BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CENTRALIZED CRYPTOCURRENCY EXCHANGE WITH TRUSTED COMPUTING

CENTRALIZOVANÁ SMĚNÁRNA KRYPTOMĚN S DŮVĚRYHODNÝM HARDWAREM

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ SASÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. IVAN HOMOLIAK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

148262

Institut:	Department of Intelligent Systems (UITS)
Student:	**Sasák Tomáš, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Centralized Cryptocurrency Exchange with Trusted Computing**
Category:	Security
Academic year:	2022/23

Assignment:

1. Study models of existing cryptocurrency exchanges, including centralized and decentralized ones.
2. Study principles of trusted computing, especially Intel SGX.
3. Analyze security issues and other problems of centralized and decentralized cryptocurrencies.
4. Propose a model and protocol of a centralized cryptocurrency exchange, which will utilize trusted computing to store private keys and orchestrate their secure usage. You can inspire your work with Tesseract and Aquareum.
5. Implement the proposed solution.
6. Make a security analysis of the proposed solution.
7. Discuss possible extensions and future work.

Literature:

- Bentov, Iddo, et al. "Tesseract: Real-time cryptocurrency exchange using trusted hardware." *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019.
- Homoliak, Ivan, and Pawel Szalachowski. "Aquareum: A centralized ledger enhanced with blockchain and trusted computing." *arXiv preprint arXiv:2005.13339* (2020).
- Oosthoek, Kris, and Christian Doerr. "From hodl to heist: Analysis of cyber security threats to bitcoin exchanges." *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* . IEEE, 2020.
- Kim, Chang Yeon, and Kyungho Lee. "Risk management to cryptocurrency exchange and investors guidelines to prevent potential threats." *2018 international conference on platform technology and service (PlatCon)*. IEEE, 2018.

Requirements for the semestral defence:
Items 1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Homoliak Ivan, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	3.1.2023

Abstract

In this thesis, we propose, design, implement, and analyze a centralized cryptocurrency exchange using trusted computing. Most popular exchanges work in a way that requires the customer to fully rely on the exchange operators. Decentralized exchanges work on top of existing blockchains, which limits the number of tradable coins. Using a trusted computing platform, Intel SGX, it is possible for operators to make code open source, and the customer can attest that the exchange platform is running the published exchange code. Also, exchange private keys are encrypted in memory and safely stored on disk using SGX sealing. Combining this approach with a smart contract platform on a public blockchain, we can achieve non-equivocation of the exchange and create a better layer of security and transparency between the customer and the exchange. The exchange was implemented as a proof-of-concept using the programming language Go, using the EGo framework to build SGX enclave applications. For data storage, we combine key-value storage PebbleDB with relation database PostgreSQL, and exchange attempts to secure the integrity of these data. The exchange smart contract was implemented in Solidity and deployed on the Ethereum development environment. Our concept implementation is able to handle around 35 deposits per second and around 23 bids matched per second. We performed a security analysis on this model from outside, but also from inside. The single ledger update from the exchange in the smart contract costs 0.00255 ETH, which in total costs 3.6742 ETH per day.

Abstrakt

Táto práca sa zaoberá návrhom, implementáciou a analýzou centralizovanej zmenárne kryptomien, ktorá pracuje na platforme trusted computing. Aktuálne centralizované zmenárne fungujú na základe plnej dôvery používateľa v operátorov zmenárne. Na opačnej strane, decentralizované zmenárne využívajú jeden blockchain a tým sú limitované iba na mince na danom blockchaine. Použitím trusted computing platformy Intel SGX je možné kód zverejniť (open source) a zákazník si dokáže overiť pomocou SGX atestácie (remote attestation) že tento kód beží na platforme zmenárne. Kombináciou so smart contract platformou na verejnom blockchaine zmenáreň dosahuje odolnosť voči nejednoznačnosti a vytvára lepšiu vrstvu bezpečnosti a transparentnosti pre používateľa. Zmenáreň bola implementovaná ako proof-of-concept pomocou programovacieho jazyka Go, za použitím frameworku EGo pre tvorbu SGX enkláv. Pre ukladanie dát bola použitá databáza PebbleDB a relačná databáza PostgreSQL. Smart contract zmenárne bol implementovaný v jazyku Solidity a nasadený na Ethereum. Implementácia je schopná vyhodnotiť približne 35 vkladov mincí za sekundu a 23 ponúk na výmenu mincí za sekundu. Aktualizácia poslednej verzie ledgeru na kontrakte stojí približne 0.00255 ETH, pri tendencií aktualizácie každú minútu deň behu zmenárne stojí približne 3.6742 ETH.

Keywords

cryptocurrency, exchange, Intel SGX, cryptography, blockchain, trusted computing, encryption, consensus, attestation, smart contract, decentralized identity, verifiable credentials

Kľúčové slová

kryptomena, zmenáreň, Intel SGX, kryptografia, blockchain, trusted computing, enkrypcia, konsensus, atestácia, smart contract, decentralizovaná identita, verifiable credentials

Reference

SASÁK, Tomáš. *Centralized Cryptocurrency Exchange with Trusted Computing.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ivan Homoliak, Ph.D.

Rozšírený abstrakt

Centralizované zmenárne sú pre obyčajného zákazníka dvere do sveta digitálnych peňazí. S rastúcim počtom útokov zo strany autorov, sú používatelia viac skeptický voči týmto službám. Nedávny autorský útok na zmenáreň FTX [16] vzbudil vlnu otázok týkajúcich sa miery transparentnosti medzi zákazníkom a zmenárňou. Táto práca, sa zaoberá návrhom, implementáciou a analýzou bezpečného protokolu centralizovanej zmenárne, nad ktorou autor nemá kompletnú moc. Protokol zmenárne sa zameriava na bezpečnosť, open-source a jednoznačnosť jej stavu. Bezpečnosť zmenárne sa dosiahne použitím zásad trusted computing, presnejšie, špecializovanej platformy Intel SGX a jej prostriedkami ako enkláva, vzdialená atestácia a šifrovanie. Vložením kódu zmenárne do enklávy, sa vytvára bezpečná vrstva medzi operátorom a zmenárňou. Šifrovaním, zmenáreň ochraňuje svoje privátne kľúče od operátora v rámci pamäte ale aj ukladania na disku. Jednoznačnosť stavu zmenáreň dosahuje vytvorením, nasadením a spravovaním smart contractu na verejnom blockchaine. Spomínaný smart contract, funguje ako snímok stavu zmenárne, ktorý enkláva dosiahla a tento stav je transparentný pre všetkých zákazníkov.

Zmenáreň sa skladá z troch elementárnych častí, žurnál $\mathbb{L}$ (štruktúra history tree), stav účtov $\mathbb{A}$ (štruktúra Merkle-Patricia tree) a indexovaná databáza $\mathbb{I}$. Všetky tieto komponenty, sú spravované iba bezpečnou enklávou a sú mimo dosah operátora $\mathbb{O}$.

Žurnál $\mathbb{L}$ sa skladá z mikroblokov zmenárne a používa podobný prístup ako centralizovaný blockchain Aquareum [26]. Mikroblok sa skladá z mikrotranzakcií, ktoré sú vykonané enklávou v rámci určitého časového okna a sú usporiadané v štruktúre Merkle tree. Mikrotranzakcia je zaobalenie, jednej z 3 používateľských akcií v rámci zmenárne a to vklad mincí, výmena mincí alebo výber mincí. Vykonanie tejto mikrotranzakcie mení stav účtov $\mathbb{A}$. Pretože žurnál $\mathbb{L}$ je history tree štruktúra, zmenáreň poskytuje podpísané inkrementálne dôkazy o tom, že mikroblok predchádza aktuálnemu, najnovšiemu mikrobloku. Hash stavu účtov $\mathbb{A}$, je taktiež zakomponovaný v mikrobloku, čiže mikroblock zachytáva aj snímok stavu účtov.

Smart contract $\mathbb{S}$ pre zaručenie jednoznačnosti zmenárne, je nasadený enklávou pri prvom spustení na blockchain Ethereum. Kontrakt ukladá nasledujúce, verejný kľúč operátora na verejnom blockchaine $PK_{\mathbb{O}}^{PB}$, verejný kľúč enklávy na verejnom blockchaine $PK_{\mathbb{E}}^{PB}$, verejný kľúč enklávy pre SGX TEE schému $PK_{\mathbb{E}}^{TEE}$, DNS záznam alebo IP adresu zmenárne $A_{\mathbb{E}}$ a hash (snímok) poslednej verzie žurnálu $H_{\mathbb{L}}$. Pretože kontrakt, obsahuje snímok poslednej verzie žurnálu $H_{\mathbb{L}}$ na jednotnom verejnom mieste, snímok je nespochybniteľný dôkaz, že enkláva dosiahla tohto stavu. Snímok verzie v kontrakte je ochránený a jediná enkláva svojim podpisom ho môže aktualizovať. Pretože každá operácia v rámci kontraktu stojí poplatky na danej sieti, je nutné nájsť kompromis medzi bezpečnosťou (finalitou) a cenou údržby zmenárne.

Zmenáreň používa decentralized identifier DID v kombinácií s verifiable credentials VC pre správu používateľov. Budúci zákazník, potrebuje získať identitu tohto typu, vygenerovať a vložiť do neho svoj verejný kľúč jeho účtu na zmenárni. Dvojica tohoto verejného a privátneho klúču, slúži pre autentizáciu používateľa.

Pre otvorenie účtu v zmenárni, budúci zákazník vykoná vzdialenú atestáciu enklávy, vyžiada si adresu enklávy určenú pre vklad na danej sieti a vytvorí transakciu ktorá pripisuje vklad na adresu enklávy. Po tom, čo transakcia je sfinalizovaná na blockchaine, zákazník vytvorí členský dôkaz o tom, že transakcia sa nachádza v danom sfinalizovanom bloku. Enkláva, zvaliduje tento dôkaz pomocou vnútorného SPV klienta a validátora. Ak dôkaz je správny a transakcia sfinalizovaná, enkláva vytvorí účet v rámci stavu účtov $\mathbb{A}$ a pripíše mu jeho vložené mince v rámci zmenárne.

Pre výmenu mincí, každá ponuka je vybavená virtuálne v rámci enklávy. Zákazník, zverejňuje novú ponuku, poskytnutím hodnoty koľko mincí chce vymeniť a koľko mincí chce za to dostať. Enkláva sa pokúsi spárovať opozitné ponuky (plne alebo čiastočne), ktoré vyhovujú tejto ponuke. Ak existujú, enkláva uzavrie výmeny medzi účtami odčítaním a pridaním z bilancií zákazníkov.

Pretože výmeny a všetky ostatné akcie, prebiehajú virtuálne v enkláve, zmenáreň poskytuje výber mincí späť z zmenárne, na svoju adresu. Enkláva periodicky zverejňuje blokchainove transakcie s viac výstupmi, kde jeden výstup reprezentuje jednu akciu výberu mincí v rámci zmenárne. V rámci enklávy, sú tieto mince zmrazené a nemôžu byť použité. Po tom, ako je transakcia sfinalizovaná, zmenáreň uvoľňuje zmrazené mince z účtu.

V porovnaní s existujúcim riešením Tesseract [7], toto riešenie nepotrebuje časovo zamknuté vklady na nízkom leveli v daných blockchainových transakciách. Taktiež, nie je nutné vykonávať denne vyúčtovacie transakcie (settlement transaction) a tým pádom nie je nutnosť pre vyúčtovací protokol medzi viacero blockchainami (cross-chain settlement protocol). Navyše, toto riešenie ponúka spôsob, akým zákazník si môže overiť svoj dôkaz o vlastnom účte v rámci stavu účtov a jeho mikrotranzakcie zverejnené v rámci žurnálu, pomocou smart contractu.

Implementácia, bola vytvorená v jazyku Go, za použitím frameworku pre enklávy EGo. Pre indexovaciu databázu sa použila relačná databáza PostgreSQL. Zmenáreň vie aktuálne pracovať s mincami Bitcoin a Litecoin. Pretože zmenáreň ukladá dáta mimo enklávu do databáze, implementácia rieši aj tento aspekt zaručenia integrity dát a dostupnosti iba pre enklávu.

Daná implementácia riešenia, je schopná spracovať približne 35 vkladov za sekundu a približne 23 ponúk za sekundu, pre jednu dvojicu mincí. Existujú dva zdroje, kde zmenáreň spomaľuje. Prvým dôvodom, je, že aktualizácia štruktúry Merkle-Patricia tree nemôže byť paralelná, a musí byť prepočítaná exkluzívne, pre každú jednu zmenu účtu.

Druhý dôvod, je integritná schéma pre ukladanie dát mimo enklávu. Pretože enkláva používa PostgreSQL, jednotlivé riadky musia byť chránené pred úpravou operátorom. Enkláva, musí neustále hashovať a šifrovať jednotlivé upravené riadky, čo vytvára spomalenie.

Na záver, v práci diskutujeme potencionálne útoky a analyzujeme, ako je zmenáreň odolná voči týmto útokom zo strany operátora ale i zákazníka. Práca ma stále veľa priestoru na zlepšenie, ako z pohľadu rýchlosti, funkcionality a aj z pohľadu smart contractu. Taktiež, by bolo možné zmenáreň rozšíriť aby ponúkala proof-of-solvency [9] schému, zmenáreň by generovala dôkazy že má vždy vo vlastníctve tolko mincí aby pokryla každého zákazníka pri výbere.

Centralized Cryptocurrency Exchange with Trusted Computing

Declaration

I hereby declare that this Master's thesis was prepared as an original work under the supervision of Ing. Ivan Homoliak, Ph.D. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Tomáš Sasák
May 17, 2023

Acknowledgements

I want to thank my supervisor Ing. Ivan Homoliak, Ph.D., for helping and guiding me with the thesis. My thanks also goes to my family and friends who supported me during this journey.

Contents

Chapter 1

Introduction

For the last few years, cryptocurrencies and blockchains have started to rise in interest among people for their new interesting views on digital money and overall finance. For entry into this world, a person needs to obtain some amount of this currency. The original idea for this was to obtain the tokens by mining (or staking), which for a non-technical person would be a great hassle. In this case, the exchange comes into play as the simplest way of entering this world. For an average customer, exchanges are nowadays the standard way to obtain these currencies and exchange them, usually for other currencies or fiat money.

As exchanges represent some new abstract layer on top of these blockchains, there come many new technical problems that engineers need to solve, problems like security, atomic cross-chain swapping, account management, etc. A customer of an exchange like this needs to fully trust this party since his information and especially his funds are in full control of the exchange. The low emphasis on the security of an exchange like this can lead to exploits from a malicious external party or even insiders, which can result in stolen funds from customers and other involved parties. There are already numerous cases of insider and external party attacks on such exchanges as Mt.Gox — over $63,000,000 lost [48], FTX — insider attack, estimated over $1,000,000,000 lost [16] or Liquid — external party attack, more than $90,000,000 lost [46].

1.1 Motivation

Currently, centralized exchanges work in such a way that they completely own the user's funds. The customer does not have any proof that the funds are still in the exchange wallet. Also, the customer cannot see the code that runs on top of these funds.

In this thesis, we will propose a model of the centralized cryptocurrency exchange with a focus on security, open source, and non-equivocation. Security will be achieved using the Intel SGX trusted computing environment and its principles, such as remote attestation and sealing. For non-equivocation, we propose an Ethereum smart contract that will function as a holder of a snapshot of the state of the exchange and provides transparency between the exchange and the customer.

There are already existing models of exchanges using SGX such as Tesseract [7], but they have some disadvantages such as non-open source code or high overhead for settlement transactions, which results in high fees and too much computing processing.

1.2 Organization

In chapter 2, we discuss the cryptography fundamentals necessary for blockchain to function, such as encryption and digital signatures.

Next, in chapter 3, we debate the foundations of blockchain and cryptocurrency, in particular, block, transaction, and consensus. There is also a need to explain real implementations and their security concerns. Closely connected to the blockchain are decentralized identity and verifiable credentials, which will be explained in the following.

Then, in chapter 4 we explain the existing centralized and decentralized exchange models and discuss their security concerns.

Since we propose an exchange based on trusted computing, we explain its fundamentals in chapter 5. The selected platform for implementation is the Intel SGX, in this Section, we examine the architecture, attestation mechanisms, and sealing.

After that, in chapter 6 we need to discuss and take into account the already existing blockchain and exchange solutions using trusted computing. These solutions contribute to our design.

We propose a secure centralized exchange design in chapter 7. We will explain the high-level overview of this exchange, which is composed of a smart contract on a public blockchain and an Intel SGX enclave. After that, we inspect the components of this exchange in more detail.

Then, in chapter 8 we discuss our implementation of the proposed design from the previous section. We dive into the actual exchange logic and tools that helped achieve the proposed proof-of-concept.

There is a need to focus on security analysis, precisely attack vectors, and analyze how exchange can mitigate such attacks. Also, there must be a focus on the feasibility and performance benchmarking of such an exchange, since regular exchanges are capable of handling thousands of trades every day. We focus on this topic in chapter 9.

Finally, we complete this text by describing the conclusion in chapter 10.

Chapter 2

Cryptography

Cryptography is a field in computer science closely coupled with mathematics and its concepts. This field focuses mainly on protecting information and achieving secure communication so that only those for whom the information is intended can access and process it. This can be achieved by using special cryptography algorithms. There are five main cryptography objectives that we want to achieve [49]:

- **Confidentiality**: Protecting personal privacy and proprietary information, that is, preserving authorized restrictions on access and disclosure of information.
- **Integrity**: Protecting against improper modification or destruction, including ensuring non-repudiation and authenticity.
- **Availability**: Ensuring reliable access to collect and use of information.
- **Authenticity**: Process of verification and trust. The confidence in the validity of the information is genuine.
- **Accountability**: Actions of an entity must be uniquely traced to that entity. This supports non-repudiation.

Cryptography offers many algorithms that can be combined to achieve these objectives and create many protocols. We will discuss only the most important ones, to better understand and implement secure blockchain, cryptocurrency, and exchange.

2.1 Hash Function

A hash function is a special mathematical function H that accepts any long block of data M as input and produces a fixed size value h, we call this value a hash of the given input. The function can be defined as:

$$h = H(M). \tag{2.1}$$

A good quality hash function will for a lot of inputs produce a large set of outputs that are evenly distributed and random, which means that for two very similar outputs, the hashes will be completely different.

For a hash function to become a cryptographic hash function, it must provide the following security properties [49]:

- **Pre-Image Resistance**: For a given hash h it is computationally infeasible to find the input M for which $h = H(M)$.
- **Second Pre-Image Resistance**: For a given input M_1, it is computationally infeasible to find M_2 for which $H(M_1) = H(M_2)$.
- **Collision Resistance**: The finding of M_1 and M_2 such as $H(M_1) = H(M_2)$ is computationally infeasible.

Normally, we want the cryptographic hash function to satisfy all properties. However, for some specific use cases, we can use functions that satisfy only some of these properties.

Cryptographic hash functions are the key to achieving some of the features of the blockchain, such as digital signatures or authenticity.

2.2 Encryption

Encryption is a cryptography technique that is used to conceal some information of any size. There are two main types of encryption, symmetric and asymmetric [49].

In encryption, the original message is called plaintext P, which is also the input to one of these algorithms. The encrypted (coded) message is called the ciphertext C, which is also an output. The transformation process from plaintext to ciphertext is called encryption. The transformation process from ciphertext to plaintext is called decryption.

2.2.1 Symmetric Encryption

Symmetric encryption, also called single-key encryption, was the only type before the invention of asymmetric encryption. Symmetric encryption is still widely used. Most of the time, this type of encryption is used for the encryption of messages or longer data blocks because symmetric encryption algorithms are faster than asymmetric ones. This can be deciding for some real-time applications. The symmetric encryption has three main components [49]:

- **Secret Key**: K, second input beside plaintext for the encryption algorithm, it is independent of plaintext, and the algorithm will produce different ciphertext based on a different secret key.
- **Encryption Algorithm**: E, performs various operations (such as substitutions or permutations) on the plaintext, which results in the ciphertext.
- **Decryption Algorithm**: D, the reverse of the encryption algorithm, produces plaintext from ciphertext and secret key.

Then the act of encryption can be defined as:

$$C = E(K, P), \tag{2.2}$$

and also the act of decryption as:

$$P = D(K, C). \tag{2.3}$$

There are two requirements for the secure use of this type of encryption:

1. The malicious party should not be able to decrypt the ciphertext or calculate the secret key, even if he knows the algorithm and has multiple plaintexts with the corresponding ciphertexts.

2. Both ends must have obtained (and also kept) the copy of the secret key correctly and secretly.

The principal security problem in this algorithm is keeping the secret key in secrecy for all parties.

2.2.2 Asymmetric Encryption

The discovery of asymmetric encryption was a breakthrough in the world of cryptography. This type of encryption is based on several mathematical functions and number theory, instead of permutations and substitutions, as used by symmetric encryption. The concept of asymmetric encryption was created as an attack on some of the disadvantages of symmetric encryption. The first is the secret key distribution. How to securely deliver the key to multiple parties, how should the parties keep the key secure, and what are the consequences if some party leaks the key?

The second is familiar to the first, that is, the existence of digital signatures. If symmetric encryption should be adopted by the public, how can a user prove that the message was sent by him. Creating secret keys for each end of symmetric encryption is nearly impossible.

The asymmetric key encryption is based on two keys, the private key SK and the public key PK. These two keys are complementary and have different use cases. The main important feature is that the private key is computationally infeasible to calculate from the public key. The public key should be used mainly for decryption, while the private key should be used mainly for encryption. Some algorithms such as RSA allow both keys to be used, where the one is for encryption and the second is for decryption. There are now more additions to the components [49]:

- **Private and Public Key**: Pair of keys where one is used for encryption and the second for decryption.
- **Encryption Algorithm**: Performs the mathematical operations with the provided key on the plaintext and returns the ciphertext.
- **Decryption Algorithm**: Performs the mathematical operations with the provided key on the ciphertext and returns the plaintext.

The parties will obtain the keys by the following. Every party generates its private key, which needs to be kept safe and private. From the private key, the party generates a public key that can be published anywhere, so that the other parties can obtain it.

Let there be two parties A and B, where each of them has its own generated private-public key pair (SK_A, PK_A, SK_B, PK_B). The party A wants to send secret information only to the party B, the act of encryption can be defined as [49]:

$$C = E(PK_B, M). \tag{2.4}$$

This results in a ciphertext that only party B can decrypt since only party B is the holder of the private key. The party B then decrypts the message as defined:

$$M = D(SK_B, C). \tag{2.5}$$

We should bear in mind that this does not confirm that this message was sent by the party A. The malicious party also has the same access to the public key of the party A as others.

Previously, we mentioned that some encryption (and decryption) algorithms, such as RSA, allow users to choose which key from the pair will be used for encryption (and the opposite key from decryption). Let us say that the party B would like to respond to the previous message but the response does not need to be kept secret and must be non-repudiable, this can be considered as a digital signature which can be defined as [49]:

$$C = E(SK_B, M). \tag{2.6}$$

This will result in a ciphertext that works as a digital signature. Anyone who has a public key PK_B can decrypt the message and read it, but it is proof that the sender is a party B because only he has the private key SK_B. The party A can then see the response as defined:

$$M = D(PK_B, C). \tag{2.7}$$

Again, this approach does not achieve confidentiality, as any holder of the public key of the party A can decrypt the message.

2.3 Digital Signature

In the previous subsection, we mentioned the existence of a cryptographic construct such as a digital signature in asymmetric encryption. This approach is not optimal because there is a big trade-off where confidentiality is swapped for authenticity. The other disadvantage of this approach is that the entire message must be the input of an asymmetric encryption algorithm.

Using the power of the cryptographic hash function and asymmetric encryption, we can define a different model for the digital signature S. Let party A create and send a signed message M to party B, this can be defined as [49]:

$$S_{MA} = E(SK_A, H(M)). \tag{2.8}$$

Signature S_{MA} can be appended at the end of message M. Then the other party B can verify the message by doing the following. First, the party B will calculate a cryptographic hash H_{MA} of the message M:

$$H_{MA} = H(M), \tag{2.9}$$

then this hash must be equal to the value of the hash buried inside the signature of party A

$$D(PK_A, S_{MA}) = H_{MA}. \tag{2.10}$$

If the hashes are equal, the signature is correct. This proves that the message was created by the party A because only the party A has control of the private key SK_A. Furthermore, this approach guarantees integrity, the malicious party is unable to recalculate the signature without holding a private key SK_A, and any tampering with the message results in the difference in hashes.

It is very important to correctly select the hash function for this usage. The potential hash function for the signatures must be cryptographically secure and must satisfy the requirements mentioned in section 2.1.

Chapter 3

Blockchain and Cryptocurrency

Blockchain is an instance of distributed ledger technology. The distributed ledger is the consensus of shared, replicated, and synchronized data in some logical data structure. By distributed ledger, we achieve that we do not have a single point of failure and potential decentralization [12].

Blockchain represents such an append-only logical data structure on a distributed ledger technology. The blockchain is divided into data sections called blocks, where every block points to the previous block. The blocks linked together then create some form of a linked-list structure. Each block, its content, and the pointer to the previous block are secure by cryptographic principles.

We have visualized a simple generic blockchain in Figure 3.1. Each block contains data such as transactions and is represented as a hash value of the hash function. The following block has this hash value of the previous block stored inside. Since blockchain is append-only, it forms a chain of trust from the latest block to the first block (also called the genesis block).

With this approach, it is not possible to modify any of the data in any of the previous blocks. Doing this would change every hash value of the block from the modified block to the latest block, since the hash of the previous block would change in every next block.

Blockchain has potential in many sectors such as healthcare, IoT, supply chain, and many more. The largest public adoption and an explosion of popularity is due to its usage in banking and finance. Especially in a way that blockchain can represent digital money

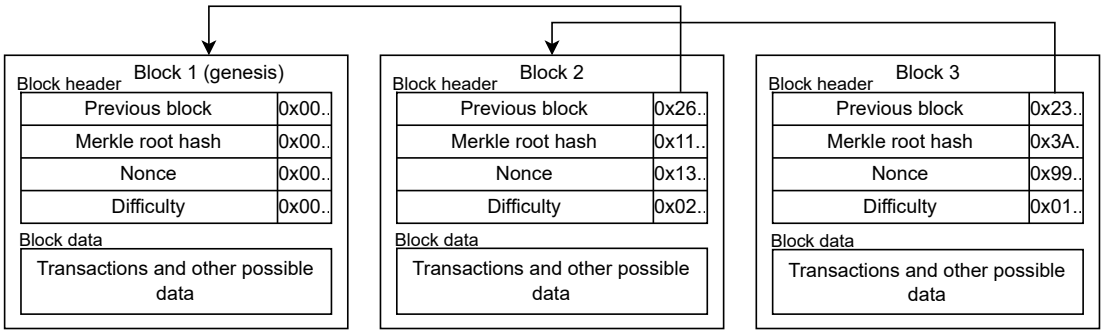

Figure 3.1: The example blockchain with 3 blocks, genesis block, and latest block.

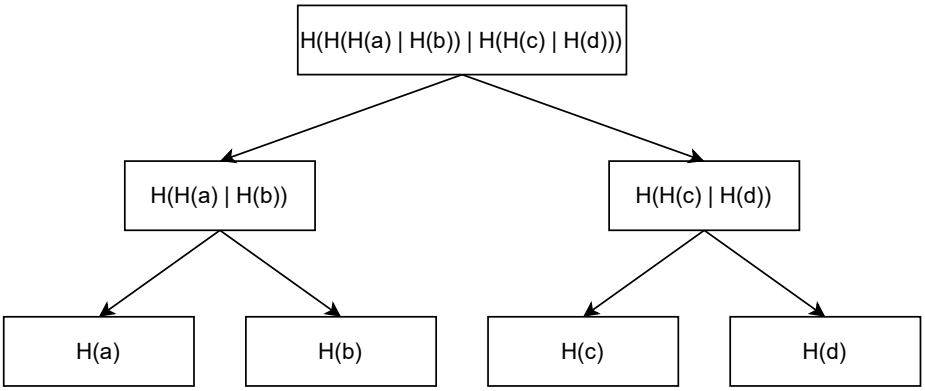

Figure 3.2: The example of Merkle tree with 4 leaves.

(the so-called cryptocurrency). From now on in this thesis, we will focus only on the usage of blockchains for cryptocurrency.

3.1 Block

Block can be represented as a single chain checkpoint. Each block is composed of data such as creation timestamps, pointer to the previous block, nonce, root hashes, etc. But the most important data in the block are the transactions. Transactions are usually grouped into some logical structure.

In most cases, the block is made up of two parts, the block header, and the block data. The header of the block usually contains the most important information of the block, which can then be used by other nodes or users without transmitting the whole block data, which is much larger than the header. On the other hand, block data can contain transactions, account states, contract states, receipts, etc.

Since transactions (or states) need to be accessed multiple times, need to be cryptographically protected, and fast for lookup, the most used data structures used for this are Merkle Tree, Merkle-Patricia Tree, or History Tree.

3.1.1 Merkle Tree

Merkle Tree is a binary tree that incorporates the use of the cryptographic hash function. Each leaf node of this tree contains some data, and every node calculates a hash, which is recursively dependent on all data inside its own subtree. At first, the leaf nodes calculate the hash of its own data, and every other parent node calculates a single hash of its children by concatenating the hashes of the children from left to right [14]. Such an example is shown in Figure 3.2.

In this way, the root node of the tree results in a single hash, which is called the root hash. This root hash depends on all of the data stored inside every node, thus changing any data inside this tree will immediately change the root hash.

One of the most important features of this data structure is that a party can prove to another party that the data are present inside the tree. This proof is called the proof of membership. To prove that the data are included in the tree, the following data are required:

- Root hash of the tree.

- The data or only its hash (if the other party can calculate the hash of the data and compare).
- A path from the root node to the leaf node that is wanted to be proven. Every member of the path needs to contain the hash value of the sibling node with an indicator if the sibling is on the left side or right side (for correct hash concatenation).

The verifying party will then use the hash of the data along with the sibling hashes from the path to compute the hash of all nodes in the path up to the root node. The final hash is the calculated root hash, which is then checked for equality with the provided root hash from the asking party. If the hashes are equal, then the proof is valid, and the data are indeed inside the tree [47].

This proof is in the logarithmic depth of the tree size, and also verifying the proof requires computing the number of hashes equal to the logarithm of the number of leaf nodes in the tree [51].

Applying this structure on a block, a single block can store a number of transactions, where a single transaction hash is stored inside the single-leaf node of the tree. The root hash can then be stored inside the block header, from which a single hash is calculated, called a block hash, which represents an identifier of this block.

With this technique, the queries on the chain from users or apps can be lighter in terms of the number of downloaded data. For example, if some app working on the blockchain would like to verify that the transaction from the user is actually stored on the chain with given proof of membership, it can easily request a node to provide the block header and use its data to verify the proof, as we mentioned previously.

3.1.2 History Tree

History tree uses a versioned computation of hashes over the Merkle tree to prove that different log snapshots represented by the Merkle tree with distinct root hashes make consistent claims about the past. The history tree can be understood as an upgrade of the Merkle tree [13].

As in the Merkle tree, the history tree stores data at the leaf nodes, and the hash at the top root leaf is a tamper-evident summary of the leaf contents. The Merkle tree is used as a static data tree, especially in blockchains, the history tree allows the developer to add new data (leaf node), which results in the history tree being altered, and the root node hash acts as some snapshot of the data. Each snapshot we call a version N of the given history tree. A filled history tree of depth D stores (as a binary tree) 2^D leaf nodes (data) [13].

In Figure 3.3 we can see a history tree of version 3 which contains 3 data leaves. Figure 3.4 shows four more added data leaves, thus evolving the history tree to version 7. Both figures show only the history tree with a depth of three, but a real implementation can store any arbitrary number of leaf nodes by extending the depth by one when the tree leaf nodes are fully filled.

The history tree has the very powerful property of being able to reconstruct the three lower versions and recalculate the tree commitments. A given tree in Figure 3.4 with version 7 can reconstruct and recalculate the tree of version 3 in Figure 3.3 by pretending that the right subtree with root node $H_{2,2}$ does not exist. Then, the hashes of the nodes are recalculated, and the root hash is the commitment of version 3.

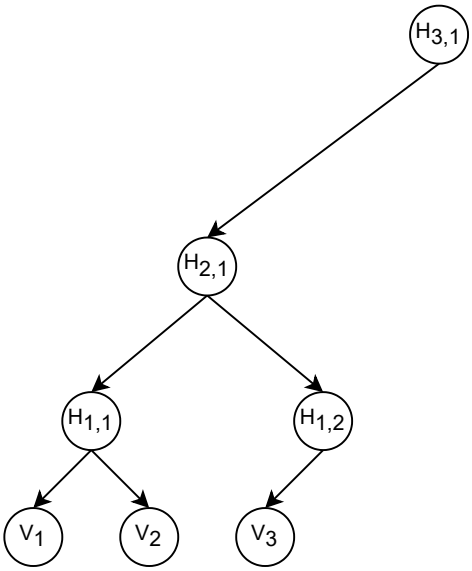

Figure 3.3: A history tree of version 3, with commitment $H_{3,1}$.

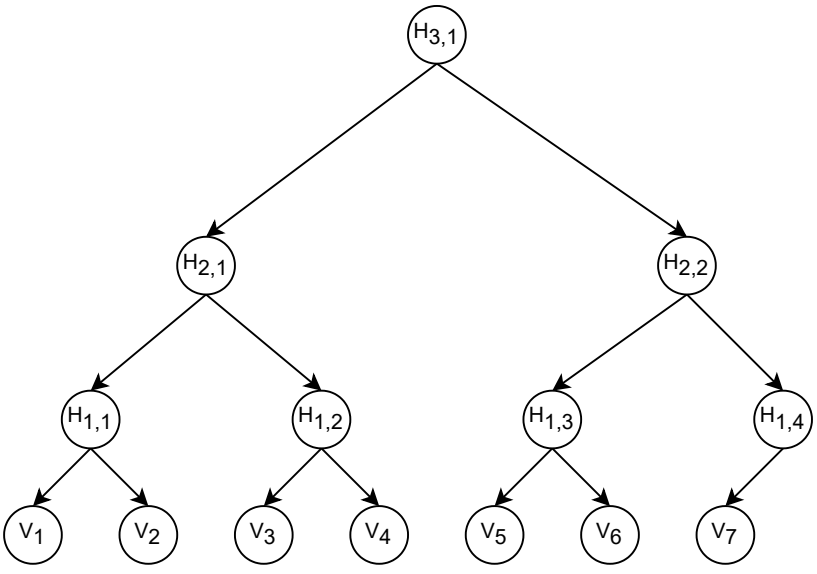

Figure 3.4: A history tree of version 7, with commitment $H_{3,1}$.

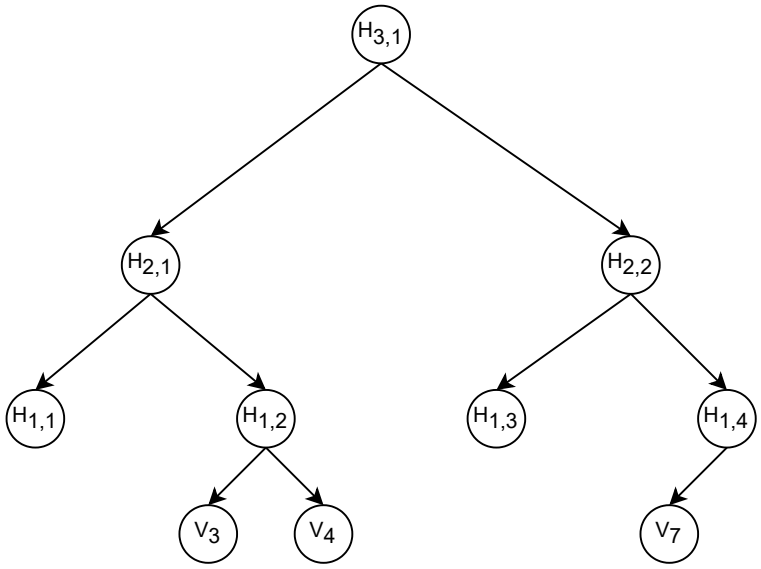

Figure 3.5: An incremental proof P of the version 3 and version 7.

As a regular Merkle tree, the history tree supports the provisioning of the leaf-node membership proof. But by combining the above-mentioned property of reconstructing the history tree is also able to provide incremental proof.

An incremental proof between two versions (commitments) V_X and V_Y $(X \leq Y)$ proves that the two versions commit to the same shared history. If the auditor only has two commitments of version V_3 (from Figure 3.3) and version V_7 (from Figure 3.4), and wants to calculate the proof that they commit to the same history, he can request an incremental proof for these two versions. The history tree creates an incremental proof, pruned history tree P, illustrated in Figure 3.5. Not-needed subtrees are replaced with a stub that contains their subtree root hash.

The hashes of the data nodes up to the root node $H_{3,1}$ are recomputed, which is the proof commitment for version 7. If the known commitment of version 7 is equal to the recomputed commitment of the proof P, the nodes of the history tree pruned by the proof must be equal to the nodes of the original tree of history (the tree from which the proof was created) [13].

Using the reconstruction property of the history tree, for the version 3 commitment, we must reconstruct the pruned proof tree P to version 3 as previously described. Then again, the reconstructed tree commitment (root hash) is calculated. If the commitment for the reconstructed version 3 is equal to the known commitment of version 3, the pruned history tree nodes in the proof must be equal to the nodes in the original tree.

If both commitments are equal to the recalculated commitments from the proof P, we can conclude that both reconstructed history trees commit the same history (leaf values) even when some of the concrete leaf values are unknown to the auditor [13].

3.1.3 Merkle-Patricia Tree

The Merkle-Patricia tree is a persistent data structure that provides a mapping from binary data of arbitrary length to binary data of arbitrary length. In most use cases, it is used to

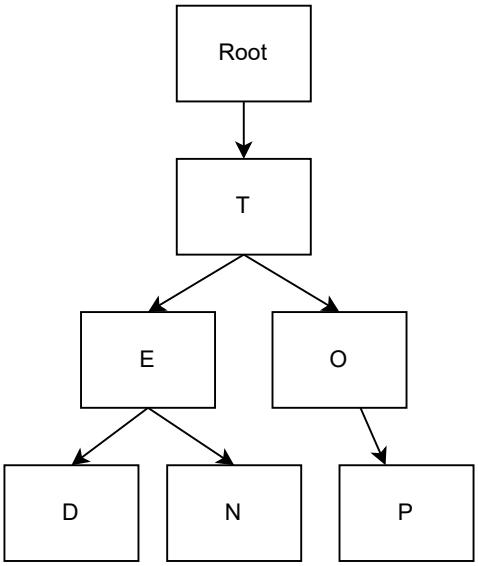

Figure 3.6: The trie example with 3 words, „ted“, „ten“, and „top“.

map from 256-bit long byte arrays to arbitrary length byte arrays, typically used as some database.

Merkle-Patricia tree is based on the trie data structure. Trie is a tree, key-value-based data structure. Each edge inside of the trie represents a single character on a given index of the key. Thus, searching for the value by a given key, the trie is traversed depth-first by following the edges of nodes that represent each character in the key. With this approach, every node of the trie can have the maximum number of children nodes equal to the number of characters allowed from the trie alphabet, as shown in Figure 3.6.

The Merkle-Patricia tree (now just called trie) optimizes this disadvantage by introducing the keys as byte arrays, which can be represented as a list of nibbles (value from $0_{(16)}$ up to $F_{(16)}$). The trie is then traversed by these nibbles. That means that a single trie node can have up to 16 children's nodes. An example of such a trie is visualized in Figure 3.7.

Introducing this feature creates a performance problem. For a single key of size N, the search algorithm must access at least the number of N nodes to reach the leaf node with the value. Due to this problem, trie introduces a special node type that works as a shortcut node. This node can skip multiple nodes that have only one child after each other. Then the trie can have 3 possible node types [52]:

- **Leaf Node**: A two-item structure, where the first item is a suffix of the key corresponding to the value (these nibbles are remaining and can have any length). The second item is the value.
- **Extension Node**: A two-item structure, where the first item is a series of nibbles (size greater than 0) that are an infix for at least two distinct keys from the current traversed key (path). The second value is a pointer to the next node.
- **Branch Node**: A 17-item structure. The first 16 elements represent an array of pointers to the next node, where the index represents the next node key nibble. The 17th item is the value corresponding to the key.

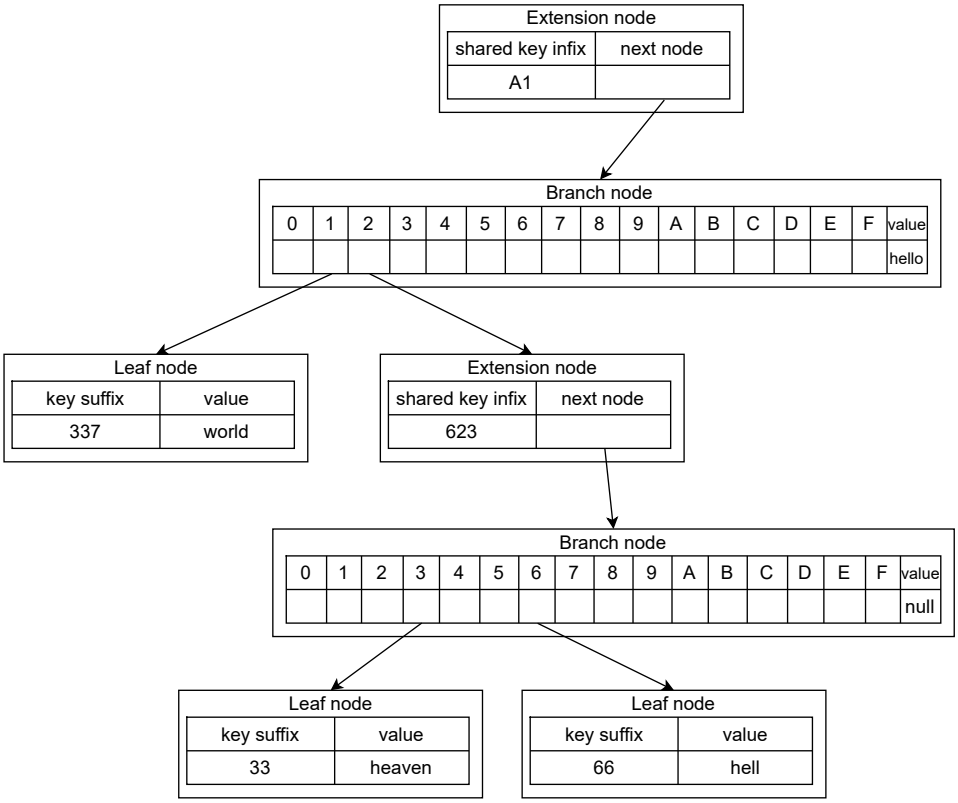

Figure 3.7: The example of Merkle-Patricia tree with 4 items, with keys „A11337“, „A12623333“, „A12623666“ and „A1“.

Since we mentioned that this structure is persistent, we need to introduce the next functional part of the trie which is called the database. The database is an abstract construct for the trie and needs to fulfill 3 properties. It needs to be able to store data on a disk, read the data, and remove them from the disk. In real implementations of trie, authors often use key-value such as LevelDB, which uses log-structured merge trees [52].

Using the database approach in the trie, we can introduce the cryptographic features of the Merkle tree from subsection 3.1.1. Instead of the pointer to the next node, the trie uses a hash of the next node, which is used as a key to the internal database. Using these hashes, we can again introduce the single-aggregation root hash construct from subsection 3.1.1. We need to alter the computation of node hashes in the following way:

- **Leaf Node**: Hash will be calculated from the byte concatenation of the remaining key nibble suffix and value in this node.
- **Extension Node**: Hash will be calculated from the byte concatenation of the shared key infix and the hash of the next node.
- **Branch Node**: Hash will be calculated from the byte concatenation of 16 hashes pointing to the next nodes and a value in this node.

In the same fashion, the party can create a proof of membership in the trie for the other party. The concept of an internal database also has one important advantage. Instead of storing all data in the process memory, the process can only hold a root node inside the memory and some cache of recent nodes (in a more sensible implementation) [52].

3.2 Transaction

The transaction can be defined as a single cryptographically signed instruction (digital signature) constructed by a party using the blockchain. The specifics of the transaction data are blockchain-specific, but usually in the blockchain with the support of digital money, the transaction is a data structure that encodes an act of the value transfer between two or more parties. The transaction life cycle can be defined in a few phases.

The transaction can be created by anyone, even if the creator is not a holder of the account. To make the transaction valid for the blockchain, it must be signed. Only the holder of the account can sign this transaction using a digital signature [3].

The signed transaction can be sent to the blockchain for processing by sending this transaction to a single node that participates in the given blockchain protocol. Since the transaction is already signed, there is no need for additional communication with the node, such as authentication of the account holder, etc. A receiving node will validate and propagate the transaction to all of its known connected node neighbors, called peers. These nodes will perform the same procedure until the transaction does not traverse the entire network of connected nodes. This process is called flooding.

We can then define two types of transaction by their representation of the transfer value between accounts.

3.2.1 UTXO-based Model

The unspent transaction output is an indivisible chunk of the given blockchain currency locked to one or multiple accounts. Every node tracks a record of all available (unspent) transaction outputs. The account balance is then represented as a set of UTXOs scattered throughout the transactions in multiple blocks through a whole chain locked to a given account. To acquire the current balance of the account, the wallet (or other client software) must scan the entire blockchain to find all the unspent UTXOs and make a sum of their values [3].

A transaction in this model can usually range in size from 300 to 500 bytes of data [3]. Such transaction is visualized in Figure 3.8 and contains the following:

- **Input**: Set of references to the UTXO in previous transactions that will be consumed (mark spent).
- **Output**: Set of new UTXOs that will be produced by adding a transaction to the final block, and thus they are new, unspent.

The complicated transaction with a custom locking script and multiple inputs and outputs (ex. sender wants to send coins to multiple receivers) can end up much larger than the mentioned size. The final size is the crucial factor for the amount of fee that the sender needs to pay, since most blockchain transaction fees depend on the size of the transaction data.

The UTXO is locked through a mechanism called a locking script. The locking script is written in the blockchain-specific programming language. This script defines the conditions that must be satisfied for the account to unlock this UTXO and spend it as an input for the new transaction. The locking script is an encumbrance always placed on every single UTXO in the output section of the transaction.

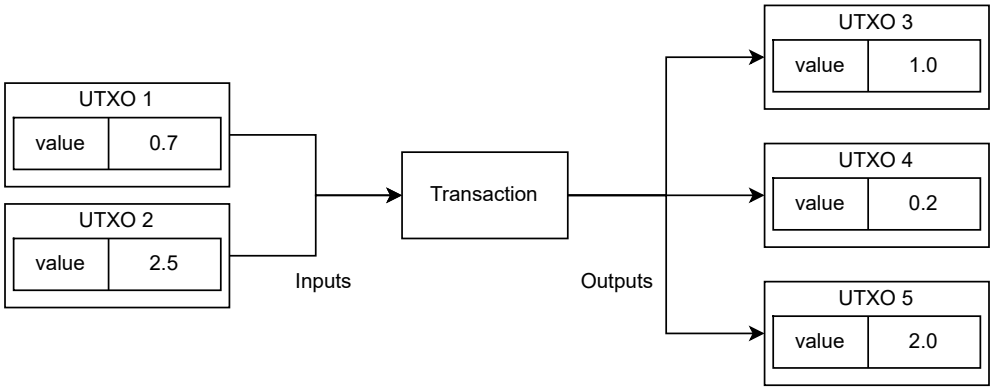

Figure 3.8: The example of two input UTXOs transformed into three new UTXOs.

Unlocking the UTXO is done through the unlocking script. The unlock script is a script that satisfies the conditions specified by the lock script and is provided by the account in the transaction input section along with a reference for every single previous UTXO.

The blockchain node needs to execute the unlocking script alongside the locking script and assess whether the locking script is satisfied. If the script is satisfied, the transaction is valid and can be inserted in the new block. The input UTXOs are spent, and the new UTXOs are ready for receiver use.

3.2.2 Account Balance Based Model

In this model, the accounts together represent part of the state of a chain. This state is saved in a block on the chain. Inside the account, there is an attribute that holds the current account balance. An example transaction in this model is visualised in Figure 3.9 and contains the following [52]:

- **Recipient**: Address of the transaction recipient.
- **Value**: A scalar value equal to the number of coins transferred to the recipient.
- **Sender**: Signature of the transaction, which is used to determine the sender of the transaction.

The transaction then needs to specify from which account and how much coin value will be transferred to another account, along with other blockchain-specific attributes.

This approach is simpler than the UTXO approach and can be easier to grasp. The only downside is that the blockchain needs to prevent multiple attacks, such as a replay attack. It is also more prone to bugs because there can be two or more concurrent transactions.

Client software (e.g. wallet) for this blockchain can find the balance simply by locating the account balance attribute in the latest block state. In addition, more memory is saved by this approach.

3.3 Wallet, Address, and Account

A wallet is a client-side application that maintains accounts on the blockchain, addresses, transactions, and pairs of public-private keys. We can categorize and review existing key management solutions [25]:

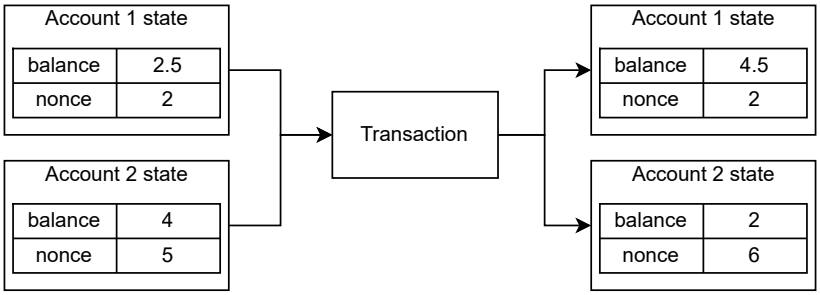

Figure 3.9: The example of an account-based model transaction between two accounts, in which account 1 sent 2 coins to account 2.

- **Keys in Local Storage**: Private keys are stored in plaintext form on the local storage of the machine. Thus, provide $(1 + 0)$-factor authentication.
- **Password Protected Wallets**: Require a user-specified password to encrypt a private key stored on the local machine. Thus, provide $(1 + 1)$-factor authentication.
- **Password Derived Wallets**: Can compute a sequence of private keys from only a single mnemonic string or password (seed). Thus, provide $(1 + X_1)$-factor authentication, where usually the $X_1 = 1$.
- **Hardware Storage Wallets**: These wallets include a device that can only sign transactions with private keys stored inside sealed storage, while the keys never leave the device. Thus, providing $(1 + 1)$-factor authentication.
- **Server-Side Wallets**: An example of this wallet is the Binance exchange. The wallet is hosted on a given service, the user logs in and authenticates himself against the Binance server using a password, and obtains a code from SMS or the authenticator service. When the service uses two-factor authentication, the wallet provides $(0 + 2)$-factor authentication. With disabled 2-factor authentication and only a password authentication wallet provides $(0 + 1)$-factor authentication.

An account in the blockchain is usually represented as a combination of keys, in most implementations, these keys are the public-private key pair mentioned in subsection 2.2.2. These key pairs are never stored on the network and are completely independent of the protocol. Typically, key pairs can be generated outside the Internet, which provides greater security. This approach enables many security properties, such as decentralized trust and control built on top of the cryptographic-based security model.

The address is a string that is produced from the public key of a user's key pair. In most cases, it is the result of combining multiple hash functions. This address can then be provided to another party as a representation of the owner of the public-private key pair, and the other party will then include this address in a transaction as an identification of the funds-receiving party.

3.4 Consensus and Network

Consensus in the context of a decentralized system can be defined as a set of principles and techniques that allow nodes in a distributed network to achieve perfect agreement on

shared data. In the context of the blockchain, consensus is used to achieve the same chain state among all participating nodes.

More precisely, the consensus algorithm needs to determine when and who will publish the next block in the chain between mutually distrusting nodes.

3.4.1 Proof-of-Work

In the proof-of-work model, a participating consensus node is called a miner. The next block is published by a miner who is the first to solve a computationally difficult puzzle. The act of finding the solution is proof that they have done their job, which means that they have provided the resources for the chain to work correctly. The puzzle is built in such a way that the solution is hard to find but easy to verify.

The most widely used puzzle method requires that the hash of the block header be less than a target value (called difficulty). Participating nodes will first build the block by adding valid transactions submitted by the users mentioned in section 3.2. The participating mining nodes will then attempt to change the value inside the header block (called nonce) so that the hash is less than the difficulty. Repeating this attempt many times is a computationally intensive process.

Since the blockchain works in such a way that the block is produced every T seconds. The difficulty must change with time to maintain this block publishing time T because the more computing power available increases over time. Usually after N blocks, participating nodes recalculate the difficulty based on the average time block for the last N blocks [53].

Once the node finds the hash solution, it will send the block to its known peer nodes, which can easily verify the puzzle solution. If it is correct and the block is valid, the peers will stop mining on the current block, append this block to the local copy of the chain, and resend this block to their node peers, this way every node acknowledges the existence of the new block.

3.4.2 Proof-of-Stake

Proof-of-Stake consensus is based on the idea that the more a user has invested in the chain, the more likely it is that he wants it to succeed, and the less likely he will be malicious. The nodes that participate in this consensus are called validators.

The invested value is called the stake, and it is usually a sum of cryptocurrency for a given chain. This stake is locked into a contract or other programmable transaction and cannot be spent for some given time. These consensus chains use the stake amount as a determining factor in selecting the validator that will be next to create the new block. Therefore, the probability of being selected to promote the new block is related to the ratio of the staked currency proportional to the total amount of the staked currency [53].

The approaches to using stake can vary, and there are many proposed algorithms for this, such as:

- **Random Selection of Stake Users**: Single validator is selected based on its stake amount.
- **Byzantine Fault Tolerance Proof-of-Stake**: Multiple validators will be proposed to create a new block, with several rounds of voting on a final block choice. Every node must be connected to each other, which causes a bad scaling of the network [27].

- **Coin Age Proof-of-Stake**: Validators obtain a cooldown timer on stake coins after promoting a new block.

This approach for consensus is less resource intensive since the validators do not need to compute any puzzle, thus not doing any work. The trade-off is the higher complexity and thus more space for errors and exploits. Furthermore, the global allocation of funds must be different and planned before the deployment of blockchain. Currency must be already distributed among users from the start rather than being progressively generated for each new block like in proof-of-work.

3.4.3 Incentive

For consensus algorithms to work, there is a need for an incentive for nodes to be reliable, so there are always more legitimate nodes than malicious ones. Each participating node is motivated by a financial reward, not by the well-being of other nodes or the whole blockchain itself.

In most cases, the consensus systems in blockchain promote non-malicious behavior by rewarding the miners or validators with their native cryptocurrency. The average reward for a participant must always be higher for non-malicious behavior than the malicious one, otherwise, the node will act maliciously to maximize its profit [53].

3.4.4 Network Model

As previously defined, the blockchain is a distributed decentralized ledger. All participants are connected through a peer-to-peer network without the need for any centralized identity. The nodes communicate through the protocol defined by the blockchain specification.

There are usually two main types of participants in this protocol, the node and the miner (validator). The node works as an entry point to the network, it replicates and extends the chain with the fresh blocks from miners, and any client can request this node for information about the chain. The node only stores the data.

There are two main types of nodes. The light and full node, light nodes keep only the block headers without all of the chain data. The full node has complete information about the chain [38]. The miner works to progress and extend the chain, as previously mentioned in consensus algorithms.

3.5 Security Concerns

Multiple security concerns must be taken into account when working with or designing applications around the blockchain.

3.5.1 Forking

In proof-of-work consensus, in decentralized networks, there is the possibility that two or more miners will simultaneously generate two or more valid blocks (solving the puzzle). Since the propagation of this block is done through flooding, it is not deterministic which of the new blocks will arrive sooner at the other mining nodes. This causes the node to obtain two possible versions of the chain, called a fork.

In this situation, all mining nodes must start working simultaneously (search for the puzzle) on all forks. If the next block is found, the longest chain rule is applied. The rule

can be defined as a chain that is longer is judged as the authentic one. Then all of the miners will switch to the longer branch. Discarding the fork means that the transactions of the users in a given block will be reverted, which can be critical to the use of the blockchain [53].

This property is a great disadvantage because the miners need to use part of their computing power to solve the puzzle on the chain fork, which in the future will be discarded.

3.5.2 Finality

Finality refers to the time from which it is impossible or highly non-probabilistic to rollback a block previously added to the chain, thus reversing the transactions. Finality can be defined as a time value (seconds) or as a number of confirmations, where confirmation can be defined as the number of blocks on top of the given examined block.

Most crypto-asset-based blockchains guarantee an eventual finality with some probability. The probability of reverting to a given block decreases as more blocks are added on top of a given block. The finality is tightly coupled with latency, propagation delay, and possible attacks on the consensus of the blockchain [2].

3.5.3 51 % Attack

Attack tightly connected to the proof-of-work consensus mechanism. If a malicious party has the computing power (hashing power) of the entire network greater than or equal to 51 %, it is capable of arbitrarily manipulating and modifying blocks (transactions) on a blockchain. With 51 % power, malicious party can execute multiple attacks such as [4]:

- **Double-Spending Attack**: Spending the same coins multiple times and deceiving the other party.
- **Transaction Reordering**: Modify the order of the transactions in the block. Also known as a miner/maximum extractable value (MEV) [15].
- **Transaction Censorship**: The malicious party can prevent the confirmation of given transactions.

This attack is hard to execute in well-established networks (such as Bitcoin and Litecoin), since acquiring that much computational power is expensive. However, it is very possible to run this on smaller networks that are less known. Attacks like this occurred multiple times on networks such as BitcoinSV or Ethereum Classic.

This attack is also possible to define for proof-of-stake consensus. In this situation, the malicious party would need to own 51 % of the total supply. On well-established networks (such as Ethereum), this would also be difficult to execute. Buying the 51 % currency total supply would become expensive as the high demand for the coin increases its price. Furthermore, this attack is possible to mitigate from the launch of the network, by locking 51 % of the supply or distributing the total supply to different parties or use cases.

3.6 Implementations

There are already multiple implementations of blockchains, of which the most popular and used are Ethereum and Bitcoin.

3.6.1 Bitcoin

Bitcoin is one of the first successful blockchain implementations that started to use the blockchain for digital money.

Bitcoin for account management (keys) uses an elliptic-curve-based approach for asymmetric cryptography. More precisely, it uses the secp256k1 parameters of the elliptic curve. To ensure that funds can only be spent by their rightful owners, Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) for the digital signature [3].

Regarding transactions, Bitcoin uses a UTXO-based approach. This approach is tightly coupled with programmable transactions. For each UTXO in Bitcoin, it must be locked and unlocked as described in subsection 3.2.1. For transaction locking, Bitcoin proposed and uses Script. The script is a simple, stack-based, and intentionally not Turing-complete scripting system [3].

Bitcoin uses a proof-of-work approach for consensus, which is starting to get heavily criticized because of its environmental impact (huge waste of electricity). Each block in the chain contains the list of transactions organized within the Merkle tree for easy proof creation. A block should be published every 10 minutes. Bitcoin uses a probabilistic finality, since the deeper the block, the more unlikely it is for it to be reverted. With this in mind, the finality in Bitcoin is considered 6 blocks (60 minutes). This number is based on the assumption that an attacker is unlikely to acquire more than 10 % of the mining power and that a risk less than 0.1 % is acceptable [38, 42].

3.6.2 Ethereum

Ethereum can be defined as an alternative protocol (blockchain) for the development and implementation of decentralized applications. Ethereum does this by building an abstract foundational layer, a blockchain with a built-in Turing-complete programming language. Any user (account) can write its own smart contract.

The same as Bitcoin, Ethereum uses secp256k1 elliptic-curve-based asymmetric cryptography for account keys and ECDSA for digital signatures. Regarding the consensus, Ethereum recently switched from the proof-of-work model to the proof-of-stake model [20].

From a transaction perspective, Ethereum uses an account balance approach. The whole state (blocks) is made up of account objects and state transitions (transactions). Ethereum uses a more sophisticated analogy, instead of blockchain, it portrays itself as a distributed transaction-based state machine. Rules that specify how the state changes from block to block are defined by the Ethereum virtual machine (EVM) [52].

Ethereum uses a Merkle-Patricia tree for storing the state and mapping addresses to the account objects. Then the block can store only its root hash and transactions in a given epoch. There are now two types of account, externally owned accounts which are controlled by the public-private key pairs (user), and contract accounts which are controlled by their smart contract code. The account then consists of [10, 52]:

- **Nonce**: A scalar value equal to the number of transactions sent from this address (preventing replay attacks).
- **Balance**: A scalar value equal to the number of Ethereum coins owned (also called Wei).

- **Storage Root**: A root hash of the Merkle-Patricia tree that encodes the storage contents of the account. The tree maps a hash to the integer value. Represents the memory of the contract.
- **Code Hash**: Hash of the code of account if it is a contract account, otherwise empty.

Transactions within Ethereum can only be constructed by externally owned accounts (users). The transaction can result in the creation of a smart contract (contract account, contract deployment) or the message. The message can result in a transfer of funds between externally owned accounts or in an invocation of a smart contract. To prevent denials of service attacks, Ethereum uses a fee to run a transaction similar to Bitcoin called gas. For every byte of transaction data, 5 gas is required.

Smart Contract

Since Bitcoin UTXO scripting is a weak implementation of the concept of smart contracts, Ethereum creates a more robust and less limited way to create decentralized code on the blockchain. The smart contract can be defined as an entity inside the Ethereum blockchain (execution environment) that can only execute specific code. As we have previously mentioned, a smart contract is represented inside the state as a special type of account that has its balance and memory (storage tree).

The contract code is executed inside an EVM. EVM is a runtime environment for smart contracts, it is a Turing-complete, stack-based architecture. Contracts are compiled from specific contract programming languages (such as Solidity) into an EVM bytecode that consists of instructions. The memory within the contract can be temporary and persistent. The temporary memory is only available at the execution of the contract, and after its run, it is wiped. Persistent memory uses the mentioned storage trie within the account object in a global state and can be used during multiple contract runs [52].

Because the EVM is Turing complete, to prevent infinite loops or other computational waste, the EVM uses gas as a fundamental unit of computation. A single operation can cost 1 gas or more, depending on the difficulty of computation. Also, the usage of storage cost gas.

The contract can send messages to another contract. Messages are virtual objects only inside the EVM. Like a transaction, a message invokes a contract that runs its code. Thus, the contracts can depend on the other contracts.

Reverting the message works the same as a transaction, if the execution of the message runs out of gas, then that message execution and all other executions triggered by this message are reverted.

3.7 Decentralized Identity

With the growing use and availability of blockchains, a new approach to digital identity is possible. Decentralized identity is a new instance of self-sovereign identity (SSI), a type of identity management that allows people to control their own digital identity without being dependent on the given identity provider. The decentralized identity is built on three pillars: verifiable credentials, decentralized identifiers, and the blockchain.

3.7.1 Decentralized Identifiers

Standard developed by the World Wide Web Consortium (W3C). Decentralized identifiers are identifiers that can be used to transfer credentials and authenticate. The ownership of the given identifier is proven by demonstrating the possession of the private key associated with the identifier.

The term DID is defined by the W3C as an instance of a uniform resource identifier (URI) in the following format [43]:

```
did:<method-name>:<method-id>
```

Every DID needs to be unique in the whole space since a single DID is bound to a single subject. The name of the method specifies how to create, update, and remove DIDs using that method. A single DID is bound to the DID document, which needs to be resolved by the DID resolver, and contains data, such as public keys, and personal information about a given DID subject. The document can also describe the mechanisms through which the DID subject is reached and how to engage in secure communication with him. The document is usually stored in JSON format in a public, reachable, and resolvable place [33, 44].

The subject can have multiple DID identifiers bound to him at the same time, which are used for different purposes. The subject can then authenticate himself at other parties by providing them the DID bound to him and a digital signature made by a public key bound to the DID. Then, the party resolves the DID document and verifies the required set of claims and digital signatures using a public key from the document.

In a real-life example, a subject can create his own decentralized identity on the Ethereum blockchain using his public key generated from his wallet. The decentralized identifier can be defined as:

```
did:ethr:<eth-public-key>
did:ethr:0xb9c5714089478a327f09197987f16f9e5d936e8a
```

The subject will upload a DID document on the chain, filled in with verifiable credentials and other information inside a specific smart contract memory. The subject can then provide the signature made by the public key and the DID to the authenticating party.

3.7.2 Verifiable Credentials

Because a decentralized identifier can be self-signed and easily forged, it is necessary to have a certified third-party issuer, which can issue cryptographically signed information bound to the DID. It is used to build trust between the parties involved in an SSI ecosystem [33].

The verifiable credential (VC) is issued to make claims about a subject. A single credential can hold many claims about the subject. The issuer is responsible for creating and specifying the credentials content, and most importantly, for the verification method of these claims. The issued credential is then maintained by the subject, which binds the credential inside the selected DID.

The credential is made up of three main parts that describe it. Credential metadata describes the issuer, expiration, date, and other information. Second, it contains statements about the subject in the form of one or more claims expressed as property-value pairs. The last is proof that the credential is valid through a digital signature. All of these data are serialized within the JSON structure [44].

Since verifiable credentials are used in conjunction with DID to make an attestation about the DID subject, the verifier party must establish trust with the issuer. If trust is

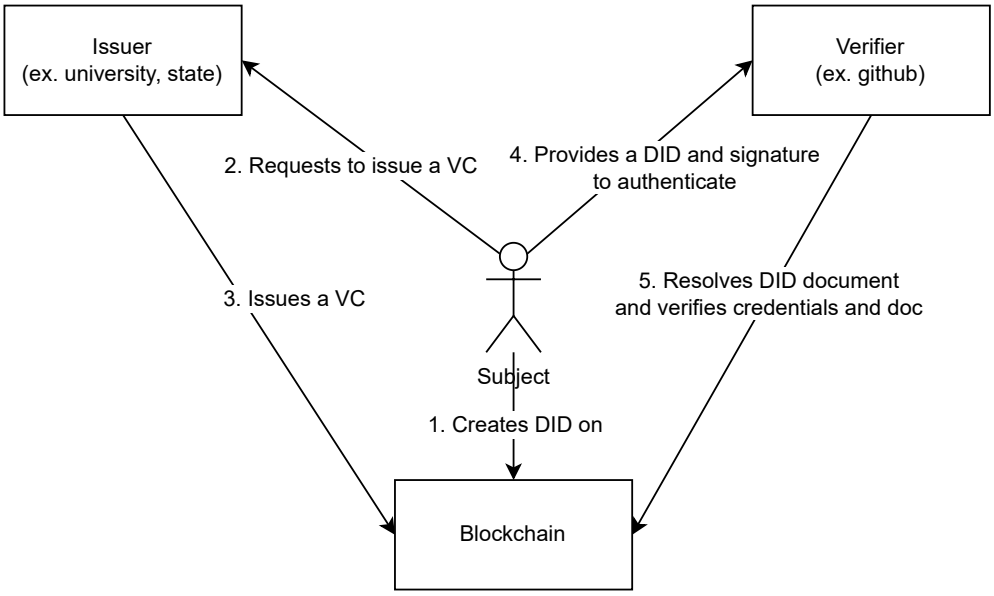

Figure 3.10: Full DID and VC authentication workflow, using blockchain as a storage of DID documents.

established, the subject can provide the DID, and the verifier checks the credentials and their signatures if they are signed by the issuer. If the signatures are correct, the verifier can trust the claims in the credentials about the subject [44].

The most important advantage of this approach is that the issuer does not need to be present when authentication occurs between the subject and the verifier. The complete authentication workflow, which uses blockchain as storage of DID documents, is visualized in Figure 3.10.

Chapter 4

Cryptocurrency Exchange

Exchange is the first way to obtain cryptocurrency for the usual user. Users can exchange their cryptocurrency similarly to customers who can exchange fiat currency in a bank. Every exchange offers specific coins and provides a service in which users can make transactions that will or will not be reflected on the blockchain later on. Usually, if the user follows the security policies of a given exchange, he can buy coins without having a deep knowledge of blockchains.

Depending on how much the third party is involved, there are two main types of cryptocurrency exchange.

4.1 Centralized Exchange

Such exchange is a platform maintained by a third party or an intermediary to sell and buy digital assets. The customer needs to trust the given third party that the funds are safe. In most of the implementations, the party has full access to the customer funds in the exchange wallet, and the amount of customer funds is virtually reflected inside the exchange system as a simple number.

The main advantage of this type of exchange is that there is a possibility to implement any type of coins and tokens to be exchangeable. Since trades are being executed inside the exchange system, there is no need to execute the trades between the chains or immediately reflect the state of the customers' funds on the given chain, and everything can be kept off-chain.

Exchange can reflect the state of the customer on the chain only at the customer's request. For example, when the customer wants to withdraw funds, the exchange needs to create the transaction on the given blockchain and execute it. In addition, this way trades can be real-time and executed as fast as possible with much fewer fees, which in the end benefits both exchange operators and customers, as the exchange can define its fee system, which can take a smaller percentage of funds than a fee on blockchain for miners.

One of the largest centralized exchanges is Binance, which provides over 600 cryptocurrency coins and tokens, and it has one of the lowest fees for trades. Binance has a multi-cluster architecture with high processing throughput, which can process hundreds of orders per second.

Binance and other popular centralized exchanges are required to perform the first stage of anti-money laundering due diligence, called „know your customer“ (KYC). When a new customer wants to use the exchange, he is immediately requested to provide his identity

in the form of an official document (such as an ID) and some other personally identifiable information. Exchanges do this to mitigate potential legal and regulatory risks. It is possible to purchase currency without performing KYC, some exchanges allow trading currency without KYC, and implement some sort of daily limit limitation for users without it.

4.2 Decentralized Exchange

This type of exchange allows the buyer to buy and exchange cryptocurrencies without intermediaries, which means that the user does not need to deposit his funds in a third-party wallet. Most of the time, the exchange is implemented on a smart contract platform of a given blockchain.

The main disadvantage of smart contract-based exchange is the fact that there is only a possibility to trade the coins implemented on top of the smart contract layer (tokens). This is caused because the smart contract layer does not have any information about other blockchains (ex. new transactions, blocks). After all, smart contract evaluating machines run in isolated environments. Furthermore, this smart contract approach would need to implement a cross-chain atomic swap protocol, which would be impossible to implement without information about other blockchains. Users are also not able to trade in real-time since a single transaction inside of this exchange needs to be reflected in a given blockchain transaction, the timing of the transaction depends on the blockchain transaction finality and fees.

One of the most popular decentralized exchanges based on smart contracts is called Uniswap, which is built on top of the Ethereum smart contract platform. More precisely, Uniswap is an automated liquidity protocol that does not provide any order book with high censorship resistance. Uniswap can only swap Ethereum-based ERC-20 coins.

4.3 Security Concerns

Since the exchange holds all customer funds, a lot of effort must be put into the security of such a platform. A security breach of such an exchange potentially decreases the value of given currencies and the reputation of the exchange. Therefore, we need to define the most common attack vectors for such exchanges.

In centralized exchanges, potential threats can come from internal and external parties and are identical to the threats for server-side hosted wallets mentioned in section 3.3, since centralized exchanges provide these wallets. So far, centralized exchanges have been the most attractive target for malicious parties that caused huge financial losses. There are many security countermeasures proposed such as 2+ factor authentications, a split of exchange funds between hot and cold wallets, and usage of multisignature wallets [27].

4.3.1 Stolen Credentials

In one of the very early attacks, the malicious actor breaches the exchange platform and continuously withdraws funds using stolen privileged credentials. The feasibility of this type of attack is password reuse and the overall bad habits of storing passwords. In traditional finance services, stolen user credentials are mostly used to steal funds from an individual given user.

In the context of a centralized exchange, these accounts have administrative powers that affect the funds of multiple customers. In recent years, this occurrence of attacks has decreased, implying that the security of exchanges around secret storage has increased [39].

In the context of a decentralized exchange, the stolen account has only access to his funds, thus an attacker can steal only the funds of a single customer.

4.3.2 Insider Attack

This type of attack is tightly connected to a centralized exchange. In this attack, someone with legal access and permission within a system causes damage using those legal permissions. It may not be possible to detect or mitigate risk without allowing an insider to act on its own. This form of threat is only possible if the insider has enough control over the system to do real damage. There are many potential instances of this attack, for example:

- Ownership of large amounts of cryptocurrency that the exchange supports.
- Ownership of an account on the exchange that can access user funds.
- Permission inside the exchange to access wallet cold or hot wallet private keys.

Potential insider attacks are created, but can also be mitigated by the design choices of the exchange. For example, open-sourcing the code, not allowing insiders special privileged accounts, not allowing insiders access to the private keys, etc.

4.3.3 Abuse of Functionality

Attack where the attacker effectively abuses the native functionality of the exchange platform or the exchange hosting platform, this functionality was implemented to meet a specific use case. This attack vector was dominant between 2011 and 2014 [39] and is possible for both types of exchanges.

An example of this attack is the MyBitcoin eWallet incident. A malicious party was able to forge Bitcoin transactions and withdraw confirmed older bitcoins on the eWallet. This was caused by a human error and misunderstanding of how Bitcoin transactions work, programmers were assuming that 1 confirmation is enough for a transaction [50].

4.3.4 Abuse of Exploit

An attack vector that abuses the presence of unintentional functionalities in the exchange code. In most cases, this may be a bug, such as race conditions, deadlocks, etc. Furthermore, exploits of malicious backdoors placed in open-source software dependencies can be exploited [39]. This attack vector possibly exists on both centralized and decentralized exchanges.

Chapter 5

Intel Software Guard Extensions

Since the last few decades, the adoption of cloud computing by corporations has steadily increased. The cost and technical advantage bundled with these services resulted in outsourcing the infrastructure. In most cases, these companies that use these services have very few legal contracts with respect to resistance to manipulation of the hardware or virtualization layer.

A trusted execution environment (TEE) is a security feature with the promise of minimizing the possible surface against powerful attackers (for example, the cloud provider). The environment is done through hardware, in most cases through the CPU. TEEs use a root of trust throughout the hardware to enable memory access through fine-grained access control, protection of executed code, and the state of the machine [24].

In addition, TEEs offer a mechanism called attestation to establish a trusting relationship with a specific software by verifying its authenticity and integrity. There are two main types of attestation, called remote and local. Throughout remote attestation, one party ensures that it is communicating with a specific, attested program remotely [37].

There are multiple hardware and software implementations of TEEs by multiple vendors such as SGX (Intel) [30], TrustZone-A/-M (Arm) [6], Keystone [35], and more. There is no clear standardization of the implementation and attestation of these TEEs. From now on, we will focus on the features of the Intel TEE implementation, the Software Guard Extensions (SGX).

5.1 Architecture

Intel introduced SGX TEE for the mass market in 2015, specifically, the Intel Skylake architecture introduced a new set of processor instructions for the creation and usage of encrypted regions of memory called an enclave.

In the Figure 5.1, we can see the high-level overview for Intel SGX. The enclave is located within the user space of the process and is completely separated from the other components. The enclave can be attested and has a call gate from which the application can call the attested code.

There are currently two instances of SGX, the client (also called SGX1, released 2015), and scalable (also called SGX2, released 2021). SGX2 focuses mainly on server-grade processors [37]. There are three main differences between these two instances, such as:

- **Available Memory in Enclave**: 128 MB (SGX1) and 512 GB (SGX2).
- **Multi-Socket Support**: It is possible to run an enclave on a multi-socket system.

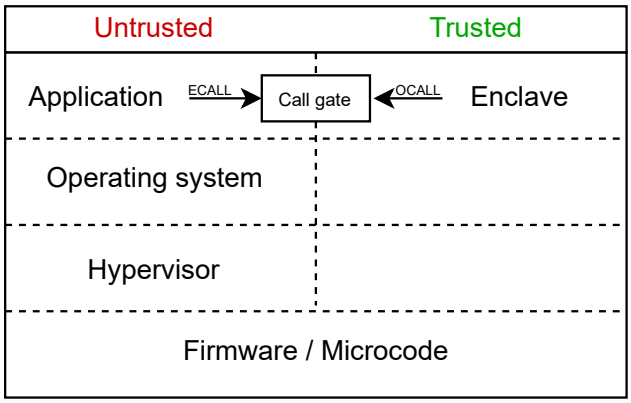

Figure 5.1: The SGX architecture and its trusted and untrusted parts, taken from [37].

- **Integrity and Replay Protections**: New ways are used to guarantee data integrity against replay attacks.

These enclave SGX instructions are implemented as their own instruction set architecture (ISA) and are grouped together with model-specific registers. A memory region is reserved at boot time to store data and enclave code. The memory region is called the Enclave Page Cache (EPC) and is inaccessible to other programs running on the same machine, including the hypervisor and the operating system (kernel). The EPC stores the checksum to ensure that the memory corresponding to the EPC was not modified by any external party to the enclave. Communication between the CPU and memory is also confidential using the Memory Encryption Engine (MEE) [28, 37].

The SGX enclave code is completely separated from the untrusted application code, as shown in Figure 5.1, using the call gate. The application can invoke the enclave code using the `ECALL` system call. The enclave is possible to call the untrusted code of the application by issuing an `OCALL` system call.

Intel provides for each SGX processor two special unique keys, called Root Provisioning Key and Root Seal Key. Both keys are engraved inside the processor at the manufacturing time. The Root Provisioning Key is randomly generated and fused into the processor at manufacturing time and is used to demonstrate to Intel that the processor is a genuine SGX device. This means that the key is also stored inside the Intel data centers. The Root Seal Key is also created at manufacturing time and fused into the processor, the difference is that this key is not saved by Intel and every trace of it is removed. As shown in Figure 5.2, this key serves as some master key for the derivation of all other keys that the enclave can generate. Enclave has a special set of algorithms to generate various cryptographic keys. A developer can request to generate a key through the instruction `EGETKEY`, and every possible cryptographic key is based on many enclave metadata such as measurement, security version number (SVN), etc [28, 32].

For the enclave to run, it needs to be signed, this is critical for ensuring security for the SGX applications. Only signed enclaves can be loaded into the EPC memory. The enclave signature structure contains a measurement of the enclave code at the build time. Another measurement is built dynamically at the enclave creation load time, and then the signature structures are compared to detect any changes that were made to the enclave. If a change is detected, the enclave cannot be run.

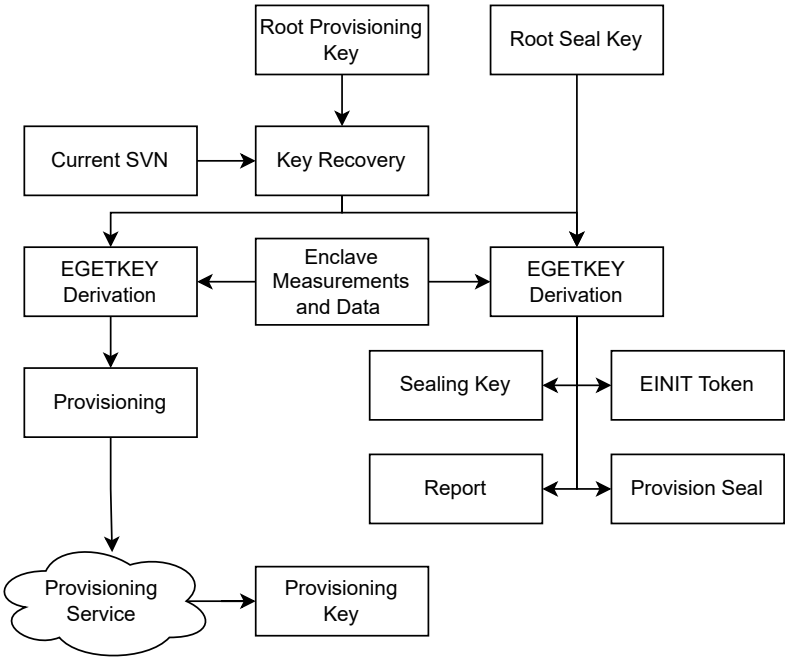

Figure 5.2: The Intel SGX Key Hierarchy, taken from [32].

An enclave measurement is a single hash that identifies the initial data and the code that is being placed inside the enclave. The order and position in which they are placed need to be exact. Any change to an order or any of these variables will result in a different hash.

In addition to this, the enclave can run in production, debug, or pre-release mode. For the enclave to run in production mode, the enclave signer must be whitelisted by Intel, otherwise, the enclave will not run. These ISV signing keys need to be stored securely, otherwise, a malicious actor could sign any enclave code and thus attest to it. For the debug (or pre-release) mode, instead of signing keys, developers can use their own generated development private key [30].

5.1.1 Architectural Enclaves

For previous architecture to work, Intel provides a few architectural enclaves. The architectural enclave is an enclave signed by Intel, has special privileges on the SGX platform, and makes important preparations for the platform to work. There are three main architectural enclaves [29, 32, 45]:

- **Provisioning Enclave**: Used in an EPID-based remote attestation scheme. Enclave contacts the Intel Provisioning Service (IPS) at the first SGX deployment to acquire the attestation key. Enclave demonstrates to the IPS that it is running on a genuine Intel SGX processor (using Root Provisioning Key) and joins the EPID group decided by the IPS. The attestation key is stored inside the enclave and sealed with the provisioning seal key.
- **Launch Enclave**: Examines whether the application enclave that is being launched is valid by checking the signature made by ISV keys (the keys must be whitelisted by

Intel for the production enclave) and measurement. The launch enclave generates a token (`EINITTOKEN`), which is passed to the `EINIT` instruction to initialize the enclave.

- **Quoting Enclave**: The enclave verifies the local attestation with the application enclave. The result of the attestation, the report (`EREPORT`) is signed by the quoting enclave and transformed into a quote (`QUOTE`), which is returned to the challenger in an act of remote attestation.
- **Provisioning Certificate Enclave**: Acts as a local Certificate Authority (CA) for local quoting enclaves, which run on the same SGX platform. Only used for DCAP-based attestation. The enclave authenticates the quoting enclave and issues a certificate-like structure for the quoting enclave.

5.2 Local Attestation

Local attestation allows a trusted environment to prove its identity to other trusted environments hosted on the same system, on the same CPU and only if the security secret for the attestation is linked to the CPU. The environment which receives the local attestation request makes sure the issued proof is genuine, usually, the proof is based on symmetric encryption using a message authentication code.

This attestation can be used to establish secure communication between enclaves, e.g. for secure delegation of tasks. In an Intel SGX remote attestation, the local attestation is used to sign proofs from another enclave via a secure communication channel.

The proof provided by the enclave is called the report, and it is a cryptographic proof that the enclave exists on the same platform. The enclave report consists of the following [29]:

- Measurement of code and data in the enclave.
- User data.
- Hash of the public key in the ISV certificate.
- Other security-related information.
- Signature block from previous data.

The verifying enclave can easily verify that this report was created on the same platform because there is only a single device key for a given platform. The downside is that only enclaves can verify this, and not the code outside the enclave. The algorithm for local attestation, in which the two enclaves authenticate each other can be defined as [29]:

1. Application X hosts enclave A and application Y hosts enclave B. Applications establish a communication path between the two enclaves.
2. Enclave B sends its measurement to enclave A.
3. Enclave A asks the SGX hardware to produce a report for enclave B using the measurement value received from enclave B and sends the report to enclave B.
4. Enclave B requests the SGX hardware to verify that enclave A is on the same platform, so the report is valid.

5. Enclave B asks the SGX hardware to create a report from the measurement value received from enclave A (in the report) and sends the report to enclave A.

6. Enclave A verifies that the report from enclave B exists on the same platform.

5.3 Remote Attestation

Remote attestation is used to establish trust between different platforms and provides cryptographical proof that the specific executed software is being run genuine and is untampered. Once the platform attested is proven to be genuine, the verifying party can proceed to communicate safely with the attested platform and provide it with confidential data and computations [37].

In the SGX architecture, Intel implemented two remote attestation methods, EPID-based and DCAP-based.

5.3.1 EPID-based Attestation

EPID-based attestation is Intel's first attempt and implementation of remote attestation. This method is based on the already existing proof of knowledge protocol called SIGMA and is extended into the enhanced privacy ID protocol [37].

EPID allows an SGX platform to sign data without uniquely identifying the platform or without linking different signatures. Each SGX platform belongs to an EPID group, and the verifying party is using the group public key to verify the signatures. In the context of SGX, the EPID group is a set of processors of the same type, the typical size of the fully populated group ranges from thousands to a few million [32].

The algorithm for EPID-based attestation needs to use both Intel architectural enclaves, the quoting enclave, and the provisioning enclave. The EPID algorithm can be defined as:

1. The verifier sends a challenge to the application enclave.

2. The application enclave creates an ephemeral public key and performs a local attestation with the quoting enclave.

3. The quoting enclave signs the report (`REPORT`) by requesting the EPID attestation key through the instruction `EGETKEY`. This key must be already resolved by the provisioning enclave before proceeding, otherwise, the attestation cannot progress.

4. The quoting enclave transforms the report into a quote (`QUOTE`) by signing the report with the attestation key and sending it back to the application enclave.

5. The application enclave sends this quote along with the public key to the verifier. The ephemeral public key enables the creation of a confidential communication channel.

6. The verifier contacts the Intel Attestation Service (IAS) and provides that quote for verification.

7. IAS verifies whether the quote signature is valid thanks to the previously agreed EPID attestation key.

8. If deemed trustworthy, the verifier can start communicating with the application enclave using the ephemeral public key and provide sensitive data and operations.

There are multiple disadvantages to this attestation schema:

- Applications running in a peer-to-peer fashion are dependent on a single point of verification (IAS).
- The infrastructure cannot run entities in environments where the Internet service cannot be reached at runtime.
- Entities can be risk averse in outsourcing trust decisions to a third party (IAS).

5.3.2 DCAP-based Attestation

DCAP-based attestation tries to address the main disadvantages of EPID-based attestation. The third-party using Intel SGX and this attestation may now build their attestation infrastructure, using asymmetric cryptography algorithms such as RSA or ECDSA. In this way, the usage of IAS drops off.

The algorithm is similar to the EPID-based attestation in subsection 5.3.1. The quoting enclave can generate its attestation key, using the preferred algorithm. Then the enclave provides the attestation key to the provisioning certificate enclave, which establishes the local attestation with the quoting enclave. The provisioning certificate enclave issues a certificate-like structure identifying the quoting enclave and the attestation key. The certificate is signed by the Provisioning Certificate Key (PCK). Intel provides and publishes certificates for these keys for all genuine SGX platforms. These steps result in a full chain of trust from the quote to the Intel Certificate Authority. The resulting quote can be verified by a party that needs to hold the entire certification chain, starting from the PCK certificate to the Intel CA [45].

5.4 Sealing

The secrets inside the EPC are lost when the enclave is closed. For the secret to be preserved for future use in the enclave, it needs to be stored outside the boundary of the enclave before the enclave is disabled. The secret before saving on disk needs to be encrypted by a key that can only be obtained by an enclave. This process of encrypting and decrypting with a given enclave key is called sealing and unsealing.

The key for sealing is called the seal key, and the main master key for generating this key is the root seal key. There are two possible ways to seal the data, to the current enclave or to the enclave author (ISV).

Sealing to the current enclave uses application enclave measurement when requesting a seal key through the operation `EGETKEY`. Only the enclave with the same measurement can unseal the data. This means that any changes to the enclave will result in different seal keys, thus the updated enclave will not be able to unseal these data.

Sealing to the author of the enclave uses the signature of the enclave author (using ISV keys), which the enclave stores inside the register at the enclave inception time. A signature is used as a parameter to the operation `EGETKEY`. The key is also bound to the Product ID of the enclave. The main advantage of this sealing is that the enclave allows the author to upgrade and does not include a complicated upgrade process for sealed data migration from the previous enclave. In addition, it allows enclaves from the same author to share the data. A simple seal algorithm can be defined as [29]:

1. Verify that the input is valid.

2. Populate the key request structure used in the `EGETKEY` operation by a given sealing type, to obtain the seal key.
3. Call `EGETKEY` to obtain the seal key.
4. Call the encryption algorithm to perform the encryption with the seal key. Intel recommends encryption with AES-GCM, such as `Rijndael128GCM`.
5. Delete the sealing key from memory.
6. Save the sealed data structure, including the key request structure to the disk.

Then, the reverse unseal algorithm can be described as:

1. Verify that the input parameters are valid.
2. Retrieve the previous key request structure.
3. Call `EGETKEY` with the given key request structure to obtain the seal key.
4. Call the decryption algorithm for the given ciphertext and the seal key.
5. Delete the seal key from memory.
6. Confirm that the hash created by decryption matches the hash created by encryption.

Enclave developers should be aware that the SGX platform does not provide any detection of the same enclave running multiple times. This means that two or more same enclaves can access each other's sealed data.

Chapter 6

Related Work

In this section, we will discuss the existing solutions for secure centralized exchange. Furthermore, we will inspect another proposed solution for centralized cryptocurrency using a smart contract, which provides interesting design patterns that we will use for our exchange solution.

6.1 Tesseract

Tesseract [7] is a secure cryptocurrency exchange, designed to eliminate funds theft and offer real-time trading using trusted computing. Exchange supports cross-chain trading where assets are exchanged between the different blockchains, and prevents possible eclipse attacks (a malicious party presents exchange forged blockchain data, such as blocks). This is achieved by checkpointing trustworthy blocks and verifying the difficulty of newly produced blocks.

Using trusted computing enclaves allows customers to ensure confidentiality and code integrity. This enables Tesseract to behave like a trusted third party. The most interesting part of this design is exchange deposits, settlement transactions, and front-running prevention. [7].

6.1.1 Deposits

Tesseract enclave $\mathbb{E}$ generates and attests a public key $PK_{\mathbb{E}}^{X}$ for each supported cryptocurrency blockchain X. From these keys, a deposit address is calculated for each supported blockchain.

When new users want to open an account on the exchange, they need to deposit a specific amount of a given currency to the Tesseract deposit address. If the deposit transaction is valid and confirmed, the future customer must create proof of the transaction and send this proof to the enclave. The proof consists of the following [7]:

- Deposit transaction.
- Merkle path or other structure (such as Merkle-Patricia tree) proof that the transaction is inside of the block.
- Block index.

The enclave at the launch fetches for each supported cryptocurrency its blocks from the genesis or the specified checkpoint block and verifies all consensus rules (such as difficulty)

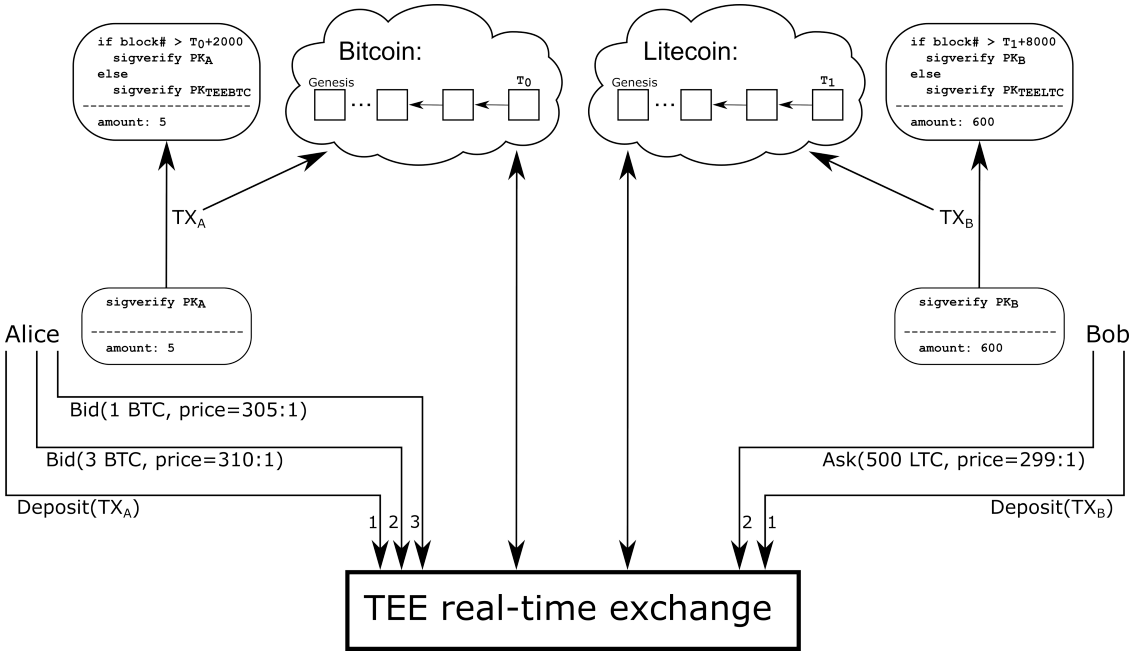

Figure 6.1: Tesseract design, portraiting deposit action, and block fetching, taken from [7].

as displayed in Figure 6.1. This prevents the malicious party from forging and feeding low-difficulty blocks. At the same time, the enclave saves the latest N blocks in a FIFO queue. The enclave can then easily verify the validity of the deposit transaction using this FIFO queue, and if the proof is valid, open the account and add the deposit funds to the account inside the enclave [7].

The user-deposit transaction on a specific blockchain must be time-locked. Before time is reached, only the enclave can spend funds in this transaction. Once the time limit is reached, the user can acquire control of the UTXO. This deposit method ensures that funds are safely restored even after the exchange disappears, but can still be stolen if exchange keys get leaked. On the Bitcoin blockchain, the UTXO can be locked using the `CHECKLOCKTIMEVERIFY` instruction specified inside the locking script [7].

This feature bears a disadvantage for real implementation, since the enclave needs to remember, for each user deposit, when it is going to expire and correctly reflect the transaction state on every blockchain inside of the exchange balance of the customer. Wrong and overdue synchronization of exchange later after blockchain could lead to possible vulnerabilities, such as double spending, where the client could spend the exchange deposit UTXO and at the same time he could make a trade inside of the enclave and withdraw the obtained funds on the other blockchain from the exchange.

6.1.2 Settlement Transactions

Because of time-locked deposits, Tesseract proposes to periodically synchronize the trades within the enclave on the blockchains it supports, these transactions are called settlement transactions.

Each 24 hours Tesseract proposes to create the settlement transaction, which must reflect every account that made a trade (within this 24 hour time window) on every supported blockchain [7]. For example, if 50,000 exchange users performed a trade that would change

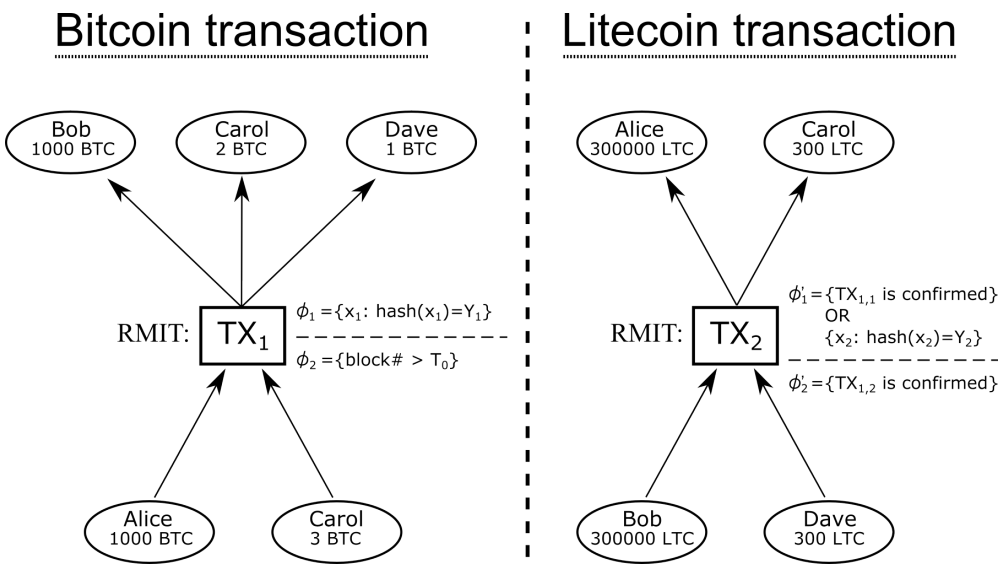

Figure 6.2: The cross-chain swap problem between two chains, taken from [7].

their balance within the exchange, at the end of the day, the enclave would need to create a transaction on at least two blockchains that would contain more than 50,000 swaps between accounts. This could be expensive in fees, since transaction fees are calculated based on the size of the data.

Since Tesseract supports trades between multiple blockchains that are independent of each other and reflects these trades periodically on-chain, a protocol that supports this atomic cross-chain swap is needed. The problem that the protocol resolves is described in Figure 6.2. Otherwise, the malicious party could steal funds from legitimate customers.

The authors of Tesseract define this problem as an all-or-nothing settlement. Given the transactions T_1^A and T_2^B for blockchains A and B, an all-or-nothing cross-chain settlement is a protocol that guarantees [7]:

1. Both T_1^A and T_2^B will become confirmed on the blockchain A and B.
2. Otherwise, neither T_1^A nor T_2^B will get confirmed on the blockchain A and B.

Tesseract proposes a practical protocol for all-or-nothing settlement that distributes trust between N servers running a trusted computing platform. The main idea of this protocol is to securely deliver all signed settlement transactions (TX_A^S, TX_B^S) and settlement cancelation transactions (TX_A^C, TX_B^C) to these exchange servers. The servers will attempt to execute the first settlement transaction on the first blockchain A, if the transaction was committed or canceled, the servers will try to do the same on the second blockchain B [7].

The cancellation transactions (TX_A^C, TX_B^C) are transactions that invalidate the generated settlement transactions. For example, in Bitcoin, this cancelation transaction can be implemented as a transaction that spends one of the inputs inside the settlement transactions into a new identical output [7].

This protocol attempts to maintain the property that the transaction TX_B^S remains secret (unsubmitted) within the server enclaves until the transaction TX_A^S is not yet confirmed [7].

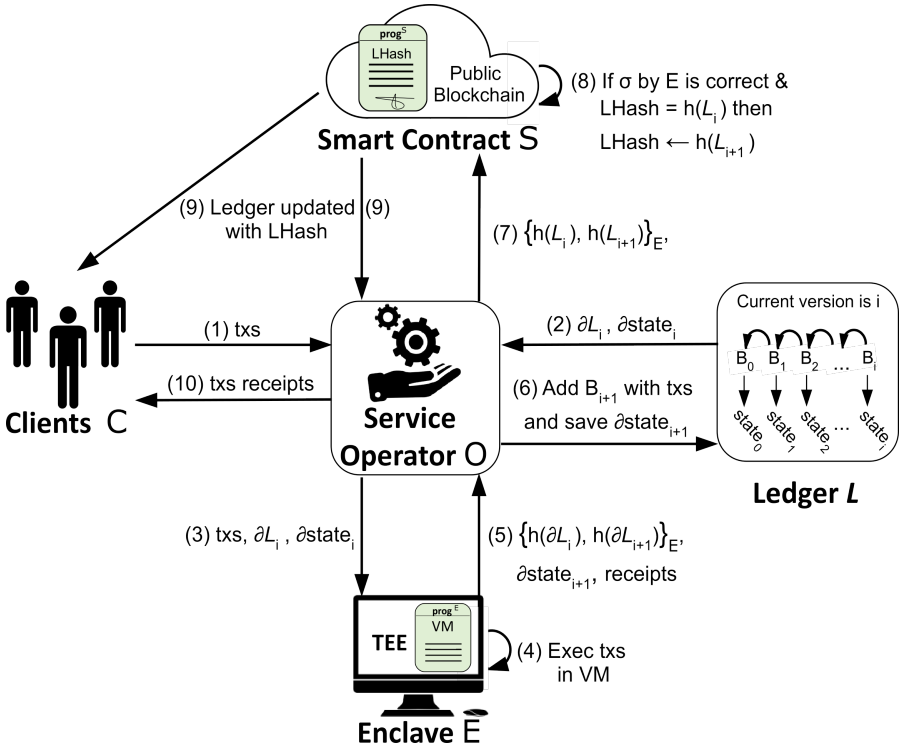

Figure 6.3: The Aquareum design, showing the operation procedure, taken from [26].

6.1.3 Front-running Prevention

For customer communication with an exchange, Tesseract requires users to use a secure channel (for example, TLS). In this way, malicious parties cannot inspect the communication flow between the customer and the exchange. Without encryption, a malicious party could inspect the communication flow and thus inspect the next bids sent by the customer. Knowing the bid, the malicious party could implement a front-running attack and make profits through arbitrage [15].

6.2 Aquareum

Aquareum [26] is a novel framework for centralized ledgers, eliminating their main limitations, such as the lack of efficient verifiability and equivocation. It combines public blockchain and trusted computing platforms to provide a publicly verifiable, non-equivocating, and high-performance ledger. The ledger is integrated with a Turing-complete virtual machine, allowing more complex transaction processing logic to be added, including smart contracts or custom tokens on top of the ledger [26].

The framework uses a history tree [13] for a ledger $\mathbb{L}$, which the operator $\mathbb{O}$ persists outside the enclave $\mathbb{E}$. Each block stored within the ledger $\mathbb{L}$ contains a list of transactions and a list of execution receipts from the virtual machine running inside the enclave $\mathbb{E}$. Although $\mathbb{O}$ stores the ledger $\mathbb{L}$, the enclave $\mathbb{E}$ maintains and operates on top of this ledger by executing submitted transactions from clients $\mathbb{C}$ inside of its virtual machine. The operator $\mathbb{O}$ cannot directly modify the ledger $\mathbb{L}$ since only $\mathbb{E}$ is allowed to do this. The enclave $\mathbb{E}$ stores the last header produced and the current root hash of the history tree of

$\mathbb{L}$, allowing $\mathbb{E}$ to verify that the ledger $\mathbb{L}$ was not modified by a dishonest operator $\mathbb{O}$ and is consistent with its history [26]. The full architecture is visualized in Figure 6.3.

Transactions sent from clients $\mathbb{C}$ to operator $\mathbb{O}$ are processed in batches, and this batch is sent to the enclave $\mathbb{E}$. The enclave $\mathbb{E}$ updates the ledger $\mathbb{L}$ by executing the transactions, and returns the new version of the ledger $\mathbb{L}$ to the operator $\mathbb{O}$ signed by the enclave $\mathbb{E}$. The signed version change is then sent by the operator $\mathbb{O}$ to the smart contract $\mathbb{S}$, which accepts the new version (root hash of $\mathbb{L}$) and stores it in its memory. The customer $\mathbb{C}$ can then request the contract for the latest version and verify its correctness [26].

The authors of Aquareum successfully implemented and deployed this platform using a minimalistic Ethereum Virtual Machine eEVM [36] for support of Turing-complete smart contracts. Aquareum implementation was able to process more than 450 transactions per second on a regular machine, while including the overhead of verifications and updates [26].

Chapter 7

Secure Centralized Exchange Design

In this chapter, we will propose a centralized exchange that uses the power of trusted computing, especially Intel SGX primitives such as remote attestation and sealing combined with the public blockchain smart contract platform.

7.1 High-Level Overview

Operator $\mathbb{O}$ before the deployment of the exchange generates his own pair of asymmetric keys $(PK_{\mathbb{O}}^{PB}, SK_{\mathbb{O}}^{PB})$ used for the public blockchain. This pair of keys will be used to maintain some of the public smart contract variables. Then, the operator $\mathbb{O}$ deploys the exchange $\mathbb{E}_{\mathbb{X}}$, which runs on special SGX hardware.

The operator provides the exchange of his public key on the public blockchain $PK_{\mathbb{O}}^{PB}$ as input. The exchange $\mathbb{E}_{\mathbb{X}}$, creates an SGX enclave called the exchange enclave $\mathbb{E}$. The enclave $\mathbb{E}$ on the first run generates its own asymmetric key pair $(PK_{\mathbb{E}}^{PB}, SK_{\mathbb{E}}^{PB})$. This key pair will be used to deploy and maintain public blockchain platform objects maintained by the exchange enclave, such as smart contracts or transactions. Furthermore, the exchange enclave generates the second asymmetric key pair $(PK_{\mathbb{E}}^{TEE}, SK_{\mathbb{E}}^{TEE})$, which is used for the SGX trusted scheme. Both pairs of keys will be securely stored on the disk using SGX sealing for later retrieval. For every blockchain B coin in exchange to be swappable, the exchange enclave also generates an asymmetric key pair $(PK_{\mathbb{E}}^{B}, SK_{\mathbb{E}}^{B})$, which will be used to generate transactions and support other exchange-customer actions.

As shown in Figure 7.1, the enclave $\mathbb{E}$ initializes on the first run an empty ledger $\mathbb{L}$ and an empty Merkle-Patricia tree called the account state trie $\mathbb{A}$. For each registered exchange client, there is a leaf node inside the account state trie $\mathbb{A}$ that contains the account object, which reflects a single customer state (balance, nonce, etc.). The ledger $\mathbb{L}$ contains microblocks grouped within the history tree [13]. Each microblock contains a list of transactions executed in a specific time window within the exchange enclave, which changes the state of the accounts within the account state trie $\mathbb{A}$.

The exchange enclave $\mathbb{E}$ then deploys a smart contract $\mathbb{S}$ on the public blockchain, which is uniquely identified within the public blockchain by its address. The smart contract is initialized with the tuple $(PK_{\mathbb{O}}^{PB}, PK_{\mathbb{E}}^{PB}, PK_{\mathbb{E}}^{TEE}, A_{\mathbb{E}}, H_{\mathbb{L}})$, where $A_{\mathbb{E}}$ is the IP or DNS address of the exchange, and $H_{\mathbb{L}}$ is the root hash of the exchange ledger $\mathbb{L}$.

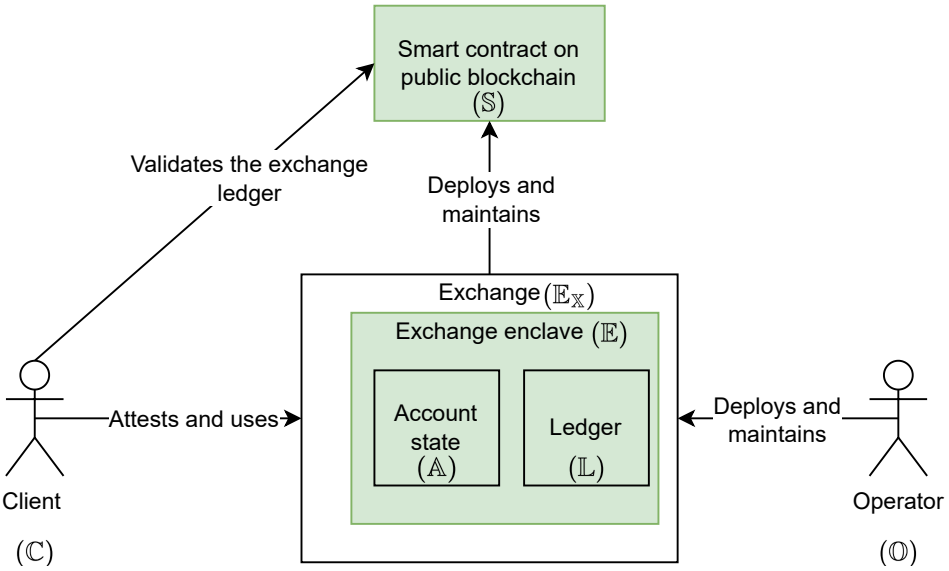

Figure 7.1: High-level view, shows the relationship between actors, exchange, and contract. The green color signifies a trusted environment.

The contract $\mathbb{S}$ keeps the root hash of the latest version of the ledger $\mathbb{L}$, which is provided to the exchange customer $\mathbb{C}$ as an indisputable proof that the exchange reached this state previously, since only the exchange enclave $\mathbb{E}$ is the holder of the public blockchain $(PK^{PB}_{\mathbb{E}}, SK^{PB}_{\mathbb{E}})$ key pair. The exchange enclave $\mathbb{E}$, using the key pair, periodically updates the contract variable that holds the root hash of the ledger $\mathbb{L}$. Only the exchange enclave $\mathbb{E}$ can update this root hash.

7.2 Smart Contract on Public Blockchain

The role of the smart contract is to provide transparency (non-equivocation) to customers. To achieve these properties, the contract holds the following variables within its persistent memory:

- Public key of the operator in the public blockchain. $(PK^{PB}_{\mathbb{O}})$
- Public key of the exchange enclave on the public blockchain. $(PK^{PB}_{\mathbb{E}})$
- Public key of the exchange enclave for the SGX TEE scheme. $(PK^{TEE}_{\mathbb{E}})$
- Exchange DNS record or IP address. $(A_{\mathbb{E}})$
- Root hash of the exchange history tree ledger. $(H_{\mathbb{L}})$

All of these variables are publicly available and can be retrieved by customers. The exchange for each new ledger microblock (or each N blocks) periodically synchronizes the smart contract root hash variable $H_{\mathbb{L}}$ with the latest ledger root hash.

This root hash of the exchange is stored inside the storage provided by a third party, which is decentralized and secured by the node consensus (in the case of using a smart contract platform like Ethereum or Algorand). In this way, the non-equivocation property

is fulfilled since there is no way for the exchange to forge and provide different false root hashes for different customers.

We define a group of methods on top of the mentioned variables inside the contract. These methods can only be successfully called by the exchange enclave, thus they always need to accept the enclave signature using $PK_{\mathbb{E}}^{PB}$, $SIG_{\mathbb{E}}^{PB}$:

- $\texttt{Init}(PK_{\mathbb{O}}^{PB}, PK_{\mathbb{E}}^{PB}, PK_{\mathbb{E}}^{TEE}, A_{\mathbb{E}}, H_{\mathbb{L}})$: The smart contract constructor, it will be run only once on the contract initialization.
- $\texttt{SetRoot}(SIG_{\mathbb{E}}^{PB}, T)$: Updates the root hash of the ledger $H_{\mathbb{L}}$. T is the transition pair, which is a tuple $(H_{\mathbb{L}}, H_{\mathbb{L}+1})$, where $H_{\mathbb{L}}$ is the current root hash in the contract (safety check) and $H_{\mathbb{L}+1}$ is the new root.

Then, we need to define a group of methods that only the exchange operator $\mathbb{O}$ can execute using the signature $SIG_{\mathbb{O}}^{PB}$:

- $\texttt{SetA}_{\mathbb{E}}(SIG_{\mathbb{O}}^{PB}, A)$: Updates the address of the exchange.
- $\texttt{SetPK}_{\mathbb{E}}^{PB}(SIG_{\mathbb{O}}^{PB}, PK^{PB})$: Updates the public key of the exchange enclave on the public blockchain.
- $\texttt{SetPK}_{\mathbb{E}}^{TEE}(SIG_{\mathbb{O}}^{PB}, PK^{TEE})$: Updates the public key of the Intel SGX TEE exchange enclave scheme.
- $\texttt{SetPK}_{\mathbb{O}}^{PB}(SIG_{\mathbb{O}}^{PB}, PK^{PB})$: Updates the exchange operator public key.

These update operations for the operator are needed in case of failures or other critical situations, such as enclave hardware malfunctions, migration of the operator, change of DNS record, etc.

Designing the exchange with the usage of smart contracts causes the exchange to inherit some of the properties of a given public blockchain. The finality of the update of the new exchange ledger version will be the same as the finality of the public blockchain. Choosing a slow public blockchain can decrease the attractiveness of the exchange.

Another concern is the fees, since operating the contract costs money, there needs to be a correct balance between the frequency of root hash update and cost. This concern can be mitigated by implementing fees inside the exchange, which can effectively cover public blockchain fees.

There is a possibility to implement a compensation fund in case of exchange malfunction or suspicious operator behavior (ex. late published blocks, outages). The funds would be filled in by the operator when the creation of the smart contract takes place and deposited at the contract address. Additionally, a percentage of the exchange fees could be converted to the currency supported by the public blockchain and credited to the compensation fund as trades are completed.

7.3 Exchange and Enclave

The exchange code will be open-source and publicly accessible, so clients can freely track potential changes within the code and perform remote attestation. Using remote attestation and open-source code, the client can fully verify that the code running on the exchange platform is the same as the code published by the exchange authors.

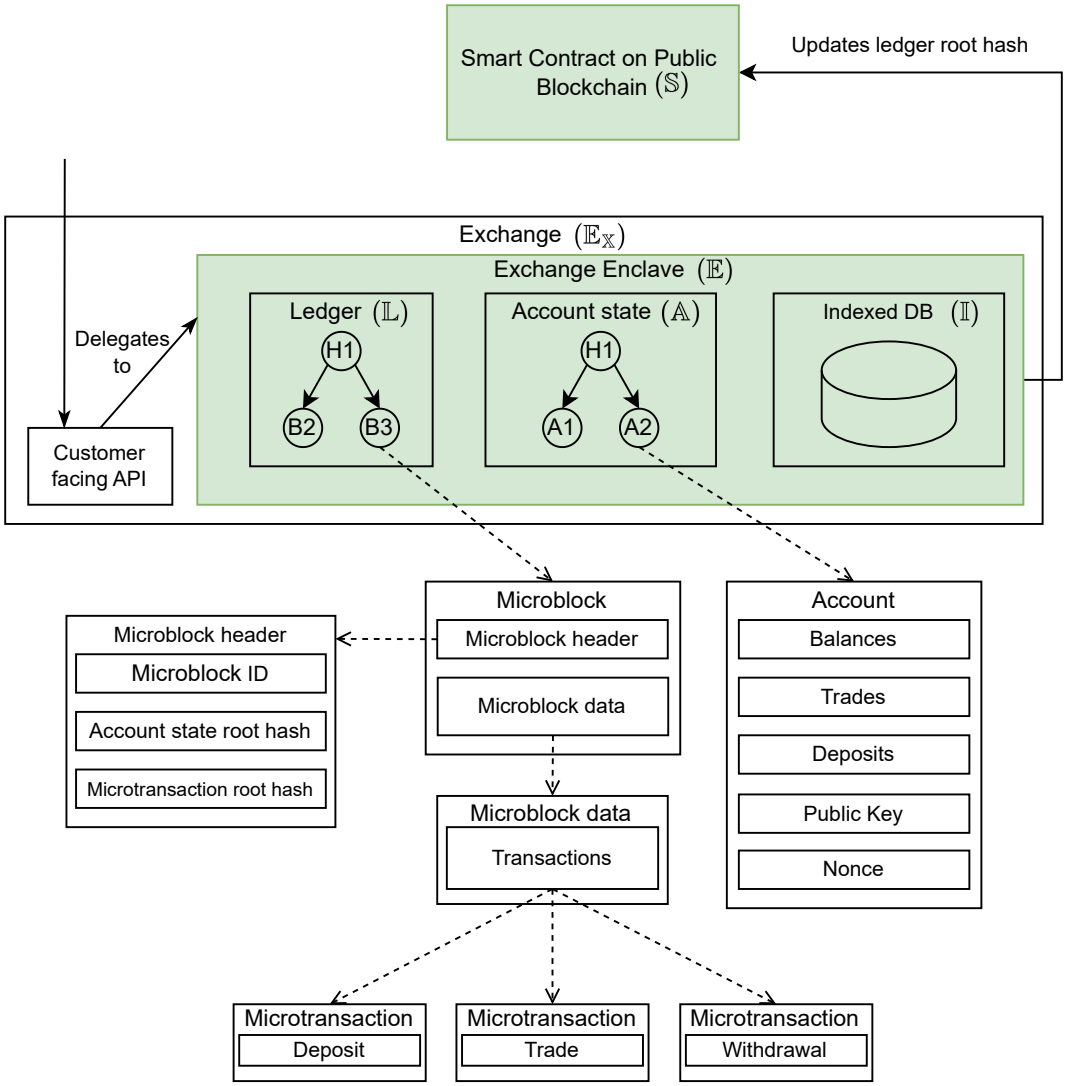

Figure 7.2: The exchange internal design, describing ledger, account state, and its objects.

7.3.1 User Control

For user control, the exchange uses a decentralized identity. The DID document contains a user-generated public key $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ marked for identification in exchange. If a new user wants to use the exchange, he needs to generate an asymmetric key pair and add the public key to his DID document. Using digital signatures with a given private key $SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$, the customer will then authenticate his requests.

Using a decentralized identity with a combination of verifiable credentials also gives an advantage in policies such as KYC, age verification, etc. Exchange can outsource the verification process of the customer to a trusted third party. Then the algorithm for user authentication can be defined as:

1. The client sends a request to the exchange to perform an action, inside of the request is a DID identifier.
2. Exchange uses the DID resolver to resolve a given DID identifier to a DID document.
3. Exchange verifies that the DID document contains requested verifiable credentials, such as age (user is old enough), country of origin (taxes), etc. Issued by one of the trusted parties.
4. Exchange finds the customer exchange public key $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ inside the DID document.
5. The exchange validates that the request is signed by a given account private key $SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$.

If this algorithm succeeds, the client request is authenticated and can be performed on the exchange.

7.3.2 Exchange Enclave

The exchange enclave $\mathbb{E}_{\mathbb{X}}$ performs and maintains all trades, bids, withdrawals, customers, and smart contract. It consists of three main parts:

- Ledger $\mathbb{L}$.
- Account State Trie $\mathbb{A}$.
- Indexed Database $\mathbb{I}$.

The complete and overall architecture of this exchange is visualized in Figure 7.2.

Ledger

A ledger $\mathbb{L}$ is stored within a history tree structure, and each member of the tree is a microblock. A single microblock contains data and microtransactions (data) performed within the exchange. The header of the microblock contains the following:

- **Microblock ID**: Serves as a unique identifier of the block.
- **Account State Root Hash**: A root hash of the account state trie after the microtransaction (in this microblock) executions.
- **Microtransactions Root Hash**: A root hash of a Merkle tree that contains microtransactions.

And the microblock data contains:

- **Microtransactions**: All microtransactions within the microblock are ordered as they are ordered in the Merkle tree, from which the root hash is calculated.

A new microblock is published every N seconds or if the number of unpublished transactions is greater than M transactions. The given microblock is created inside the enclave and stored inside the ledger, and it can also be empty. After this sequence, the ledger root hash can be published to the proposed smart contract in section 7.2. This depends on the allowed number of not yet synced blocks at the same time between the contract and the exchange enclave.

A microtransaction is a single operation executed within the exchange enclave and can be one of the following:

- **Deposit**: The customer deposited funds in the exchange enclave wallet address on the exchange-supported blockchain.
- **Trade**: A single customer submitted bid that fulfills an opposite bid made by another customer, trade was executed inside of the exchange enclave.
- **Withdrawal**: The customer requested a withdrawal of the funds inside the exchange enclave to their address on the supported blockchain, and the transaction was completed on the given blockchain.

Account State Trie

The account state trie $\mathbb{A}$ is a Merkle-Patricia tree, which maps the public key of the customer on the exchange $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ to the account object in exchange. A single account object stores the account data inside the exchange and consists of the following:

- **Balances**: A map structure that maps a coin supported by the exchange to the value of how much the customer owns inside the enclave.
- **Trades**: A map structure that maps a supported coin pair to the array of trades (microtransaction hashes) in which the customer has participated.
- **Deposits**: A map structure that maps a coin supported by the exchange to the highest block height in which this customer made an on-chain deposit transaction.
- **Public Key**: Public key of the customer $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ used for authentication.
- **Nonce**: Replay attack protection, which is incremented for each microtransaction executed by the customer.

Indexed database

The indexed database $\mathbb{I}$ is used to index and store exchange information. This is needed primarily for two reasons, the preparation of customer responses and the storage of unfulfilled bids. The selection of the database is an implementation detail and is up to the author.

Because a single exchange microblock is designed to store all transactions from all users in a specific time window, searching for specific customer transactions can be computationally hard. Exchange can link each microtransaction to the respective customer and other metadata into the indexed database.

The unfilled bids are not immutable, and cannot be stored inside a ledger microblock. They are stored inside the database for quick retrieval of the order book and customer manipulation. The moment the bid is transformed into a trade, it can be deleted from the database and prepared to be published inside the new microblock.

Methods

We define a group of methods that the enclave supports for exchange customers:

- `GetOrderBook`$(P, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns a signed order book for a selected coin pair P in the enclave. The returned order book is signed by the enclave with $SK_{\mathbb{E}}^{TEE}$.
- `GetDeposits,GetWithdrawals`$(P, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns a list of deposits or withdrawals for the selected coin C, in the enclave.
- `GetBalance`$(S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns a map with all balances of the customer inside the enclave.
- `GetBids,GetTrades`$(P, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns a list of bids or trades for the selected coin pair P in which the customer is or was the participant, in the enclave.
- `GetCoinInfo`(C): Returns an object with information about the selected coin C, such as the exchange enclave deposit address.
- `DeleteBid`$(B_{ID}, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: The exchange enclave removes the bid from the order book, identified by the bid identifier B_{ID}. The signer must be the author of this bid.
- `ProofDeposit`$(P_X, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Used for an exchange customer to prove that he deposited the funds in the exchange enclave wallet. The caller must send proof P_X that this blockchain transaction is published within the specific block. The proof is usually in the form of a Merkle tree path (e.g. Bitcoin) or another format specific to the supported public blockchain X and must be signed by the sender's address private key $SK_{\mathbb{C}}^{X}$, on a given blockchain X.
- `SubmitBid`$(A_X, A_Y, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: The exchange customer wants to submit an order to sell an amount A_X of coins X and obtain an amount A_Y of coins Y, in the exchange enclave.
- `RequestWithdrawal`$(A_X, R, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: The exchange customer wants to withdraw the amount A_X of coin X from the exchange enclave to address R.
- `GetLatestVersion`$(S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns information about the latest version of the ledger $\mathbb{L}$ and its mined microblock.
- `GetVersionProof`$(V, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns an incremental proof that the version V of the ledger $\mathbb{L}$ identified by its version number, commits to the same history as the latest version of the ledger $\mathbb{L}$.
- `GetVersion`$(V, S_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}})$: Returns information about the version V of the ledger $\mathbb{L}$ and the microblock content.

Some of the methods require the signature of the customer's request $\mathbb{C}$, with his account private key $SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$. This is needed for the authorization of the customer.

Chapter 8

Implementation

In this chapter, we describe the implementation and other design choices of the exchange. As for the programming language, we chose Go, as it offers a good abstraction for working with the network layer and also great CSP-style concurrency (paradigms, such as channels).

The choice of this language may seem odd to a skilled SGX developer, as SGX is natively supported only in C++. We need to introduce an EGo framework [18], which is used to build confidential applications in Go. EGo attempts to bridge the gap between cloud-native and confidential computing. EGo provides a modified Go compiler with additional tools and a Go library for Intel SGX enclaves, remote attestation, and sealing. These libraries are mostly wrappers for the Open Enclave [40] attestation or sealing APIs.

ERTGo compiler [19] is included in Edgeless RT [17], which is an SDK for TEE built on top of Open Enclave [40]. Edgeless RT attempts to add support for modern programming languages for easy porting of existing applications into the enclaves. RT offers more control, such as linking an additional Go code to the C++ app. The compiler attempts to wrap and mock many instructions, such as RDRAND, to generate secure random numbers.

As we mentioned previously in section 5.1, the application is composed of trusted (invoked by ECALL) and untrusted (invoked by OCALL) code, EGo takes a different approach to this, where the application runs fully in an enclave. We can speculate on the reason for this design. The first one is that the Go language does not offer such low-level functionality to switch between OCALL and ECALL, and it would be complicated to emulate. The second reason is that one of the main EGo promises is that an existing Go software should be easy to migrate to the enclave and the developer should be less concerned about the enclave settings. This can be problematic for developers running the now deprecated SGX1 enclaves, as these can only provide 128 MB of memory (section 5.1), which is mitigated by SGX2. Also, this may be the reason why EGo only supports DCAP-based remote attestation.

8.1 Implementation Structure

In Figure 8.1, we show the implementation structure of the exchange. The solid box represents a single module of the exchange enclave that performs some exchange enclave tasks. The dashed box represents an outside dependency that the exchange enclave uses to work correctly. In this chapter, we will describe the functionality, purpose, and implementation of the mentioned modules, which together form the exchange.

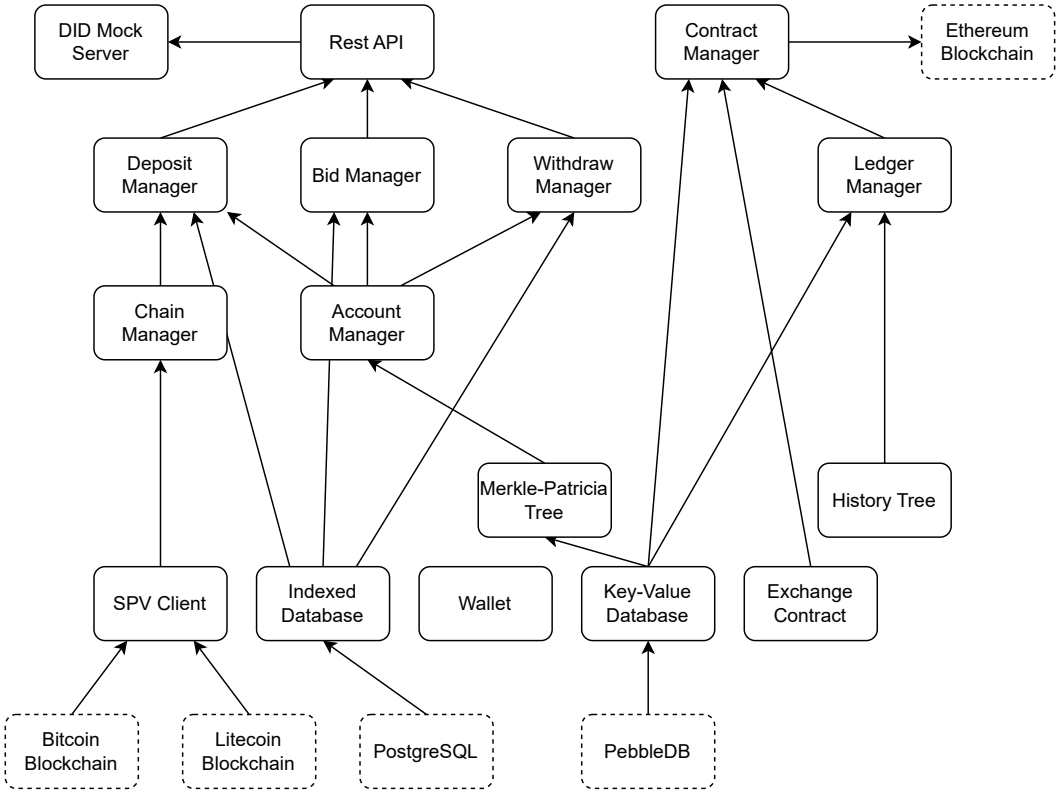

Figure 8.1: The exchange implementation structure.

8.2 Key-Value Database

Key-value database is a simple wrapper for the chosen key-value storage. For implementation, we selected to use a PebbleDB [11] which is a LevelDB-inspired key-value store focused on performance. PebbleDB does not need network communication, data are stored purely on disk, and storage is natively implemented in Go.

8.3 Indexed Database

An indexed database is another simple wrapper for the chosen database. We chose PostgreSQL [41], which is a powerful object-relational database system. The main reason for choosing this system is that the data we want to store inside this database are suitable for relational design. Also, PostgreSQL offers great synchronization principles which will be discussed later, such as advisory locks and row-specific locks.

On the code side, the exchange enclave uses the Gorm [31] ORM database library, which can easily map the table to the Go structure and also provides a set of interesting hooks.

8.3.1 Coin Value Representation

As for the values of coins (bids, deposits, etc.), we use the 64-bit integer value representation to mitigate the truncation errors created by using the floating-point data type. Inspecting the ER diagram in Figure 8.2, tables with values have the PostgreSQL `BIGINT` data type for this representation.

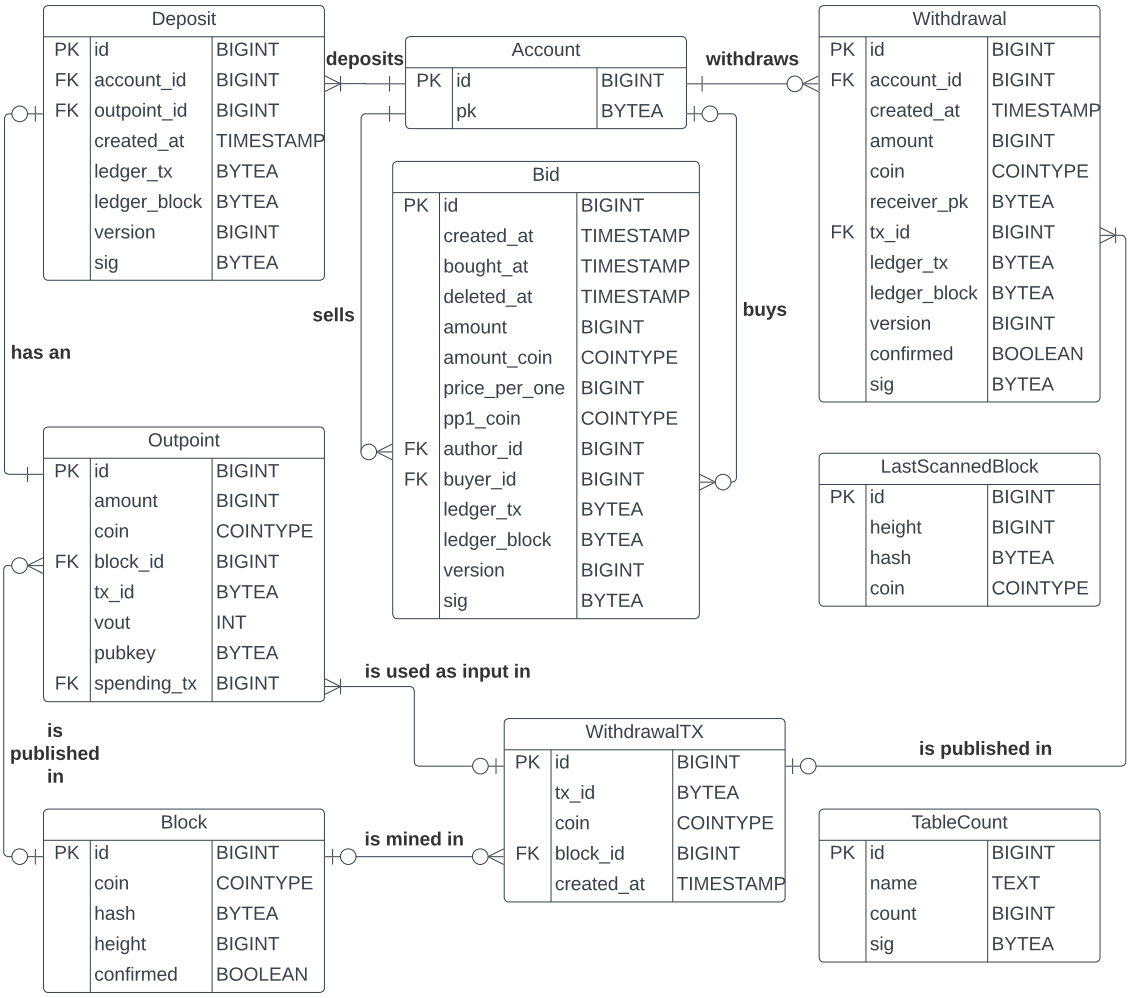

Figure 8.2: ER diagram for the exchange indexed database.

The integer represents an imaginary fixed type floating point, the exchange allows a maximum of 8 decimal places, and the rest is reserved for the whole number part and is limited by the maximum 64-bit integer value. We can present some examples of floating-point numbers converted to our representation:

$$(1)_{10} = 100000000, \tag{8.1}$$

$$(12.2456)_{10} = 1224560000, \tag{8.2}$$

$$(0.00034444)_{10} = 34444. \tag{8.3}$$

Because the highest number that the type `BIGINT` can represent is $(9223372036854775807)_{10}$, the highest value that the exchange object can hold is 92233720368.54775807. The exchange accepts the minimum amount of value in action $(0.00010000)_{10}$.

8.3.2 Data Integrity

Since the exchange stores data outside the enclave, data integrity is very important in this situation. An exchange makes decisions based on data stored in the indexed database, and also uses these data in blockchain requests, etc. Leaving the integrity out of the data could lead a malicious exchange operator to edit the data on a disk, which in the end could be beneficial for him. For example, the operator could edit the price of the specific bids, lowering their price, and then immediately buy them.

To mitigate this vulnerability, we need to implement an integrity scheme for the rows in the tables. For mitigation of potential `INSERT` and `UPDATE` operations from malicious operator, we introduce a new column called `sig`. This column stores a sealed hash for a given row content, and this hash is calculated before each creation or edit of this row. In code, this is achieved using GORM hooks for both operations. The sealed hash is calculated by following these steps:

1. Create a byte array B.
2. For each value in a row, serialize it and push it to the array B.
3. Set this array B as input to the hash function.
4. Seal the hash of the row with the SGX seal key.
5. Store this sealed hash in the `sig` column of that row.

The exchange after each retrieval (`SELECT`) for each row recalculates the hash, unseals the sealed hash in the row, and compares the hashes. If the exchange discovers the row that was tampered with, it refuses to work with that row.

Since this integrity scheme only mitigates the potential editing and creation of rows, we need to introduce one more step to achieve protection also against the `DELETE` command. In the current implementation, the exchange does not delete any rows from integrity-protected tables:

- **Deposit**: The single deposit row never needs to be deleted, the information about the deposit needs to be kept for the whole exchange lifecycle.
- **Withdrawal**: The same applies as to the deposit.

- **Bid**: When the bid wants to be deleted by its author, instead of a real deletion in the database, it is marked as deleted by using the `deleted_at` timestamp field. The exchange in front of the customer pretends that the bid does not exist.
- **TableCount**: Exchange never deletes from this table, since this table keeps information about the previous three tables.

We can take advantage of this feature for protection against deletion by introducing a new table called `TableCount`. This table for each integrity-protected table keeps a row that records the count of rows in that table. Before each row is inserted into the protected table, the exchange increases the number of rows in the column `count`.

Surely, the rows inside the `TableCount` must be protected against the same `INSERT`, `UPDATE`, and `DELETE` commands. For `INSERT` and `UPDATE`, we can use the same integrity scheme for each row as for the protected tables. As for the `DELETE`, the enclave expects the table to have a single row for each table, if the row does not exist, the enclave refuses to perform any operations connected to the protected table. These rows are created only by the exchange enclave on its first run. The complete algorithm to select from the protected table T can be described as:

1. Select the row R from the protected table T.
2. Unseal the sealed hash of row R with the SGX seal key.
3. Serialize and calculate the hash of row R.
4. Compare the calculated hash with the unsealed hash for equality.
5. Select a row P from the table `TableCount` for a given protected table T. It must exist.
6. Unseal the sealed hash of the row P with the SGX seal key.
7. Serialize and calculate the hash of the row P.
8. Compare the calculated hash with the unsealed hash for equality.
9. Select the count of rows in the protected table T.
10. Compare the calculated count of rows with the count stored in the row P for equality.

8.4 SPV Client

SPV Client is a low-level implementation of the given blockchain protocol. For an exchange to support the deposit and withdrawal of coins, it needs to implement this client interface. There is a universal implementation of the Bitcoin protocol, which can be used for communication with the Bitcoin blockchain and nearly every fork of Bitcoin (such as Litecoin, Bitcoin Cash, etc.). For the client to work with the selected Bitcoin fork, the developer needs to specify these variables:

- Magic number of the network.
- Client service flag.

- Client user agent.
- IP of the node.
- Port of the node.

The client for Bitcoin and its forks works on the Peer-to-Peer network protocol layer and does not use RPC. Using a P2P network instead of RPC was chosen because it allows the exchange to keep the connection alive by responding and sending ping and pong type messages. The connection is then used to fetch new blocks and their headers or to send new transactions to the chain.

It is highly recommended that the exchange operator deploys and maintains its own Bitcoin (and other forks) nodes dedicated to exchange, since it should not rely on random unknown blockchain nodes. In this way, the operator can only deploy an SPV node.

With the Ethereum blockchain, the client situation is very different. As Ethereum recently switched to Proof-of-Stake, Geth (the official implementation of the Ethereum protocol) currently does not support a light client (as of April 2023). This leaves the moderator with one of the two options:

- **Deploy self owned full node**: Requires a lot of disk space, over 300 GB.
- **Use a third-party Ethereum API services**: More potential vulnerabilities.

8.4.1 Chain Verifier

Chain verifier leverages the implementation of the SPV client and is a higher-level abstraction of the blockchain. The exchange also needs to implement this interface to support the deposit and withdrawal of specific blockchain coins.

The verifier always keeps the header blocks from the last two weeks in the FIFO queue. These headers are provided for higher-level exchange components for various operations. To be more precise, on the Bitcoin blockchain, every new block is mined on average every 10 minutes, so two weeks of Bitcoin state can be described in 2016 blocks. Since a single block header is 80 B in size, a two-week period of block headers takes 162 KB of memory. For every blockchain being synced, the verifier must know the consensus rules so it is able to correctly verify the validity of newly synced blocks.

For the Ethereum blockchain, this approach takes more space, since the Ethereum network proposes a new block every 12 seconds, so two weeks of Ethereum state are represented by 100,800 block headers. Another complication is that the Ethereum block headers are larger in size (minimum of 512 B [52]).

We can easily overcome these two problems by shrinking the block header size by selecting only the fields that are being used by the exchange. For example, the deposit action for exchange only needs two fields, the height of the block, and the root hash of the transaction Merkle tree. In this way, the block header only takes 40 B (32 B transactions root hash and 8 B for 64-bit integer). Then, the two-week period of Ethereum blocks takes 4.1 MB.

To exchange to use the verifier for a specific blockchain, the developer needs to implement a specific interface, which provides the following:

- **Block Sync Period**: Duration in seconds, how often the chain should check for a new block.
- **Block Window Size**: How many blocks represents two weeks of a given blockchain state.

- **Genesis**: Genesis block hash.
- **Finality**: Finality of the chain in the number of blocks.
- **Block Hash Calculation Algorithm**: Function that takes the block header as input and returns Proof-of-Work hash and block hash.
- **Coin Name**: The coin name.
- **P2PKH Address ID**: Returns the blockchain-specific ID of Bitcoin (and its forks) for the calculation of the P2PKH address.

The verifier at construction time syncs and verifies the chain starting from the specified genesis block up to the latest blocks. Then, the verifier starts a goroutine (Go-specific green thread) that every specified block sync period asks the underlying blockchain node for a new block. For the Bitcoin (and its forks) blockchain, this is done using the `GETHEADERS` protocol message by providing the latest known block hash as the start hash and leaving the stop hash empty to sync all missing blocks.

For each new block header, its hash is validated (recalculation and comparison), the previous block is checked if its correct, and block difficulty is calculated and verified, by the rules of given chain consensus. If the block is valid, it is added to the header two weeks window. The verifier also provides a callback for new blocks (observer pattern), which will be used later.

If the synced chain seems invalid, the verifier attempts to resync the whole chain, if it is not successful, it panics and ends execution.

8.5 Wallet

A wallet is an implementation of a simple coin wallet. For each supported exchange coin, a wallet must exist. This wallet is used to sign on-chain transactions and generate an address for customer exchange deposits.

At the first start of the exchange, a wallet is generated for each supported coin (Bitcoin and Litecoin) and also for Ethereum, to be able to deploy and maintain the smart contract.

Each wallet for each coin securely generates a random private key using the enclave. To be consistent with private keys and addresses, the wallet is sealed by the SGX seal key and stored persistently on the disk. On every subsequent exchange launch, the exchange enclave attempts to read the wallet files from the disk, unseal the private keys, and loads them into encrypted enclave memory.

8.6 Exchange Contract

Exchange contract is implemented in Solidity [21], a statically typed language to implement smart contracts on the Ethereum blockchain. The contract is compiled through the solidity compiler, the resulting EVM application binding interface (ABI), and the binary is handled as input to Abigen [22]. Abigen is a special binding generator that interacts with Ethereum contracts through the Go language. Abigen abstracts the hassle of contract binding, data encoding, and decoding and directly creates contract methods natively in Go.

8.6.1 Contract Implementation

The contract on deployment remembers the address of the deployer (exchange enclave) and each property in section 7.2. Each property is available for customers to query through the Ethereum nodes.

The update methods specified in section 7.2, are protected by the solidity modifiers. The contract defines two modifiers, one for the exchange enclave, and the other for the exchange operator, these modifiers are very simple, and before each modified contract method invocation, they check if the invoker address is equal to the address stored inside the contract. If the address is the same, the invocation is correct, and the method continues to run. Otherwise, the invocation fails.

The contract also defines a set of events that can notify subscribers through the Ethereum node websocket. For each contract property, an event is defined that is triggered when the property is updated. Also, there is a special event that is fired when the exchange enclave ledger root hash is updated (contract obtained a new valid transition pair).

Normally, the customer (or exchange) wants to query all properties of the contract. The number of requests that the customer (or exchange) would need to make is equal to the number of properties. Because of this overhead, we introduced special solidity public view function for requesting all variables through one call.

8.7 Merkle-Patricia Tree

Merkle-Patricia Tree (trie) is implemented according to the specification in subsection 3.1.3. As specified in the Ethereum Yellow Paper [52], trie uses the implemented key-value storage from section 8.2. An instance of trie can be easily created by invoking its constructor and providing an instance that implements a node database interface.

The trie provides three methods, storing, reading the value (byte array) stored into the trie under a specific key, and the third method for retrieving the current trie root hash. The store method works as an upsert, if a given value under a specified key exists, it gets updated. The retrieval method returns a negative Boolean value when a value is not found under the key.

The trie is not able to perform parallel inserts (and updates), since every single change in every node needs to be specially rehashed from the leaf node up to the root node. This introduces the need for goroutine (green thread) locks, such as mutex. The reading operation from the trie can be parallel, in a way that multiple threads can read from the trie at the same time but only one exclusive thread can write at the same time. We can take advantage of this property and use the read-write mutex provided by the Go native library. RWMutex allows the lock to be held by an arbitrary number of readers or a single writer. This allows us to use at least some level of parallelization, which is key when providing customer-facing APIs.

As mentioned in subsection 3.1.3, trie keys are always nibbles, thus retrieving and storing methods need to convert the key (byte array) to nibble array (also byte array). This key-byte array can easily be converted to a nibble-byte array by bitwise logical operations, such as `AND` and byte shift.

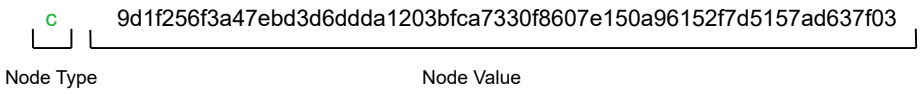

Figure 8.3: Example of a serialized node with example type of value „c“.

8.7.1 Low-Level Node Storage

The node database interface provides two methods to store and read a single internal trie node. There is currently only one implementation of a specific interface, using the key-value database. To store, the key (node hash) and node are provided as input, the node gets serialized to a byte array, and depending on the node type (branch, extension, leaf, empty), a single byte is appended at the start of the node raw byte array. This array is stored as a value, where the node hash is its key in the key-value storage. The method is visualized in Figure 8.3.

For node retrieval under the specific key (node hash), the raw node byte array is read from the key-value database, the first byte is parsed into the node type, and returned with the remaining raw node byte array. This raw node byte array is then deserialized into a node object.

The current trie implementation is not fully sane, every node must be read from the key-value database over again, which means that each operation on the trie level descend needs to read the node and deserialize it from the disk.

For a trie node to be stored and loaded, it needs to implement a serializable and deserializable interface. The serializable interface expects a single method that turns a node into a serialized byte array. The deserializable interface provides a single method, which expects a byte array, parses, and populates the given node object. Every trie node type implements this interface, so that the node database is able to save them on disk. For serialization and deserialization, the nodes use a concise binary object representation (CBOR) format [8]. The processing time of the CBOR library in Go is lower than that using the JSON library, and the message sizes are also smaller, which is useful for this use case.

Every loaded node inside the trie is validated by hashing the node and comparing the original hash (key) with the calculated fresh hash. Before every save inside the trie, the node is hashed, which is the resulting node key. In the current implementation, trie always uses the Keccak256 (SHA-3 family) hashing algorithm for every hash operation. This can be easily changed by implementing another structure that implements the hashing interface.

8.8 History Tree

The history tree is implemented according to the definition from subsection 3.1.2. The history tree provides both the tree and proof creation (both incremental and membership). The tree within the implementation is represented in two parts, which are presented in Figure 8.4.

The first part is that every history tree non-leaf level (inner layer) is represented as an array of hashes (byte arrays) and all non-leaf levels are stored inside an array of levels. The second part represents the leaf level (leaf layer) as an array of values (byte arrays).

In this way, the insertion of the tree is simple and is composed of two steps. The first step is to add the value to the leaf layer. The second step is to recursively recalculate the hashes on the path up to the root hash. The index i of the hash that needs to be recalculated

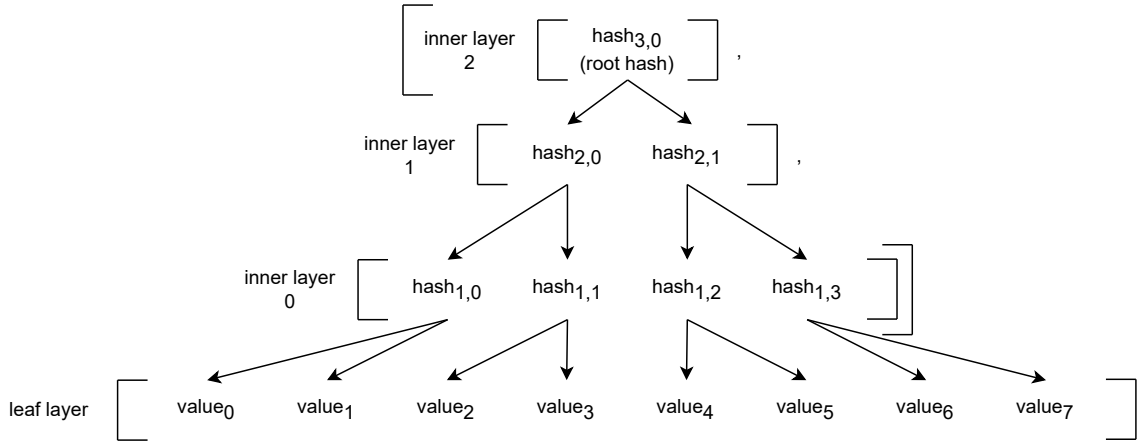

Figure 8.4: Visualisation of storing the history tree inside the memory.

next based on the current recalculated value of index n can be easily computed using the following formula:

$$i = n \div 2. \tag{8.4}$$

The creation of an incremental proof consists of two main steps. First, a proof skeleton is created from the higher-specified version. A proof skeleton is a path from the leaf node to the root node. This path must be filled with calculable nodes (nodes that will be recalculated as verifying the proof, for example, node $hash_{2,1}$ in Figure 8.4, when the proof skeleton is created for a leaf node in this subtree) and node stubs (frozen nodes, for example, node $hash_{2,0}$ in Figure 8.4, when the proof skeleton is created for node not in this subtree).

The second step is to transform the proof skeleton into an actual incremental proof. We achieve this by traversing the tree from the root node to the value given by the lower version, and adding the traversed nodes to the existing skeleton proof. On the way to the leaf layer, every encountered stub previously created in the skeleton algorithm needs to be unfrozen by converting it from stub to the calculable node. The final proof is returned to the caller.

The returned proof structure also implements the verification of both the membership and the incremental proof exactly by a verification algorithm specified previously in subsection 3.1.2.

8.9 Account Manager

Account manager is a higher level exchange component that maintains accounts, account locks, and provides methods for other components for account manipulation. We can divide the manager functionality into two main sections.

8.9.1 Account Object

A single account object defines a single exchange customer and is the main structure that defines a single user. It is made up of the exact same attributes as mentioned in section 7.3.2.

8.9.2 Account Storing

The manager keeps the accounts in the account state trie from section 8.7. The trie does not provide any protection against tampering by someone else other than the enclave. Because of this, the exchange enclave needs to seal the latest account state trie root hash with the SGX seal key and store this latest root hash inside the key-value database under a specific key.

This allows the exchange enclave to verify that the trie was not modified by someone else other than the enclave on every load or store from the trie. This introduces the great advantage that a single hash can represent the integrity of every single account in the exchange. A disadvantage of this approach is that the enclave on every store or load needs to seal or unseal the account trie root hash and store it on the disk.

When the exchange is relaunched, it attempts to fetch the latest trie root hash from the key-value storage under a specific key and load the given account trie. On the first launch, if the root hash of the account trie does not exist in the key-value storage, the trie is created.

To be more precise, the enclave stores only the latest account object hash inside the trie under the key, which is the public key of the user, used for authentication with the exchange. When the manager requests an account from the trie, it needs to do one more key-value storage lookup for the account that is stored under the account hash returned from the trie. The loading, serialization, and deserialization of the account are handled completely in the same way as the nodes of the trie specified in section 8.7.

8.9.3 Account Locking

Another important responsibility of the manager is to control access to accounts, since by combining multiple exchange actions, an account can become a critical section. We also want the exchange to work concurrently to achieve better performance, so locking is needed.

The manager provides methods for locking the account, unlocking the account, and unlocking the account with additional indexed database rollback. Since the exchange uses PostgreSQL for indexed database implementation, we use PostgreSQL row-level locks to lock user accounts. In the Figure 8.2, we can see that the indexed database also has a table `account` for exchange accounts with their corresponding public keys. When the enclave wants to lock an account for writing, it needs to first start a PostgreSQL transaction by keyword `BEGIN` and then select the specific row from the table `account` with specified clause `FOR UPDATE`. This select issues a `ROW SHARE` specific lock on this row, thus every other thread attempting to lock the account will need to wait until the transaction is committed. All other enclave operations that change the state of an account must first acquire this lock.

Using PostgreSQL application-controlled locking instead of code locking adds a potential for exchange to be replicated. In fact, no additional code is required to maintain locks, and we leverage PostgreSQL memory outside the enclave.

8.10 Deposit Manager

Deposit Manager is another high-level component of exchange, its main role is to handle customer deposits to exchange. To run a manager correctly, a chain manager must be implemented for each specific coin. Each exchange deposit costs a fee of 0.1 % of the deposited amount.

8.10.1 Chain Manager

The chain manager needs to implement a verification of the exchange enclave deposit transaction on-chain. Currently, there is only one implementation of verification, that is, for the Bitcoin (and its forks) blockchain, and it does the following.

The algorithm expects as input the blockchain-specific transaction T, a membership proof P of the transaction T being included in the block B:

1. Recalculate the Merkle membership proof path up to root and obtain the transactions Merkle tree root hash for a given block B.
2. Obtain the block header using the verifier (subsection 8.4.1).
3. Validate that the block B is finalized (block is older than the finality value).
4. Compare the recalculated transactions Merkle root hash from the proof with the Merkle root hash from the block header, they must be equal.
5. For each transaction T unspent output U:
   (a) Verify that the output U pubKeyScript is a pay script addressed to the exchange enclave.
   (b) The value credited from the output U is the amount deposited in the exchange enclave.
6. Return the sum of every output U addressed to the exchange enclave as the total amount deposited by the user.

Currently, the exchange only supports the P2PKH pay script. PubKeyScript validation of the output is verified by doing the following:

1. Verify that the pubKeyScript length is 25 bytes.
2. Extract the receiver public key hash from the fourth to the 24th byte of the pubKeyScript.
3. Hash the public key of the enclave deposit address.
4. Compare the extracted receiver public key hash with the hashed deposit address public key, must be equal.

8.10.2 Deposit Validation

To fully validate the user's exchange deposit for the Bitcoin and Litecoin blockchain, the algorithm expects the following:

- **Block Hash** B: Hash of the block in which the blockchain deposit transaction was mined.
- **Transaction** T: Full blockchain deposit transaction.
- **Merkle Path** M: Membership proof of the transaction T being mined in the block B.
- **Account Public Key** PK_E: Public key of the user on exchange.

- **Address Public Key** PK_A: Public key bound to the address that sent the deposit transaction T.
- **Request Signature** S_A: Signature of the values provided previously, created by private key SK_A.

The algorithm then performs these steps:

1. Validate the request signature against the provided address public key PK_A.
2. Validate the transaction T being in block B using the algorithm from subsection 8.10.1.
3. Create an account if it does not exist:
   (a) Create and save a new account object in the key-value database under a key defined by its hash.
   (b) Insert the hash of the account inside the account state trie under the key of the user's account public key PK_E.
   (c) Insert the account row inside the `account` table in the indexed database.
4. Start an indexed database transaction T_I.
5. Lock the account through the account manager using transaction T_I and obtain the account from the trie.
6. Validate if the block number is strictly higher than the last deposited block number height for a given account.
7. Subtract the fee from the deposited amount (currently the fee is 0.1 %).
8. Add this deposited balance to the account balance map, for the given deposited coin.
9. Increment account nonce.
10. Update the last deposit block height for a given coin for this account.
11. Upsert the coin block in the `block` table.
12. Insert the outpoint (UTXO index number and transaction hash) into the `outpoint` table.
13. Save the account to the account state trie.
14. Commit the transaction T_I (unlocks account lock and commits indexed database changes).
15. Send a notification to the ledger manager that a new microtransaction of exchange occurred.

The exchange only allows users to deposit coins with a strict increase in block height. This means that the user cannot perform two or more deposits in a separate transaction in a single block.

It is now possible to notice the disadvantage of using the P2P Bitcoin protocol instead of the RPC wallet. The exchange enclave needs to implement some features of a typical

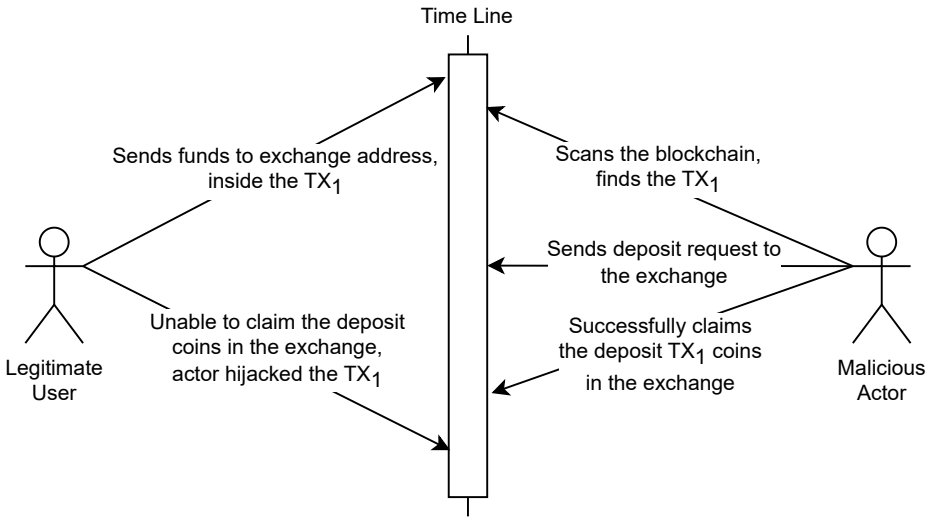

Figure 8.5: Visualisation of deposit transaction hijacking vulnerability, without using the request signature.

wallet (such as storing and maintaining blocks and unspent output) to be able to spend coins (for withdrawals).

The customer needs to provide a signature S_A of the deposit request, created using the private key SK_A, to prove that this customer is the holder of the private key bound to the address, respectively, to prove that this customer was the one who sent the on-chain transaction. This mitigates potential vulnerability, since without the request signature, it would be possible for a malicious actor to hijack the deposit transaction of a legitimate customer. Since in the current implementation there is a single exchange enclave address per coin for all deposits, the malicious actor could potentially scan the blockchain. If the malicious actor finds the on-chain transaction used as a deposit of the legitimate customer and knows the public key of the address, he can redeem the deposited coins on the exchange. This scenario is visualized in Figure 8.5.

Someone could argue that since the exchange currently only accepts P2PKH transactions scriptPubKey, the malicious actor is unable to decode the public key of a legitimate user from the blockchain address, since the address is calculated from the hash of the public key. Even if this is true, it is a bad practice to rely on the secrecy of a public key. Also, if an address is reused in previous transactions, it is possible to find the public key by scanning previous scriptSigs.

8.11 Withdrawal Manager

The withdrawal manager is another high-level exchange enclave component. Its main purpose is to receive, handle, and publish customer withdrawal requests and transactions. Because blockchain transactions are not instant, we have to divide the action into 3 main parts:

1. **Withdrawal Request**: Handling of customer withdrawal requests and preparation from the enclave side.

2. **Transaction Publish**: Submission of a transaction which transfers customer requested funds to a customer on a given blockchain.

3. **Confirmation of the Withdrawal Transaction**: Scanning of withdrawal transactions on the blockchain and confirming withdrawal at the exchange.

Each withdrawal request from the exchange costs a fee of 0.1 % of the withdrawal amount. Currently, withdrawal functionality is also only implemented for Bitcoin and Litecoin.

8.11.1 Withdrawal Request Handler

The withdrawal request handler is invoked on every new customer's withdrawal request. The handling algorithm can be described in a few steps:

1. Start the indexed database transaction T_I.

2. Acquire the account lock using T_I, through the account manager for the requested account.

3. Delete every non-filled and non-deleted bid created by a given account for the coin being withdrawn.

4. Get the balance of the account for the given coin.

5. Verify that the account owns the requested withdrawal balance amount.

6. Subtract fee from the withdrawal amount.

7. Increase the account nonce.

8. Create a withdrawal row in the table `withdrawal` in the indexed database.

9. Commit the transaction T_I (unlock the account lock).

These steps ensured that we have validated and created a new withdrawal request in our database for our customer, which must be completed later.

8.11.2 Transaction Publisher

Since the Bitcoin (and its forks) blockchain uses a UTXO model for transactions, it is not feasible to immediately send a single transaction per withdrawal for a single customer. We first collect withdrawal requests and then use a method called transaction batching to include multiple withdrawals in a single transaction.

Transaction batching is a technique to save fees (precisely to minimize the number of transactions), when the sender is in a situation where he needs to send funds to different recipients. The UTXO transaction approach allows us to include outputs bound to different receivers in the same transaction, as visualized in Figure 8.6. A typical transaction that takes a single P2PKH input and provides two P2PKH outputs (single output for the receiver and single output as the remaining change from the input) has a size of 226 B. A transaction that takes a single P2PKH input and provides three P2PKH outputs (two outputs for two receivers and a single output as the remaining change from the input) has a size of 260 B. It is significantly cheaper to send a single withdrawal transaction for two or more customers in size of 260 B, than to send two separate transactions in size of 452 B.

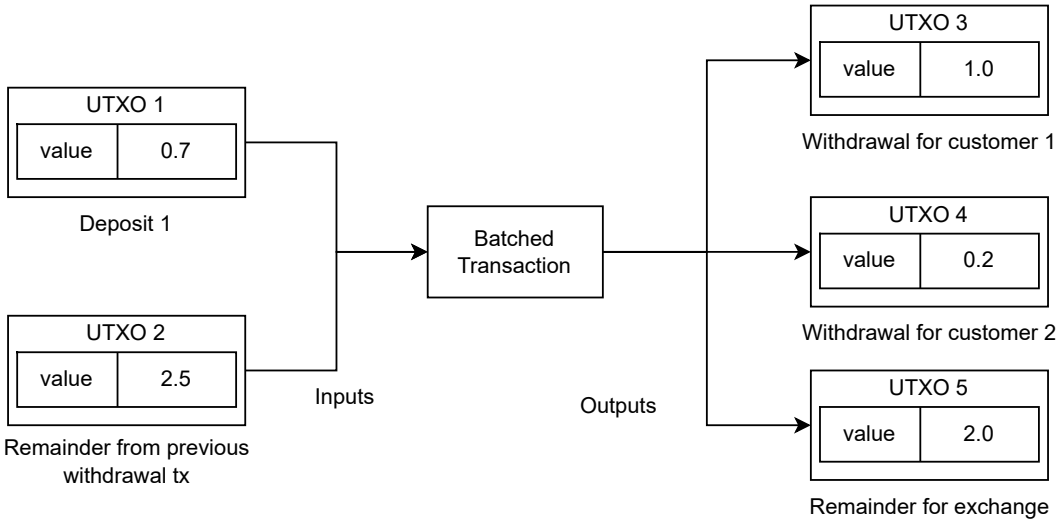

Figure 8.6: Transaction batching example for exchange withdrawal for two customers.

The transaction publisher is a special goroutine (green thread) that gets invoked every 3 minutes. The publisher sends withdrawals on each specific supported blockchain in the created on-chain transaction. When the transaction publisher is invoked, the algorithm performs these steps for each UTXO-based exchange-supported coin C:

1. Begin the indexed database transaction T_I.
2. Select and lock withdrawals using T_I, which have not yet been submitted to a blockchain transaction for a given coin C.
3. Calculate the total sum of the selected withdrawals S_W.
4. Select and lock unspent outpoints (UTXO, from table `outpoints` in the indexed database), which have not been spent on a given blockchain C, and their sum is greater than or equal to the sum of withdrawals S_W.
5. Calculate the total sum of the unspent outpoints S_U.
6. Build a blockchain transaction T_C for a given coin:
   (a) Append every unspent selected outpoint (UTXO) as input to the transaction.
   (b) For every requested withdrawal, add a transaction output that addresses the requested withdrawal amount to the customer, and lock it with the P2PKH script by the customers' blockchain public key hash.
   (c) Calculate the fees F for the forged transaction.
   (d) If there is a remaining balance ($S_W + F < S_U$) in the transaction, add a transaction output O_E, which locks the remaining funds ($S_U - (S_W + F)$) with the P2PKH script addressed to the public key hash of the exchange enclave.
7. Sign the transaction using the exchange enclave wallet.
8. Publish the transaction to the blockchain node.

9. Mark the withdrawal request as submitted to the blockchain.

10. Mark locked outpoints as spent in the indexed database.

11. Insert the withdrawal transaction hash in the indexed database (`withdrawal_tx` table).

12. If so, insert the transaction output O_E into the indexed database as a row in the table `outpoint`, this outpoint is new for future use.

13. Commit the transaction T_I (unlock locks).

Step 4. is very critical for the exchange, the sum of the withdrawals S_W cannot be strictly higher than the sum of the available outpoints S_U. If this happens, it would mean that the exchange has less currency available than customers can withdraw, which would be a fatal error of the exchange.

8.11.3 Block Scanner

The block scanner is the last part of the withdrawal process, which functions as a callback. This callback is provided to the chain verifier component and is invoked when the component syncs a new blockchain block header. The main task of the scanner is to confirm inside the exchange enclave that the withdrawal was successful, and that the withdrawal on-chain transaction published by the transaction publisher is finalized. The callback expects the queue of the latest block headers C and the height H (index in a queue) of the new block. It does the following:

1. Check whether the scanner has already scanned this block, by selecting the last confirmed scanned block hash and height (from table `last_scanned_block`), skip if it was previously scanned.

2. Begin the transaction T_I in the indexed database.

3. If the height of the new unconfirmed block was previously scanned (there is a different block hash with the same height for a given coin inside the table `block`), this old block and its descendants must be rolled back in exchange:

   (a) Remove the linkage of the forked block and its descendants with the submitted withdrawal transactions (table `withdrawal_tx`).

   (b) Remove the linkage of the forked block and its descendants with the outpoints (table `outpoint`).

   (c) Remove the forked block and its descendants from the table `block`.

4. Select and lock the withdrawal on-chain transactions that were not yet mined in a block.

5. Retrieve the full block B from the node through the SPV client component.

6. For each new transaction T, within the new block B:

   (a) Check if the transaction hash is one of the withdrawal transactions.

   (b) If so, link the block to the withdrawal transaction within the indexed database.

(c) Link the block with the exchange-unspent outpoint (if exists) in the indexed database.

7. Select and lock the blocks from the indexed database that are marked as not confirmed.

8. For each unconfirmed block B_U, do the following:

   (a) If the unconfirmed block height is equal or lower than the new block height minus the given blockchain block finality, the block is confirmed, otherwise skip this block.

   (b) Confirm the block within the indexed database.

   (c) Select the withdrawals that are linked to the on-chain transactions in this newly confirmed block.

   (d) For each withdrawal, do the following:

      i. Obtain and lock the account to which the withdrawal was addressed.

      ii. Subtract the withdrawn amount from the customers' balance.

      iii. Confirm the withdrawal in the indexed database.

      iv. Store the account in the account state trie.

      v. Create a savepoint in the indexed database transaction T_I.

   (e) Update the last scanned block in the indexed database and update it to this confirmed block hash.

9. Commit the transaction T_I.

10. For each confirmed withdrawal, create a microtransaction and notify the ledger manager.

We can analyze that the block scanner, the new block callback, is composed of three main steps. The first step is to analyze whether the new block is the successor of the previous block, which is done in step three and its sub-steps. These steps check if some of the previous synced blocks that are not yet confirmed (finality has not yet passed) got rolled back, which means that a fork occurred on the blockchain. There is a need to revert everything connected to the forked block inside the exchange enclave.

The second step is that the block scanner needs to link the withdrawal on-chain transaction and exchange outpoints with the block, this is done in step six and its substeps.

The third step is the complicated one, the exchange needs to sync the withdrawal on-chain transactions. When the withdrawal transaction on the blockchain gets finalized, the exchange enclave needs to confirm the withdrawals. This is a signal that the transaction cannot be reverted and that the customer obtained his funds. For every withdrawal within the confirmed transaction, the exchange enclave must subtract the amount from the account balance.

There is also an optimization mechanism implemented to not repeatedly rescan previously scanned confirmed blocks, since the exchange enclave on restart always attempts to sync the blockchain from the genesis (or checkpoint) block and attempts to call the new block callback. In this way, the exchange enclave on restart skips most of the blocks for a scan.

8.12 Bid Manager

Bid manager is a high-level exchange component of the exchange enclave that maintains and evaluates exchange bids and transforms them into trades.

8.12.1 Bid Representation

We can interpret single bid from the customer's point of view as:

- I want to sell X amount of coins A and I want to obtain at least Y amount of coins B.

A single bid like this is stored within the indexed database (table `bid`), the field `amount` represents the amount X that the customer is trying to sell, and the field `amount_coin` represents the specific coin A. The field `price_per_one` signifies the minimum price at which the seller wants to sell a single coin. This price P is in the coin B, that the author wants to buy (column `pp1_coin`), and is calculated using the following formula:

$$P = \frac{Y}{X}. \tag{8.5}$$

This price per one P is a metric used to compare opposite bids, which sells the coin B for the coin A. It can also be understood as a bid normalizing value, since there can be tons of bids that sell and buy unlimited amounts of coins, and this ratio will always be comparable between these bids.

8.12.2 Bid Matching

On each new submission of the bid, the following matching algorithm is executed. The algorithm takes as input the amount S to sell coins C_S and the amount B to buy coins C_B. Since the opposite bids (selling the coin C_B for the coin C_S) have our normalization value (price per one P) in the coin C_S, we need to calculate the maximum price P_A that the submitting customer is willing to pay (in coin C_S) for a single coin C_B, using the following formula:

$$P_A = \frac{S}{B}. \tag{8.6}$$

The algorithm then performs these steps:

1. Begin the transaction T_I in the indexed database.
2. Lock the partition of the table `bid`, that contains non-filled and non-deleted bids that sell coin C_B for coin C_S (opposite bids).
3. Select the bids, for which $P \leq P_A$ and select only the bids that the submitter can actually buy based on the amount of coins being sold (calculate the cumulative sum). We call these bids the buyable bids. Lock these bids, their account authors, and the bid submitter account using the transaction T_I.
4. Check if the customer has more or equal balance than he is attempting to sell, if not, commit the transaction T_I and return an error.
5. For each buyable bid, subtract the value of the field `amount` from the amount B. Let us call this the remaining value R.

6. If the remaining value R is strictly less than zero, the buyer can fill all the buyable bids except the last. This last bid can only be purchased partially:
   (a) Alter the last partially buyable bid amount to $A = A - (-R)$, where the A is the amount being sold in the bid.
   (b) Create a leftover bid where the author is the author of this edited bid. The reversed (negative) remaining value R, is the amount remaining to sell in that bid $A = -R$.
7. Otherwise, if the remaining value R is strictly higher than zero, the buyer can fill all buyable bids, but at the same time there are not enough buyable bids to completely fill his order:
   (a) Create a leftover bid, where the author is the bid submitter. The remaining value R, is the amount remaining to sell in that bid.
8. For each bid O from the buyable bid set:
   (a) The amount B_A to add to the bid submitter balance is equal to the amount O_A sold in the bid O, in coin C_B.
   (b) The amount B_S to subtract from the submitter balance O is equal to $B_S = O_A \times P_A$, in coin C_S.
   (c) The amount A_A to add to the balance of the author of the bid O is equal to B_S.
   (d) The amount A_S to subtract from the balance of the author of the bid O is equal to B_A.
   (e) Calculate the fees for both accounts from the values A_A and B_A.
   (f) Swap the coins by adding and subtracting the calculated amounts B_A, B_S, A_A, A_S from both accounts.
   (g) Mark the bid as bought inside the indexed database. This transforms the bid into a trade.
   (h) Increase the nonce of both accounts.
   (i) Save both accounts in the account trie.
   (j) Create a savepoint inside the transaction T_I of the indexed database.
9. If the leftover bid was created, insert it into the indexed database.
10. Commit the transaction T_I.
11. For each trade, create a trade microtransaction and notify the ledger manager.

We can see that the bid matching contains three main steps. The first step is to lock the partition of the bid table that is used for resolution. When resolving a bid, the partition needs to be exclusively locked for this thread so that no other thread can add a new buyable bid, which would be skipped by the first thread. The late buyable bid and the author's leftover bid would then be left unresolved, even if they could match.

The second step is to find the bids that can meet the new bid conditions. For this, we use our normalization metric (price per one). Using PostgreSQL, we can select and lock the bids (and accounts) whose metric P is lower than or equal to the new bid P_A and the

submitter is able to buy them by calculating the cumulative sum and comparing them to the amount being bought, these are buyable bids. This means that the selected bid authors are selling their coins cheaper (or equal) to the price at which the new bid submitter is attempting to buy.

Because there can be a situation where the new bid author buys fewer coins, there can be more supply than the new bid demands. In this situation, we completely fill the new bid, but we are only filling a subset of the buyable bids and we also allow the bid to be partially filled. Here, we split the bid into a trade and the remaining bid. In this situation, the author of this bid is the split bid author. The reverse situation can happen, when the new bid author is buying a lot of coins, when there is less supply than the bid demands. Here, we need to create a leftover bid for the new bid author.

The third and last step is to perform the swap between the accounts. We calculate the amount of coins that are subtracted and added to both accounts. The trading fees are always calculated in the asset that the customer is receiving, which means that both customers are paying the exchange fees. The fee is 0.1 % for both trade participants. Then we recalculate the account balance, transform the bid into a trade, and save the accounts into the account-trie.

Here we are using PostgreSQL transaction savepoints as disaster recovery. Because the accounts got saved in the account state trie with the swapped balance, we cannot let some future error in this function rollback our indexed database changes. The complete rollback would make the bids not marked as filled, meanwhile, the trie would be saved with the traded balance. This could lead to potential replaying of the bid swap with someone else, completely corrupting the account balance in the account state trie. This is a disadvantage of using two different databases, as one needs to keep them synced correctly.

Investigating Step 8. of the algorithm, sub-step B., we can see that for the calculation of the bid O author balance addition A_A, the algorithm uses the original maximum amount P_A of what the submitter is willing to pay for a single coin. This leads to a situation in which the author of the original buyable bid can receive more coins than he requested. This means that the author sold his coins at a better price than he originally proposed.

For example, let there be a non-filled bid O that wants to sell 5 BTC for 7 LTC by the customer T:

$$O_A = 5, \tag{8.7}$$

$$P = \frac{7}{5} = 1.4, \tag{8.8}$$

this bid aims to sell the amount O_A of 5 BTC, where the price per one BTC is 1.4 LTC (normalization value).

Then, a new submitter submits a bid in which he wants to sell 20 LTC for 2 BTC, customer M:

$$P_A = \frac{20}{2} = 10. \tag{8.9}$$

The maximum amount P_A that M is willing to pay for a single BTC is 10 LTC. The bid O satisfies the condition $P \leq P_A$, so this bid is buyable. Because customer M cannot buy the entire 5 BTC offered by customer T, our amount value is now limited to the maximum value that customer M can buy, which is 2 BTC. The algorithm is splitting the bid O:

$$O_A = 2. \tag{8.10}$$

Then we calculate the swap additions and subtractions (we ignore the calculation of the fees):

$$M_A = O_A = 2, \tag{8.11}$$

$$M_S = O_A \times P_A = 2 \times 10 = 20, \tag{8.12}$$

$$T_A = M_S = 20, \tag{8.13}$$

$$T_S = M_A = 2. \tag{8.14}$$

This results in the customer M partially fulfilling the bid O and obtaining 2 BTC and losing 20 LTC. But the customer T will result in selling his 2 BTC for 20 LTC. Profiting 17.2 LTC more than awaited.

In the real-world use case, the exchange operator wants to maximize his profit. To be a more greedy exchange, we can alter the calculation of the bid author A balance addition by the following:

$$A_A = O_A \times P. \tag{8.15}$$

In this way, the new bid submitter always buys the coins from the author of the buyable bid for his own original price (price per one). From the previous example, the customer M would still sell 20 LTC for 2 BTC. However, the customer T would result in the sale of 2 BTC and the gain of 2.8 LTC. The resulting 17.2 LTC would fall into the hands of the exchange. This move is completely valid, since both customers obtained what they requested.

It is not completely clear whether leaving this potential profit for the first customers who submit a fillable bid adds some incentive to submit quality bids first, or whether this engages some malicious behavior by spamming a lot of most likely unfillable bids, which exist just for the purpose of profit.

8.13 Ledger Manager

A ledger manager is a high-level exchange enclave component for managing the exchange ledger. Its main task is to obtain microtransactions, publish microblocks, and update the ledger.

In previous algorithms, there was a single step to create a microtransaction and pass it to the ledger manager. This is done by creating a Go channel that accepts only a transaction structure interface. Go channel is a special type for intercommunication between goroutines. The ledger manager acts as a consumer of this channel, accepting every microtransaction produced by the three producers, the deposit, bid, and withdrawal manager. The consumer algorithm does the following:

1. For a new microtransaction on the channel, add it to the queue.
2. If the queue size is larger than or equal to the maximum number of microtransactions in a microblock, trigger the block publishing algorithm.
3. Otherwise, if a new microtransaction was not obtained through the channel for the last 5 seconds, trigger the block publishing algorithm.

Then the invoked block publishing algorithm does the following:

1. Check that there are one or more microtransactions in the queue.

2. Pop the microtransactions from the queue, but not more than 1000.
3. Move the microtransactions to the new microblock body.
4. Build a Merkle tree from the microtransactions and calculate the root hash.
5. Build the microblock header.
6. Assemble the header and the microblock body and calculate the new block hash.
7. Add the new microblock hash to the ledger (history tree).
8. Begin a transaction T_I in the indexed database.
9. For all the microtransactions (deposit, trade, withdrawal) that are being marked by this microblock, update their microblock in the indexed database.
10. Save the microblock in the key-value database under its block hash.
11. Seal the new ledger root hash with the SGX seal key and save it to the key-value database under the latest ledger version specific key.
12. Commit the transaction T_I.
13. Notify the contract manager about the new published ledger version number and its root hash.

This algorithm is pretty straightforward, it takes the oldest not yet published microtransactions, puts them in microblock, and publishes them in a ledger. There is also a need to update the microtransactions in the indexed database and link them with their mining microblock to be able to return to the customer which transactions were published in which microblock (or version).

So that the exchange can be persistent, it stores all the microblocks (versions) and seals the latest ledger root hash on the disk into the key-value database. On every start, the exchange loads all the microblock hashes and builds the ledger again. The recalculated ledger root hash with the last sealed ledger root hash are compared, if they are equal, the tampering did not occur, and the exchange can proceed. Without this, the microblock could be edited, and the ledger recalculated.

The manager also provides all the methods to query the ledger. It can create membership and incremental proofs. In addition, all microblocks can be loaded on demand and returned to other components.

8.14 Contract Manager

The contract manager is a high-level exchange enclave component whose main task is to bind and maintain the exchange contract from section 7.2.

8.14.1 Contract Deployment

The manager attempts to deploy the contract on the first exchange run. The deployment algorithm is as follows:

1. Initialize the connection to the Ethereum node through the RPC client.

2. Sign the contract deployment transaction using the Ethereum enclave wallet.
3. Attempt to publish the transaction that deploys the contract.
4. Wait for 5 seconds. Then if the contract deployment fails, wait for 5 seconds, and jump to step 3.
5. Obtain the deployed contract address.
6. Seal the contract address with an SGX sealing key and store it in the key-value database under a specific contract key.

Because every exchange wallet is generated randomly (cryptographically secure) on the first run. The enclave does not have enough funds to deploy the contract immediately in the first run. The Ethereum exchange address is printed on the output, and the operator is required to send the Ethereum coins to the exchange address. Additionally, the moderator must periodically refill the address so that the exchange can update this contract throughout its lifetime. The exchange will not proceed until the contract is deployed, and the exchange is bound to it by the contract bindings. When the contract is deployed and the exchange is bound to it, it will request the contract info status contract view and refresh the current contract variables in the exchange enclave memory.

The contract address is sealed by the SGX enclave key and stored inside the key-value database again for the persistence of the exchange and also for tampering protection against malicious moderator. The malicious moderator could potentially forge a contract that is partially equal to the original exchange contract ABI, thus the exchange would bind to it. The malicious contract could have additional logic that benefits the moderator.

8.14.2 Contract Maintanence

As we mentioned in the previous section 8.13, the ledger manager is waiting for a new version of the ledger through the Go channel. The contract update algorithm has the following steps:

1. Listen for a new ledger version channel until a new version arrives.
2. Create a new transition pair structure:
   (a) Add a new ledger root hash and version number to the transition pair.
   (b) Add the previous ledger root hash and version number to the transition pair.
3. Call the `SetRootHash` contract method, and save the Ethereum transaction hash that makes this call.
4. Periodically request an Ethereum node for this transaction until its state is not pending.
5. Verify that the transaction receipt has a successful status, otherwise jump to step 3.
6. Update the contract variables within the contract manager through the info status contract view.

For now, the contract manager updates the latest ledger version of the exchange contract on every newly mined microblock. As the contract method updates the contract storage, it costs a certain gas to execute. The exchange enclave should not proceed with any newly published blocks until the contract is transitioned to a newer available ledger version.

```
Authorization:
did=did:ethr:0xb9c5714089478a327f09197987f16f9e5d936e8a;
sig=89eb8db7d4fe712aa1a501df752e369f062a1225fcfd12ddacd2d62dfee24ae7
```

Figure 8.7: Example of signature in exchange HTTP request using authorization header.

8.15 Rest API

Rest API is a high-level exchange component that implements a customer-facing API to work with the exchange. Currently, the implementation uses only HTTP requests, but in a production environment combined with remote attestation, it would be using HTTPS for encrypted communication. We use the minimalistic Go Web framework Echo [34], to implement the API.

8.15.1 User Control

For user control, the exchange does not use any authentication methods followed by cookies and embraces the statelessness of the REST API principles. Every request from a specific customer needs to be signed by his exchange private key stored in the DID document resolved by the third-party DID provider. The signature is forged by following these steps [1, 5]. The signing algorithm for customer $\mathbb{C}$ takes as input an HTTP request R:

1. Create a raw byte array A.
2. In the array A insert the selected HTTP method of the request R.
3. Insert in the array A the selected HTTP URL.
4. If the request has a non-empty body:
   (a) Insert the `Content-Type` header value into the array A.
   (b) Insert the body value into the array A.
5. Calculate the hash of the array A using the SHA-256 hashing function.
6. Sign this hash using the ECDSA signature using the SECP256K1 customer account private key $SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$, creating the signature S.
7. Create a new authorization header with the customer DID and signature S (example Figure 8.7).

The exchange verifies this signature S by deciphering the signature by the account public key $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$, recreating the request byte array, hashing it, and then comparing the two hashes. Maintenance of the public-private key is in the hands of the customer and the DID provider.

8.15.2 User Registration

For a customer $\mathbb{C}$ to open an account, a customer must do the following:

1. Generate an exchange SECP256K1 key-pair ($SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$, $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$).

2. Register and obtain DID and DID document at the DID provider.
3. Include the public key of the account ($PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$) in the DID document.
4. Include the public key of the address on Bitcoin ($PK_{\mathbb{C}}^{B}$) in the DID document.
5. Include the public key of the address on Litecoin ($PK_{\mathbb{C}}^{L}$) in the DID document.
6. Request the specific coin deposit address from the exchange enclave $\mathbb{E}_{\mathbb{X}}$.
7. Create a transaction that transfers funds to the exchange enclave address.
8. Send a request to the exchange deposit endpoint, with all required deposit proofs, and a signed request using the generated account private key $SK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$.

8.15.3 DID Mocking

Since this is a proof-of-concept implementation, we need to mock the DID storage. The exchange on start launches a DID mock listening on port 9999. This mock is only for local development and testing use. Mock is a simple REST API that provides the following endpoints:

- **GET /v1/diddoc/<did>**: The user provides a DID that he wants to resolve (to obtain the DID document). The endpoint returns the DID document in JSON format or HTTP code 404 if it does not exist.
- **POST /v1/diddoc**: Automatically create a DID document, the user provides account public keys and DID that he wants to use. Mock creates a DID document and stores it.

The mock store the documents inside the document map, where the key is the DID. The map is stored only in memory, and when the exchange is restarted, the map is erased.

8.15.4 DID Resolution

The DID resolution is performed by the exchange enclave when a signed request is obtained. The exchange contacts the DID provider (currently, the DID mock) by using the rest API endpoint to query the DID. The algorithm to search and verify the public key of the account of customer $\mathbb{C}$ is as follows:

1. Request the DID document at the provider, using the DID provided in the customer signature.
2. Obtain the document, for each verification method in the `verificationMethod` document section:
   (a) Check if `ID` of the verification method has anchor `#exchPubKey`.
   (b) Decode the SECP256K1 public key of the account $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ using Base58.
   (c) Parse and verify the public key $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$.
   (d) Verify that the public key $PK_{\mathbb{C}}^{\mathbb{E}_{\mathbb{X}}}$ can be used for authentication by searching for the `ID` of the public key of the account in the DID document `Authentication` section.

The returned public key is then used by the exchange for authentication, accessing the trie, etc.

8.15.5 Endpoints

The exchange API is listening on port 8000. We can group the endpoints of the exchange by their functionality. An endpoint marked with an asterisk means that the request must be authenticated (the signature must be provided by subsection 8.15.1).

Deposit

- **POST /deposit/btc***: Deposit Bitcoin or Litecoin funds and open an account if it does not exist. It expects a request body with specified coin, block hash, transaction, Merkle membership proof, a public key that belongs to the private key that signed the on-chain transaction, and a signature of this request by the private key that signed the on-chain transaction.
- **POST /deposit/btc/info**: Obtain a Litecoin or Bitcoin exchange deposit address. The coin is selected in the body of the request.

Bid

- **POST /bid***: Submit a new bid to exchange coins. The endpoint expects an amount of coins to sell, the name of the coin to sell, the amount of the coin to buy, and a name of the coin to buy.
- **DELETE /bid/{id}***: Delete the submitted bid from the order book. The ID of the bid is searchable through the account bids endpoint.
- **POST /orderbook***: Request an order book for a specific coin pair. A coin pair is a string in the format `coin_owned-coin_wanted`. For example, if a customer wants to sell Bitcoin and buy Litecoin, the coin pair is `BTC-LTC`.

Withdrawal

- **POST /withdraw/btc***: Withdraw a specific amount of Bitcoin or Litecoin from the exchange. Expects an amount to withdraw, and a specified coin.

Account

- **GET /account/balance***: Returns the customer's balance map, filled for each coin.
- **GET /account/deposits/{btc|ltc}***: Returns the customer's list of deposits for the specified coin.
- **GET /account/bids/<coinpair>***: Returns the customer's non-filled and non-deleted bids for specified coin pair.
- **GET /account/trades/<coinpair>***: Returns the customer's trades (filled bids) for specified coin pair.
- **GET /account/withdrawals/{btc|ltc}***: Returns the list of customer withdrawals for the specified coin.

Ledger

- **GET /ledger/contract***: Get the smart contract address on the Ethereum blockchain.
- **GET /ledger/latest***: Returns the latest ledger version and its details.
- **GET /ledger/confirmed***: Returns the ledger version, which is currently the latest in the smart contract, and its details.
- **GET /ledger/version/<version>***: Returns the specified ledger version and its details.
- **GET /ledger/block/<version>***: Returns the whole block published for a given ledger version.
- **GET /ledger/<version>/proof***: Generate incremental ledger proof from the specified version up to the latest version.
- **GET /ledger/<version>/confirmedproof***: Generate incremental ledger proof from the specified version to the latest confirmed version in the smart contract.

Chapter 9

Security Analysis and Benchmarks

In this chapter, we perform a security analysis and benchmark of exchange, precisely its components and actions, such as ledger, account state trie, deposit, and bidding.

9.1 Security Analysis

We define and analyze a set of potential attack vectors that could be executed against the exchange and its components. These attack vectors were selected, because they can portray attacks from malicious operator side (insider attack) and attack from outside.

The following vulnerabilities and their countermeasures are based on the fact that the Intel SGX hardware is ideal. In reality, there were several cases of existing vulnerabilities in this architecture, such as address translation vulnerabilities, cache cpu vulnerabilities, or hardware vulnerabilities [23].

9.1.1 Leaking the Exchange Wallet Private Keys

Malicious operator could attempt to read the private keys of the enclave exchange, which are used for cryptocurrency wallets. An operator would not be successful, since every private key on the disk is sealed by the SGX sealing keys.

9.1.2 Tampering with Account State Trie

A malicious operator could attempt to change the state of the account objects inside the account state trie. The attacker would be successful to edit some account in the trie since the objects stored in key-value database are not sealed. But attackers would not be successful in recalculating the root hash of the account state trie, since this root hash is sealed by the SGX sealing keys.

Enclave, upon retrieval of this corrupted account from the trie, would check and detect that the root hash of the account state trie is not equal to the sealed trie root hash. The current implementation on this tampering discovery refuses to perform any action when the trie root hashes are different.

9.1.3 Tampering with Ledger and its Microblocks

A malicious operator could attempt to change the contents (microtransactions) of the microblock. The attacker would be successful in editing the microblock, but it would not mean

much, since these microtransactions are historical and were already properly executed by the enclave. However, an attacker would not be successful in recalculating the ledger root hash of the microblocks, since this root is also being sealed by the SGX sealing key.

Even if the operator were somehow successful in deceiving the enclave, the root hash calculated from the incremental ledger proof (proof created from the tampered version up to the latest confirmed version in the contract) would not be equal to the root hash stored in the smart contract. In addition, the enclave at each launch reconstructs the ledger from the microblock hashes and compares the root hash with the sealed root hash. If the root hashes are not equal, it refuses to continue.

9.1.4 Spoofing of the Contract Address in Enclave

A malicious operator could attempt to change the address of the smart contract, to deceive the enclave to call a different one. This different contract could benefit the malicious operator.

The operator is also not able to achieve this, since the address is also sealed by the SGX seal key when stored on disk.

9.1.5 Providing Malforged Blockchain Blocks to Enclave

Since the exchange enclave synchronizes blocks from external specific blockchain nodes maintained by the operator, a malicious operator could attempt to forge fake, non-existing blocks.

The operator would not be successful, since the enclave has hardcoded genesis block hashes in the code and also keeps up with the blockchain consensus rules. Keep in mind that this means that the 51 % attacks on a given chain would be successful against the exchange.

9.1.6 Changing Environment Variables and Launch Options

A operator could attempt to change the environment variables, which would cause the exchange to connect to the different nodes of the blockchain. This is mitigated by the same behavior as mentioned in the previous subsection.

9.1.7 Tampering with Indexed Database Records

Malicious operator could attempt to edit the values of the columns in the indexed database to benefit. He could attempt to alter the data for specific bids, and then buy these cheaper by being the first who submits a new bid.

As we defined in subsection 8.3.2, a single edit of a row would be detected by verifying the SGX sealed hash stored also in this row.

9.1.8 Tampering with Sealed Values in Key-Level Database

Sealed values are encrypted by the SGX seal keys and their hash, the malicious operator cannot change this value without knowing the SGX seal key.

9.1.9 Authorizing as a Different Customer

This attack is not possible, since the customer never submits his account private key to the exchange. In the case of a leaked account private key, the funds can be stolen. Of course, this is not a problem of the exchange enclave, but an error of the customer.

9.1.10 Replaying Customers Requests

Replaying attacks are also not possible, since the enclave should provide a secure channel for communication using the SGX remote attestation, and the enclave should provide nonces, which will be used in the customer signatures. Also, for every account, the enclave keeps a nonce field in the account object.

9.1.11 Double Spending Attacks

We can propose two double spending scenarios, which we need to take into account:

1. From a deposit action side.
2. From the withdrawal action side.

Deposit

In the first mentioned attack, the attacker could attempt to send an on-chain transaction with funds to the address of the exchange enclave. Then, on the first confirmation of the transaction, the attacker would attempt to claim the deposit into his account inside the exchange enclave. If the fork (rollback) occurred on that specific chain for that specific deposit transaction, the attacker would be left with funds credited in the exchange and also funds in his wallet.

This attack is also not possible, since the enclave does the deposit validation only for confirmed blocks. This fact could discourage future customers from using the exchange since a single customer needs to wait a specific amount of time for the block with the deposit transaction to become finalized.

For example, in Bitcoin, the finality of the block is 6 confirmations (6 blocks), which results in 60 minutes of the customer waiting for his deposit transaction to be finalized and accepted by the enclave.

Withdrawal

In the second mentioned attack, the attacker could request a withdrawal of funds from the exchange and then attempt to use these funds to bid inside the enclave, performing double spending.

This attack is also not possible to execute. The exchange enclave, before submitting the withdrawal requests, attempts to delete all unfilled bids of the attacker for the withdrawn coin. Even if the enclave was at the same time in the act of performing a trade between this attacker and another customer, his account and bids would be locked for this period. This would result in the thread, which is submitting the withdrawal request to wait until the trade is done. Then the withdrawal thread would detect that the attacker does not have enough funds for his submitted withdrawal.

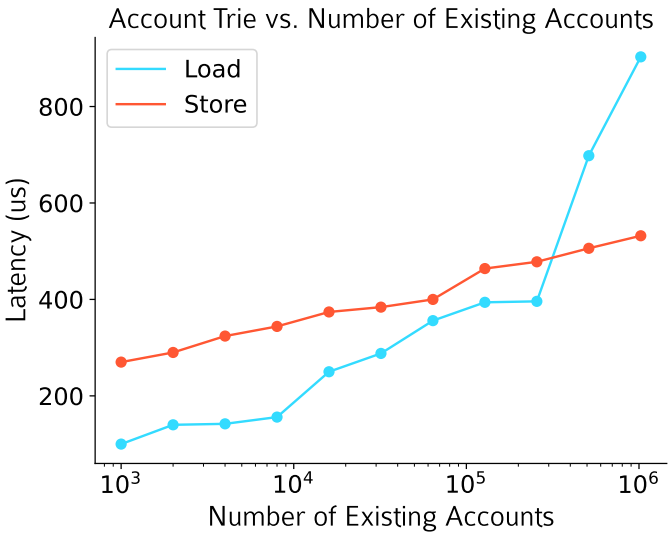

Figure 9.1: Account trie benchmark, comparing load and storage action latency vs a number of existing accounts.

After the withdrawal is submitted, the withdrawal funds are frozen inside the enclave, with which the attacker is unable to do anything. These funds are frozen until the withdrawal on-chain transaction is finalized.

9.1.12 Rollback on the Smart Contract Platform

A fork that occurs on the smart contract platform should not cause any security issues for the exchange, since the enclave only updates this contract with newer values. The customer should be acknowledged that he needs to take into account the finality of this smart contract platform, before verifying the ledger incremental proofs.

9.2 Benchmarks

We benchmark the most customer-critical operations such as depositing, bidding, and withdrawing, but also the low-level components such as the Merkle-Patricia tree or history tree. All evaluations are run on the machine using an Intel i7-8650U (4-core) CPU, 16 GB RAM, and SSD storage disk.

9.2.1 Account State Trie

Since account state trie is one of the integral components of the exchange, we benchmark the performance of this structure. We measure the average latency of the load and storage action of a single account compared to the increasing number of accounts.

Averaging the latencies of the benchmark in Figure 9.1, we conclude that the average latency for an account load from the trie of size 1,000 up to 1,000,000 account objects takes around 348 us. For account storage, the average latency is around 397 us. The account state trie scales well with an increasing number of accounts.

The most important disadvantage of this structure is the need for an exclusive access lock when a store action is being made. This means that at an average rate of 500 us for

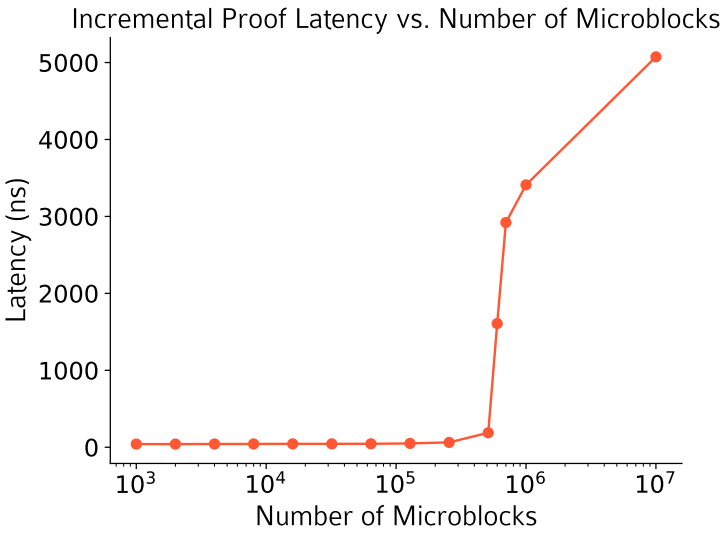

Figure 9.2: Ledger benchmark, comparing latency of incremental proof creation vs a number of microblocks in ledger.

a single store, the exchange can update at most 2,000 accounts per second. Keep in mind that this number only represents an ideal situation. In a more realistic situation, when a bunch of customers requests at the same time their balance, there will be some time spent as load action holds the reader lock.

9.2.2 Ledger

Since the ledger is an integral part of the exchange, we benchmark the ability of the exchange to provide incremental proofs and the action of inserting a new microblock hash, thus creating a new version.

The ledger has no problem scaling with a lot of new blocks, of the size of 1,000 to 10,000,000 microblocks, the average time to create an incremental proof was around 0.973 us, as shown in Figure 9.2. The history tree is able to create incremental proofs in parallel by acquiring the reader lock, but cannot create proofs when a new block is being inserted.

The benchmark measuring the average insertion latency can be seen in Figure 9.3. The average insertion latency when working with the history tree of sizes $1,000$ up to $10,000,000$ blocks is around 21 us.

9.2.3 Bid

Since this action is integral for an exchange, we tested two specific benchmarks. The first benchmark opened an account for 100 users, and then requested to submit a new bid at the same time. With around 50,000 and more already existing available bids in exchange, the exchange was able to resolve around 23 bids per second for a single coin pair.

The second benchmark in Figure 9.4 was focused on examing the degradation of the bid resolution latency, as the exchange increases the amount of available bids.

Investigating the logs and profiler, the main bottleneck of this action is caused by Step 2 of the algorithm at subsection 8.12.2. This lock causes exclusive access to a table `bid`

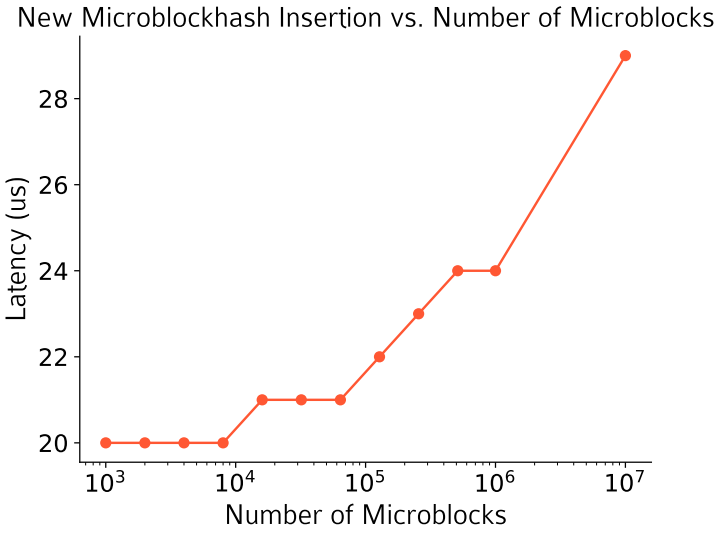

Figure 9.3: Ledger benchmark, comparing latency of new microblock hash insertion vs a number of microblocks in ledger.

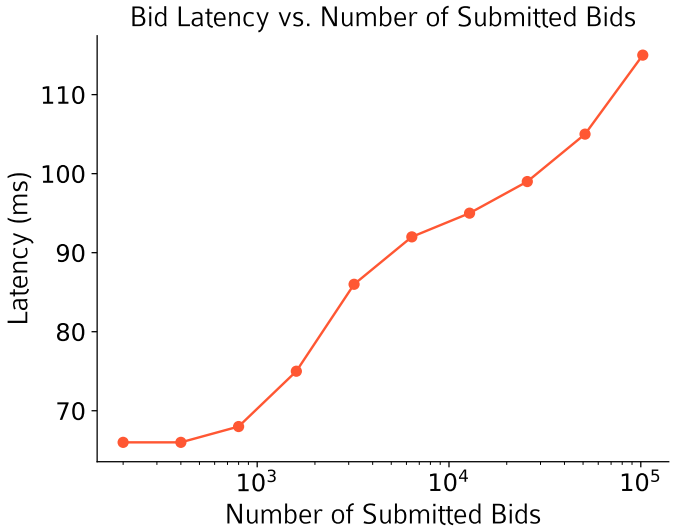

Figure 9.4: Bid benchmark, comparing latency of bid resolution vs a number of submitted bids at an exchange.

partition for a single coin pair. Without this lock, there is room for a data race, where two bids would not get matched, even if they were supposed to. This could lead to bids getting stuck unmatched.

The second source of bottleneck is the integrity scheme mentioned in subsection 8.3.2. Every selected, updated, or inserted bid must be rehashed and ciphered with the SGX sealing key, an exchange cannot update the bids in bulk using PostgreSQL `UPDATE` operation.

The third source of the bottleneck is the account state trie. A Merkle-Patricia tree cannot be updated in parallel, which becomes a problem when performing a coin swap between the two accounts in Step 8., Substep i of the algorithm in subsection 8.12.2. Since an update of a single account can be performed exclusively by only one thread, swaps between the accounts are serialized. The situation can be portrayed as follows. If there exist a group of small unfilled bids and a new big opposite bid gets submitted (someone with a lot of funds wants to sell for a good price), then all these small bids will be filled. To be exact, if there are N bids to be filled, we need to do a swap between the new bid author and at most N other accounts. This means that the exchange would need to perform at most $2 \times N$ saves to the trie, one at a time. At an average rate of 3 ms to load and store from the account trie, then at 5 swapping bids, where each bid is from a different customer, it would take $2 * 3 * 5 = 30$ ms to perform the swap between accounts.

9.2.4 Deposit

The next benchmark attempts to measure the average throughput of the deposit action. This benchmark was performed by grouping more than 100 future customers who attempted to send a deposit request at the same time, which opens an account. Before these deposits were executed, there were more than 10,000 accounts already existing in the account state trie. On average, the exchange is able to verify and commit 35 valid deposits per second.

The second benchmark in Figure 9.5 was focused on examing the degradation of the deposit resolution latency, as the exchange increases the number of customer accounts in the account state trie.

When inspecting the profiler results, a single deposit that opened an exchange account took 33 ms. The execution time of the deposit can be divided into these steps:

- DID resolution and verification of the user control signature. (13 ms)
- Deposit on-chain transaction proof validation. ($\leq$ 1 ms)
- Check and insert a new account object into the account state trie and indexed database. (3 ms)
- Account locking. (2 ms)
- Deposit insert into the indexed database. (14 ms)
- Update and save the account in the account state trie. ($\leq$ 1 ms)

Looking at the logs of the indexed database and the profiler, the database signature scheme (subsection 8.3.2) plays a role in the impact on performance. Since we use the table `TableCounts` a single row update locks the row until a transaction is committed (until a deposit is completed). This situation creates a bottleneck, where updates of the row belonging to the protected table `deposit`, need to be performed one by one. We converge to the well-known dilemma of speed versus security.

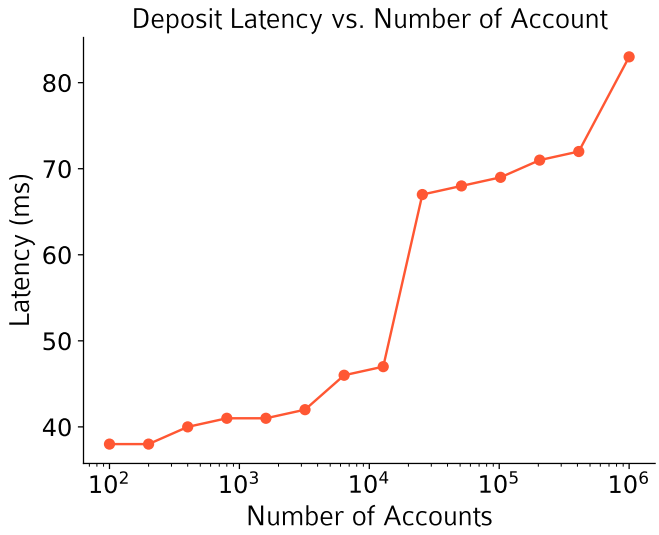

Figure 9.5: Deposit benchmark, comparing deposit latency vs a number of already opened accounts.

9.2.5 Contract Gas Usage and Exchange Finality

For the current smart contract deployment on the Ethereum blockchain, the transaction consumed exactly 1,026,414 gas. Today (26 April 2023), a single unit of gas costs 51 Gwei. When this is converted to Ethereum, the deployment cost 0.052347114 ETH, which is $1,843.80.

Updating the root hash of the ledger in the contract using the update method costs 50,029 gas, which with the current cost of gas is 0.002551479 ETH, precisely $4.77. At the rate of publishing a new microblock every 5 seconds, the exchange would publish exactly 17,280 blocks daily. Translating every block publication into a single contract update, the daily cost that the enclave would spend is $82,426. This cost is not realistic to provide, thus we need to find a compromise between exchange security and cost.

Even if the exchange cannot be rollbacked (a fork cannot happen), customers can have a need to verify the microblocks as soon as possible. For this reason, we can update the ledger root hash in the contract every 12 blocks (1 minute), which ends up in 1,440 contract updates per day, which costs 3.6742 ETH, precisely $6,899 daily.

Chapter 10

Conclusion

The main objective of this thesis was achieved as we proposed a centralized exchange model that uses the Intel SGX platform, so the customer can use the power of attestation and there is a secure gap between the customer and the operator using the sealing. In combination with a smart contract on a public blockchain, we achieved the desired non-equivocation and exchange snapshotting. We also reviewed existing solutions to this problem and other technologies closely related to it, such as Tesseract [7] and Aquareum [26].

We discussed important preliminary parts of typical exchange environments, such as blockchains, trusted computing, and exchanges, and, most importantly, we mentioned possible attack vectors for these exchanges and blockchains.

We implemented the proposed protocol as a proof-of-concept using the Go programming language, PebbleDB as key-value data storage, and PostgreSQL as our relational database. In combination with the SGX sealing primitives, we described the need for a data integrity solution outside of the enclaves. We resolved problems such as user control, bid matching, deposit validation, and withdrawal of funds, and we properly defined their algorithms.

Next, we performed a security analysis of possible vulnerabilities from both the insider and outsider attacker perspective, and showed their mitigations. As for benchmarking, the implementation was able to resolve around 35 deposits for a single coin per second and around 23 bids for a single coin-pair per second. We also benchmark the scaling of both the Merkle-Patricia tree and the history tree, where both showed that they can scale well up to millions of nodes. The main source of the bottleneck was the integrity scheme and exclusive edit access to the Merkle-Patricia tree. We also measured that at the frequency of each minute update of the ledger contract, the exchange requires 3.6741 ETH daily.

There is still a lot of room for improvement in this work, such as integrating a non-UTXO based blockchain coin or the possibility of implementing a proof-of-solvency [9] proofs. A proof like this shows that the exchange owns enough funds on-chain to cover all liabilities of the customers. From the optimization perspective, there could be a new and faster approach to account state storage, such as only updating the Merkle-Patricia tree when the new block is being published. From the contract perspective, the contract could be enhanced to accept a new ledger transition pair in the form of history tree incremental proof, instead of the current simple transition pair.

From a personal perspective, I got to know trusted computing platforms, and I have solidified my knowledge about blockchain, especially very specific implementation details such as transaction scripting.

Bibliography

[1] Amazon Web Services Inc.. *Signing AWS API requests*. Available at: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-signing.html.

[2] Anceaume, E., Pozzo, A., Rieutord, T. and Tucci Piergiovanni, S. On Finality in Blockchains. arXiv. 2020. DOI: 10.48550/ARXIV.2012.10172. Available at: https://arxiv.org/abs/2012.10172.

[3] Antonopoulos, A. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, 2017. ISBN 9781491954362.

[4] Aponte Novoa, F. A., Orozco, A. L. S., Villanueva Polanco, R. and Wightman, P. The 51% attack on blockchains: A mining behavior study. *IEEE Access*. IEEE. 2021, vol. 9, p. 140549–140564.

[5] Apple Inc.. *Apple News Security Model | Apple Developer Documentation*. Available at: https://developer.apple.com/documentation/apple_news/apple_news_api/about_the_apple_news_security_model.

[6] Arm Ltd.. *Trustzone for cortex-M – ARM®*. 2016. Available at: https://www.arm.com/technologies/trustzone-for-cortex-m.

[7] Bentov, I., Ji, Y., Zhang, F., Breidenbach, L., Daian, P. et al. Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1521–1538. CCS '19. DOI: 10.1145/3319535.3363221. ISBN 9781450367479. Available at: https://doi.org/10.1145/3319535.3363221.

[8] Bormann, C. and Hoffman, P. *Concise Binary Object Representation (CBOR)* [Internet Requests for Comments]. STD 94. RFC Editor, December 2020.

[9] Buterin, V. *Having a safe CEX: proof of solvency and beyond*. November 2022. Available at: https://vitalik.ca/general/2022/11/19/proof_of_solvency.html.

[10] Buterin, V. et al. A next-generation smart contract and decentralized application platform. *White paper*. 2014, vol. 3, no. 37, p. 2–1.

[11] Cockroach Labs. *PebbleDB, key-value based storage.* 2023. Available at: https://github.com/cockroachdb/pebble.

[12] Crosby, M., Pattanayak, P., Verma, S., Kalyanaraman, V. et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*. 2016, vol. 2, 6-10, p. 71.

[13] Crosby, S. A. and Wallach, D. S. Efficient data structures for tamper-evident logging. In: *USENIX security symposium*. 2009, p. 317–334.

[14] Dahlberg, R., Pulls, T. and Peeters, R. Efficient sparse merkle trees. In: Springer. *Nordic Conference on Secure IT Systems*. 2016, p. 199–215.

[15] Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X. et al. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: IEEE. *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, p. 910–927.

[16] De, N. FTX Files for Bankruptcy Protection in US; CEO Bankman-Fried Resigns. *Coindesk*. November 2022. Available at: https://www.coindesk.com/policy/2022/11/11/ftx-files-for-bankruptcy-protections-in-us/.

[17] Edgeless Systems. *Edgeless RT*. 2022. Available at: https://github.com/edgelesssys/edgelessrt.

[18] Edgeless Systems. *EGo*. 2022. Available at: https://github.com/edgelesssys/ego.

[19] Edgeless Systems. *ERTGo, Go compiler for Edgeless RT*. 2022. Available at: https://github.com/edgelesssys/ertgo.

[20] Ethereum Foundation. *Proof-of-stake (POS)*. 2020. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[21] Ethereum Foundation. *The Solidity Contract-Oriented Programming Language*. 2022. Available at: https://github.com/ethereum/solidity.

[22] Ethereum Foundation. *Abigen*. 2023. Available at: https://github.com/ethereum/go-ethereum/tree/master/cmd/abigen.

[23] Fei, S., Yan, Z., Ding, W. and Xie, H. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)*. ACM New York, NY, USA. 2021, vol. 54, no. 6, p. 1–36.

[24] Geppert, T., Deml, S., Sturzenegger, D. and Ebert, N. Trusted Execution Environments: Applications and Organizational Challenges. *Frontiers in Computer Science*. 2022, vol. 4. DOI: 10.3389/fcomp.2022.930741. ISSN 2624-9898. Available at: https://www.frontiersin.org/articles/10.3389/fcomp.2022.930741.

[25] Homoliak, I., Breitenbacher, D., Hujnak, O., Hartel, P., Binder, A. et al. SmartOTPs: An air-gapped 2-factor authentication for smart-contract wallets. In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 2020, p. 145–162.

[26] Homoliak, I. and Szalachowski, P. Aquareum: A Centralized Ledger Enhanced with Blockchain and Trusted Computing. arXiv. 2020. DOI: 10.48550/ARXIV.2005.13339. Available at: https://arxiv.org/abs/2005.13339.

[27] Homoliak, I., Venugopalan, S., Reijsbergen, D., Hum, Q., Schumi, R. et al. The security reference architecture for blockchains: Toward a standardized model for studying vulnerabilities, threats, and defenses. *IEEE Communications Surveys & Tutorials*. IEEE. 2020, vol. 23, no. 1, p. 341–390.

[28] Intel. *Intel®Software Guard Extensions Programming Reference.* 2014.

[29] Intel. *Intel® Software Guard Extensions Developer Guide.* Intel Corporation, 2016.

[30] Intel. *Overview on Signing and Whitelisting for Intel® Software Guard Extension (Intel® SGX) Enclaves.* Intel Corporation, 2018.

[31] Jinzhu. *GORM, ORM library for Golang.* 2022. Available at: `https://github.com/go-gorm/gorm`.

[32] Johnson, S., Scarlata, V., Rozas, C. V., Brickell, E., McKeen, F. X. et al. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. In:. 2016.

[33] Kubach, M., Schunck, C. H., Sellung, R. and Rossnagel, H. Self-sovereign and Decentralized identity as the future of identity management? *Open Identity Summit 2020.* Gesellschaft für Informatik eV. 2020.

[34] LabStack. *Echo, High performance, extensible, minimalist Go web framework.* 2022. Available at: `https://echo.labstack.com/`.

[35] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K. and Song, D. Keystone: An Open Framework for Architecting Trusted Execution Environments. In: *Proceedings of the Fifteenth European Conference on Computer Systems.* 2020. EuroSys'20.

[36] Microsoft. *Microsoft/EEVM: Enclave Ready EVM (eevm), an open-source, standalone, embeddable, C++ implementation of the Ethereum Virtual Machine.* 2020. Available at: `https://github.com/microsoft/eEVM`.

[37] Ménétrey, J., Göttel, C., Khurshid, A., Pasin, M., Felber, P. et al. *Attestation Mechanisms for Trusted Execution Environments Demystified.* Springer International Publishing, 2022. DOI: 10.1007/978-3-031-16092-9_7. Available at: `https://doi.org/10.1007%2F978-3-031-16092-9_7`.

[38] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review.* 2008, p. 21260.

[39] Oosthoek, K. and Doerr, C. From hodl to heist: Analysis of cyber security threats to bitcoin exchanges. In: IEEE. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* 2020, p. 1–9.

[40] Open Enclave. *Open Enclave SDK.* 2022. Available at: `https://github.com/openenclave/openenclave`.

[41] PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* 2023. Available at: `https://www.postgresql.org`.

[42] Rosenfeld, M. *Analysis of Hashrate-Based Double Spending.* arXiv, 2014. DOI: 10.48550/ARXIV.1402.2009. Available at: `https://arxiv.org/abs/1402.2009`.

[43] Sabadello, M., Guy, A., Reed, D. and Sporny, M. Decentralized Identifiers (DIDs) v1.0. July 2022. Available at: `https://www.w3.org/TR/2022/REC-did-core-20220719/`.

[44] Satybaldy, A., Subedi, A. and Nowostawski, M. A Framework for Online Document Verification Using Self-Sovereign Identity Technology. *Sensors*. MDPI. 2022, vol. 22, no. 21, p. 8408.

[45] Scarlata, V., Johnson, S., Beaney, J. and Zmijewski, P. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. *White paper*. Intel Corporation. 2018.

[46] Sebastian, Sinclair and Eliza, Gkritsi. Japan's Liquid Global Exchange Hacked; $90M in Crypto Siphoned Off. *Coindesk*. August 2021. Available at: https://www.coindesk.com/markets/2021/08/19/japans-liquid-global-exchange-hacked-90m-in-crypto-siphoned-off/.

[47] Shadders, S. *Merkle Proof Standardised Format - Bitcoin SV Technical Standards*. Bitcoin Association, Jun 2021. Available at: https://tsc.bitcoinassociation.net/standards/merkle-proof-standardised-format/.

[48] Southurst, J. Mt. Gox Files for Bankruptcy, Claims $63.6 Million Debt. *Coindesk*. February 2014. Available at: https://www.coindesk.com/markets/2014/02/28/mt-gox-files-for-bankruptcy-claims-636-million-debt/.

[49] Stallings, W. *Cryptography and Network Security: Principles and Practice, Global Edition*. Pearson Education, 2018. ISBN 9781292158594.

[50] Williams, T. MyBitcoin Incident Report - August 5th 2011. august 2011. Available at: https://archive.is/LUMzs#selection-267.122-267.306.

[51] Wong, D. *Real-World Cryptography*. Manning, 2021. ISBN 9781638350842.

[52] Wood, G. et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2014, vol. 151, no. 2014, p. 1–32.

[53] Yaga, D., Mell, P., Roby, N. and Scarfone, K. Blockchain technology overview. National Institute of Standards and Technology. October 2018. DOI: 10.6028/nist.ir.8202. Available at: https://doi.org/10.6028%2Fnist.ir.8202.

Appendix A

Contents of the Storage Medium

- `root`
  - `poster.pdf` ..... Poster submitted for Excel@FIT conference
  - `src` ..... Source codes for the exchange implementation
    - `exchange` ..... Exchange implementation
      - `contract_src` ..... Smart contract implementation
  - `thesis` ..... LaTeX source codes for this thesis
  - `xsasak01.pdf` ..... This thesis in PDF
  - `README.md` ..... Guide for running the exchange and smart contract