DATA ESCROW WITH TRUSTED COMPUTING

LU XI
(M.Comp, NUS)

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

DEPARTMENT OF SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2020

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.

LU XI

Lu Xi
08 April 2020

Table of Contents

SUMMARY

Critical industries (e.g. healthecare and government) are using data escrow service to balance data privacy and sharing request across multiple transaction parties. Data escrow service requires data owner to trust other parties and share sensitive data with others. Therefore, a trustworthy escrow solution that protects the confidentiality and integrity of data will be enssential to escrowed data in use, in transit and at rest. Many reaserchers proposed Trusted Platform Module (TPM) as data escrow hardware solution. However, we identify TPM is vulnerable to platform reset attack and cannot act as a single point of trust. Firstly, we present an authenticated data escrow solution which combines blockchain and Intel SGX as a Trusted Computing Base (TCB). Blockchain acts the front-end of access control validating integrity and authenticity of data escrow record, while Intel SGX verifies secure code execution and provides confidentiality and integrity of data escrow processes as a trusted hardware-based platform.

Sencondly, we describe the design architecture with security model, and also report the implementation of our solution features on Intel SGX to prevent single-entity manipulation and secure network communication. The solution enables encrypted data at rest and in use to be secured within Intel SGX enclave, while data in transit to be protected under TLS/HTTPS communication channel.

Lastly, we discuss the limitations and propose countermeasures and enhancement for future work so that the data escrow solution can be deployed to meet real-world requirements.

LIST OF FIGURES

1. INTRODUCTION

1.1.Motivation

Information technology continues improving the quality and productivity of our lives, there is a growing trend of security concern for implementing technologies in our societies. Safeguarding data is prioritized by both public and private sectors. In the past decades, classic methods to keep data secure and private, e.g. hash and encryption, have been raised to meet CIA triad of information security – balanced protection of confidentiality, integrity and availability. In this information era, sharing data with clients and third parties is inevitable, but may violate data confidentiality and integrity, so that bring huge loss to organizations. The method "data escrow" has been introduced to balance individual data security and sharing request across entities [1]. In data escrow model, data will be temporarily held in escrow, which is the custody of a third party, for a short period. During the period, data can be accessed by other authority. None of transaction parties "owns" the data. As a result, data escrow can increase confidence on protecting individual privacy as well as the confidentiality and integrity of data [2]. In previous literatures, data escrow is discussed to be applied in many critical industries [3, 4, 5, 6]. For examples,

- Government agency may use data escrow for background check and crime investigation;
- Hospital may use data escrow to share patient health records under emergency with authorization from patient;

- Real Estate stakeholder may use data escrow to verify property ownership and register property with government;
- IT company may use data escrow to store business critical source code and grant the access to authorized users.

They tend to have a consensus that whether data escrow is applicable depends on the trust on the escrow from all transaction parties, while leveraging cutting-edge technology, like highly secure hardware or verifiable audit log, can increase the public trust. An escrow which has higher public visibility, less manual manipulation, and secure network transmission framework may lead to increased confidence on trust for data owner. This project will demonstrate a practical Trusted Computing Base (TCB) solution to improve trust level of data escrow.

1.2.Trusted computing for data escrow

1.2.1. Existing solution

The implementation of trusted computing in a secured hardware component has been introduced for a decade. One solution called Trusted Platform Module (TPM) is developed as hardware security base for data escrow by ensuring the escrowed data to be encrypted and not to be opened. TPM is a microcontroller to achieve the goal of trusted computing system by securing secret storage and reporting platform state [7]. TPM contains many platform configuration registers (PCR) whose value can be extended and reset to initial state. PCR is a special TPM primitive. When keys are locked to a certain PCR value, i.e. state, the usage command of keys can only execute under this specific PCR value [8]. TPM can seal

data in the envelope, and also can perform remote attestation by signing data with the TPM key on a specific PCR value. As a result, TPM plays a role as a hardware digital envelope mechanism [1]. The secure cryptographic function calls provided by TPM also can run secure software solutions on TPM platform to form a root of trust.

Blockchain technology as a decentralized trust model is suitable for data escrow by validating the integrity and authenticity of each publicly recorded transaction. Once a blockchain transaction is verified by the majority of nodes within the blockchain network, the transaction log is publicly appended and very hard to be changed. Therefore, transaction log can be considered as audit log for data escrow request. Additionally, cryptography on blockchain smart contract is able to verify the identity of transaction sender, so that it can enforce access control policies. As a result, blockchain can act as front end of data escrow by controlling access policies and providing evidence with access log. The transparency and resilience of blockchain allows trust distribution across the blockchain network and ensures the integrity of smart contract [9].

1.2.2. Limitations on existing solution

Firstly, TPM or blockchain cannot be the single point of trust in TCB system for data escrow. If TPM is the single trusted entity for data access control, additional evidences are needed to prove that the relevant TPM state i.e. PRC value is not tampered; data access request is not modified or deleted; and data requestor did not collude with TPM system. Blockchain standing alone will not protect any key storage to keep data private, thus, it

is not a single trusted escrow entity. Therefore, the previous studies suggested to combine TPM and blockchain into a trusted computing base by storing data securely in TPM hardware and verify data access control on blockchain smart contract.

Secondly, TPM as hardware trusted computing component is vulnerable to platform reset attack. An attacker can reset and forge platform configuration registers (PCR) whether processes are started statically or dynamically. The vulnerability is identified in specification design flaws and implementation defect [10]. When TPM receives a hardware reset, it will believe the system has been rebooted, and then return to uninitialized state by resetting PCR value. An attacker can use malicious device to drive hardware reset to re-initialize TPM in an untrusted configuration, so that break the entire trusted boot process [11].

In our data escrow model, when data requestor applies to access data, TPM is initialized and records the “open” data request. After granting access, TPM will update to “seal” state to record the requestor has requested the data access. At the same time, if TPM is compromised by platform reset attack, then TPM is reset and re-initialized. When data requestor submits the second access request, TPM will start over again with “open” data request instead of “seal” state. Therefore, to protect the trust chain in data escrow from platform reset attack on TPM, we switch to another more reliable trusted computing hardware component – Intel SGX.

1.2.3. Intel SGX and Blockchain as TCB

Based on the previous project and studies, we introduce a new Trusted Computing Base (TCB) system for data escrow – the combination of Intel SGX and blockchain as root of trust. Instead of TPM, we adapt Intel's Software Guard Extensions (Intel SGX) as hardware security solution. Intel SGX is able to provide code and data protection based on an isolated hardware-based platform with memory encryption. It is able to verify code is securely executable and protect the confidentiality and integrity of processes. Different from TPM, Intel SGX is a built-in feature in Intel CPU chip rather than a microcontroller chip. More details will be introduced in Section 2. Background, 2.2. Intel SGX. In our solution, Intel SGX generates encryption keys and enforce secure execution, while blockchain provides high public visibility for transaction log. Meanwhile, data flow is protected under TLS trusted communication framework and the end-to-end data escrow process minimizes human interaction. As a result, this secure solution can increase the trust of data owner.

1.3.Contributions

- We study the existing work on data escrow using TPM protocol and introduce a new hybrid TCB.
- We report and develop an end-to-end implementation of data escrow on Intel SGX hardware backend with blockchain smart contract as front end. This TCB solution addresses security barriers for authenticated data flow across trusted and untrusted entities.

- We present techniques on secure code execution and size management on Intel SGX applications. With the security nature of Intel SGX, trusted function libraries need to be securely imported into Intel SGX. The enclave page cache size only allocates less than 100MB to be used by applications under current SGX specification. We demonstrate practical implementation to address these technical challenges.

2. BACKGROUND

Section 2 provides fundamental background on four security properties of main technology in the project. They are blockchain smart contract, Intel SGX, dual key encryption and TLS/HTTPS. The first two consist of an integrated TCB solution; while the later are two key technical challenges that this project needs to overcome for implementation on the TCB.

2.1.Blockchain and Smart Contract

This project uses smart contract on Ethereum blockchain for data escrow front end system. Ethereum is one of the largest public distributed blockchain networks. Smart contract is application code to self-execute specific functions within Ethereum blockchain when certain conditions are met [12]. The decentralized nature of blockchain ensures high availability and fault tolerance of data on chain, as these data trails are distributed on each node. Additionally, all information stored on
the blockchain is publicly accessible thus publicly auditable. In our TCB design, Ethereum blockchain provides a proof of audit trails for data

request activities. The smart contract will perform the transaction and generate transaction log for each data escrow request. As the previous block is immutable, the transaction log is written via appending new block only. User can invoke a smart contract by sending blockchain transaction to a 160-bit contract address, which is an identifier of the smart contract. After the recipient smart contract accepts a new transaction, everyone on the mining network can execute the transaction payload as input at current stage of the blockchain [13].

2.2.Intel SGX

Intel Software Guard Extensions (SGX) was introduced by Intel in 2015 for trusted computing functionality. It is a set of built-in security instruction codes in Intel Central Procession Units (CPUs). Intel SGX enables highly secured code execution based on a set of protected memory region – enclaves, which is isolated from operating system and other applications and does not allow any external process (e.g. write, read) outside itself. An Intel SGX application allows more than one enclave to exist inside. Intel SGX application contains two logical components - untrusted part and trusted part. The untrusted part is used to start enclave and interact with the external parties; while the trusted part, i.e. enclave, accesses the secret and executes trusted code. They communicate via “ecall” and “ocall”. To be specific, data within enclave only can be accessed and processed by calling enclave functions. However, creating enclave and calling trusted function can be done by untrusted part [14]. As a result, Intel SGX meets the security requirement of running application

on a remote third-party user side and ensures the integrity and confidentiality in the following respectively [16]:

- Remote user cannot alter the application in execution;
- Application can process encrypted data which is inaccessible to remote operating system but only can be decrypted inside Intel SGX.

SGX internal features. Here is a brief introduction of security features which are used in enclave. We leverage these features to achieve the integrity and confidentiality of sensitive data in our solution design. The implementation details are in Section 4. Implementation.

- Sealed Storage

  When enclave is closed, data provisioned in enclave is lost. To save secret data outside enclave and ensure secret provision for future use, sealing functions are enable to store the encrypted data and a unique key can be retrieved for the enclave. The enclave needs to retrieve its unique seal key using EGETKEY instruction [17]. Enclave application use the seal key to encrypt data to platform (sealing) and to decrypt the data from platform (unsealing). Sealed storage protects the integrity and confidentiality of secret data [14].

- Remote Attestation

  This feature builds a trusted channel between enclave and third-party client. Enclave is able to prove to a remote third-party its identity and it is running the given program securely and has not been tampered with. Remote attestation consists of the application's enclave and two

specific enclaves, Intel-provided Quoting Enclave (QE) to provide quoting service and Provisioning Enclave (PvE) to conduct provisioning process [17]. QE is able to sign the report with Provisioning Key and transform it into a quote. Intel SGX remote attestation and provisioning protocol follows Enhanced Privacy ID (EPID) which is an anonymous signature schema. Remote attestation guarantees the authenticity of platform and verifies the integrity of enclave [14].

SGX external features. Here is a brief introduction of communication functionalities of Intel SGX. These features enable the secure communication between Intel SGX enclave and the relay Proxy App in our design, so that data can be further transmitted between other parties e.g. smart contract DApp and Authority App. The implementation details are in Section 4. Implementation.

- Enclave Call (ecall)

  Ecall refers to a trusted function call which can enter enclave [17]. Invocation is from application to enclave. When an application wants to pass parameters or pointers to its shared memory, it invokes an ecall as a pre-defined function inside the enclave [15].

- Outside Call (ocall)

  Ocall refer to an untrusted function call which can call unsecured functions from enclave [17]. When code within enclave wants to call external functions in untrusted memory, it performs an ocall to a pre-

defined function inside the application. Typical services require ocall is requesting OS syscalls [15].

Intel SGX developers can specify the ecall and ocall interfaces in Enclave Definition Language (EDL) file.

2.3.Dual Key Generation and Encryption

Using Intel SGX as hardware TCB has another security shield for cryptography key management. Secret keys (private keys) are the core of secure data escrow for sensitive information, thus the TCB needs to make sure the secret keys will not be compromised during end-to-end data escrow process. Asymmetric key cryptography uses pairs of keys i.e. public keys and private keys, to protect a secret message from unauthorized access by encryption and decryption. In our project, as the sensitive data needs to be escrowed across multiple parties including the Authority, data requester (user) and data owner, we choose to adopt dual key encryption to ensure the data confidentiality by eliminating single-entity decryption. Therefore, secret sharing cryptography scheme is best suitable for our data being maintained privately and stored securely. Additionally, the key must be stored geographic separately, so that data cannot be decrypted when a single system is compromised. Within Intel SGX enclave application, one private key splits into two shares of secret key for separate distribution. Dual secret key encryption allows two shares of key to be stored separately, while one share of key is securely maintained by Intel SGX through enclave's seal feature, the other one is maintained by the third-party Authority. For example, when a user request

to access sensitive data, firstly the user needs to obtain one share of key from the Authority; secondly, the user needs to submit the access log to the blockchain smart contract, and then obtain the other share of key from Intel SGX after transaction verification. A completed decryption requires both shares of the key.

Shamir's Secret Sharing (SSS) algorithm. SSS is a form of secret sharing across multiple entities. It illustrated that a secret can be divided into multiples parts (shares) and each share is held by one individual [18]. A minimum number of shares (threshold) is needed for reconstructing the secret. We implement SSS for dual key encryption. As Intel SGX does not support SSS as a native secure crypto function, importing its external library is a technical challenge of our project. This will be discussed later in details in Section 4. Implementation.

2.4.TLS and HTTPS

As the proxy is considered as untrusted component, it is essential to verify the integrity of sensitive data during the relay between Intel SGX enclave and other untrusted parties. HTTPS protocol also relies on asymmetric cryptography. It consists of unsecure HTTP protocol over the transport layer security (TLS) to establish a secure communication channel over untrusted network environment. TLS layer is able to handle secure communication between SGX enclave and other components, while HTTP layer handles interaction with web servers [19]. TLS provides the confidentiality and integrity for data in transit.

3. ARCHITECTURE AND SECURITY DESIGN

3.1.Basic architecture design of the solution

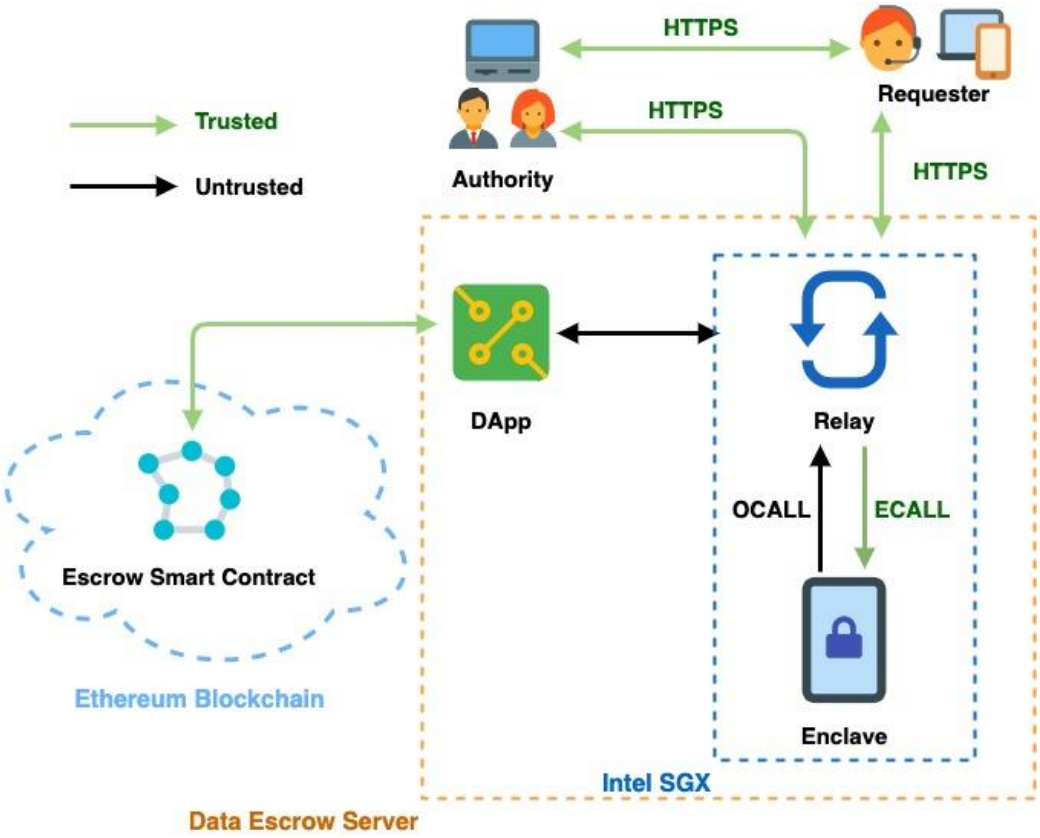


Figure 1 Architecture Diagram

Figure 1 demonstrates the basic architecture design of our project. Within the trusted data escrow processes, data at rest and data in use are secured by Intel SGX enclave, while data in transit is secured by TLS/HTTPS communication framework. The untrusted data escrow process is vulnerable to Man-in-the-Middle (MitM) attack.

Escrow Smart Contract. The smart contract on Ethereum will record data escrow request and verify data access request by access control list (ACL). The request is sent by adding a new transaction on the blockchain. This part is implemented by the previous project.

Data Escrow Server contains the following components.

- **DApp**. Decentralized web application provides a blockchain front-end for data escrow server. The blockchain transaction is trusted and authenticated by a signed wallet account, thus, communication between DApp and smart contract is trusted. This is implemented by the previous project.
- **Relay**. A backend proxy application to relay data between enclave and other entities, providing interface functions for enclave. It sends input from other entities to enclave applications via trusted ecall and receives transaction output from enclave to other entities via untrusted ocall. It also handles bidirectional network communication on behalf of enclave.
- **Enclave**. The key trusted component in Escrow server to allow secure function execution. It will address these functions in protected memory: 1) perform remote attestation to verify the integrity of enclave; 2) generate key pairs and secure key storage; 3) verify access request on blockchain transaction; 4) establish secure communication sessions.

Authority. The identity provider (IDP) for the data requester. It is responsible for data requester authentication. Authority will sign Json Web Token (JWT) to ensure the authentication status of data requester is not tampered during transmission.

Requester. The data requester who owns their wallet account on smart contract and shares their IP address for secure data transmission.

3.2.Security model

Escrow Smart Contract. We assume it will behaves honestly because its public visibility.

Authority and Requester. We assume Authority is trusted. However, the communication between Authority and Requester should be protected under TLS/HTTPS, which is out of scope of our project.

Enclave and Relay. We assume Enclave behaves honestly and only Enclave knows the private key for all enclave-generated key pairs. Relay is untrusted. However, even Relay is compromised, sensitive data still can be protected by: 1) using private key generated and stored in Enclave to verify data is not modified by Relay; 2) using remote attestation to prove integrity of Enclave; 3) using dual key encryption in Enclave to split private key into 2 shares and distribute one share via secure TLS/HTTPS communication while seal the other share within enclave, to ensure the integrity and confidentiality of private key.

Blockchain communication. Communication between DApp and Relay is not trusted but can be protected from: 1) generating key pairs for wallet account in enclave; 2) signing the transaction by Elliptic Curve Digital Signature Algorithm (ECDSA) in enclave; 3) verifying signature of each transaction in Escrow Smart Contract. We assume communication between DApp and Escrow Smart Contract is trusted because each transaction is verified by a signed wallet account and ACL rules which ensure the authenticity and integrity.

Intel SGX communication. Relay is untrusted but cannot alter enclave's behaviors. As explained in Section 2.2, ocall is not trusted while ecall is trusted. Refer to the previous statement in Enclave and Relay, even Relay may be an adversary, sensitive data can still be protected. Relay securely transmits private key under TLS/HTTPS protection to other entities i.e. Requester and Authority on behalf of Enclave.

3.3.Technical challenges

Leverage libraries outside enclave. Enclave functions support C/C++ programming languages but are restricted by many limitations. Only some C/C++ runtime trusted libraries can be directly use in internal enclave trusted functions. User need to specifically import other trusted libraries to make sure they align with the same rules as original trusted functions. For example, in our project, Shamir's Secret Sharing (SSS) algorithm library need to be carefully partitioned into enclave as trusted library.

Errors in using enclave functions such as memory corruption. As SGX will not automatically secure all libraries and functions, even functions work in external environment, they may not be able to run in secured enclave environment. There are many bugs and errors when compiling enclave functions, such as segmentation error due to array pointing to wrong memory address in C/C++ programming language. Additionally, C/C++ is also exploitable to OS memory attack. Therefore, to increase programming efficiency, we use additional Rust SGX SDK to write memory-safe secure enclave with Rust programming language [20]. Rust SGX SDK is recommended by Intel as a high proficiency SGX SDK.

Minimize the size of TCB. To minimize code size for Intel SGX, we need to identify the resource and code to be protected and carefully import trusted libraries. BIOS reserves only 128MB physical memory size of CPU for enabling Intel SGX while only 93MB of them can be used for applications [21]. Therefore, it is difficult to manage code size of functions and libraries in Intel SGX to support all requirements for Data Escrow Server. We have make use of libraries in Rust SGX SDK and carefully tune functions in use to minimize the code size.

4. ESCROWED DATA FLOW

We present Figure 2 as a preliminary escrowed data flow through Intel SGX. Escrowed data is under secure network transmission between trusted and untrusted components, with key pair generation, storage and usage. For simplicity, Relay and Enclave are combined into one part to represent Intel SGX while Relay only passes data.

Data Owner will initialize data escrow service by submitting escrow request with DataID (hash of escrowed data) and ACL (a list of approved requesters and access rules). Relay will accept the escrow request and then Enclave will perform a set of key generation, split, sealed storage and transmission. To be noted, with SSS, even if Authority is compromised, escrowed data cannot be decrypted by only one share of SK_D (private key for escrowed data).

Once this data escrow is registered on blockchain, Data Owner will also check $Quote_{RA}$ (remote attestation quote generated by Enclave) and

securely send escrowed data encrypted by PK_D (public key for escrowed data) and DataID to Authority. This process is out of this project scope.

Data Access Requester is authenticated user. They may be from Authority's organization. When they request escrowed data, smart contract will perform a set of verification on their identity. This is implemented by the previous project. Upon successful verification, Enclave will receive and sign PKrequester (public key for data access requester's wallet account), new BlockID and DataID, and then send back to smart contract for verification. Once smart contract successfully verifies this BlockID is from Requester, Enclave starts to give another share of SK_D. Requester also needs to ask for the first share of SK_D and encrypted data from Authority. This is out of this project scope.

Important denotations in escrowed data flow are explained as below.

PK_D public, SK_D private key: for escrowed data submitted by data owner;

PKenc public key, *SKenc* private key: for Enclave's wallet account on Ethereum blockchain;

PKrequester public key, *SKrequester* private key: for data access requester's wallet account on Ethereum blockchain.

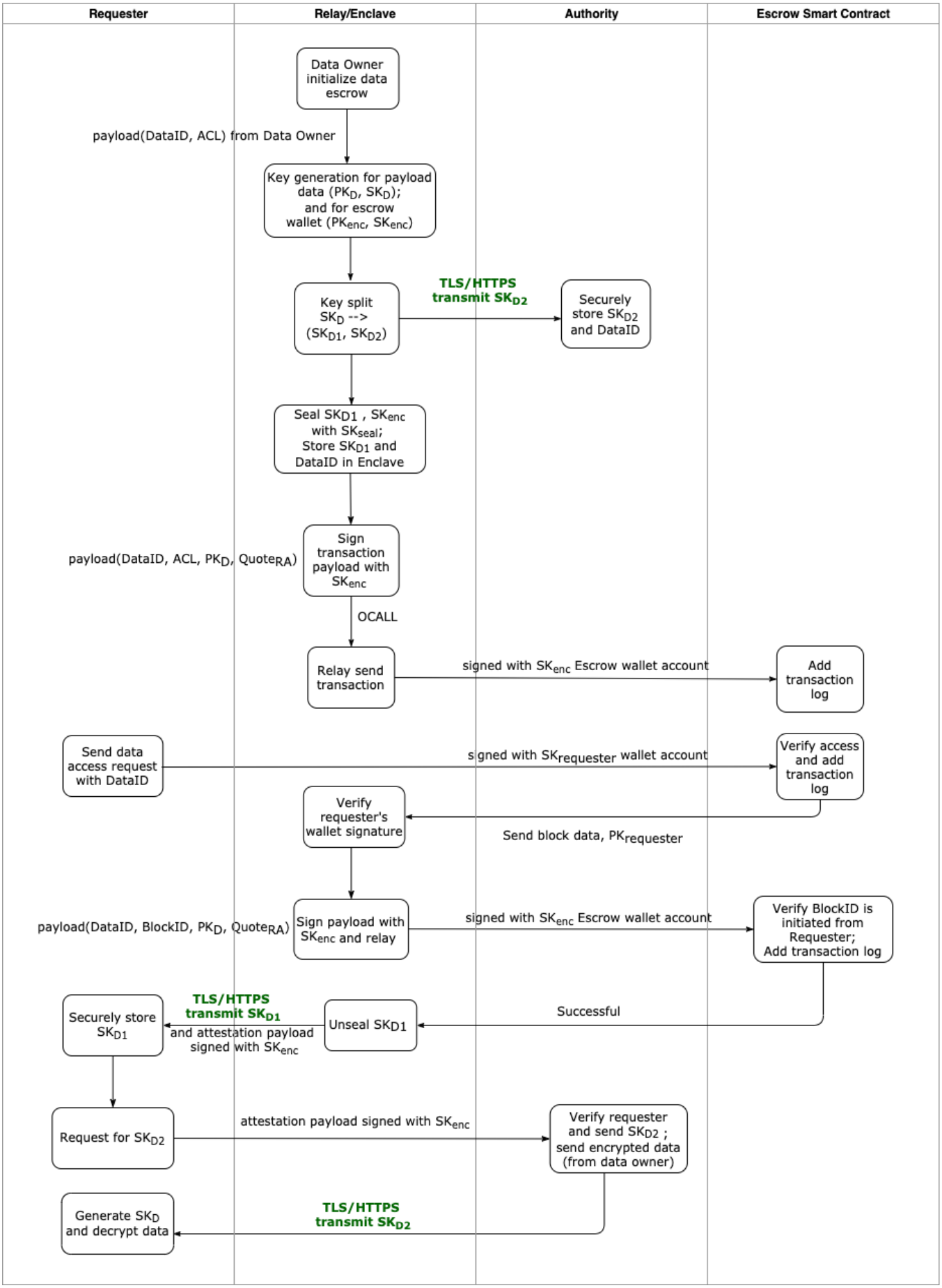


Figure 2 Escrowed Data Flow

5. IMPLEMENTATION

The majority of implementation of this project is for Intel SGX part corresponding to Enclave and Relay in Data Escrow Server. According to Intel SGX developer reference, the software stack contains SGX SDK and Platform Software (PSW) need to be configured as suggested test environments. We also implement with the help of Rust SGX SDK as explained in the previous section 3.3. Technical challenges.

5.1. Dual key generation in Intel SGX

As discussed earlier, in case of any single point of trust, we adopt dual key encryption in data escrow security model. To split the private key into two shares i.e. SK_{D1} and SK_{D2}, we implement a Shamir Secret Sharing (SSS) algorithm and its mathematics [23] that works within the enclave.

First, we need to create the secret share by instantiating the *SecretData*. We instantiate a new Secret with *with_secret* function. The private key is passed in as a string, and the threshold is the number of keys it takes to reconstruct the secret.

```
pub fn with_secret(secret: &str, threshold: u8) -> SecretData {
    let mut coefficients: Vec<Vec<u8>> = vec![];
    println!("In SecretData");

    let rand_container = vec![0u8; (threshold - 1) as usize];

    for c in secret.as_bytes() {

        thread_rng().fill_bytes(&mut rand_container);
        let mut coef: Vec<u8> = vec![*c];
        for r in rand_container.iter() {
            coef.push(*r);
        }
        coefficients.push(coef);
    }
    SecretData {
        secret_data: Some(secret.to_string()),
        coefficients,
    }}
```

Once the secret is initiated, we could call the *get_share* method to obtain two shares *sk_d1* and *sk_d2*.

```
pub fn get_share(&self, id: u8) -> Result<Vec<u8>, ShamirError> {
        if id == 0 {
            return Err(ShamirError::InvalidShareCount);
        }
        let mut share_bytes: Vec<u8> = vec![];
        let coefficients = self.coefficients.clone();
        for coefficient in coefficients {
            let b = (SecretData::accumulate_share_bytes(id,
coefficient))?;
            share_bytes.push(b);
        }

        share_bytes.insert(0, id);
        Ok(share_bytes)
    }

     fn accumulate_share_bytes(id: u8, coefficient_bytes: Vec<u8>)
-> Result<u8, ShamirError> {
        if id == 0 {
            return Err(ShamirError::InvalidShareCount);
        }
        let mut accumulator: u8 = 0;

        let mut x_i: u8 = 1;

        for c in coefficient_bytes {
            accumulator = SecretData::gf256_add(accumulator,
SecretData::gf256_mul(c, x_i));
            x_i = SecretData::gf256_mul(x_i, id);
        }

        Ok(accumulator)
    }
```

Enclave will then call the following functions in the main function.

```
    let secret_data = SecretData::with_secret(private_key, 2); //
denotes two keys are required to reconstruct the secret
    let sk_d1 = secret_data.get_share(1).unwrap(); // getting first
key
    let sk_d2 = secret_data.get_share(2).unwrap(); // getting second
key
```

Second, we need to reconstruct the shared secret (private key) from the two shares. The secrets are parsed as a vector to the *recover_secret* method. The method will validate the shares and attempt the

reconstructing of the share. Within the reconstructing phase, there is a method called *full_lagrange* which constructs the Lagrange basis polynomial. Code implementation is in Appendix.

```
pub fn recover_secret(threshold: u8, shares: Vec<Vec<u8>>) ->
Option<String> {
        if threshold as usize > shares.len() {
            println!("Number of shares is below the threshold");
            return None;
        }
        let mut xs: Vec<u8> = vec![];

        for share in shares.iter() {
            if xs.contains(&share[0]) {
                println!("Multiple shares with the same first
byte");
                return None;
            }

            if share.len() != shares[0].len() {
                println!("Shares have different lengths");
                return None;
            }

            xs.push(share[0].to_owned());
        }
        let mut mycoefficients: Vec<String> = vec![];
        let mut mysecretdata: Vec<u8> = vec![];
        let rounds = shares[0].len() - 1;

        for byte_to_use in 0..rounds {
            let mut fxs: Vec<u8> = vec![];
            for share in shares.clone() {
                fxs.push(share[1..][byte_to_use]);
            }

            match SecretData::full_lagrange(&xs, &fxs) {
                None => return None,
                Some(resulting_poly) => {

mycoefficients.push(String::from_utf8_lossy(&resulting_poly[..]).to_
string());
                    mysecretdata.push(resulting_poly[0]);
                }
            }
        }

        match String::from_utf8(mysecretdata) {
            Ok(s) => Some(s),
            Err(e) => {
                println!("{:?}", e);
                None
            }
        }}
```

5.2. Data transmission between trusted and untrusted components

Regarding to TLS/HTTPS secure data transmission, which plays the most important role in data escrow security model, we present the implementation on a TLS Handshake process. The handshake protocol is built based on a modified version of the *rustls* library [22]. An important point to note about Rust is the use of unsafe function. The Rust handbook has adequately covered the subject of why unsafe functions are necessary. In our implementation, as static analysis is difficult and underlying computer hardware is inherently unsafe, declaring *unsafe* functions relaxes the compiler checks and allows low-level programming with Intel SGX enclave.

We have a TLS Server and a TLS Client to perform a handshake. For example, when Enclave initialize to transmit SK_{D2} to Authority, Enclave is TLS Client while Authority is TLS Server. Similarly, when Enclave transmits SK_{D1} to Requester, Requester is regarded as server.

TLS Client has the following structure.

```
struct TlsClient {
    enclave_id: sgx_enclave_id_t,
    socket: TcpStream,
    closing: bool,
    tlsclient_id: usize,
}
```

On the TLSClient, we declared the following trusted functions (ECALL).

```
public size_t tls_client_new(int fd, [in, string]char* hostname,
[in, string] char* cert);
public int tls_client_read(size_t session_id, [out, size=cnt] char*
buf, int cnt);
public int tls_client_write(size_t session_id, [in, size=cnt] char*
buf, int cnt);
public int tls_client_wants_read(size_t session_id);
public int tls_client_wants_write(size_t session_id);
public void tls_client_close(size_t session_id);
```

Whenever we set up a new connection, the client first initiates itself, which calls the following trusted function within the enclave.

```
pub extern "C" fn tls_client_new(fd: c_int, hostname: * const
c_char, cert: * const c_char) -> usize {
    let certfile = unsafe { CStr::from_ptr(cert).to_str() };
    if certfile.is_err() {
        return 0xFFFF_FFFF_FFFF_FFFF;
    }
    let config = make_config(certfile.unwrap());
    let name = unsafe { CStr::from_ptr(hostname).to_str() };
    let name = match name {
        Ok(n) => n,
        Err(_) => {
            return 0xFFFF_FFFF_FFFF_FFFF;
        }
    };
    let p: *mut TlsClient =
Box::into_raw(Box::new(TlsClient::new(fd, name, config)));
    match Sessions::new_session(p) {
        Some(s) => s,
        None => 0xFFFF_FFFF_FFFF_FFFF,
    }
}
```

The trusted function retrieves *certfile* and *hostname* and instantiates a *name*. Then, it allocates a memory on the heap by using *Box* function for the TLSClient. Once allocated successfully, the client attempts to initiate a new session.

```
    fn new_session(svr_ptr : *mut TlsClient) -> Option<usize> {
        match GLOBAL_CONTEXTS.write() {
            Ok(mut gctxts) => {
                let curr_id = GLOBAL_CONTEXT_COUNT.fetch_add(1,
Ordering::SeqCst);
                gctxts.insert(curr_id, AtomicPtr::new(svr_ptr));
                Some(curr_id)
            },
            Err(x) => {
                println!("Locking global context SgxRwLock failed!
{:?}", x);
                None
            },
        }
    }
```

The instantiating of the *new_session* from here onwards is completed with a *rustls* library, which performs the following step to establish a master secret.

```
pub fn new(randoms: &SessionRandoms,
           hashalg: &'static ring::digest::Algorithm,
           pms: &[u8])
           -> SessionSecrets {
    let mut ret = SessionSecrets {
        randoms: randoms.clone(),
        hash: hashalg,
        master_secret: [0u8; 48],
    };

    let randoms = join_randoms(&ret.randoms.client,
&ret.randoms.server);
    prf::prf(&mut ret.master_secret,
             ret.hash,
             pms,
             b"master secret",
             &randoms);
    ret
}
```

The *master_secret* generated is based on random values generated during the pre-master phase. The *master_secret*, which is 48-bytes in length will be used by both the client and server to symmetrically encrypt the data. When a client wants to send data, they will parse the data as a buffer of const *c_char* type. Code implementation is in Appendix.

Once completed, the client will close the connection session using the following method.

```
fn close(&self) {

    let retval = unsafe {
        tls_client_close(self.enclave_id, self.tlsclient_id)
    };

    if retval != sgx_status_t::SGX_SUCCESS {
        println!("[-] ECALL Enclave [tls_client_close] Failed {}!",
retval);
    }
}
```

TLS Server has the following structure. This code binds together a TCP listening socket and an outstanding connection.

```
struct TlsServer {
  server: TcpListener,
    enclave_id: sgx_enclave_id_t,
    cert: CString,
    key: CString,
    mode: ServerMode,
    connections: HashMap<mio::Token, Connection>,
    next_id: usize,
}
```

First, we need to initialize a new TLS server in enclave. TLSServer's *main.rs* function initializes a new enclave.

```
fn init_enclave() -> SgxResult<SgxEnclave> {
    let mut launch_token: sgx_launch_token_t = [0; 1024];
    let mut launch_token_updated: i32 = 0;
    let mut misc_attr = sgx_misc_attribute_t {secs_attr:
sgx_attributes_t { flags:0, xfrm:0}, misc_select:0};
    SgxEnclave::create(ENCLAVE_FILE,
                       debug,
                       &mut launch_token,
                       &mut launch_token_updated,
                       &mut misc_attr)
}
```

The server sets up a few configurations i.e. certificate, key and a monitoring service for TCPStream. We use poll to monitor a large number of event types, until one of them is ready for READ or WRITE.

```
let mut poll = mio::Poll::new().unwrap();
poll.register(&listener,
              LISTENER,
              mio::Ready::readable(),
              mio::PollOpt::level()).unwrap();
```

Relay App initializes a new TLSServer in enclave by calling:

```
let mut tlsserv = TlsServer::new(enclave.geteid(), listener,
ServerMode::Echo, cert, key);
```

Within the enclave, the TLSServer will be initiated with a new session created for the configuration.

```
fn new(fd: c_int, cfg: Arc<rustls::ServerConfig>) -> TlsServer {
    TlsServer {
        socket: TcpStream::new(fd).unwrap(),
        tls_session: rustls::ServerSession::new(&cfg)
    }
}
```

Second, the poll engine will pick up the event and accept the connection whenever there is a new connection. Code implementation is in Appendix. The server will establish a connection with the client. A connection consists of the following structure.

```
impl Connection {
 fn new(enclave_id: sgx_enclave_id_t,
           socket: TcpStream,
           token: mio::Token,
           mode: ServerMode,
           tlsserver_id: usize)
           -> Connection {
        let back = open_back(&mode);
        Connection {
            enclave_id: enclave_id,
            socket: socket,
            token: token,
            closing: false,
            mode: mode,
            tlsserver_id: tlsserver_id,
            back: back,
            sent_http_response: false,
        }
    }
  fn read_tls(&self, buf: &mut [u8]) -> isize {
    // omitted for brevity reasons
  }
  fn write_tls(&self, buf: &[u8]) -> isize {
    // omitted for brevity reasons
  }
  fn tls_close(&self) {
    tls_server_close(self.enclave_id, self.tlsserver_id);
  }
}
```

If the client is sending data, the server will attempt to read the data from Relay App *main.rs* function. The TLS server will read the data sent within the enclave. As the TLSServer is run by Authority, Authority should check for the TLS Request and store DataID and SK_{D2} within the enclave.

```
pub extern "C" fn tls_server_read(session_id: size_t, buf: * mut
c_char, cnt: c_int) -> c_int {
    if let Some(session_ptr) = Sessions::get_session(session_id) {
        let session = unsafe { &mut *(session_ptr) };
        if buf.is_null() || cnt == 0 {
            // just read_tls
            session.do_read()
        } else {
            if !rsgx_raw_is_outside_enclave(buf as * const u8, cnt
as usize) {
                return -1;
            }
            // read plain buffer
            let mut plaintext = Vec::new();
            let mut result = session.read(&mut plaintext);

            // process the retrieval of DID and sk_d2 and store them
            store_data_from_datareg(&plaintext);

            if result == -1 {
                return result;
            }
            if cnt < result {
                result = cnt;
            }
            rsgx_sfence();
            let raw_buf = unsafe { slice::from_raw_parts_mut(buf as
* mut u8, result as usize) };
            raw_buf.copy_from_slice(plaintext.as_slice());
            result
        }
    } else { -1 }
}
```

A successful handshake shows as below.

```
root@cd6c23a03efe:~/sgx/tlsserver/bin# ./app
[+] Init Enclave Successful 45805826211842!
[+] Tls client established in enclave
[+] TlsServer new "end.fullchain" "end.rsa"
[+] TlsServer new success!
```

6. FUTURE WORK

Side-channel Attack. Some papers explored Intel SGX is vulnerable to side-channel attacks. An adversary may gather processing statistics (e.g. power use, information on pages accessed) from CPU and analyze characteristics of an executed application [24, 25, 26]. For example, interface-based side-channel attack can imply the information of enclave input data [24]; controlled channel attack may allow untrusted OS to extract documents which may infer information in enclave [25]; control flow trace of enclave may be revealed by branch shadowing attack [26]. However, performing these attacks to exploit Intel SGX is challenging in reality. Side-channel attack is not a specific threat to Intel SGX architecture but the enclave developers need to prevent side-channel attack based on Intel's security objectives with Intel side-channel detection tools [27]. However, TLS communication between enclave and other systems can also become side-channel attack surface because of the exploitable vulnerabilities in SSL/TLS libraries, including OpenSSL and mbedTLS [28]. The attack leverages specific errors on encrypted SSL/TLS packets, different control flows of decryption code may reveal chosen-ciphertext oracles. Although our implementation uses unpopular Rustls library, countermeasures like patching vulnerabilities in SSL/TLS libraries are need to prevent control-flow side-channel attack.

Intel SGX Clock. In addition to the generation of the key pair i.e. PKD and SKD, key rotation is also important to decrease the attack risk. A new key pair can be reinitialized after a certain time period. Our project still

lacks of an Intel SGX clock. Intel SGX SDK offers a function sgx_get_trusted_time which allow enclave to measure valid session period via an offset clock time. In the future implementation, a trusted clock function with timestamp can be added to set a valid interval for certain functions. For example, if the total time spent exceeds the pre-defined round-trip time of one data escrow request service, this service needs to be cancelled. A clock also can check TLS certification expiration so that increase the trust of TLS connection sessions.

7. CONCLUSION

We have introduced a trusted computing solution – Intel SGX and blockchain, to facilitate highly secure data escrow services. The blockchain smart contract offers access control and front-end service while Intel SGX plays as a trusted hardware backend, so that the solution is able to increase the trust from data owner by providing high public visibility, preventing single-entity manipulation and securing network communication. We have presented the project design, security model, data escrow flow and implementations on Intel SGX. Our solution demonsdtrates the enficiency and sustainability to overcome technical challenges on implementation. We also have discussed future work with possible countermeasures and improvement on limitations of our project. We believe this solution offers a practical and secure method to address trust concerns for data escrow services in real-world.

Acknowledgements

My thanks to the previous contributor of data escrow project. This project is the continuous work on the top of an existing project – Data Escrow with Trusted Computing and Blockchain, 2019 from Du Xue. The “previous project” in the report refers to Du Xue’s work.

References

[1] K. Ables, M. D. Ryan. Escrowed Data and the Digital Envelope. In Proc. 3rd International Conference on Trust and Trustworthy Computing – TRUST, 2010. Pages 246–256.

[2] D. M. Schaeffer, P. C. Olson. Securing Confidence with Data Escrow. International Journal of Management & Information Systems. Vol 22, Number 2, Dec 2018.

[3] D. E. Denning, W. E. Baugh Jr. Key Escrow Encryption Policies and Technologies. Vol 41, Issue 1, 1996.

[4] H. M. Krumholz, J. Kim. Data Escrow and Clinical Trial Transparency. Ann Intel Med, Vol 166, Issue 12, Jun 2017.

[5] N. Kirit, P. Sarkar. EscrowChain: Leveraging Ethereum Blockchain as Escrow in Real Estate. International Journal of Innovative Research in Computer and Communication Engineering, Vol 5, Issue 10, Oct 2017.

[6] Software Escrow for Dummies, Iron Mountain Edition. Published by John Wiley & Sons, Inc. 2012.

[7] Replacing Vulnerable Software with Secure Hardware the Trusted Platform Module (TPM) and How to Use It in the Enterprise. Trusted

Computing Group. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Overview.pdf, 2018.

[8] S. Delaune, S. Kremer, M. D. Ryan, G. Steel. Formal Analysis Of Protocols Based On TPM State Register. IEEE 24th Computer Security Foundations Symposium, June 2011.

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

[10] S. Han, W. Shin, JH. Park, HC. Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. 27th USENIX Security Symposium, Aug 2018.

[11] E. R. Sparks. A Security Assessment of Trusted Platform Modules Computer Science Technical Report. TR2007-597, June 2007.

[12] A. Ramachandran, M. Kantarcioglu. Using Blockchain And Smart Contracts For Secure Data Provenance Management. arXiv:1709.10000, Sep 2017.

[13] L. Luu, DH. Chu, H. Olickel, P. Saxena, A. Hobor. Making Smart Contracts Smarter . CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, October 2016. Pages 254-269.

[14] A. Adamski. Overview of Intel SGX - Part 1, SGX Internals. https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html, 2018.

[15] A. Adamski. Overview of Intel SGX - Part 2, SGX Externals. https://blog.quarkslab.com/overview-of-intel-sgx-part-2-sgx-externals.html, 2018.

[16] JP, Aumasson, L., Merino. SGX Secure Enclaves in Practice: Security and Crypto Review. Kudelski Security, July 2016.

[17] Intel(R) Software Guard Extensions SDK Developer Reference for Linux* OS. Developer Reference. Intel Corporation. https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf, 2016.

[18] A. Shamir. How To Share A Secret. Communications of the ACM. https://doi.org/10.1145/359168.359176, Nov 1979.

[19] F. Zhang, E. Cecchetti, K. Croman, A. Juels, E. Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. The 23rd ACM Conference on Computer and Communications Security, Oct 2016.

[20] Rust SGX SDK. https://github.com/apache/incubator-teaclave-sgx-sdk#rust-sgx-sdk

[21] T.D. Ngoc, B. Bui, S.B. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, D. Hagimont. Everything You Should Know About Intel SGX Performance on Virtualized Systems. Proc. ACM Meas. Anal. Comput. Syst. 3, 1, Article 05, March 2019.

[22] Mesalock-Linux. Rustls Library. https://github.com/mesalock-linux/rustls, 2019.

[23] E. Rafaloff. Shamir's Secret Sharing Scheme. https://ericrafaloff.com/shamirs-secret-sharing-scheme/, 2018.

[24] J. Wang, Y, Cheng, Q. Li, Y. Jiang. Interface-Based Side Channel Attack Against Intel SGX. Cryptography and Security, Oct 2018.

[25] Y. Xu, W. Cui, M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. Security and Privacy (SP). 2015 IEEE Symposium on, pages 640–656, 2015.

[26] S. Lee, M.W. Shih, P. Gera, T. Kim, H. Kim, M Peinado. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. 26th USENIX Security Symposium. USENIX Security, pages 16–18, 2017.

[27] S. Johnson. Intel® SGX and Side-Channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, Feb 2018.

[28] Y. Xiao, M. Li, S. Chen, Y. Zhang. Stacco: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves, 2017.

Appendices

1. More Implementation Code Examples

Function *full_lagrange* which constructs the Lagrange basis polynomial:

```
fn full_lagrange(xs: &[u8], fxs: &[u8]) -> Option<Vec<u8>> {
        let mut returned_coefficients: Vec<u8> = vec![];
        let len = fxs.len();
        for i in 0..len {
            let mut this_polynomial: Vec<u8> = vec![1];

            for j in 0..len {
                if i == j {
                    continue;
                }

                let denominator = SecretData::gf256_sub(xs[i],
xs[j]);
                let first_term =
SecretData::gf256_checked_div(xs[j], denominator);
                let second_term = SecretData::gf256_checked_div(1,
denominator);
                match (first_term, second_term) {
                    (Some(a), Some(b)) => {
                        let this_term = vec![a, b];
                        this_polynomial =

SecretData::multiply_polynomials(&this_polynomial, &this_term);
                    }
                    (_, _) => return None,
                };
            }
            if fxs.len() + 1 >= i {
                this_polynomial =
SecretData::multiply_polynomials(&this_polynomial, &[fxs[i]])
            }
            returned_coefficients =
                SecretData::add_polynomials(&returned_coefficients,
&this_polynomial);
        }
        Some(returned_coefficients)
    }
```

TLS Client initializes itself:

```
fn new(enclave_id: sgx_enclave_id_t, sock: TcpStream, hostname:
&str, cert: &str) -> Option<TlsClient> {

      println!("[+] TlsClient new {} {}", hostname, cert);

      let mut tlsclient_id: usize = 0xFFFF_FFFF_FFFF_FFFF;
      let c_host = CString::new(hostname.to_string()).unwrap();
      let c_cert = CString::new(cert.to_string()).unwrap();

      let retval = unsafe {
          tls_client_new(enclave_id,
```

```
                            &mut tlsclient_id,
                            sock.as_raw_fd(),
                            c_host.as_ptr() as *const c_char,
                            c_cert.as_ptr() as *const c_char)
        };

        if retval != sgx_status_t::SGX_SUCCESS {
            println!("[-] ECALL Enclave [tls_client_new] Failed {}!",
retval);
            return Option::None;
        }

        if tlsclient_id == 0xFFFF_FFFF_FFFF_FFFF {
            println!("[-] New enclave tlsclient error");
            return Option::None;
        }

        Option::Some(
            TlsClient {
            enclave_id: enclave_id,
            socket: sock,
            closing: false,
            tlsclient_id: tlsclient_id,
        })
    }
```

Client parse the data as a buffer of *const c_char* type:

```
pub extern "C" fn tls_server_write(session_id: usize, buf: * const
c_char, cnt: c_int)  -> c_int {
    if let Some(session_ptr) = Sessions::get_session(session_id) {
        let session = unsafe { &mut *(session_ptr) };

        // no buffer, just write_tls.
        if buf.is_null() || cnt == 0 {
            session.do_write();
            return 0;
        }

        rsgx_lfence();
        // cache buffer, waitting for next write_tls
        let cnt = cnt as usize;
        let plaintext = unsafe { slice::from_raw_parts(buf as * mut
u8, cnt) };
        let result = session.write(plaintext);

        result
    } else { -1 }
}
```

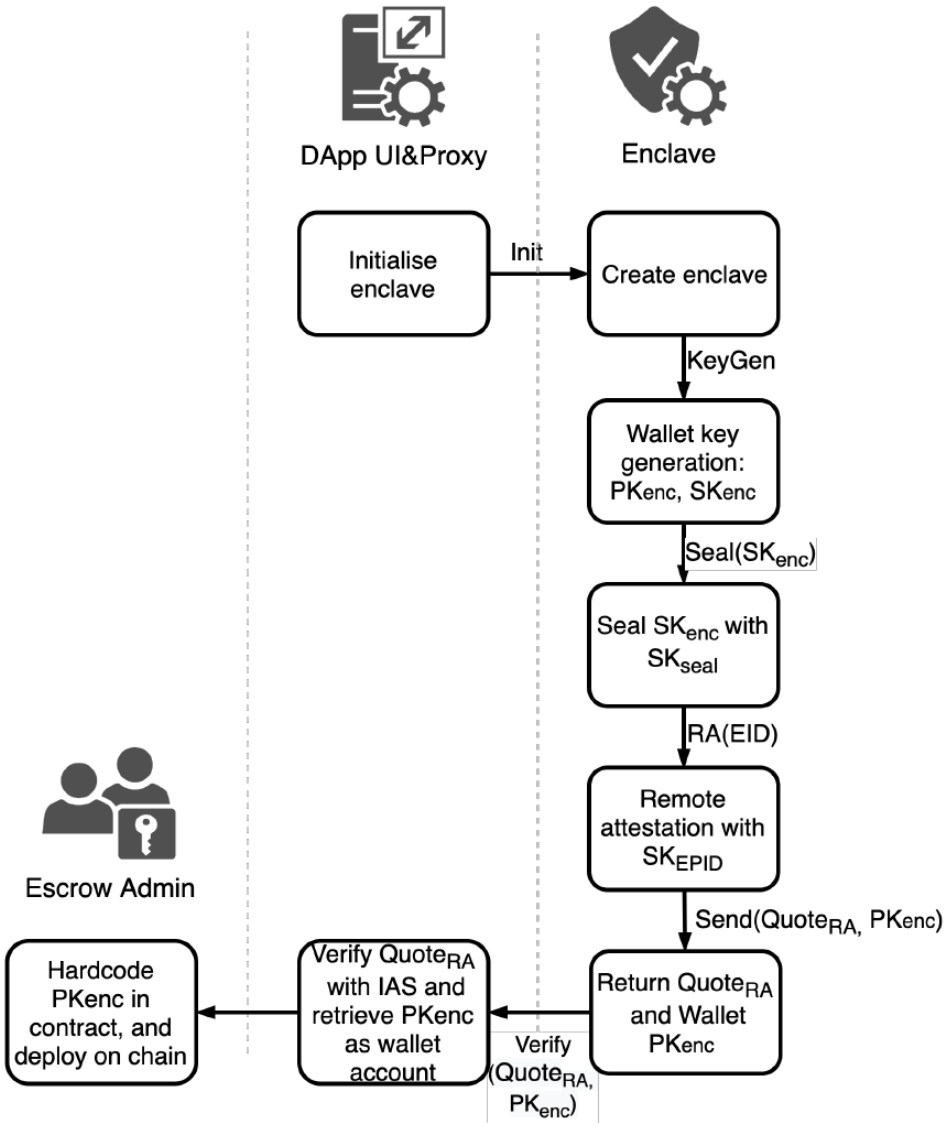
Poll engine will pick up the event and accept the connection:

```
// acceting an event
 'outer: loop {
     poll.poll(&mut events, None)
         .unwrap();
```

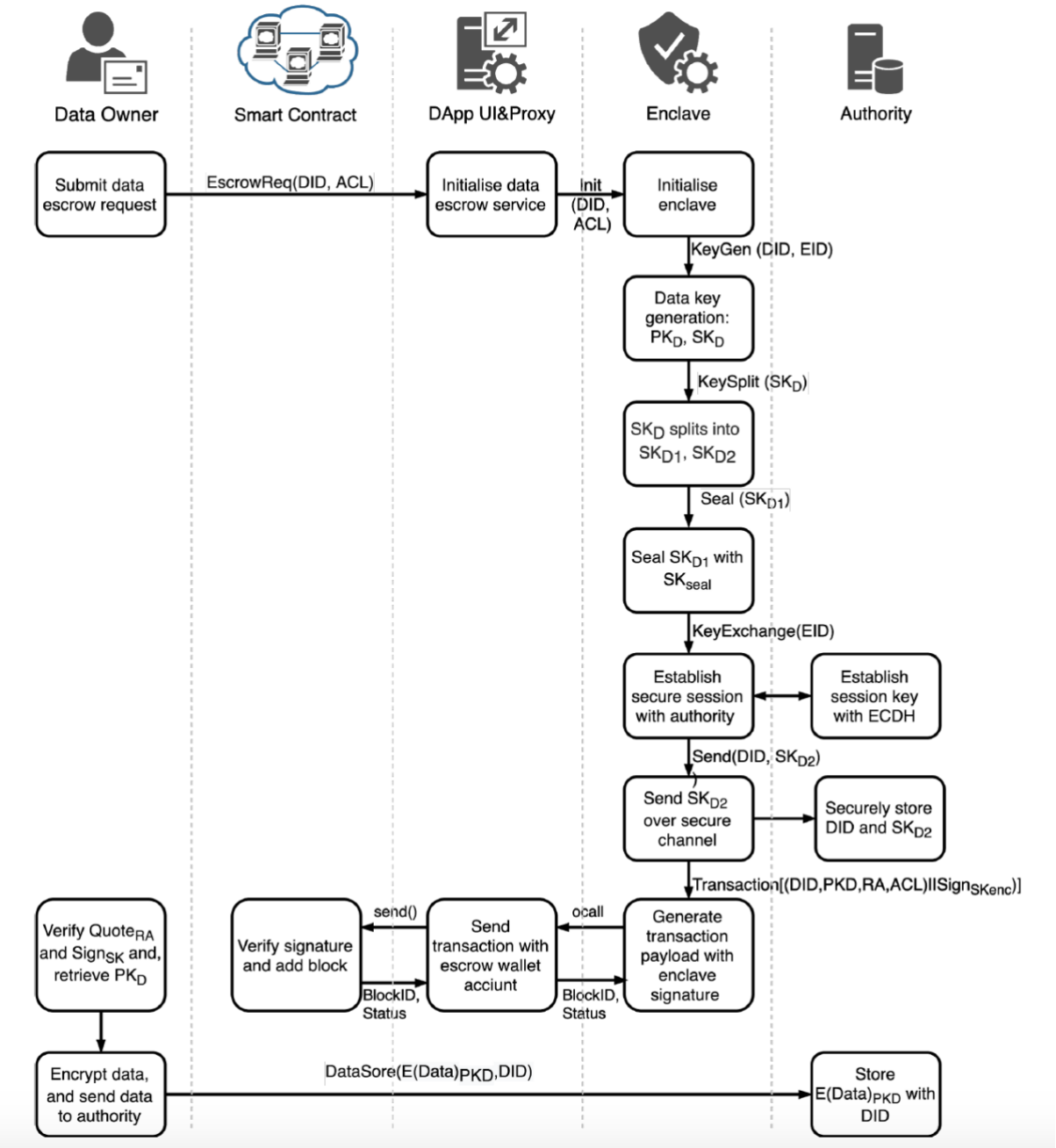
```
    for event in events.iter() {
        match event.token() {
            LISTENER => {
                if !tlsserv.accept(&mut poll) {
                    break 'outer;
                }
            }
            _ => tlsserv.conn_event(&mut poll, &event)
        }
    }
}


fn accept(&mut self, poll: &mut mio::Poll) -> bool {
    match self.server.accept() {
        Ok((socket, addr)) => {

            println!("Accepting new connection from {:?}", addr);

            let mut tlsserver_id: usize = 0xFFFF_FFFF_FFFF_FFFF;
            let retval = unsafe {
                tls_server_new(self.enclave_id,
                               &mut tlsserver_id,
                               socket.as_raw_fd(),

self.cert.as_bytes_with_nul().as_ptr() as * const c_char,

self.key.as_bytes_with_nul().as_ptr() as * const c_char)
            };

            if retval != sgx_status_t::SGX_SUCCESS {
                println!("[-] ECALL Enclave [tls_server_new] Failed
{}!", retval);
                return false;
            }

            if tlsserver_id == 0xFFFF_FFFF_FFFF_FFFF {
                println!("[-] New enclave tlsserver error");
                return false;
            }

            let mode = self.mode.clone();
            let token = mio::Token(self.next_id);
            self.next_id += 1;
            self.connections.insert(token,
Connection::new(self.enclave_id, socket, token, mode,
tlsserver_id));
            self.connections[&token].register(poll);
            true
        }
        Err(e) => {
            println!("encountered error while accepting connection;
err={:?}", e);
            false
        }
    }
}
```

2. Initial Design of Escrow Workflow in the previous project

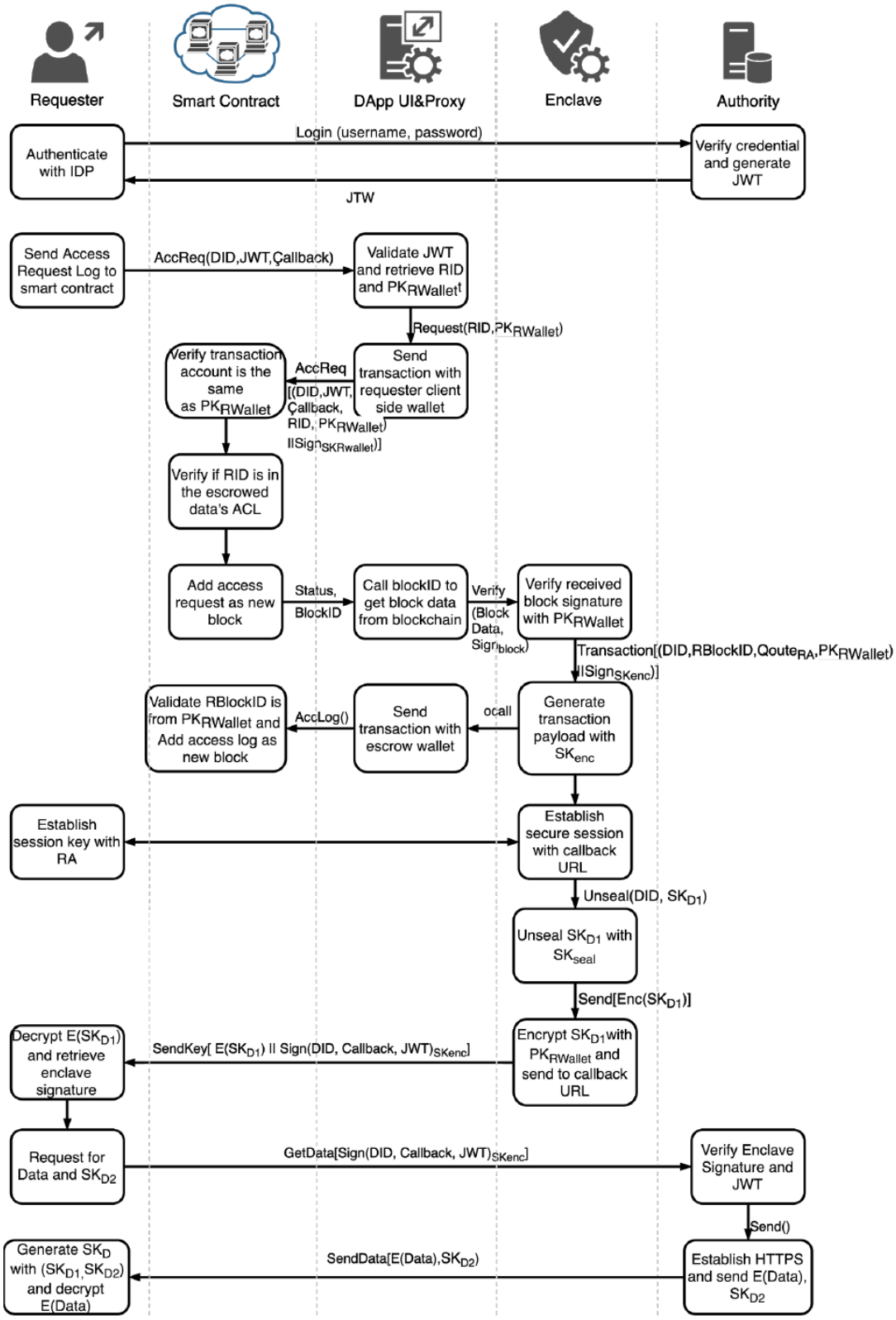
Wallet and Enclave Initialization workflow



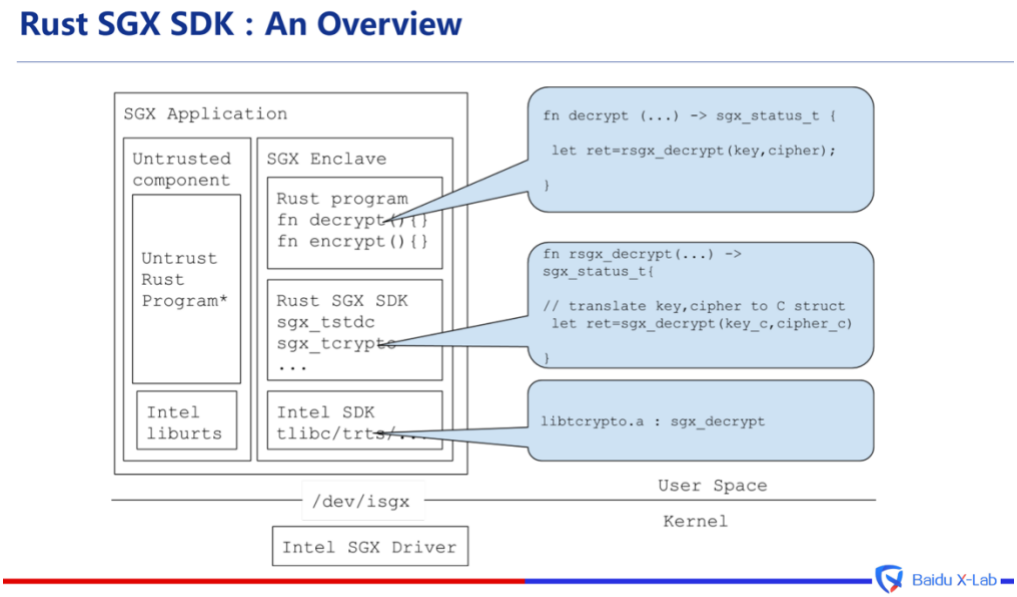
Data Registration



Data Access Request Workflow



3. Rust SGX SDK Design



Y. Ding, R. Duan , L. Li , Y. Cheng , L. Wei. Rust SGX SDK: Towards Memory Safety in Intel SGX. Baidu X-Lab, 2017.