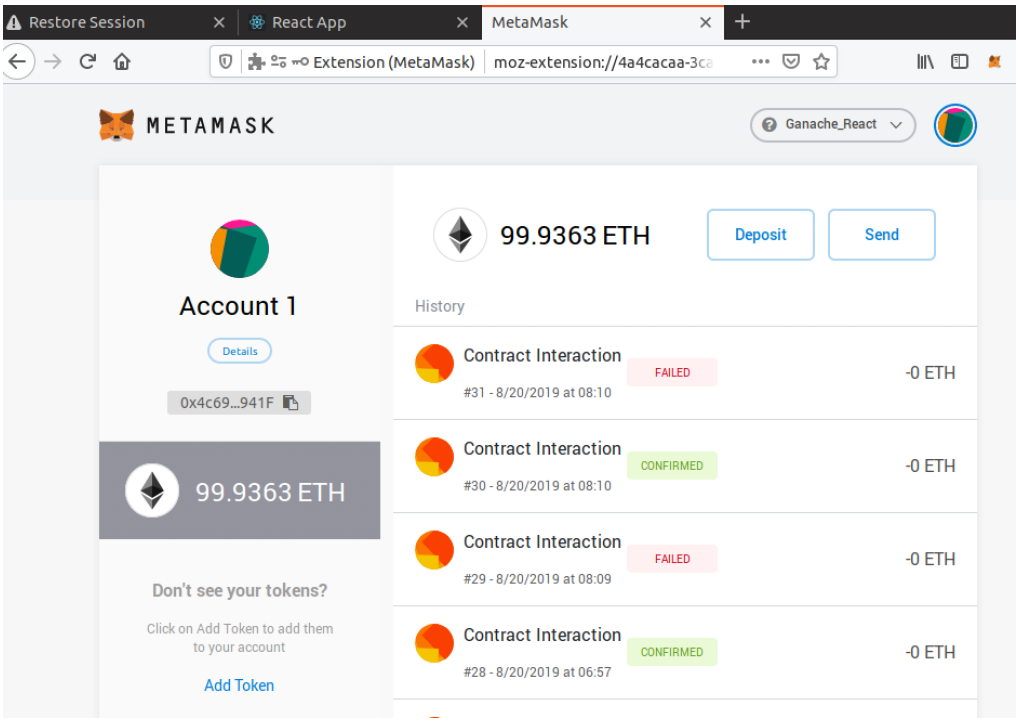
The DApp UI is demonstrated in below screenshot which has no difference form a normal web application.



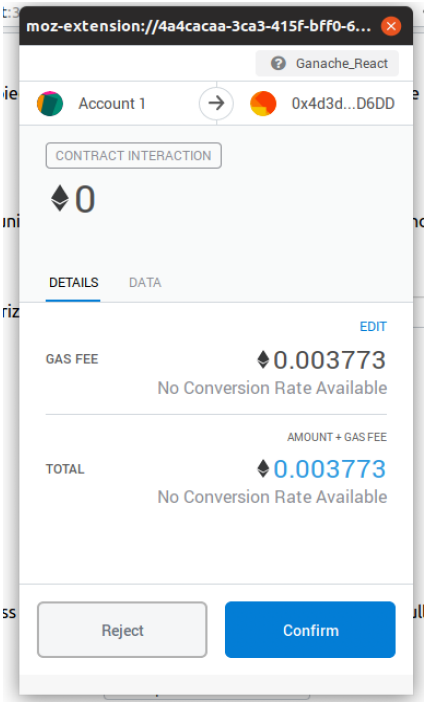
- Wallet

As discussed in section 3.3.1, the transaction can only be inserted into the blockchain from a wallet. The wallet actually manages cryptographic keys for an account which represents the digital identity of the sender of a transaction. The wallet also keeps the blockchain address. As the wallet private key will be used for transaction signature and maintain the crypto currencies in it. A

secured solution (such as vault) should be adopted by the data owner and requester to properly protect wallet private key at rest. In our implementation, we leverage on MetaMask which provide a secure identity vault and wallet functions with a UI [32]. MetaMask works as a browser add-on and can be integrated with Ganache network and injects its own web3 instance in the DApp.



When a transaction is initiated from DApp using the user's wallet account, MetaMask browser add-on pops up for user's confirmation on the transaction and the gas for this operation.



- Development, testing and deployment tools

Truffle provide a framework that integrates the tedious operations including but not limited to contract compilation, contract migration to Ethereum, Ganache network integration, and web3 instance detection [33]. Our implementation of smart contract and DApp heavily relies on Truffle framework.

- To integrate with Ganache, we need to update the truffile-config.js with the Ganache RPC network address.
- To compile a contract, we need to install truffle suite and run the compilation command to generate the ABI file.

```
osboxes@osboxes:~/DataEscrowDapp$ truffle compile

Compiling your contracts...
===========================
> Compiling ./contracts/DataEscrow.sol
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/SimpleStorage.sol
> Artifacts written to /home/osboxes/DataEscrowDapp/client/src/contracts
> Compiled successfully using:
   - solc: 0.5.8+commit.23d335f2.Emscripten.clang
```

- To deploy the contract, we need to run the migration command for deployment.

```
osboxes@osboxes:~/DataEscrowDapp$ truffle migrate

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.


Starting migrations...
======================
> Network name:    'ganache'
> Network id:      5777
> Block gas limit: 0x6691b7


1_initial_migration.js
======================

   Replacing 'Migrations'
   ----------------------
   > transaction hash:    0xcad0b2401929ccf959382fd6b191ac10fe0b88516c5c59104980084ef6985df2
   > Blocks: 0            Seconds: 0
   > contract address:    0x63E0DF8B50daef1AA02ac8C1e0f9136dFF7a0B4a
   > block number:        74
   > block timestamp:     1572856140
   > account:             0xE53736Cb429d5ad2578a27b67eaD50348a495dA8
   > balance:             99.841717088
   > gas used:            261393
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00522786 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00522786 ETH
```

```
2_deploy_contracts.js
=====================

   Replacing 'DataEscrow'
   ----------------------
   > transaction hash:    0x0cd6a46bfca296d66365b571ec54aed231ab9d185fbdf4b84b44729e48f1a60f
   > Blocks: 0            Seconds: 0
   > contract address:    0x1CEC8567971E750adA6defBeb4A64d0257147ef7
   > block number:        76
   > block timestamp:     1572856141
   > account:             0xE53736Cb429d5ad2578a27b67eaD50348a495dA8
   > balance:             99.830123968
   > gas used:            537633
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.01075266 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.01075266 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.01598052 ETH
```

4.2 Proxy App

As discussed in section 3.3.2.2, proxy app is used to relay data between DApp and enclave. In order to interact with enclave application, proxy app requires to have IntelSGX SDK, installed for the set of API, libraries and tools [25].

One of the key challenges is to develop the relay between two different technology stacks: DApp is developed with JavaScript and Solidity, while the

enclave application requires low level language such as C or C++. To address this issue, the proxy app is developed with C++ and using Crow micro web framework for DApp integration [35]. For simplicity of the implementation, we assume the DApp and proxy app is communicated locally on the same server. For future work, mutual authentication needs to be established between the DApp and proxy app server to ensure the security of the communication.

The proxy app performs majority of the application processing, as well as prepares inputs for ecalls defined in enclave edl library and processes ocall outputs from enclave. The major challenge in the implementation is to integrate and compile the enclave edl, and enclave app generated header files with the proxy app. The compilation process is tedious and prone to error with numerous dependencies in Makefile and involves manual configuration when the code gets complicated. In the later part of implementation, we found that using the right tool - Visual Studio Professional version 2015 or 2017, will make the compilation efficient as it has a special function to build enclave project with all the header files auto-generated placed in the correct folder. By importing the enclave app inside the proxy app, it will import the correct header files from enclave app as well. However, Visual Studio Professional version is only compatible with Windows OS and requires hardware support for IntelSGX driver for enclave project function [38]. We switched the development environment to a Windows machine and started over the proxy development.

With different development environment for DApp (Ubuntu) and proxy app (Windows), the integration becomes challenging as well. Both apps have a large number of dependencies to install which may not be cross platform compatible. We plan to containerize the DApp in a docker image and run the docker container on the Windows platform [36, 37].

The pseudocode for the proxy application is documented as below. For wallet and smart contract initialization phase, proxy initializes the enclave and verifies the enclave remote attestation with following functions:

- initialize_enclave() initializes enclave parameters and invokes sgx_create_enclave() to create the enclave.
- ra_verify() performs remote attestation verification including following steps: sgx_get_extended_epid_group_id() to get the public key from Intel attestation server, and sgx_ecdsa_verify() to verify the quote validity.

In the data registration phase and data access request phase, proxy is mainly working as the relay between enclave app and DApp.

4.3 Enclave App

Enclave App provides a list of functions for proxy app to invoke as ecall. The pseudocode for enclave app is defined for the 3 workflows in section 3.4 heavily leveraged on the API and libraries in IntelSGX SDK [25].

4.3.1 Wallet and Enclave Initialization

Upon enclave creation, enclave performs following functions to initialize the escrow wallet keys and generate data key.

- sgx_ecc256_create_key_pair() generates the PK_{enc} and SK_{enc} key pair on the ECC curve for the escrow wallet account.
- sgx_seal_data() uses AES-GCM algorithm to encrypt the SK_{enc} twith the enclave SK_{seal}.
- sgx_ra_init() generates the Remote Attestation $Quote_{RA}$ for enclave and the PK_{enc} with $SK_{EPID.}$
- ocall_send_ra_pk() invoke ocall in proxy app to send the $Quote_{RA}$ and PK_{enc}.

4.3.2 Data Registration

When a data owner submits a data request for escrow service, following enclave functions are performed:

- sgx_ecc256_create_key_pair() generates the PK_D and SK_D key pair on the ECC curve for data encryption.

- sss_key_split() uses Shamir secret sharing algorithm to split SK_D into SK_{D1} and SK_{D2}. However, there is no ready to use libraries in IntelSGX SDK for this algorithm. IntelSGX SDK has a list of secured versions of crypto libraries that can be used inside the enclave. In order to use external downloaded libraries, the library can either be invoked using ocall, or port the external library into enclave code compliant to enclave function rules for secure usage. Invoking the key split function using ocall indicates the SK_D will be transmitted to proxy app in cleartext. This defeats our design goal. Therefore, we will need to port the library functions into the enclave and remove unsecured functions.
- sgx_seal_data() uses AES-GCM algorithm to encrypt the SK_{D1} to the enclave with the enclave SK_{seal}.
- sgx_ra_get_keys(), as part of the remote attestation process, this function gets the negotiated key $SK_{session}$and form a secure exchange session.
- sgx_rijndael128GCM_encrypt() encrypts SK_{D2} with $SK_{session}$ using AES-GCM 128 bits.
- generate_trasaction() function is a series of self-developed functions to form the transaction block following Ethereum protocol payload. Including form_transaction() to construct the payload, keccak() to compute SHA3 hash, and gx_ecdsa_sign () from SGX SDK for block signing with SK_{enc}.
- ocall_transaction() invokes ocall in proxy app to send the transaction to DApp.

4.3.3 Data Access Request

Enclave performs following functions and exposes the ecall in edl to interact with proxy app:

- sgx_ecdsa_verify() verifies the input block signature with the $PK_{RWallet}$.
- generate_trasaction() discussed in 4.3.2 will be reused to send the access log verified assertion to smart contract for additional verification.

- sgx_ra_get_keys() establishes a secure session with requester app to for key delivery.
- sgx_unseal_ data() retrieves the SK_{seal} and encrypted SK_{D1}. It decryptes SD_{D1}
- sgx_rijndael128GCM_encrypt() encrypts SK_{D1} with $SK_{session}$ using AES-GCM 128 bits.
- ocall_send_data_key() invokes ocall in requester app to deliver the enclave assertion and $E(SK_{D1})_{SKsession}$.

5 Limitations and Future Work

5.1 Data Availability Concern

The data stored on blockchain is maintained on majority of nodes. It is very expensive to run a 51% attack to tamper the records or launch a Denial of Service (DoS) attack on the main Ethereum network. However, it is still possible. Our design does not prevent 51% attack on blockchain.

DoS attack is feasible at the network connection points such as routers, gateways, or firewall. In addition, although the proxy application in the current design cannot access or tamper the communication between the enclave and external parties, enclave is utilizing the proxy's OS capability for network access. If the proxy is malicious or compromised, it is able to drop the packets or delete the enclave to launch a DoS attack to the solution.

To provide an end-to-end secured data escrow solution, we would recommend future studies on the recommended network level protection and resource redundancy assigned for the proxy server should be hardened and secured. The server running the IntelSGX enclave should be protected physically in a secured restricted areal and behind firewall. The data escrow provider should adopt a defense in depth approach to compliment the current design.

5.2 Requester Side Data Protection

We assume the requester would perform the due diligence to ensure the decrypted data and the secret key is securely stored and not shared with

anyone else. Without an end to end solution to protect the client data handling, it is hard to enforce security policies on the requester side.

For future work, requester side SDK could be developed to securely connect to proxy's server to view the required data. Data download should be disabled.

5.3 Lack of Key Rotation for Escrowed Data

In current design, the PK_D and SK_D for the data submitted to escrow service do not have a defined cryptoperiod. As recommended in NIST.SP.800-57, the keys for a given system should implement a cryptoperiod and perform key rotation [42].

For future work, the enclave could regenerate a new key for the data entry after a defined number of access requests fulfilled or after a defined cryptoperiod, whichever is shorter. And use this new key to wrap previously encrypted data.

6 Conclusion

Escrowed data technique is a solution to balance public security/functionality with individual privacy. The combined use of blockchain and IntelSGX seems a promising solution to address the flaws in existing escrowed data design and improve the access control. We proposed detailed system design and process workflow to achieve the security properties required for a secure data escrow solution. However, the proposed solution has limitations, and we discussed the proposed future works to explore and research on to eventually deploy the solution in real-world applications.

7 Bibliography

[1] King Ables and Mark Dermot Ryan. 2010. Escrowed Data and the Digital Envelope. In Proc. 3rd International Conference on Trust and Trustworthy Computing - TRUST. 246–256.

[2] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. MedRec: Using Blockchain for Medical Data Access and Permission Management. In Proc. 2nd International Conference on Open and Big Data - OBD. 25–30.

[3] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew P. Martin. 2014. TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms. In Proc. 19th International Symposium on Formal Methods - FM. 110–126.

[4] Carl M. Cannon. 2018. The Personal Privacy vs. Public Security Dilemma. https://www.realclearpolitics.com/articles/2018/07/26/the_personal_ privacy_vs_public_security_dilemma.html, visited at 14 Aug. 2019.

[5] Natmin Pure Escrow. 2019. Escrow Services powered by Blockchain Technology. https://www.natmin.io/docs/NatminDeck.pdf, visited at 14 Aug. 2019.

[6] Ethereum. 2017. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper visited at 14 Aug. 2019.

[7] Escrow Europe. 2019. Benefit of software escrow. https://www.escroweurope. co.za/benefits-of-software-escrow/, visited at 14 Aug. 2019.

[8] Jon Evans. 2018. Personal Privacy vs. Public Security: Fight! https://techcrunch. com/2018/05/06/personal-privacy-vs-public-security-fight/, visited at 14 Aug. 2019.

[9] Navin Ramachandran James Brogan, Immanuel Baskaran. 2018. Authenticating Health Activity Data Using Distributed Ledger Technologies. 16 (2018), 257–266.

[10] Harleen Kaur, Mohd. Afshar Alam, Roshan Jameel, Ashish Kumar Mourya, and Victor Chang. 2018. A Proposed Solution and Future Direction for BlockchainBased Heterogeneous Medicare Data in Cloud Environment. J. Medical Systems 42, 8 (2018), 156:1–156:11.

[11] Harlan M. Krumholz and Jeanie Kim. 2017. Data Escrow and Clinical Trial Transparency. Annals of internal medicine 166 12 (2017), 893–894.

[12] Julia Lane and Claudia Schur. 2009. Balancing Access to Health Data and Privacy: A Review of the Issues and Approaches for the Future. https://www.ratswd.de/ download/RatSWD_WP_2009/RatSWD_WP_113.pdf, visited at 14 Aug. 2019.

[13] Li Li, Naipeng Dong, Jun Pang, Jun Sun, Guangdong Bai, Yang Liu, and Jin Song Dong. 2017. A Verification Framework for Stateful Security Protocols. In Proc. 19th International Conference on Formal Engineering Methods - ICFEM. 262–280.

[14] Iron Mountain. 2019. Registry Data Escrow Service FAQ. https: //www.ironmountain.com/resources/data-sheets-and-brochures/r/registrydata-escrow-faq, visited at 14 Aug. 2019.

[15] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https: //bitcoin.org/bitcoin.pdf, visited at 14 Aug. 2019.

[16] TK. Sharma, 2017. How Blockchain can be Used in Escrow & How it Works?https://www.blockchain-council.org/bitcoin/blockchain-can-usedescrow-works/, visited at 14 Aug. 2019.

[17] J. Leah, 2003, The Create Debate: Security vs. Privacy. https://www.ecmag.com/section/your-business/great-debate-security-vs-privacy, visited at 14 Aug. 2019.

[18] E. Lim, Data Escrow with Blockchain Blockchain Platform, Solidity Code Design and Implementation, National University CP5102 MComp InfoSec Project Report, Apr 2019.

[19] F. Zhang, E. Cecchetti, K. Croman, E. Shi, "Town crier: An authenticated data feed for smart contracts", Proc. ACM SIGSAC Conf. Comput. Commun. Security, pp. 270-282, 2016.

[20] K. Lau, Data Escrow System using Smart Contract, National University CP5102 MComp InfoSec Project Report, Apr 2019.

[21] A gentle introduction to smart contracts, [Online]. Available: https://bitsonblocks.net/2016/02/01/gentle-introduction-smart-contracts/

[22] Signing and Verifying Ethereum Signatures, Yos Riady, 16 Nov 2018, https://yos.io/2018/11/16/ethereum-signatures/

[23] I. Anati, S. Gueron, S. P. Johnson and V. R. Scarlata, "Innovative Instructions for Attestation and Sealing," 2013. [Online]. Available: https://software.intel.com/enus/articles/innovative-technology-for-cpubased-attestation-and-sealing.

[24] SGX Secure Enclaves in Practice: Security and Crypto Review, JP Aumasson, Luis Merino — Kudelski Security July 29, 2016, https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review-wp.pdf

[25] Intel(R) Software Guard Extensions Developer Reference for Linux* OS https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Reference_Linux_2.1.3_Open_Source.pdf

[26] Shamir, Adi (1979), "How to share a secret", Communications of the ACM, 22 (11): 612–613, doi:10.1145/359168.359176.

[27] T, Sascha, Ethereum Wallet in a Trusted Execution Environment / Secure Enclave, Jun 2018, https://medium.com/weeves-world/ethereum-wallet-in-a-trusted-execution-environment-secure-enclave-b200b4df9f5f

[28] JSON Web Tokens, retrieved from: https://auth0.com/docs/jwt

[29]Introduction to JSON Web Tokens, retrieved from: https://jwt.io/introduction/

[30]Ethereum Development Walkthrough, retrieved from: https://hackernoon.com/ethereum-development-walkthrough-part-2-truffle-ganache-geth-and-mist-8d6320e12269

[31]ETHEREUM OVERVIEW, retrieved from: https://www.trufflesuite.com/tutorials/ethereum-overview

[32]MetaMask, retrieved from: https://metamask.io/

[33]Truffle overview, retrieved from:: https://www.trufflesuite.com/docs/truffle/overview

[34]web3.js - Ethereum JavaScript API, retrieved from: https://web3js.readthedocs.io/en/v1.2.2/index.html

[35]Crow framework, retrieved from: https://github.com/ipkn/crow

[36] Tutorial: Create C++ cross-platform projects in Visual Studio, retrieved from:https://docs.microsoft.com/en-us/cpp/build/get-started-linux-cmake?view=vs-2019

[37]Running Truffle in a Docker container, retrieved from: https://www.jitsejan.com/truffle-in-docker.html

[38]Intel® Software Guard Extensions (Intel® SGX) Web-Based Training, retrieved from: https://software.intel.com/en-us/articles/intel-sgx-web-based-training

[39]Simon Johnson, Vincent Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel software guard extensions: EPID provisioning and attestation services. ser. Intel Corporation (2016).

[40]Hiie Vill. 2017, SGX attestation process, https://courses.cs.ut.ee/MTAT.07.022/2017_spring/uploads/Main/hiie-report-s16--17.pdf.

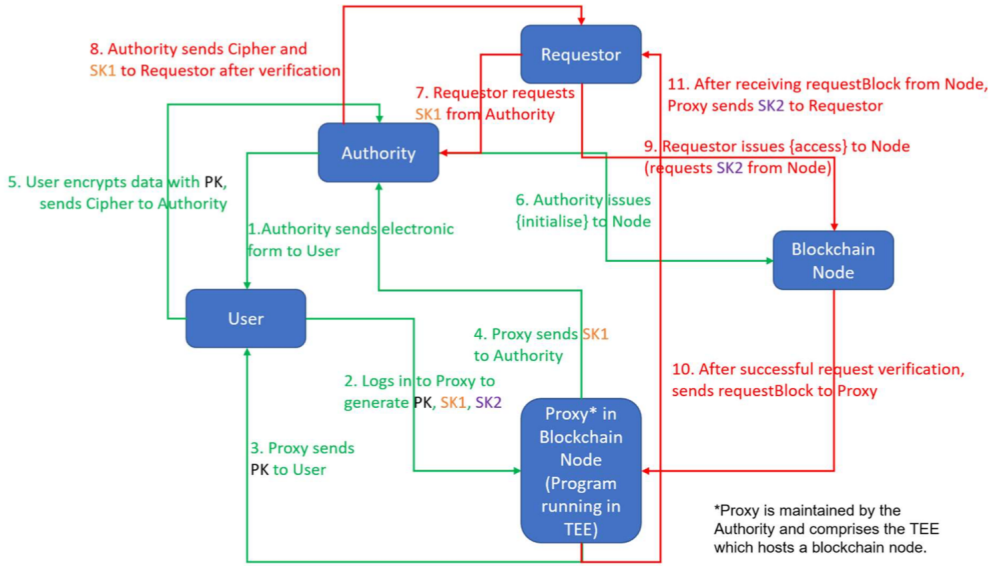
[41]Enterprise Security: Putting the TPM to Work, Trusted Computing Group, 2008, https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Applications-Whitepaper.pdf

[42]Barker, E.: NIST Special Publication 800–57 Part 1, Revision 4. http://dx.doi.org/10.6028/NIST.Spp.800--57pt1r4

8 Appendices

Appendix 1 Initial Design Flowchart

The following is the graphical representation of the proposed workflow:



Source: screenshot from [20]