Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Deparment of Telecommunications and Media Informatics

Supporting multi-stakeholder industrial processes with blockchain technology

Scientific Students' Association Report

Author:

Bence Oláh

Supervisors:

Dr. Pál Varga
Attila Frankó

2021

Contents

Kivonat

A többszereplős ipari folyamatokban gyakran szükség van adatok gyors, biztonságos megosztására. Ebben a kérdéskörben az egyik aktuális trend az ipari digitalizációban a blokkláncok használata – azaz egyelőre ennek a lehetőségnek a vizsgálata.

A blokklánc technológiák utat nyitottak az elosztott, biztonságos és megmásíthatatlan adattároláshoz. Egyes blokkláncok lehetővé teszik a fejlesztőknek, hogy kódot írjanak, amiket gyakran okosszerződésnek neveznek, majd a kódot felírják a hálózatra és ott futtassák azokat. A blokklánc technológia természetéből fakadóan semmilyen adat nem változtatható meg azután, hogy a hálózatra íródott, beleértve az okosszerződések kódját is. Mint bármely más kód, így az okosszerződések kódja is tartalmazhat hibákat, amik javításra szorulhatnak. Ezt a javítást azonban úgy kell eszközölni, hogy ez közben megfeleljen a blokklánc alapelveinek, elsősorban azok megmásíthatatlan és elosztott jellegének. Figyelembe véve a tényt, hogy sok blokklánc publikus, valamit rosszindulatú szereplők számára is hozzáférhető a kód, kiemelkedő jelentőséggel bír a robosztus okosszerződések fejlesztése, amik a lehető legbiztonságosabbak, és képesek megbirkózni a lehetséges sérülékenységekkel. Még mindig vitatott, hogy az okosszerződések frissítése elfogadható-e, mivel ez megsérti a fent említett elveket – és ha elfogadható, akkor hogyan kellene frissíteni.

Ebben a dolgozatban az okos-szerződések robosztusságát növelő megvalósítási minták mellett egy olyan módszert is bemutatok, amely kiegészíti a hibák javítását és az okosszerződések frissítését célzó, már létező megoldásokat egy szavazást megvalósító mechanizmussal. Egy művelet elvégzéséhez vagy az okosszerződés állapotának megváltoztatásához egy csoportnak szavaznia kell, hogy egyetértenek-e a javasolt változtatásokkal. Ezzel a fejlesztőknek lehetőségük nyílik, hogy egyszerűen megvalósítsanak egy szavazás-alapú döntéshozatali mechanizmust, amellyel elérhető az elosztott, demokratikus frissítési és állapotváltoztatási folyamat. A bemutatott módszer célja, hogy közelebb hozza az okosszerződések frissítési folyamatát a blokklánc technológia alapelveihez azáltal, hogy az megfelel a decentralizáltság elvének, miközben a hagyományos eljárások, amelyek során egy vagy több ember hajtja végre a frissítéseket - de szinte kizárólag egy címről valósul meg - megsértik ezt az elvet.

Abstract

In multi-stakeholder industrial processes, fast and secure data sharing is often required. In this area, one of the current trends is the usage of blockchains in industrial digitalization – more precisely, recently the examination of this possibility.

Blockchain technologies provide a way for distributed, secure, and immutable data storage. Some types of blockchains allow developers to write code, known as smart contracts or chaincode, that can be deployed to the network and are able to execute their code. Due to the nature of blockchains, no data can be changed after writing it to the network, including the code of smart contracts. However, just like any other code, smart contracts can contain bugs that need to be fixed while still adhering to the main concepts of blockchains, especially their immutability and decentralized feature. Considering the fact, that many blockchains are also public, and the code is accessible to malicious parties, it is paramount to develop robust smart contracts, that are as secure as possible and can deal with the potential vulnerabilities. It is still debated if updating contracts is acceptable since it violates the two aforementioned features, and if it is acceptable, how it should be done.

In this paper I present a method that enhances the existing solutions for fixing bugs and upgrading contracts with a voting mechanism. To perform an action or to change the state of the contract, a group of people has to vote whether they agree with the proposed changes. This allows developers to easily implement a voting-based decision-making mechanism that enables a decentralized, democratic update or state change process. The proposed method aims to bring the process of updating contracts closer to the core principles of blockchains by adhering to the decentralization concept, compared to the usual way, when updating is performed by one person or a group of people, but almost exclusively from one address, which contradicts the decentralization principle.

Chapter 1

Introduction

Since the introduction of Bitcoin [1] in 2008, blockchain technologies gained popularity among companies and tech enthusiasts alike, lately even going mainstream thanks to the success of cryptocurrencies. While individual users mainly interact with the most popular blockchains such as Bitcoin or Ethereum [2] and use them as a speculative asset or a store-hold of wealth, companies can see different values in the usage of blockchains. Even though some companies – e.g. Tesla – hold significant amounts of Bitcoin and had accepted it as a form of payment in the past, the majority of them uses a customized, mostly private blockchain to amend their existing tools or to support cooperation with other companies in their supply chains. Some of the most notable industries that benefit greatly from the usage of blockchains include healthcare, energy, financial sectors, Internet of Things (IoT), supply chains, and smart cities [3]. Multi-stakeholder processes include at least two parties, commonly from different industries, that could connect in many different ways – such as through the Arrowhead framework that is based on Service Oriented Architecture principles [4]. The players in multi-stakeholder processes are in a partnership, and cooperate in one or more fields, such as decision-making, governing, logistics, and manufacturing processes. These partnerships highly value the properties provided by blockchain technologies, namely: security, fault tolerance, high reliability, and immutability [5]. Figure 1.1 illustrates an example of a multi-stakeholder supply chain, the type of stakeholders, and the some of the services they offer to each other. Beside exchanging goods, data is also exchanged in these partnerships.

Although the main features are mostly the same for all blockchains, their level of presence might differ. Ethereum, for instance, allows developers to create smart contracts, moreoverany participant of the network is able to see every transaction that happened on the chain. This, combined with its strong immutability feature makes the impact of having vulnerabilities in the code devastating. Therefore, it is crucial for developers to create robust and secure smart contracts that are resilient to attacks and can mitigate the effects caused by the exploitation of a vulnerability. Utilizing the industry-standard best practices and patterns significantly decrease the number of possible attack vectors, however, some of them make contracts conform less to blockchain's decentralization principle, which could be a problem in industrial use cases. In multi-stakeholder scenarios, or even within the same company, decentralization, trust, and control are key factors for an efficient workflow

and maintaining long-term operation and partnership. Furthermore, accounts on blockchains are generated from a private key which, – as its name suggests – cannot be shared with anyone, therefore, it is challenging to control or limit specific abilities in industrial settings when many devices with individual accounts are under the same ownership or managing entity.

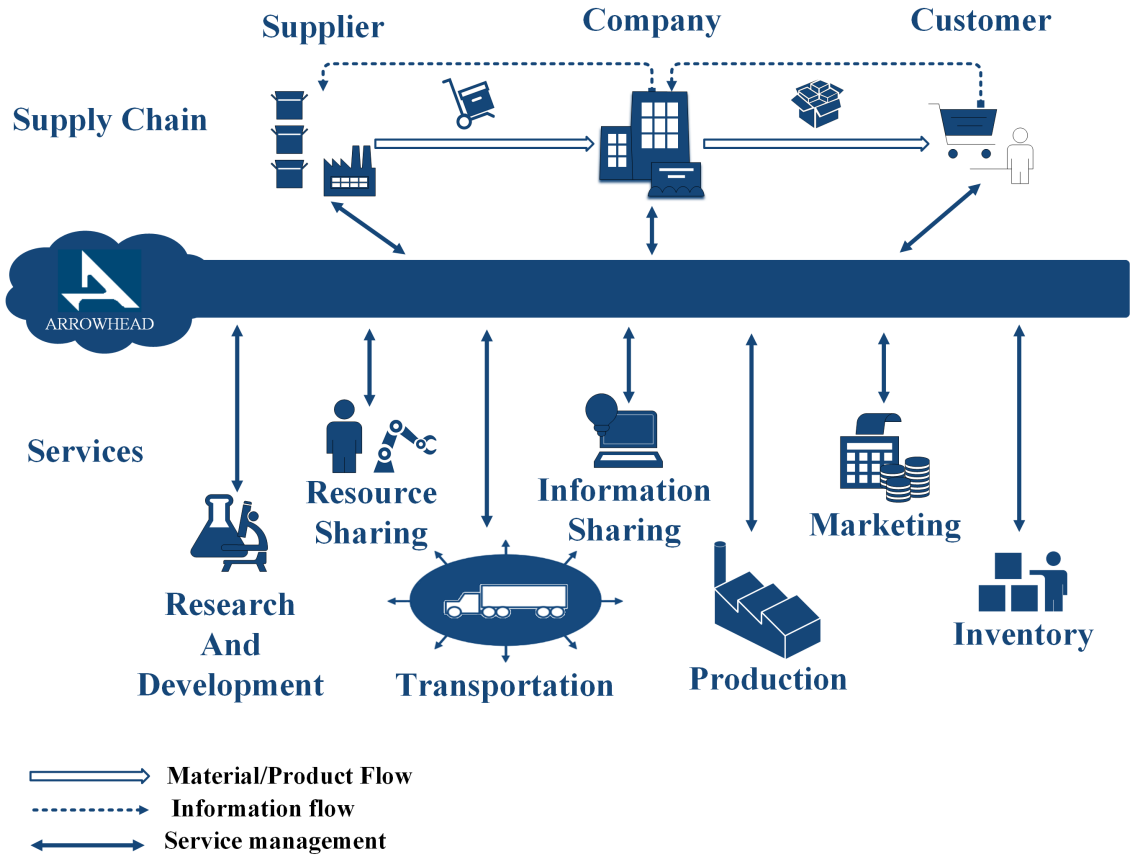

Figure 1.1: An illustrative example of a multi-stakeholder supply chain [4]

In this paper, I propose an approach that allows companies to exert control over their devices while also leaving them as a separate entity with the potential of fine-grained authorization control and spending limits. To enhance this ability, I further propose a method that gives companies the potential to utilize an on-chain decision-making mechanism that supports corporate governance by enabling owners to vote on specific proposals. This mechanism is not exclusive to companies or organizations. Furthermore, it can be useful for numerous decentralized applications and even to amend the capabilities and characteristics of existing smart contract design patterns, since their usage is of paramount importance to the development process of robust smart contracts.

The solutions have been validated and verified through use cases presented in this paper. The work presented Chapters 4 and 5 are my novel contributions, together with the actual smart contract design and implementations presented in Chapter 3 – together with the applications of design patterns aiming for robustness.

Chapter 2

Related works

2.1 Blockchain Technologies

A blockchain is a special form of a distributed database, where records, or transactions get stored by grouping individual transactions into blocks. Each block consists of a header and a body. The block header stores information about the block itself, like the creation time and the block number. Additionally, it also stores information about the preceding block by including the hash of it, thus creating a structure of consecutive blocks. Due to the fact that every information about a block is hashed and stored in the following block, tampering with any data is very difficult since it would require the recalculation of the hash value of every subsequent block, which is practically infeasible under real conditions. Each transaction is signed with the private key of the sender which makes it impossible for anyone to impersonate someone on the network or alter their transaction data.

There are many characteristics of blockchain technologies that vary widely depending on the concrete implementation. Some blockchains are public and permissionless, while others are private. There are also major and fundamental differences in terms of the applied consensus mechanisms between various blockchains. Further information, a comprehensive analysis of the characteristics, and a blockchain taxonomy can be found in [6].

2.2 Ethereum and Solidity

Ethereum [2] is one of the most popular blockchains right now. It is public and permissionless, meaning anyone can join the network without restriction. A great advantage of Ethereum is that it is not only able to record cryptocurrency transactions, but also allows developers to create applications that can be deployed to the network. These applications are known as smart contracts, and network participants can interact with them. Smart contracts are commonly written in Solidity [8], a language resembling JavaScipt and C++, and after deploying them to the network, they run on the Ethereum Virtual Machine.

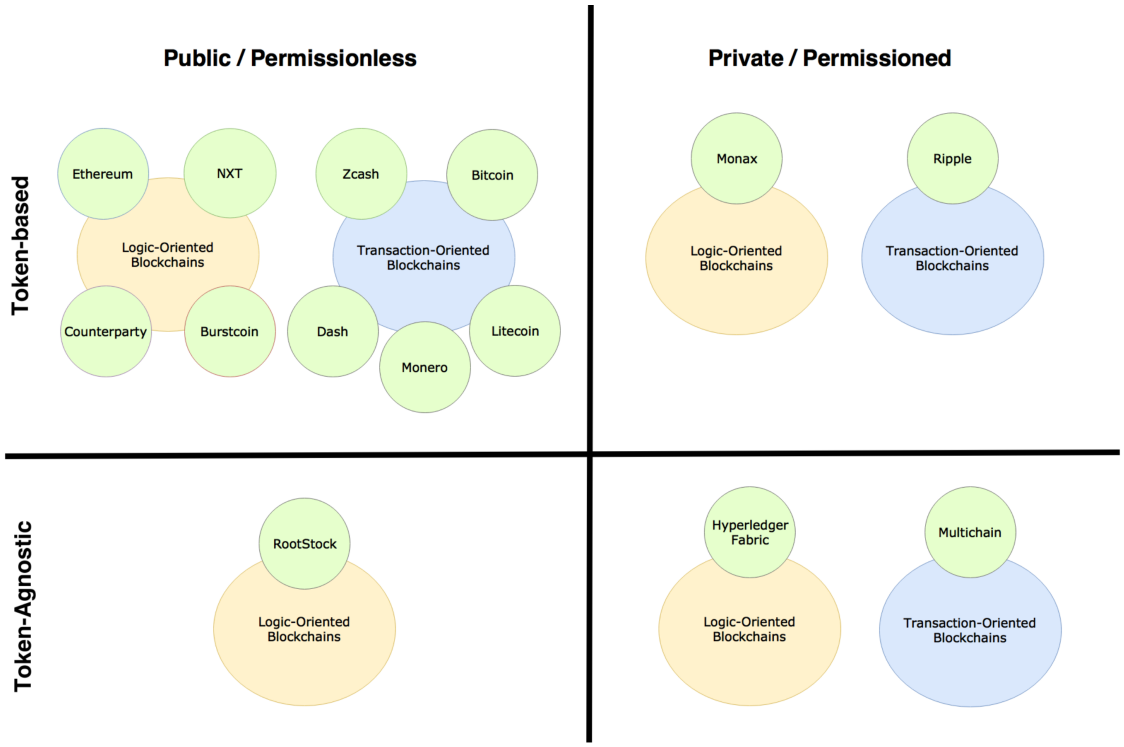

Figure 2.1: Blockchain taxonomy [7]

Ethereum distinguishes two types of accounts: everyone with a keypair on the network has an Externally Owned Account (EOA), while the second type consists of the deployed contracts. They both have a public address that can be used by other accounts to identify them, and both can receive funds, but contracts cannot initiate transactions without themselves being the recipient of another one. Transactions that change the state of the blockchain cost *gas*, which has to be paid in *ether*, the native currency of Ethereum. The amount of *gas* used is proportional to the computational resources that the particular transaction uses, so developers are incentivized to write simple and efficient code to reduce costs.

2.3 Smart Contracts

Smart contracts are self-executing codes that run on specific blockchains. Not every blockchain is capable of running smart contracts, out of those that support this feature, one of the most widely used is one Ethereum. As mentioned earlier, a very common programming language used by developers in the Ethereum ecosystem to create smart contracts is Solidity, however, there are alternatives to it, such as Vyper [9], which is a Python-like language. Any participant can create smart contracts and deploy them to the network, but deploying costs *gas* just like any other transaction does that modifies the blockchain.

The complexity and capabilities of smart contracts vary widely, ranging from those that provide very basic functionality to the immensely complex decentralized applications like Aave [10], a lending application, or Augur [11], which is a betting

platform. An analysis of decentralized finance (DeFi) applications, including decentralized exchanges, asset management, and debt markets, can be found in [12].

Ethereum also supports the creation of tokens, which are described by smart contracts compliant with one of the token standards, for example, the ERC-20 standard. Major tokens can be exchanged on decentralized exchanges like Uniswap [13]. ERC-721 tokens are non-fungible tokens, meaning every token is unique. These tokens are commonly used to represent the ownership of different assets such as property rights, paintings, artworks, collectible items, etc.

2.4 Robust Smart Contracts

It is crucial for a smart contract to be safe and bug-free , since deployed contracts cannot be updated. This means that the developers have no way to patch the already deployed smart contracts as they can release security fixes for traditional software, so the code will contain the bug forever, leading to the possibility of exploits, that can cause serious losses, just like it happened in the case of the infamous TheDAO attack [14]. A classification of the most common attacks against blockchains and smart contracts, including the one used against TheDAO, can be found in [15].

The definition of smart contracts can go wrong in various different ways – the underlying reason is mostly the lack of technical understanding. In order to avoid such defects, creators of smart contracts should follow certain best practices and rigorously validate and verify the behavior of the smart contract before unleashing it in the wild. A good reference on the possible smart contract defects can be found in [16], where the authors provide ideas on preventing such defects, as well as describing the effects of such errors in Ethereum-based blockchains. When discussing smart contracts on Ethereum, further description of best practices to avoid security vulnerabilities can be found in [17] and in [18].

In this section, I give a brief introduction and description of some of the existing and widely used patterns that are of great importance for developing robust and secure smart contracts.

2.4.1 Access Restriction

Public functions in Solidity can be called by anyone on the network, either by an account or by another contract. Obviously, this is not acceptable most of the time, since it can pose serious security risks, especially, if a contract handles funds on behalf of other accounts. To be able to exert control over the identity of the caller, or even the time of the call, the *Access Restriction* pattern is implemented in contracts. Using this pattern, developers can write such functions that ensure a transaction can only be completed if specific preconditions hold. The most common use case of this pattern is to check if a caller is authorized to call a given function: most commonly, they check if the caller is the owner of the contract, and if not, the transaction is reverted automatically. Other use cases include a time constraint, meaning, a function can only be called or an action can only be taken if a specific

amount of time has passed. Developers can check against a given state, and only allow the calling of the function if the state of a given variable is desired.

2.4.2 Checks Effects Interactions

The *Checks Effects Interactions* pattern is a best practice used when there is an external call in a function. Contracts can call other contracts, and when this happens, the called contract executes its own code. This allows an attacker to create such contracts that act in a malicious way, circumventing the intended control flow of the transaction. The most common way to do this is when the malicious contract re-enters the originally called contract before the previous transaction could have been completely executed. This allows the attacker to exploit the weakness of a vulnerable contract, which is the order in which the checks of conditions, setting variables, and calling external contracts happen. If an external call precedes the lines that check if the caller is authorized to call the function or the setting of variables, the attacker can re-enter the function as long as the gas runs out, or – in the usual case when the aim is to steal funds – the contract runs out of ether. To mitigate this risk, a function should first check the eligibility of the caller, and other preconditions that might be required to hold. Then, contrary to what the common practice might be in other cases, the state variables should be updated, even though the actions have yet to happen. Lastly, the external call should be made. This way, if a malicious contract re-enters the original, all variables are updated, thus rendering the re-entrancy attack ineffective.

2.4.3 Emergency Stop

If a vulnerability is discovered in a smart contract, it is virtually impossible to fix it. Even if some patterns allow upgradeability to some degree, creating and deploying the new version of the contract can take a significant amount of time, which is not acceptable due to the risk of serious, harmful, and destructive effects that can be caused by exploiting the vulnerability. The *Emergency stop* pattern helps to mitigate this risk by giving developers a tool that allows certain functions to be stopped by an authority in case of an unforeseen, serious event that poses a great risk to the contract and its assets. This way, critical functions can be shut down to protect assets of the contract until the vulnerability gets fixed. Depending on the implementation, the halting of the contract might be permanent if there is no way to fix the vulnerability and it poses a serious threat. A good reference point could be OpenZeppelin's [19] Pausable contract module [20] that implements the *Emergency Stop* pattern and could be used by other contracts to take full advantage of the benefits that it provides.

2.4.4 Ownership

The *Ownership* pattern is widely used by developers due to its crucial role in smart contracts. This pattern is actually a type of the *Access Restriction* pattern, in which accessing to certain functions is restricted to one address. Most of the time it is the address that deployed the contract and therefore is considered to be the owner of it. While it might be a type of another pattern, it is worth noting on its own, since it is one of the most commonly used patterns, so much so that OpenZeppelin created a contract module [21] that provides the required functionality while also allowing of the ownership role for the management.

2.4.5 Proxy

OpenZeppelin proposed a method to solve the problem of upgrading already deployed smart contracts caused by the immutability feature of the blockchain. By default, deployed contracts cannot be modified; however, changing certain variables, writing special functions, and separating logic and storage can enable developers to upgrade features or even whole smart contracts if needed. The authors proposed several ways to this method in a blog post [22] on OpenZeppelin, called *Proxy Patterns*. Every method faces the same problem of separating storage, which makes it possible to update the underlying logic without losing data or needing costly rewrites. They achieve this by using a permanent proxy contract, an intermediary between the caller of the called contract, and the called contract itself. This contract does nothing but forwards calls to the logic contract and the result back to the caller. It does not change at all, its address remains the same so the callers do not notice any change when updates happen to the logic contract. When developers update or add some functions, they deploy a new logic contract that contains the updated code, and the proxy contract will then communicate with this new contract. This way, the changing of the logic is completely transparent to the callers.

The proposed ways are *Inherited Storage*, *Eternal Storage*, and *Unstructured Storage*. Each of them solves the problem, but they have some characteristics which make one preferred over the other. Namely, the structure of the *Unstructured Storage* makes it the best choice in the majority of use cases, because it is fairly easy to implement, and the existing contracts do not need to be modified at all, they can be used with the proxy without any changes. (This approach is very similar to the *Contract Relay Pattern* found in [23], and the *Proxy Delegate* pattern found in [24], however, the *Contract Relay Pattern* is an outdated version that is not recommended to use due to its inability to return result values.) Further details and some important things about the *Unstructured Storage*, and sample contracts can be found in [25] and [26].

Chapter 3

Smart Contract use cases

The initial idea that motivates my work is that the usage of blockchain technologies brings many benefits to stakeholders in numerous industries, including manufacturing, finance, and trading.

In this chapter, I present a system of smart contracts that manages the exchange of assets, keeps ownership records of them, and handles payments utilizing an ERC20 compliant [27] smart contract. The solution presented here realizes a supply chain management tool, where an asset can be tracked throughout the whole chain, including suppliers, manufacturers, wholesalers, retailers, and customers. Blockchain's immutability, traceability, and transparency features can build trust and provide accountability in this scenario, while also speeding up payments compared to legacy solutions. It can also serve as a trading platform where both B2B, B2C, and C2C transactions can take place.

This solution – i.e. the system of smart contracts – was a part of a central bank digital currency (CBDC) project that demonstrated a CBDC system and its integration and cooperation with industrial smart contract applications. The project was funded by the Hungarian National Bank (MNB), and carried out by a selected research group at BME that I became part of, during the project.

The use case solutions presented in this chapter provides a visible, physically available environment for my research. The work presented in the upcoming chapters are my novel contributions, together with the actual smart contract design and implementations presented in this chapter – together with the applications of design patterns aiming for robustness.

3.1 Asset management

For managing and keeping track of availability, amount, and ownership of different types of assets, there is an asset contract. This contract stores every asset type and their parameters, such as the address of the owner, the available quantity, and the unit price. Anybody can add a new type of asset, view and buy existing assets, and manage their own assets. An owner can increase or decrease the quantity available for purchase, and update the unit price of a given asset that they have.

Anything that has to do with the assets happens on the asset contract, but managing payments is the responsibility of the token contract. The token contract keeps track of the balances of the individual accounts, verifies transactions, and transfers funds between accounts.

The asset contract is linked to the token contract, so every time someone buys an asset, the token contract needs to be called, and is also called by the asset contract. It is similar to when someone wants to sell their house, they ask a real estate agent to sell the house on their behalf. The agent is not the owner of the house, but they are authorized by the seller to sell it.

The following sections describe the actual functions that the stakeholders can use in the various scenarios.

3.1.1 Adding new assets

A user can add a new asset that they want to sell by calling the *createAsset* function. This function takes two parameters: the first one is the unit price, that specifies how much one unit of the new asset costs, and the second one is the quantity of the new asset that is up for sale. The new asset will have a unique ID, that identifies the asset, and the asset will be tied to the caller of the function (the seller of the asset).

3.1.2 Buying an asset

3.1.2.1 Before calling the *buyAsset* function

By default, calling only the *buyAsset* function of the asset contract will result in reverting the transaction because the asset contract is not allowed to transfer funds on behalf of the buyer. To prevent this, before buying a specific asset by calling the *buyAsset* function, the buyer must call the *approve* function of the token contract first that authorizes the asset contract to initiate the transfer of the funds from the account of the buyer to the account of the seller. The allowance is passed as a parameter of the *approve* function as well as the address of the asset contract. After calling this function, the asset contract will be authorized to spend money from the buyer's account (but not more than the allowance), so it is recommended to call the *approve* function every time before buying an asset with the exact amount given as the parameter.

3.1.2.2 Buying the asset

To buy an asset, a user has to call the *buyAsset* function with two parameters: 1) the ID of the asset, 2) the number of units he/she wants to buy. If the requested amount exceeds the available amount of the given asset, the transaction will be reverted. This function will call the *transferFrom* function of the token contract, which transfers the total amount of money that needs to be paid to the seller for the assets. If the buyer has authorized the asset contract to transfer the necessary amount from their account before trying to buy the asset, and they have sufficient

funds, the transaction will be successful, and the buyer becomes the new owner of the asset(s).

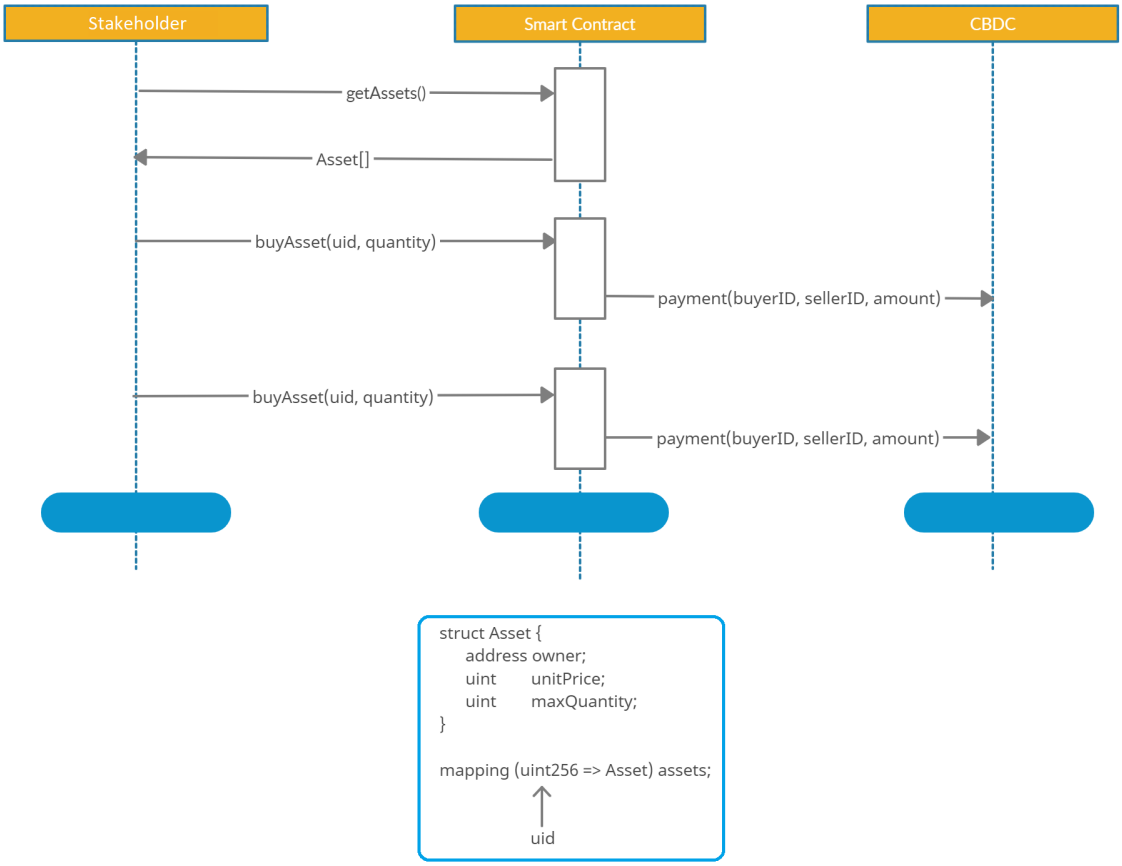

Figure 3.1: Sequence chart of buying assets

3.1.3 Increasing or decreasing quantity

It is necessary for an owner to be able to update the available quantity of an asset. It might be needed, for example, if there is a restock or some units become damaged and are no longer in a condition to be up for sale. In this case, the owner of an asset can call the *addAsset* or the *removeAsset* function to increase or decrease the quantity, respectively. Both functions take to parameters: the ID of the asset, and the difference between the old and the new quantities. In case of the *removeAsset* function, if the quantity to be subtracted is larger than the current quantity, the transaction will be reverted and the existing quantity will not be changed. The quantity of a given asset can only be change by its owner.

3.1.4 Updating the unit price of an asset

Another useful feature is the ability to change the unit price of an asset. One thing to keep in mind is that it is not possible to differentiate between units of the same asset. Changing the unit price results in every unit costing the new amount of

money thereafter. If it is needed to have a separate batch of an asset that costs more or less than the others, a new asset has to be created with the new price and the quantity of the batch.

3.2 Cooperation with the CBDC system and physical devices

In order to deploy the smart contracts in the described context, a suiting blockchain-type had to be chosen. Because of previous experiences, I developed and implemented the contracts of the system in Solidity [8] – hence I needed to deploy them to an Ethereum-based network. For this purpose, the solution that suited the given use case and requirements best was to host a private Ethereum network instead of using the Ethereum Mainnet or other public Ethereum test networks. This was mainly because of security and convenience considerations, since the participants of a private network can be limited to trusted stakeholders, and – regarding convenience – transaction fees did not have to be paid or testnet ethers required, thus allowing more scalability and cost-effectiveness. I set up and hosted the private Ethereum network in an environment that provided an API for devices and the CBDC system, thus they were able to connect to the blockchain.

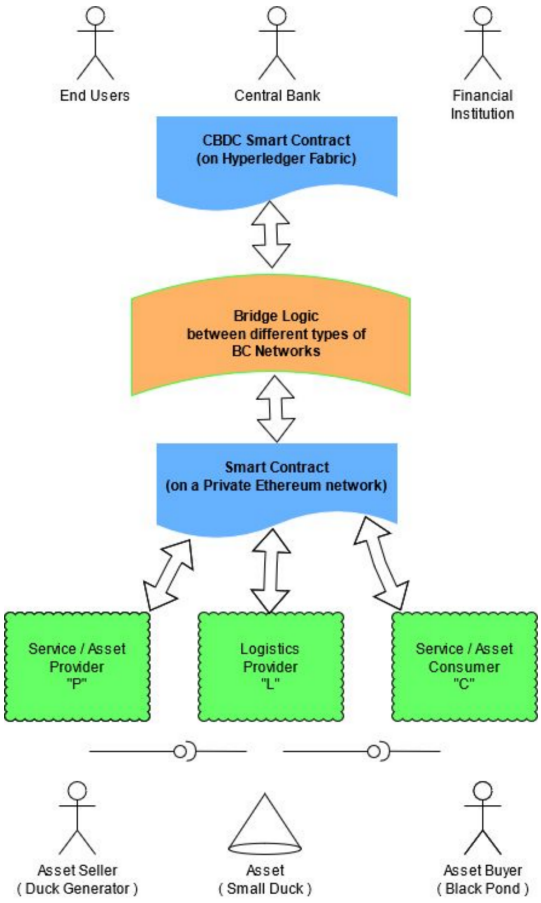

Figure 3.2: The simplified view of the complete system [28]

Figure 3.2 depicts the simplified view of the complete system, which consisted of (i) the CBDC system, (ii) the bridging logic, (iii) the asset management application, and (iv) the physical devices.

The asset management application connected to the CBDC blockchain through the bridging logic, which enabled the CBDC system to issue digital currency to the accounts on the Ethereum blockchain. These accounts were tied to the devices that handled the assets, therefore in response to a transaction, the appropriate actions could be taken.

In order to demonstrate the different scenarios, a series of interactions and transactions occurred between an asset manufacturer, a courier, and a buyer. The devices that had these roles were two robotic arms and a conveyor belt, where the arms represented the roles of the manufacturer and the buyer, while the conveyor belt served as the courier. First, the manufacturer produced and sold the asset – which was a rubber duck – to the courier, then the courier sold the asset to the buyer. In the process of the transactions, the asset management application interacted with both the physical devices and the CBDC system. As a transaction was approved, (i) the payment went through the bridging contract, (ii) the ownership of the asset got transferred in the asset management contract, and (iii) the seller's device physically moved the asset to the buyer's device.

Figure 3.3 presents the physical environment in which the stakeholders have actually exchanged the assets - in this case: rubber ducks. This physical setup provided an excellent base to my work, since each ownership change in the smart-contract had a physical effect: the provider blue robotic arm put the asset to the conveyor belt (belonging to another stakeholder), which then moved it to the buyer, which could then take it both physically and ownership-wise in the smart-contract. The creation of this physical environment was a result of another Scientific Student's Association paper, presented in the previous year by my peers in our laboratory [29].

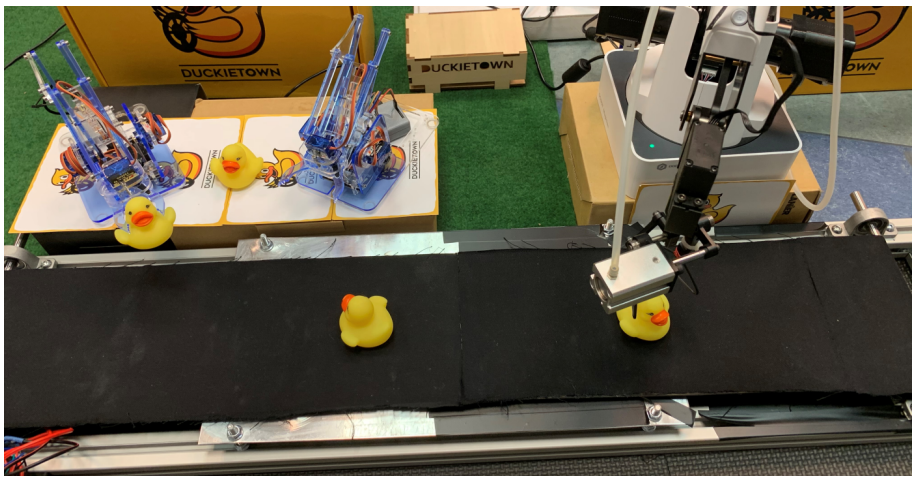

Figure 3.3: The physical demonstration environment - in this case two blue robotic arms on the producer side, and one robotic arm at the buyer side; the logistics is provided by a conveyor belt in between. This environment was provided as part of [29]

In this chapter, I presented a system that handles the exchange of assets on behalf of both individual users and companies. However, companies – that might have multiple hundreds of different devices with separate accounts – have very different requirements and use cases than individual users do, consequently, they need a solution to efficiently manage accounts that are under their supervision and administration.

In the next chapter, I am going to present an approach that gives companies the ability to manage the authorization and spending limits of those accounts that are under their control.

Chapter 4

Making transactions as a part of a company on the Ethereum blockchain using smart contracts

The benefits of using blockchain (BC) technologies for recording transactions usually outweighs the drawbacks of it. The speed, reliability, security, immutability and traceability, to name a few advantages provided, can be of great value for most companies. However, there could be obstacles in some special cases.

One of the basic principles of many blockchain scenarios is that the ledger is distributed, so there are no authorities, and every account is equal. These are usually useful, but in a business-to-business case or within some industrial setting, it might be needed to have some sort of authority over a group of accounts. Companies are usually organized hierarchically, and consist of many divisions or even have subsidiaries. Since a division is a part of a company, there are certain dependencies – and divisions are often not autonomous entities, so it is restricted what they can do. Logically, it doesn't make sense to give away accounts that are fully independent to every device the company wants to connect to the blockchain (or use its ledger).

Companies' acceptance of using blockchain technologies in their operations can only be improved if the problem of authority over their own accounts is solved, or if there is a viable alternative that overcomes the related obstacles.

4.1 Individual and shared accounts for individual but company-bound devices

Suppose there are multiple companies that are working together, or are part of a supply chain, for example, the manufacturer, suppliers, shipping companies etc. They are considering using blockchain to track where each part of the product or the final product itself is in the supply chain.

Let's look at an archetypical shipping company. They might have devices such as cranes, chain-hoists, robotic arms or conveyor belts in their parcel center, where they

want to keep track of the locations of the assets (in this case: packages). Tracking can simple mean that we know for each given point of time: which robotic arm has the asset or which conveyor belt is carrying it.

The abstract model for tracking "ownership" of assets in this setting is that when the device takes or handles a given asset, it takes the "ownership" for it as well. The device does not only create a record in an event log about this, but at the same time "pays" for the ownership, as well.

As an example: if a robotic arm picks up a package, it sends a transaction to the blockchain that it has the package, and "pays" a token at the same time (it might be the case that a robotic arm can only handle a predetermined number of packages a day). Since these devices are in the same company (or the same division), they have to be grouped together on blockchain, and establish some form of hierarchy.

4.1.1 Issues with using individual accounts

A possible solution could be that every device has an own account on the given blockchain, with an own balance. This way it is obvious where an asset is and which device handles it, since the device becomes the 'owner' of the asset. The problem with this approach is that it is not possible to deauthorize a device to handle assets, because every account (i.e., *externally owned account* on the Ethereum blockchain) has the same rights, and is independent of the other accounts. Another problem is that anybody on the blockchain can see the exact location of every asset (which device handles it). For example, the manufacturer doesn't need to know the exact location of an asset in the supplier's factory (even if it is just an account address), it only needs to know which member of the supply chain has the particular asset.

4.1.2 Issues with using shared accounts

The other approach could be that every device use the same account. It solves the problem of individual locations, but introduces many more. Obviously, an outside party could not tell the exact location, but nobody else either. The company would not be able to tell where an asset is without individual identifiers, such as wallet addresses. The other problem is even worse: since every device uses the same address, every one of them knows the private keys of the account, so they could send any transaction they wanted to, which is unacceptable and could cause serious losses.

4.2 Companies represented through smart contracts

On many BC platforms (i.e. including the Ethereum blockchain), it is possible to deploy smart contracts, which is a huge advantage compared to other blockchains

without this feature. Using smart contracts, it is possible to solve the aforementioned problems.

One possible solution is the following. Every company is represented as a smart contract on the blockchain, where it has an address, just like the externally owned accounts, making the use of smart contracts invisible to the outside world. Inside the smart contract, there are two key-value lists: the first one (i.e., *authorized*) tells if an address is part of the company and therefore allowed to initiate transaction on behalf of the company. The other one (i.e., *allowance*) stores the maximum amount each address is allowed to spend from the company's balance. The combination of the two lists makes it possible to easily manage the individual limits and rights of every device of the company.

Using this approach, every device has an own, unique address, so it is known exactly where an asset is, and it is possible to keep a history of previous transactions of each device.

The company's smart contract has to be deployed by an administrator of the company, who will authorize and manage the allowances of the accounts that are in the same logical group that the smart contract represents (e.g. a floor, office, division, the whole company).

In this initial example, the group is a company, but it can be any subset or superset as well. Initially, every individual address is excluded from the list of authorized addresses, so none of the accounts can initiate transactions on behalf of the company. The authorized addresses are added by the administrator, and by default, their allowance is 0. Allowances also have to be defined explicitly for each address to ensure that every account has the sufficient allowance and to prevent overspending. The administrator can also deauthorize addresses. In this case, the address will no longer be able to initiate transactions in the name of the company, and at the same time, the allowance of the address will be set to 0 to prevent inconsistencies in the state of the contract.

The transaction steps on how an asset's ownership gets exchanged in our model is depicted by Figure 4.1. The blockchain stores various *Asset Contracts* and *Token Contracts*. The Asset Contract stores the ownership information regarding the given asset – in this case, bound to the Company address. The Token Contract stores information of the token balances of the individual wallets as well as the balances of company contracts, and transfers the tokens between them. Besides, as the Figure shows, the company takes care of authorising and keeping track of the allowances of their "devices" (that are represented as wallets).

Those devices that are parts of a company, instead of calling the main contract (that handles transactions, assets etc), have to call the contract of the company. The individual devices do not have separate balances, instead they can spend from the company's balance, as long as their allowance is enough to pay for the asset. If the caller address is authorized in the company, and the total cost of the transaction is less than the allowance of the caller, the company contract will handle the transaction and buy the asset. In this case, the bought asset will be owned by the company contract, so the outside world will only know which company has the asset, but not the exact device inside the company. The company contract will

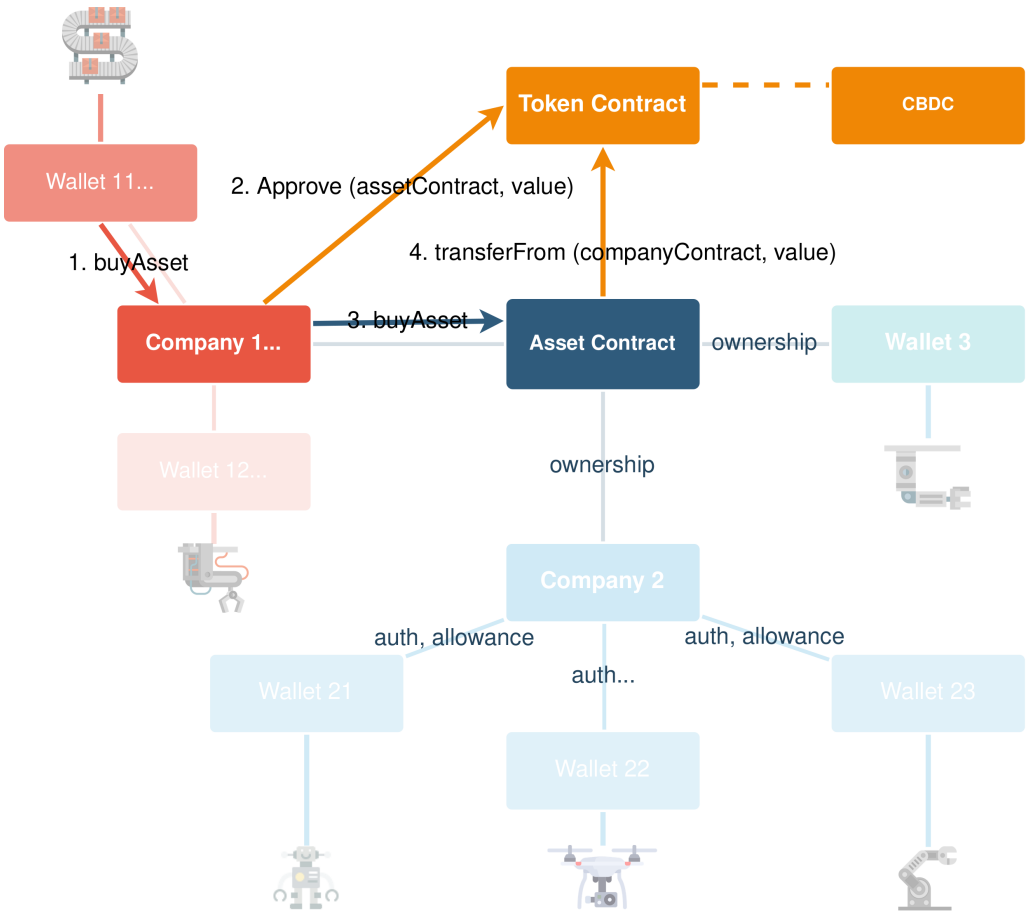

Figure 4.1: Steps of a transaction initiated by a member of a company

record the transaction and the address of the device that initiated the transaction. After the transaction, the buyer device's allowance will be lowered by the amount of the transaction cost.

Representing companies as smart contracts has multiple benefits, such as:

- Authorization management
- Set spending limits to prevent overspending
- Realistic ownership (an asset is owned by a company, not by a device)
- Hides the exact address from the outside world
- Assets can still be tracked inside the company

This method also allows that a device can be a part of multiple companies at same time, and can initiate transactions on their own, not in the company's name while being registered as a member of one or more companies. This is safe and does not present any threat to the company's balance, because the individual accounts don't have tokens (that belong to the company) tied to their addresses, so it is not

possible that an address uses the company's funds and becomes the owner of the asset instead of the company. Besides this, accounts can have own funds that they can spend however they want. This is completely independent of their allowances at specific companies.

By utilizing the presented approach, the problems discussed earlier can be solved efficiently and transparently. At the same time, it is easy to implement and ready to be incorporated into the existing workflow by only requiring the updating of the address of the recipient on devices. The result will be realistic in terms of hierarchy and ownership, and the authorization of devices can be managed quickly and easily.

4.3 Implementation

An implementation of the contract was made to demonstrate that the idea of companies as smart contracts is in fact a possible solution to the initial problem. The code was written in Solidity [8] and the 0.8.4 compiler version was used to compile the contract.

Figure 4.2 shows the class diagram of the Company contract. It can be seen that the constructor needs the address of a token contract (i.e., the address of a deployed, ERC20 compliant token contract) and the address of another contract (that (in the example) manages the ownership of assets), so they have to exist when the company contract is deployed to the blockchain. However, this implementation can be changed whenever the model changes.

Figure 4.3 shows the class diagram of the Asset contract, which the Company contract communicates with. It also needs the address of the same token contract that the Company contract uses, to ensure correct token transfers. Like in the case of the Company contract, the address of the token contract can be changed later. In the event of the Asset contract changing its token contract address, it is the responsibility of the administrators of the companies to change their addresses accordingly and keep them up to date at all times.

Figure 4.4 shows the class diagram of the ERC20 token contract. Any contract can be used as a token contract that is ERC20 compliant. The Asset contract calls the transferFrom method inside of its buyAsset method to initiate the payment for the assets. For the payment to be successful, any buyer, whether it is an externally owned account or a contract, has to approve the asset contract to transfer funds on their behalf.

The contracts were deployed to a private Ethereum blockchain, then a series of test were carried out, including buying from an unauthorized address, buying from an address that is authorized but has an insufficient allowance to buy an asset, buying from an address that is authorized and has sufficient allowance to buy an asset, calling methods from a non-admin address that can be called only from the admin address, etc.

The contract behaved as expected and every test case was successful during the test.

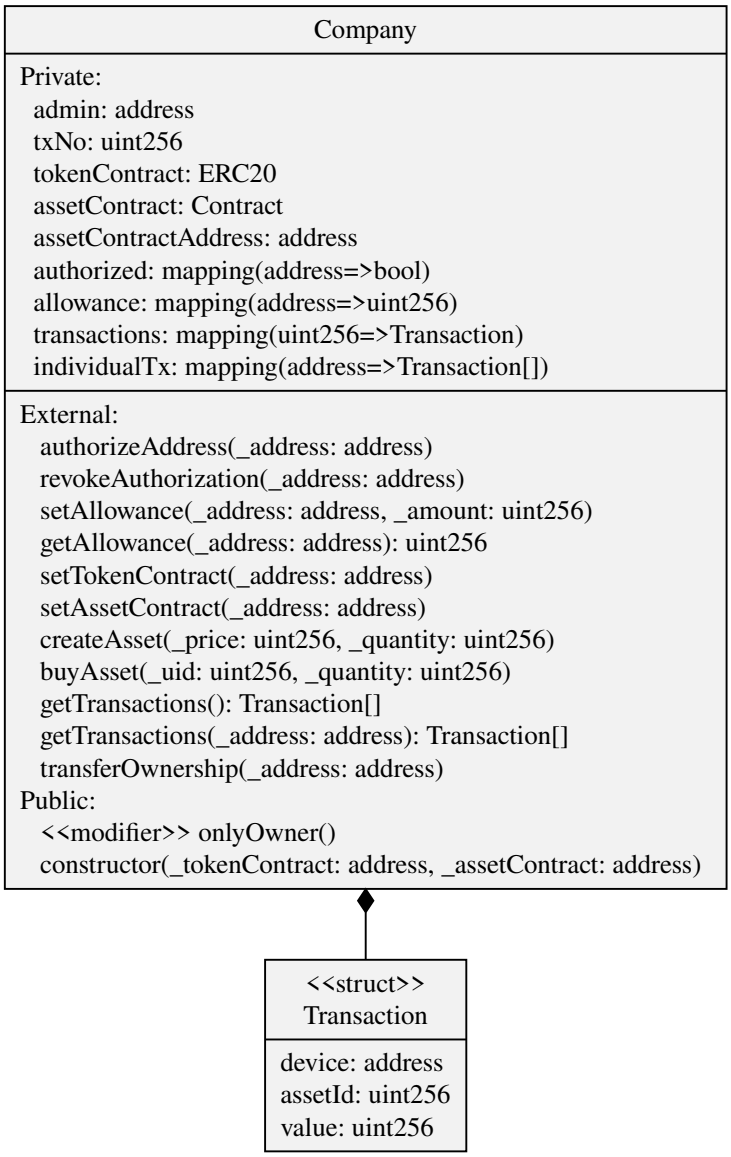

Figure 4.2: Class diagram of the Company contract

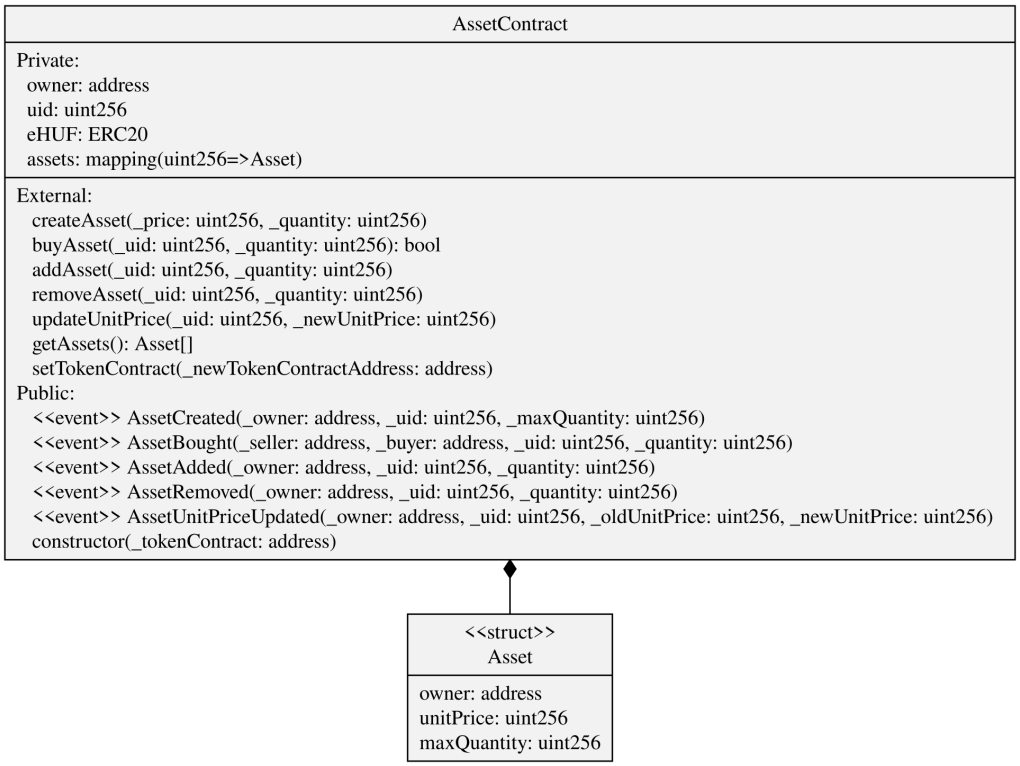

Figure 4.3: Class diagram of the Asset contract

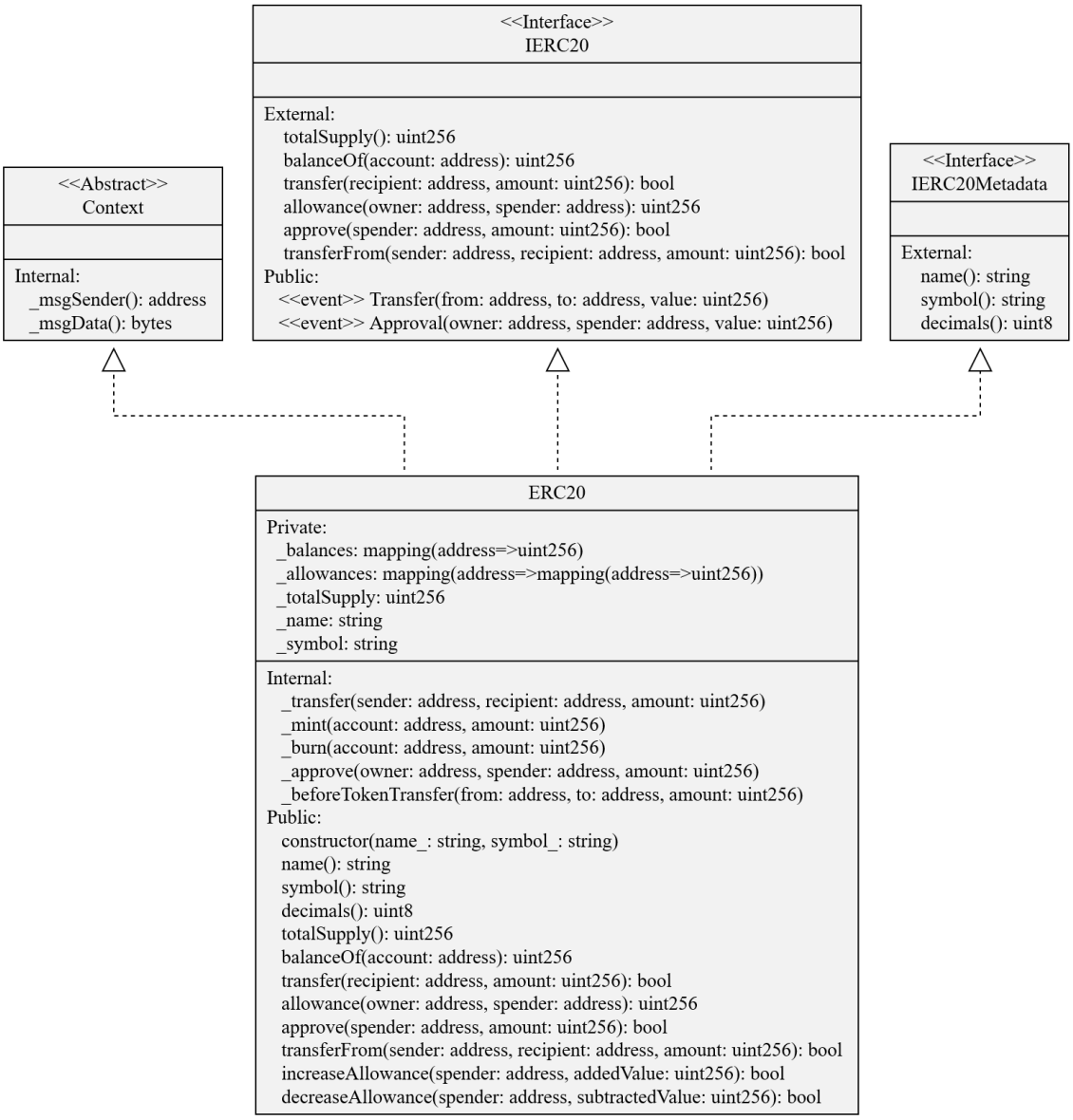

Figure 4.4: Class diagram of the ERC20 token contract

Chapter 5

Voting mechanism in company contracts

Major decisions that affect the operation or the long-term strategy of the entire company usually don't come from one person. Most of the time, a specific group, such as the board of directors, makes these decisions by voting on a given question. The board of directors usually consists of key people that represent the interests of shareholders of the company, and indirectly, the interest of its users or consumers.

In Solidity, one of the most commonly used patterns is the *Ownership* pattern. This pattern allows the creator of the contract to write functions that can be only called by one account, which is the one that represents the creator. This gives the developer nearly unlimited power and authority over the contract, and the possibility to do whatever action or modification they want. In this case, the users of the contract have no choice but to blindly trust the developers that they won't impose any changes that they don't agree with or that contradict the original goal of the contract.

This is probably unacceptable in the real world, even in the case of a company that is fairly centralized in terms of decision-making and changing its policies, yet the ownership pattern and many contracts virtually do exactly this. They – unintentionally – centralize the power to make modifications to the state of the contract or change certain variables that ultimately affect the way how a contract works or behaves.

In the following, I am going to summarize the problems associated with the existing solutions, namely, the one-address method that the *Ownership* pattern and numerous contracts use. I am also going to give a brief explanation of why it would be inadequate to use a list of owners instead of one owner without applying further control mechanisms. Then, I'm going to give a detailed explanation of the proposed method that solves numerous problems of the mentioned existing methods.

5.1 Issues with the existing methods

5.1.1 Issues with using a single address as an owner

When the owner (or admin; the two terms are interchangeable in this case, since these are not differentiated roles most of the time) of the contract is just one address, it is stored as a variable in the contract. The owner's address can either be an EOA (Externally Owned Account) or a contract address. We should look at each case separately because each of them poses different types of risks, but both of them have the same fundamental problem, that is the problem of being a single point of failure, and having too much power without any control.

5.1.1.1 Issues with an EOA as the owner

In the case of the owner being an EOA, the biggest problem we have to deal with is the human factor, meaning, we have to fully trust a single person that they will not act in a malicious way. Furthermore, it gives them an opportunity for blackmailing since they are capable of causing serious and wide ranging negative effects. Even if we ignore the human factor, and we have a very strong reason to assume that the owner can be fully trusted, the risk of the account being hijacked, or the private keys of the account becoming compromised is still there. If that happens, the attacker gains full control over the account and can do anything that the owner can, which would also lead to the aforementioned serious problems.

5.1.1.2 Issues with a contract as the owner

When the owner of a contract is another contract, the problems we face are very similar to the ones that arise when the owner is an EOA, however, there are some key differences. If the owner of the contract is an EOA, every action is directly initiated by that one address (and therefore, by one person most of the time), causing a very high level of dependency on one person, directly. But if the owner is a contract itself, the aforementioned problems affect the owned contract more indirectly, and the level of dependency is highly defined by the concrete implementation of the owner contract. The owner contract can call any function, but we can't be sure what the actual trigger is that initiates an action. Those can be wide-ranging factors from the case when one address can directly initiate the action without any other checks or actions, thus basically single-handedly controlling the owned contract through the owner contract, making the owner contract a proxy contract, to the case when there are multiple, complex mechanisms built in the owner contract that greatly reduces the significance of one person's action and leads to a more decentralized decision making when implemented correctly.

Unfortunately, this high level of uncertainty makes it very hard to trust an outside party and to give them full control over our contract, therefore there needs to be another solution that the developers have more control over, but with great decentralization features.

5.1.2 Issues with using a list of address as owners

Previously, we saw that EOAs and contracts cannot be a viable alternative as an owner because of their characteristics that give them too much power and make them a single point of failure. It is clear now that it's not sufficient if one address controls the contract, therefore an ideal solution would implement a design that somehow utilizes the power of decentralized governance, which obviously requires multiple addresses that will collectively decide on changes and proposals.

The first thing that would come to mind – instead of one address as an owner, which is stored as a variable in the contract – is to use a list of addresses that stores which accounts are in the group of owners regarding the given contract. In this case, every account that is on the list becomes an owner, where every account has the same rights and they are equal in every way. Seemingly, this achieves the desired goal of having a decentralized decision-making system, however, there are serious problems with this approach. This, in fact, achieves the goal of having multiple parties in charge instead of just one, thus making the ownership and governance more democratic, but it still has the same fundamental problem as the EOA ownership does, namely, having too much power, and in this case, multiplied by the number of owners. Every owner account has unlimited power over everything in the contract, just like if there was only one owner, including the ability to add or remove owner addresses. In this case, we assume that (i) the group of owners represents the highest level of authority in the scope of the contract, and (ii) there is no other party that appoints or removes members, which then would become the problem of the EOA or contract ownership.

If one of the owners was malicious and tried to change the state of the contract or alter it in some way, it would be quite easy for the other owners to reverse the changes and restore the original state, but they would not be able to prevent the attack in any way. If there are no events emitted when something changes that requires owner privileges, the modification of the contract state can remain unnoticed for a significant amount of time, which can lead to a state where too much contract data gets compromised, making it harder for other owners to restore the original state. Furthermore, a malicious owner could remove every other owner from the list of owners, making themselves the sole owner and taking over the contract, practically reducing the ownership to an EOA ownership. It could be argued that this is an even more severe problem than the problem of EOA since in this case there are many individual accounts that pose the same risks as the single-address ownership does, but multiplied by the number of owners, even worse, any owner can appoint new owners that further deepens the problem and raises the probability of an attack.

We can see that many serious problems arise when there are multiple owners with the same, unlimited authority, and without any built-in mechanism which would constrain the influence one owner can exert over both the contract state and the other owners. To solve these problems, specific control mechanisms have to be added to this pattern to make it as safe and robust as possible while still utilizing the power and benefits of multiple ownership.

5.2 Characteristics of the proposed voting mechanism

My proposed method aims to provide a solution to those problems both developers and users have to face when using the previously described methods.

To equip the company contract with real-world-like features and to offset the aforementioned weaknesses of the commonly used patterns, I propose a voting mechanism that enables companies to develop a contract with features resembling the role of the board of directors. As discussed earlier, the responsibility and the right to make certain decisions or modifications cannot be of a single person, instead, it has to be the competence of a group. This pattern also uses a list to store the addresses of those accounts that are the owners of a contract but additionally, complements it with some features that significantly reduce the likelihood of a successful attack. Moreover, these features prevent one or even multiple malicious owners (the exact number depends on the actual implementation and settings of the given contract) from taking control over the contract.

Developers can create their functions in such way that fully fits their requirements and necessities, meaning, they can implement the voting mechanism as a part of almost any function, and they have the ability to further customize them as needed. Often, functions that are considered to be critical, or those that can potentially have a detrimental effect on the assets, operation, or reputation of the company, will have the built-in voting mechanism, which, when activated, is going to create a proposal with a timeframe within which board members can vote on that proposal.

5.2.1 Voting contract

The core of the proposed pattern is the voting contract. As developers implement the proposed method, the functions they deem critical will be creating proposals in the form of voting contracts when an owner calls a function or initiates an action that requires validation from the other owners. This contract manages the whole procedure of voting, including keeping track of the individual votes and making sure every owner can cast their votes only once. It also determines the outcome based on the number of votes, the individual choices, and the threshold that is set by the contract in addition to taking the timeframe into account. The contract creation is done by the main contract that implements the voting mechanism, and it does not require additional user interaction. The functions of the voting contract can only be called by the host contract to prevent voter fraud or any other manipulation or interference with the voting process. After the deadline has passed or a majority has been reached, the contract does not allow any further votes to be cast and the result is final.

5.2.2 Demonstration of the voting mechanism through the company contract

I will demonstrate the capabilities, functionality, and usage of the proposed pattern through one of the company contract's functions, which is a critical function and has a very drastic effect on the operation of the contract.

A good example of a critical function could be a function called *voteForSuspension*, which has very similar functionality to the *Emergency stop* pattern presented in [24] and in [30], which halts the normal operation of a contract in case of an unforeseen event, such as discovering a vulnerability in the contract that could lead to serious losses. This pattern does this by utilizing the *Access Restriction* pattern to only allow a certain address to call the function that acts as the circuit breaker.

5.2.2.1 Halting the operation of the company

As opposed to the *Emergency stop* pattern, which poses the previously discussed risks, the *voteForSuspension* function works in the following way: the default state is the normal operation when the contract is working as it is supposed to, and there are some addresses added to the list of owners (let n be the number of owners). When an event that is so serious it threatens the contract due to its severity or uncertainty happens, and one of the owners calls the *voteForSuspension* function, the operation of the contract is halted immediately. At the same time, a proposal to stop the operation indefinitely is created with a predefined deadline, and an event is emitted to let the other owners know there is an emergency and they need to vote either for or against the halting of the operation.

The first caller of the function, who initiated the halting has already automatically voted for the proposal, so they do not need to cast a vote separately. Other owners can vote by calling the *voteForSuspension* function, just as they would do if they wanted to stop the function in an emergency. The function takes one argument – a boolean value – that denotes the intention of the caller. Given the name of the function is "voteForSuspension", calling the function with a *true* value indicates that the caller wants to stop the operation or agrees with the proposal and votes for the halting, otherwise, calling with a *false* value means the caller wants to vote against it.

For the proposal to be accepted and to make the changes of the contract state to stay in effect indefinitely, the majority of the owners have to vote in favor of the halting within the given timeframe. When this happens, the proposal is accepted, and those owners who have not cast their votes so far cannot do so anymore. If the majority of the owners voted against the proposal, the normal operation is resumed immediately as the majority is reached. If a majority decision cannot be reached within the given timeframe, the normal state will be restored. This ensures that a possible attack will be limited in time, as the remaining owners will vote against the halting of the contract as soon as possible, and even in the worst-case scenario, the operation will be restored as the proposal expires due to the lack of a majority decision.

At any given point in time, only one proposal can be active on a specific subject. The lifecycle management is done by the contract, as it makes sure that there are no overlapping proposals on the same subject.

5.2.2.2 Restarting the operation of the company

If the contract is in an emergency state, and the proposal to halt the operation has passed, the contract will remain in the same state indefinitely. In order to restart the operation, one of the owners has to call the *voteForSuspension* function, but this time, with a *false* value that indicates the motive for voting against the suspension. This also automatically creates a proposal just like when an owner halts the operation. However, in the case of restarting the operation, the initial vote does not change the state of the contract immediately. The reason behind this is that the majority of the owners had previously voted in favor of stopping the normal operation. Consequently, there is a high probability that the initiation of the emergency state was because of a legitimate threat and not as a result of an attack. If the restarting happened immediately as one of the owners called the function, it would present an opportunity for an attacker to put the contract into unsafe state, thus allowing both them and any additional attacker to exploit the vulnerability that led to the emergency state.

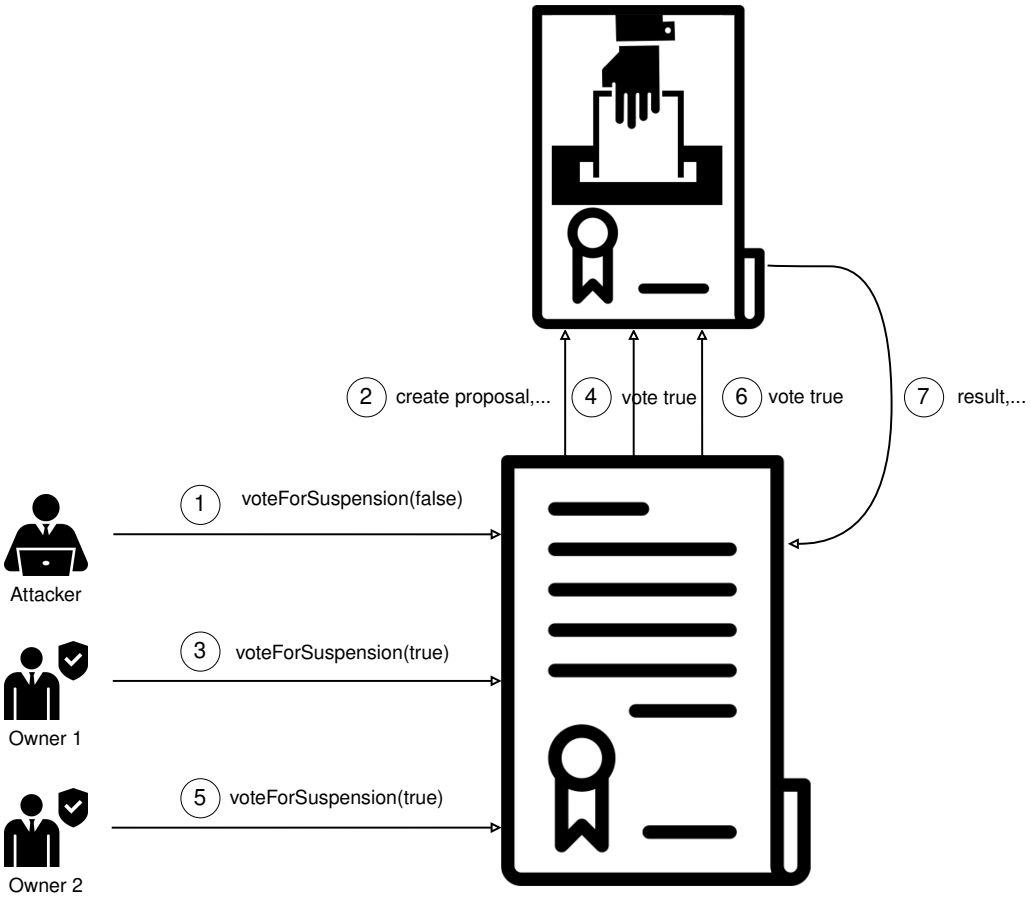

Figure 5.1: An attacker trying to restart the operation

Owners have to vote in favor of lifting the suspension with a *false* value, while if they do not agree with restarting the contract, they have to vote against the proposal by calling the function with a *true* value. If the majority of owners voted *false*, the contract will be restarted. In case the majority voted *true*, i.e. they still see it justified to keep the contract in a suspended state, or there is no majority decision in either way at the end of the given timeframe, the contract will remain suspended.

Figure 5.1 depicts a sequence when one of the owners is a malicious actor trying to restart the operation while the contract is still vulnerable due to some conditions. The attacker initiates the lifting of the suspension by calling the *voteForSuspension* function with a *false* value, which creates a proposal and automatically votes on it. As the real owners get notified about the proposal, they vote against it, thus nullifying the attack and keeping the contract in a suspended state to protect it until it can be safely restarted. Note that the other owners do not necessarily have to cast a vote against the proposal, as it will expire without a majority decision, resulting in an unchanged state. This further increases the resiliency and robustness of the contract, as it does not require immediate human interactions in case of an attack.

5.3 Comparison of the different ownership methods

Type of access control	One address		List of addresses	Voting mechanism
	EOA	Contract		
Number of malicious owners required to carry out an attack	1	1	1	$> n/2$ [1]
Severity of threat	very high	high	medium-high	low
Risk of a successful attack	high	high	medium	low
Probability of restoring normal state after attack	very low	very low	low - high	very high

Table 5.1: Comparison of capabilities of methods to deal with a potential attack

Table 5.1 provides a comparison of the four discussed ownership methods. Some of the aspects – such as severity or risk – are not necessarily quantifiable, so a value was assigned to them on a scale from "very low" to "very high" based on the impact and loss of control over the contract caused by an attack. In the case of a contract being the owner, while – in practice – it might take more than one malicious owner to carry out an attack, from the perspective of the owned contract, it is still a single point of failure and the owned contract heavily depends on it.

[1] in case of requiring a simple majority to pass the proposal, where n is the number of members of the board. The threshold can be set higher to achieve a supermajority, for example, two-thirds or even unanimous decision to further reduce the possibility of a successful attack

5.4 Comparison to Gnosis Safe

The demonstrated company contract complemented with the voting mechanism shows many similarities to the Gnosis Safe [31] Multisig wallet in terms of functionality, however, there are fundamental differences between the two approaches.

The Gnosis Safe Multisig works by utilizing the contract as an owner approach. The developers of the contract have to create a new contract which is called a *Safe*. At the time of creating the Safe, the addresses of the accounts that will serve as the owners of the main contract have to be added to the Safe, along with the threshold that is required for a transaction to be executed. After it is done, the Safe contract's address has to be set as an owner (or admin) in the main contract. From this point in time, certain functions and actions will require the owners to sign each transaction in order to execute them. The number of signers has to reach the predefined threshold, otherwise, the transactions will not be valid and cannot be executed. Note that this method does not allow the developers to differentiate between functions based on the threshold requirement. Each transaction will require the same amount of signatures to be qualified as valid and accepted.

The proposed voting mechanism allows developers to set an arbitrary threshold for each function to meet the requirements of the specific use case. This can be especially important in use cases where the contract represents the board of directors of a company or even legislative bodies where different legislation need different types of a majority to pass, for example, a simple majority, qualified majority, or even double majority. In the case of a company, different actions might not be of the same importance, e.g. halting the operation might require a simple majority, while revoking an owner's authorization would need a qualified majority (three-fifths, two-thirds, etc.). Using the voting mechanism, this can be achieved easily, while it is not as straightforward or very difficult to do with a multisig wallet.

In summary, despite the fact that the Gnosis Safe and the example use case are very similar in their functionalities, the two solutions have different goals. The Gnosis Safe aims to help manage digital assets, while the proposed method's broader goal is to support the governance of companies utilizing blockchain technologies.

5.5 Further potential applications of the voting mechanism

As the proposed method is a voting system, the demonstration contract and its use case is just a subset of all the use cases it can be applied to. To give an illustration of further use cases, let's look at a company, whose board votes on a budget or target that will be set for a specific goal, e.g. funds for an upcoming project, the production target for the next year, the amount of premium that will be paid out to shareholders, deciding what charity they will donate to, etc. These are all use cases when the choice is not a binary value, so members are not restricted to casting a yes or no vote, but they can vote with a value they see fit to the case. In this case, either the value that had the most votes will be the result that is going to be applied

to the goal (for example, deciding on the production target), or the total budget could be allocated proportionally to the result if it is a viable alternative according to the specific use case (for example, in the case of supporting a charity).

In essence, the proposed voting mechanism is a tool to support decision-making and governance in the case of multi-stakeholder applications, and it could be utilized almost any time there is a question or problem that needs to be decided on or a consensus needs to be reached on a particular matter.

Chapter 6

Validation and Verification

In this chapter, I present test results to verify that the proposed mechanism works as it is supposed to.

I implemented the proposed mechanism in Solidity, using the 0.8.4 version. The contract that has been tested is the Company contract presented in Chapter 4, complemented with the voting mechanism. The critical function is the one that suspends the operation of the contract, and it requires a simple majority to pass a proposal.

Test cases were written to test the core functionality of the contract and verify the correct behavior. I used Hardhat [32] as the development environment, with Waffle [33] as the test framework. The contracts were deployed to the local Hardhat Network which is intended for development and testing purposes.

6.1 Test cases and results

6.1.1 Access control

```
Testing the company contract
  ✓ Company contract deployed
  ✓ Check token contract address
  ✓ Check asset contract address

  Testing access control
    ✓ Trying to authorize an address from an unauthorized address (55ms)
    ✓ Trying to deauthorize an address from an unauthorized address (41ms)
    ✓ Trying to set allowance from an unauthorized address
    ✓ Trying to reauthorize an address (43ms)
    ✓ Trying to propose an owner from an unauthorized address
    ✓ Trying to withdraw an owner from an unauthorized address
    ✓ Trying to suspend operation from an unauthorized address
    ✓ Trying to change the token contract address from an unauthorized address
    ✓ Trying to change the asset contract address from an unauthorized address
    ✓ Checking if the allowance is correct when setting it (55ms)
    ✓ Checking if the allowance is correct when revoking authorization (80ms)
    ✓ Checking if a previously authorized address is unable to buy an asset after deauthorization (91ms)
    ✓ Checking if the token contract address is updated when changing it (115ms)
    ✓ Checking if the asset contract address is updated when changing it (121ms)
```

Listing 6.1: Testing basic functionality and access control

Listing 6.1 shows the results of those test cases that check whether all the necessary contracts were deployed correctly to the network, as well as the tests that verify the access control features. It covers the uses cases where the functionality of the *Access Control* pattern is essential, such as in the event of an unauthorized user, ie. not an owner, trying to invoke functions that can potentially have damaging effects, thus requiring owner privileges.

6.1.2 Non-owner functionalities

```
Asset creation
  ✓ Trying to create an asset from an unauthorized address
  Creating the asset from an authorized address
    ✓ Checking the owner of the asset
    ✓ Checking asset price
    ✓ Checking asset quantity
```

Listing 6.2: Testing the creation of assets

Test cases that can be seen in Listing 6.2 and Listing 6.3 verify that the main functionality of the contract, which is to manage asset transactions initiated by company members in the name of the company, works correctly. It also incorporates checking additional access control features to verify that only authorized company members can purchase assets and they cannot exceed their allowance in the process.

```
Asset purchase
  ✓ Trying to buy a non-existing asset (73ms)
  ✓ Trying to buy an asset from an unauthorized address
  ✓ Trying to buy an asset without a sufficient allowance (61ms)
  ✓ Trying to buy too many units of an asset (84ms)
  Buying the asset from an authorized address
    ✓ Checking the owner of the asset
    ✓ Checking if the company has paid for the asset
    ✓ Checking if the previous owner got the money
    ✓ Checking if buyer's allowance is reduced
```

Listing 6.3: Testing the functionality of purchasing assets

6.1.3 Owner management

```
Owner management
  ✓ Adding an owner (40ms)
  ✓ Removing an owner (48ms)
  ✓ Trying to remove the last owner
```

Listing 6.4: Testing the management of owners

Those test cases that aim to verify the correctness of the management of owners, specifically, adding and removing addresses from the list of owners, are shown in Listing 6.4. It also shows that the last owner cannot be removed, which would leave the contract without any owners, thus preventing a state that leads to catastrophic consequences. The tests regarding access control in owner management functions have been covered in Section 6.1.1.

6.1.4 Critical function

```
  Suspension of operation
    ✓ Checking if the Suspended event is emitted when suspending operation (58ms)
    ✓ Checking if the create function is disabled in suspended state
    ✓ Checking if the buy function is disabled in suspended state

    Suspending operation
       Waiting 5 seconds...
      ✓ Checking if the operation is resumed after the voting duration is over without enough votes
     (5043ms)
      ✓ Checking if the operation is resumed after the majority of owners voted against suspension
     (1028ms)
       Waiting 5 seconds...
      ✓ Checking if the operation is still suspended after the voting duration is over and the majority
     of owners voted for it (5029ms)

    Resuming operation
       Waiting 5 seconds...
      ✓ Checking if the operation is still suspended after the voting duration is over without enough
     votes (5050ms)
      ✓ Checking if the operation is still suspended after the majority voted against resuming (91ms)
      ✓ Checking if the operation is resumed after the voting duration is not over and the majority of
     owners voted against suspension (110ms)


  41 passing (21s)
```

Listing 6.5: Testing the critical function

Listing 6.5 shows the testing of the voting mechanism by calling the critical function, which suspends the operation of the company. The tests were carried out in a scenario where three addresses were appointed as owners, all of them being EOAs. Some test cases require a certain amount of time to pass to simulate the timeframe in which owners are allowed to cast their votes. During the tests, the timeframe for voting was set to 5 seconds.

The first series of tests verify that in a suspended state the operation is indeed suspended, thus not allowing the invoking of specific functions, namely, those that create and purchase assets in the name of the company.

The next block is to verify the correct functionality of the implemented voting mechanism in case of suspending the operation. Test cases cover the instances when the proposal expires, gets accepted, or gets rejected.

The test cases in the third block cover those situations when the contract is already in a suspended state, and the owners try to resume the operation.

```
---------------------|----------|----------|----------|----------|----------------|
File                 |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
---------------------|----------|----------|----------|----------|----------------|
 contracts/          |      100 |    81.08 |      100 |      100 |                |
  Company.sol        |      100 |    79.17 |      100 |      100 |                |
  Asset.sol          |      100 |    91.67 |      100 |      100 |                |
  EHUF.sol           |      100 |      100 |      100 |      100 |                |
  Voting.sol         |      100 |    78.57 |      100 |      100 |                |
---------------------|----------|----------|----------|----------|----------------|
All files            |      100 |    81.08 |      100 |      100 |                |
---------------------|----------|----------|----------|----------|----------------|
```

Listing 6.6: Code coverage of the tests

The code coverage for all tests can be seen in Listing 6.6. The implemented contract is a proof-of-concept to demonstrate the validity of the proposed methods, so the complete, comprehensive testing was not the goal here, hence the less than hundred percent branch coverage.

6.2 Security analysis

To discover as many vulnerabilities as possible, I ran static and dynamic security analysis tools on the implemented contracts. The tools I used for this purpose were Mythril and Slither, which are capable of detecting vulnerabilities in smart contracts, including but not limited to reentrancy vulnerabilities, unchecked tokens transfers, and functions allowing unauthorized parties to destruct the contract.

6.2.1 Mythril

Mythril is a security analysis tool that uses symbolic execution, SMT solving and taint analysis to detect security vulnerabilities in smart contracts (more specifically, in the EVM bytecode of the contract) [34]. The full list of vulnerabilities detected by Mythril can be found in its module listing [35]. By default, Mythril uses 22 as the recursion depth for the symbolic execution engine. To increase the number of explored states, therefore lowering the possibility of uncovered states and bugs remaining in the code, Mythril was run with double recursion depth compared to the default value.

```
docker run -v ~/contracts:/tmp mythril/myth analyze /tmp/Company_voting.sol --max-depth 44
The analysis was completed successfully. No issues were detected.

docker run -v ~/contracts:/tmp mythril/myth analyze /tmp/Voting.sol --max-depth 44
The analysis was completed successfully. No issues were detected.
```

Listing 6.7: Results of testing through Mythril

Listing 6.7 shows that as a result of security analysis through Mythril, there has been *zero known vulnerabilites* found in the company smart contract with 44 recursion depth.

6.2.2 Slither

Slither [36] is a static analysis framework for Solidity. It converts Solidity smart contracts into an intermediate representation, therefore it is able to preserve semantic information that would be lost in transforming Solidity to bytecode [37]. It complements the use of dynamic analysis tools, such as Mythril, it is able to find more/different vulnerabilities, and it can highlight code optimization opportunities. The list of vulnerabilities detected by Slither – along with the information about them and the detectors – can be found in [36].

Slither marked some points of the contract that could possibly be vulnerable to reentrancy attacks. This was caused by the critical function containing external calls that precede states changes. While generally, this is not a good practice due to its violation of the *Checks-Effects-Interactions* pattern, in this case, this is the contract that creates the voting contract, which also only allows the contract that created it to call its functions, thus reducing the risk of unwanted or malicious actions.

```
Compiled with solc
Number of lines: 986 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 7 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 13
Number of informational issues: 42
Number of low issues: 6
Number of medium issues: 5
Number of high issues: 0
```

Listing 6.8: Results of testing through Slither

Listing 6.8 shows the summary and the assessment of the security analysis that was conducted using Slither. The medium issues were the aforementioned, likely false-positive reentrancy vulnerabilities, and the others were mostly informational ones, a number of them coming from the testing of the related contracts. This shows that Slither found *zero serious, high-risk vulnerabilities* during testing.

6.3 Validation

The proposed approaches had been validated using thorough testing and case studies. Both the company contract and the voting mechanism met the requirements of the specifications and fulfilled their purposes, as they functioned according to their intended objectives discussed in Chapter 4 and Chapter 5, respectively.

Chapter 7

Summary

In this paper, I gave a brief introduction to blockchain technologies and smart contracts, along with the state of the art regarding their applicabilities to industrial use cases, and their provided benefits to stakeholders from different industries. I identified different challenges that these entities have to face when preparing to incorporate the usage of blockchains and smart contracts into their business operations. These challenges include the creation of robust smart contracts, and decentralized, trusted cooperation and decision-making both inside and between companies.

In Chapter 2, I presented the technical background of my work, namely, (i) blockchain technologies in general, (ii) Ethereum and Solidity, that I used for my work, and (iii) the purpose of smart contracts. I also detailed the importance of developing robust smart contracts, and presented some patterns that are commonly used by developers to increase the resiliency of their contracts.

In Chapter 3, I presented the results of my work in an industrial scenario, where I set up a private Ehtereum-based blockchain and developed a system of smart contracts that realized an asset management application that cooperated with a CBDC system on a separate blockchain through bridging between networks.

In Chapter 4, I gave a detailed explanation of the problems that may arise in industrial and corporate applications. Then, I proposed an approach that gives companies the power to manage the authorization and spending limits of the accounts individually that are under their control. I gave an illustration of how this can be incorporated into an industrial setting through the example of a manufacturing line.

Then, in Chapter 5, I detailed the problems of the existing smart contract ownership and access control methods. To provide a possible solution to these problems, I proposed a voting-based decision-making mechanism that aims to address the shortcomings of these patterns and can support industrial applications by allowing stakeholders to vote on specific proposals. Then, I assessed the characteristics and capabilities of the discussed methods.

Finally, in Chapter 6, I verified and validated my solution by demonstrating the results of testing and security analysis that was conducted to confirm its effectiveness and discover potential bugs and vulnerabilities in the contracts.

Acknowledgements

I would like to express my gratitude to my supervisors, Dr. Pál Varga and Attila Frankó for all their advice and continuous support.

The research was supported within the framework of the Cooperation Agreement between the National Bank of Hungary (MNB) and BME.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[2] V. Buterin, "Ethereum whitepaper: A next-generation smart contract and decentralized application platform," 2013.

[3] U. Bodkhe, S. Tanwar, K. Parekh, P. Khanpara, S. Tyagi, N. Kumar, and M. Alazab, "Blockchain for industry 4.0: A comprehensive review," *IEEE Access*, vol. 8, pp. 79764–79800, 2020.

[4] D. Kozma, P. Varga, and G. Soós, "Supporting digital production, product lifecycle and supply chain management in industry 4.0 by the arrowhead framework–a survey," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, pp. 126–131, IEEE, 2019.

[5] P. Varga, J. Peto, A. Franko, D. Balla, D. Haja, F. Janky, G. Soos, D. Ficzere, M. Maliosz, and L. Toka, "5g support for industrial iot applications— challenges, solutions, and research gaps," *Sensors*, vol. 20, no. 3, 2020.

[6] P. Tasca and C. Tessone, "A taxonomy of blockchain technologies: Principles of identification and classification," *Ledger*, vol. 4, 02 2019.

[7] T. Fernández-Caramés and P. Fraga-Lamas, "A review on the use of blockchain for the internet of things," *IEEE Access*, vol. 6, pp. 32979–33001, 05 2018.

[8] "Solidity programming language." `https://soliditylang.org/`. Accessed: 2021-10-23.

[9] "Vyper." `https://vyper.readthedocs.io/en/latest/index.html`. Accessed: 2021-10-24.

[10] "Aave – open source defi protocol." `https://aave.com/`. Accessed: 2021-10-25.

[11] "Augur." `https://augur.net/`. Accessed: 2021-10-25.

[12] F. Schär, "Decentralized finance: On blockchain- and smart contract-based financial markets," 2021.

[13] "Uniswap." `https://uniswap.org/`. Accessed: 2021-10-25.

[14] I. Mehar, C. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack," *Journal of Cases on Information Technology*, vol. 21, pp. 19–32, 01 2019.

[15] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020.

[16] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[17] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 2–8, 2018.

[18] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*, iiWAS2018, p. 375–380, Association for Computing Machinery, 2018.

[19] "Openzeppelin." `https://openzeppelin.com/`. Accessed: 2021-08-02.

[20] "Openzeppelin pausable." `https://docs.openzeppelin.com/contracts/4.x/api/security#Pausable`. Accessed: 2021-10-21.

[21] "Openzeppelin ownable." `https://docs.openzeppelin.com/contracts/4.x/api/access#Ownable`. Accessed: 2021-10-21.

[22] "Proxy patterns." `https://blog.openzeppelin.com/proxy-patterns/`. Accessed: 2021-08-02.

[23] M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the ethereum ecosystem," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1513–1520, 2018.

[24] "Design patterns." `https://github.com/maxwoe/solidity_patterns`. Accessed: 2021-07-28.

[25] "Upgradeability using unstructured storage." `https://blog.openzeppelin.com/upgradeability-using-unstructured-storage/`. Accessed: 2021-08-02.

[26] "Proxy upgrade pattern." `https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies`. Accessed: 2021-08-02.

[27] "Eip-20: Erc-20 token standard." `https://eips.ethereum.org/EIPS/eip-20`. Accessed: 2021-10-28.

[28] I. Kocsis, L. Gönczy, A. Klenik, P. Varga, A. Frankó, and B. Oláh, "CBDC-based smart contract ecosystems," *Technical report, BME-MNB*, 2021.

[29] T. Mrázik, K. Szabó, and B. Tóth, "Dynamic industrial workflow execution supported by service discovery," in *Conference of BME Scientific Students' Association*, 2020.

[30] "Solidity patterns." https://github.com/fravoll/solidity-patterns. Accessed: 2021-07-28.

[31] "Gnosis safe." https://gnosis-safe.io/. Accessed: 2021-10-23.

[32] "Hardhat." https://hardhat.org/. Accessed: 2021-10-25.

[33] "Waffle." https://getwaffle.io/. Accessed: 2021-10-25.

[34] "Mythril." https://github.com/ConsenSys/mythril. Accessed: 2021-10-25.

[35] "Mythril modules for vulnerability analysis." https://mythril-classic.readthedocs.io/en/master/module-list.html. Accessed: 2021-10-25.

[36] "Slither, the solidity source analyzer." https://github.com/crytic/slither. Accessed: 2021-10-25.

[37] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019.