Poznan University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Master's thesis

COMPARISON OF PROGRAMMING CAPABILITIES OF ETHEREUM AND EOS BLOCKCHAIN PLATFORMS

Krzysztof Wencel

Supervisor
dr hab. inż. Anna Kobusińska

Poznań, 2019

Contents

Chapter 1

Introduction

In 2009 an anonymous person published a Bitcoin whitepaper [Nak09] and released the first version of the Bitcoin software under the pseudonym of Satoshi Nakamoto. The paper introduced a concept of a *blockchain* – a sequence of data blocks secured by a *Proof of Work* consensus protocol and linked together using a cryptographic hash function. Bitcoin makes use of the blockchain by inputing users' transactions (understood as money transfers) into blocks and broadcasting them to other users of the software. Because of its decentralized nature, Bitcoin does not have a single point of failure. Moreover, due to the heavy use of crytography, it is very hard to tamper with transactions that were included in the blockchain. Thanks to these properties, Bitcoin became the first global decentralized currency – also referred to as a *cryptocurrency* because of its reliance on cryptography.

Since the release of Bitcoin, a surge of interest in cryptocurrencies has been observed. Not only the Bitcoin userbase was growing, but also many developers tried to improve upon the concept proposed by Satoshi Nakamoto. As a result, many alternative cryptocurrencies, called *altcoins*, started to emerge. The improvements made in various altcoins focus on many different aspects. Some of them, like *Monero* [Noe15] and *ZCash* [BCG$^+$14] implement private transactions where the sender and the recipient cannot be identified. Others, like *Decred* [Jep15] and *DASH* [DD18], target financial self-sustainability and governance, so users can democratically decide in which direction the project should be headed. *Nano* [LeM18] and *IOTA* [Pop16] are examples of cryptocurrencies which do not rely on blockchain and use directed acyclic graphs instead to achieve much higher transaction throughput. Finally, projects like *Ethereum* [W$^+$14] and *EOS* [L$^+$18] enable users to not only transfer value, but also upload code to the blockchain and give transactions possibility to interact with the aforementioned code (also referred to as *smart contract* [B$^+$14]). Considering the fact that Bitcoin allows only simple money transfers, the amount of innovations in the cryptocurrency space is really impressive.

However, the presented improvements are just a small selection of what todays cryptocurrencies have to offer. Some of them also try to address multiple problems instead of focusing on one. The Table 1.1 presents ten most popular (by market capitalization) cryptocurrencies and their primary use case.

The technological advances that were made in the cryptocurrency space over the last decade and the market data show that programmable blockchains providing support for smart contracts are the most appreciated innovation since the inception of Bitcoin. Ethereum is the second most popular cryptocurrency and EOS occupies the seventh place on the list, however, it is a much younger project. If we disregard all Bitcoin forks (which do not contribute much from the technological perspective) the picture becomes even more clear – smart contracts are one of the most desirable

Table 1.1: Top 10 cryptocurrencies by their market capitalization. Source: CoinMarketCap [coi19]

Name	Market Capitalization	Price	Purpose
Bitcoin	$184 933 252 563	$10 316.88	Simple value transfer
Ethereum	$19 429 805 512	$180.45	Smart contracts
Ripple	$11 243 727 323	$0.26	Communication medium for banks and payment providers
Bitcoin Cash	$5 480 848 949	$304.59	Simple value transfer (Bitcoin fork)
Litecoin	$4 385 578 506	$69.37	Simple value transfer (Bitcoin fork)
Tether	$4 092 713 020	$1.00	Dollar-pegged cryptocurrency used on exchanges
EOS	$3 573 276 126	$3.84	Smart contracts
Binance Coin	$3 453 752 820	$22.21	Utility currency giving benefits to Binance exchange users
Bitcoin SV	$2 342 137 356	$131.18	Simple value transfer (Bitcoin fork)
Monero	$1 308 898 738	$76.10	Private transacting

features in cryptocurrencies.

However, writing smart contracts is not an easy task. Programming languages used by the blockchain platforms, tools and libraries evolve rapidly, often involving breaking changes. For this reason, documentation is not always up-to-date and most importantly, especially from developer's perspective, the tutorials become outdated and obsolete very quickly. Moreover, to the best knowledge of the author, a comprehensive comparison of the most popular programmable blockchain platforms does not exist.

This thesis aims to solve above mentioned problem by providing an analysis and comparison of Ethereum and EOS blockchain platforms in regard to their capabilities and limitations from both user and developer perspective. For this purpose, the same smart contract (logic-wise) was designed and implemented on both platforms using the recommended and latest tools at the time of writing. The thesis also describes the difficulties that were encountered and how they were dealt with. The primary goal of this thesis is to gather the essential knowledge about the features (or lack thereof) of Ethereum and EOS, because it is not easily accessible unless someone has a hands-on experience in programming on both of these platforms. Thus, the thesis aims to make it easier for developers to choose a proper platform for their needs by explaining the differences and comparing the code written for both of them.

The thesis is structured as follows. Chapter 2 explains in detail the essential terms such as *blockchain*, *blockchain consensus algorithm*, *token* and *smart contract*. A brief description of Ethereum and EOS, as well as the smart contract logic, is presented in Chapter 3. Chapter 4 contains a comparison of both platforms and provides code examples. Final evaluation of both platforms and possible improvements to the provided smart contracts are presented in Chapter 5. Conclusions and future work is presented in Chapter 6.

Chapter 2

Theory

To be able to understand the nuances of smart contracts it is necessary to get a grasp of the underlying technology powering them.

2.1 Blockchain

Bitcoin was invented to enable global value transfers in a distributed and permissionless manner. As a result, Bitcoin does not rely on any central authority that decides which transactions are valid and who can join the network.

To accommodate these requirements Bitcoin makes use of the blockchain – a data structure that the authors of *Blockchain technology: Beyond Bitcoin.* [CPV+16] define as "a distributed database of records, or public ledger of all transactions or digital events that have been executed and shared among participating parties.". Both of these descriptions are correct, although high-level. At the low-level, blockchain is a list (also referred to as a *chain*) of blocks. Every block holds arbitrary data and contains a hash of a previous block. This design implies that any modification of an existing block will change its hash resulting in a mismatch between the new hash and the hash stored in the block's successor.

For this reason, blockchain is an append-only data structure. New blocks can be added on top of the latest block, however any attempt to modify any of the existing block can be easily detected by traversing the entire chain looking for mismatched hashes.

As for cryptocurrencies, the arbitrary data that blocks can store are transactions. Transaction is a record that usually consists of a nonce, sender, recipient, amount of cryptocurrency to send and a signature of the sender. The nonce is used to uniquely identify every transaction issued by a particular sender. Although some cryptocurrencies have a slightly different transation structure, the aforementioned fields are present in the vast majority of them.

To understand how transacting in cryptocurrencies works in detail, it is required to know how cryptocurrencies work on a lower level. Before making any transaction, every cryptocurrency user has to install a computer software provided by the cryptocurrency project they want to use. This software is usually split into two programs. The first is the *node* which is responsible for handling communication with other users' nodes. All running nodes form a cryptocurrency network, which (in case of most cryptocurrencies) is *homogeneous*, (i.e., no node has a special role in such network). The second program is a *wallet*, which the user interacts with to check their balance and make transfers. Then, the wallet forwards user's request to the node program.

2.2 Transacting

In order to start sending or receiving cryptocurrency, a user has to generate a public-private key pair (also known as *account*) using the wallet application. The public key is their account address they can receive money with. The private key is used for authorizing transactions involving the account. Following the rules of asymmetric cryptography [KMVOV96], data encrypted with a private key can be decrypted by anyone with the knowledge of the corresponding public key, thus proving that the data was approved by the sender. With this knowledge we can clarify that the *sender* and *recipient* fields in transactions are in fact public keys. Moreover, the signature is a hash of the whole transaction (excluding the signature field which is unknown at this point) encrypted by the sender's private key.

When user wants to send some cryptocurrency, they construct a transaction and broadcast it to the nodes they know about. Then, the transaction gets spread across the whole network by a gossiping mechanism.

However, just informing other nodes about the transaction is not enough to make it accepted across the network. Cryptocurrencies are distributed systems and due to the lack of a coordinator that could decide which transactions are accepted, there has to be a way to reach consensus among the nodes about the transactions that going to be included in the blockchain. Due to the properties of a blockchain, such inclusion is permanent and is tantamount to accepting the transaction.

Before explaining the actual consensus mechanism, let us discuss how do nodes come up with a subject of consensus, which is a block of transactions.

First of all, a transaction has to be valid to get accepted. Validity implies that a transaction has been properly signed and has a sequentially incremented nonce. It also requires the sender to have enough funds available to cover the transaction amount. Every node selects a fixed amount of transactions it has received by the gossip mechanism, and puts them in a temporary block in an arbitrary order. The amount of chosen transactions is limited by a block size, which can vary across different cryptocurrencies. Moreover, a chosen order of transactions must not violate validity of any of them. Ensuring validity is required because it is possible that some transaction references money that was received by another transaction and both of them were selected to be included in a temporary block. In this scenario, the receiving transaction has to be put before the transaction that spend received funds. However, until there is a clear *happens-before* relation present (like in this example), the transactions within the same block are considered to have happened at the same time. When comparing transactions included in different blocks, the transactions from the older block are considered to have happened before the transactions included in a newer block.

The temporary block becomes a node's candidate for the block to be appended on top of the blockchain. Hence, it also becomes a subject of an consensus algorithm, because one of these proposals has to be chosen and added to the blockchain.

After reaching the consensus, an author of the winning block broadcasts it across the network. Upon receiving the block, every node appends it on top of their local copy of the blockchain and update users' balances (which are stored in memory as a cache and are calculated by scanning all transactions from the local blockchain).

2.3 Consensus mechanisms

Various cryptocurrencies use different methods of reaching a consensus regarding the contents of the next block to be appended to the blockchain. Its main purpose is preventing *double spends*, which is sending more than one transaction referencing the same money. In distributed systems, where

the order of receiving transactions does not have to match the actual ordering of the transactions, a consensus mechanism is needed for every node to agree on the state of the blockchain, which determines users' balances. This section describes four of the most common consensus mechanisms used in cryptocurrencies.

2.3.1 Proof of Work (PoW)

Bitcoin Wiki [bit19b] describes *Proof of Work* as "a piece of data which is difficult (costly, time-consuming) to produce but easy for others to verify and which satisfies certain requirements". In case of Bitcoin, it requires every node that wants to participate in the block creation to perform hashing of a concatenation of the block's content and an arbitrary nonce. The requirement is to compute a hash that begins with a given number of zeros. In order to generate different hashes, the nodes can manipulate the block content, the nonce or both. If a proper hash is found, it is added to the block header as a proof of performing a time-consuming work (hence the name *proof of work*). This hash will be referenced by a future block in the form of *hash of a previous block* which is core to maintaining blockchain integrity. Then, such block is broadcasted to the network, and its author is rewarded with extra cryptocurrency for spending his computing resources. The process of finding a right hash is also referred to as *mining*, and users participating in the consensus are called *miners*.

It is worth noting, that the hashing process makes this consensus probabilistic. On average, miners with more powerful equipment will find the right hash sooner than miners with slower machines. For this reason, the probability of solving the puzzle increases with the rise of computing power. The Bitcoin Wiki [bit19a] does explain why mining has to be computationally hard: "Mining is intentionally designed to be resource-intensive and difficult so that the number of blocks found each day by miners remains steady". Maintaining a certain time interval between blocks (called *block time*) is needed to ensure that newly created blocks have enough time to propagate through the network and thus reduce the probability of some node not being aware that the block was already created by someone else in this round. Such precaution prevents unnecessary energy waste and blockchain *forking*, which is out of the scope of this thesis, but an interested reader is encouraged to read more about this issue [Vuk15].

Proof of Work is considered to be one of the most secure blockchain consensus algorithms, as it relies solely on mathematics instead of any social or game theory rules. However, it is criticized for its enormous energy consumption from the global viewpoint, and therefore its environment harmfulness. Moreover, cryptocurrencies using PoW which has low number of miners (and hence the low overall network hashrate) can easily be dominated and attacked by the powerful miners, who would possess an absolute control over the block contents. Such scenario is especially dangerous for young cryptocurrencies which has not gained traction yet, and as a result, do not have a large user base.

2.3.2 Proof of stake (PoS)

Proof of Stake is a consensus that was probably first proposed by the *bitcointalk.org* user *QuantumMechanic* [bit19c]. The author explains the idea the following way: "instead of your "vote" on the accepted transaction history being weighted by the share of computing resources you bring to the network, it's weighted by the number of bitcoins you can prove you own, using your private keys". Essentially, the Proof of Stake consensus eliminates the need for wasting electricity on brute-forcing hashes hoping to find the right one. Instead, the chances of being eligible for

producing a block are directly tied to the amount of cryptocurrency owned. The block creator gets rewarded with an extra cryptocurrency for producing a block.

Proof of Stake is often praised for its efficiency, performance and reduced transaction fees. It is however also criticized for being vulnerable to politics like vote buying, monopoly and rich becoming even richer. Fortunately, cryptocurrency projects using proof of stake and researchers are aware of these problems and try to mitigate them. One of the most notable contributions to this consensus algorithm is the Casper Protocol [BG17] proposed by Ethereum's developers which punishes the bad-behaving actors by *slashing* (taking away) their stake. Such defensive mechanism is not possible to be implemented in Proof of Work.

2.3.3 Delegated Proof of Stake (DPoS)

Delegated Proof of Stake was invented by Daniel Larimer [Lar19], who is also a main developer of EOS. The consensus works similarly to a representative democracy. Cryptocurrency users vote for *delegates* (also known as *witnesses*) who then perform consensus among themselves to agree on the contents of a next block. A user's vote is weighted by their stake and the number of witnesses is limited. The limited number of nodes actually taking part in the consensus allows them to organize themselves efficiently and arrange designated time slots for each of them to publish their block.

For this reason, Delegated Proof of Stake is much faster than proof of work and proof of stake. A hallmark of cryptocurrencies that use DPoS is a very short block time which results in transactions getting confirmed much quicker than in PoW- and PoS-based cryptocurrencies. Another advantage is virtually non-existent energy waste, as only the delegates participate in the consensus. It also has a positive social effect – it encourages users to engage with the community and follow the news regarding the project because their vote may largely influence the project growth and success.

However, DPoS share similar problems as the real world politics – the proper operation of the network requires a large number of interested and well-informed users to choose honest and diligent witnesses. A limited number of delegates could also lead to centralization of control over the network.

2.3.4 Practical Byzantine Fault Tolerance (pBFT)

Practical Byzantine Fault Tolerance is a classical consensus algorithm introduced by Barbara Liskov and Miguel Castro [CL+99]. It is rarely used in *permissionless* blockchains (open for everyone to join), as it requires exchanging messages between all participating nodes. Because of this significant message overhead, it does not scale well with thousands of active nodes that popular cryptocurrencies might have. However, it can be used in *permissioned* blockchains (joining the network requires approval) with great results. In this scenario the communication overhead is not a problem because the number of participans is much lower. The primary advantages of pBFT are energy efficiency and instant absolute transaction finality.

In the context of cyptocurrencies, the notion of finality is understood as a non-reversible and final acceptance of a transaction. It does mean that once the transaction has been included in the blockchain, it will remain accepted forever. For example, in PoW-based cryptocurrencies, it is possible to execute a *selfish mining attack* [ES18] or *51% attack*, which can render previously accepted transactions rejected. For this reason, these type of cryptocurrencies can only achieve a probabilistic finality, which level of certainty increases with every new mined block, but can never reach an absolute certainty. On the contrary, pBFT, due to the communication between every

node, can achieve an absolute finality, which results in transactions getting confirmed instantly and permanently.

2.4 Smart contracts

The term *Smart contract* has been coined by Nick Szabo [Sza97] who described a digital counterpart of a traditional law-binding contract. What differentiated it the most from a paper contract was its self-enforcement without the need of the third party and transparency. He gave an example of real-life execution of this idea:

> A canonical real-life example, which we might consider to be the primitive ancestor of smart contracts, is the humble vending machine. Within a limited amount of potential loss (the amount in the till should be less than the cost of breaching the mechanism), the machine takes in coins, and via a simple mechanism, which makes a freshman computer science problem in design with finite automata, dispense change and product according to the displayed price. The vending machine is a contract with bearer: anybody with coins can participate in an exchange with the vendor. The lockbox and other security mechanisms protect the stored coins and contents from attackers, sufficiently to allow profitable deployment of vending machines in a wide variety of areas. [Sza97]

Szabo's vision of smart contracts required a technology that would provide a safe environment for contracts execution in a trustless and objective manner. The weakness of traditional contracts lies in the human nature. Humans can lie or not obey the rules written in the contract. On the contrary, computers do not lie and follow instructions exactly as they where written. What is more, having a distributed system for storage and enforcement of the contracts with the guarantees of honest and verifiable execution, would take the idea to the next level. Researchers and programmers realized that cryptocurrencies are perfect tools for making the idea of smart contracts a reality.

2.4.1 Implementation in cryptocurrencies

In the context of cryptocurrencies, the notion of smart contract means a computer program written in a *Domain Specific Language* invented by a particular cryptocurrency developers or in an already established programming language. Then, it is compiled to the bytecode compatibile with a particular cryptocurrency and uploaded to the blockchain by sending a special transaction, which is different from the usual value transfer. The uploaded smart contract receives its own address, which has the same format as all other addresses. Essentially, the smart contract accounts share the same characteristics of the externally owned accounts with one difference. An externally owned account is controlled by a person holding its private key. Smart contract accounts are however controlled solely by the code that was uploaded to them. They are completely autonomous and the only thing defining their behavior in response to various events is the smart contract code.

The code is structured around *functions*, which can accept arguments. When a cryptocurrency user wants to interact with a particular smart contract, they send a transaction with an additional fields containing a name of the function to call and a serialized list of arguments.

When the transaction becomes confirmed, every node execute the function body, possibly altering its local blockchain as a result. However, because smart contract execution has to be deterministic, every node would alter its blockchain in the same way.

Smart contract's code can access information from only three sources. The first one is the functions' arguments – these are the way to supply a contract with an external (from a blockchain point of view) knowledge. The second one is the contract's own storage. Every cryptocurrency which supports smart contracts provisions some storage space for every uploaded contract. The storage can be used to persistently store information required by the contract's logic. The last source of information is a blockchain data, which means that any past transactions and also other contract's storage (if their code permits it) can be queried.

It is worth noting, that the contract storage can be established by applying all transactions concerning the contract. As transactions are stored in the blockchain, which is distributed among all nodes thus visible to everyone participating in the system, the contract storage cannot be assumed private. Contract developers can specify whether particular data can be queried by other contracts, however this limitation is only enforced within the contracts' execution environment. Nothing prevents someone with bad intentions to establish a current contract storage by analyzing the blockchain transactions and use this knowledge against the contract's logic. For this reason, it is crucial for developers to be aware of this fact and assume the contract storage is public. This severely limits the use cases where stored data should not be public. There are attempts to work around this limitation, however they are out of the scope of this thesis.

2.4.2 Communication with external world

In order to accommodate some use cases, smart contracts need to access data that is not stored in the blockchain. Let us consider a smart contract that executes different branches of code depending on the current dollar's exchange rate. If we allow users to input the exchange rate as a function's argument, we cannot be sure if they are honest or they are trying to fool the contract to execute the code they want to. The only solution to this problem is to have a reliable and trusted source of external data. Because of the requirement of deterministic execution, a smart contract execution environment (a virtual machine executing the bytecode) cannot access the Internet. Let us imagine a scenario when a transaction addressed to the contract gets accepted, every node starts to execute the function body, and as a result queries the same webpage for the exchange rate. The rates are changing constantly so there is a possibility that some nodes would receive slightly different values as the nodes can be processing at various speeds. What is more, if these discrepancies would result in executing different code branches for some nodes, their local blockchains could be altered in a different way, violating the assumption that every node's local copy of the blockchain is the same. That is why smart contracts can only access data from sources that are guaranteed to be deterministic. Accessing the blockchain data is deterministic, because every node has exactly the same copy of the blockchain due to the consensus mechanism. Reading the contract's storage is also deterministic as its every modification could only be done by contract's code execution which is deterministic by definition. Lastly, referencing function arguments is deterministic because they are part of the transaction that every node executing the function must have received with the same content, otherwise a transaction's sender signature would not match.

The inability to query a real-world data from a smart contract poses a huge limitation. Fortunately, there is a way to get around this problem. Since the only way to supply the contract with external data is the use of function arguments, we are forced to use them. However, we cannot trust the users that they will provide the contract with correct data. The way to solve this issue is to allow only a trusted third party to invoke the function with the needed external data. These trusted parties are called *oracles* [ora19] and act as a bridge between a secure and isolated world of smart contracts with a real world. The process of integrating the smart contract with an oracle

service is not uniform and depends on a particular cryptocurrency architecture. However, at a high level it works the following way:

- Firstly, a contract programmer splits the function that needs an external data into two functions. The first one is going to be invoked by a user that wants to interact with the contract. The second function is invoked to be called by the oracle service provider with the needed data passed as an argument.
- A user invokes a contract function, which needs an external data to execute its logic. The function is executed until it reaches a point where it needs the external data to continue.
- The function somehow signals the oracle service what type of data does it need. For instance, it could be a transaction sent to an account owned by oracle service provider. The exact implementation of this step depends on the cryptocurrency. The user-invoked contract function ends there.
- An oracle service provider constantly scans the blockchain looking for transactions requesting external data.
- An oracle service provider notices the request, fetches the data and executes the callback function of the contract providing a requested data as an argument.

This approach works but it has a few disadvantages. First of all, the contract becomes less secure as its proper execution depends on a third party that should be honest, which cannot be guaranteed. Moreover, the trusted party could stop providing the service in the future, which will render the contract non-functional. Secondly, splitting the logic into two functions has two implications – the code becomes harder to understand, and the user calling the function is not able to get its return value. The first function ends with a request for more data needed to continue so there is nothing to return yet. When the execution resumes in the callback function, the user is no longer a caller, because such a role plays the oracle service provider. For this reason, the user cannot get the return value of the callback function. However, the result can be saved to the contract storage, so the user is able to query the result afterwards.

2.4.3 Preventing denial of service

Cryptocurrencies supporting smart contracts need to take precautions to protect the newtork from badly written or malicious smart contracts. Let us consider a smart contract that has a function whose body consists of an endless loop. If nodes would start executing that function, they would block forever, stopping the whole cryptocurrency from functioning. For this reason, some cryptocurrencies only support a simple, non-Turing complete contracts without loops and recursion. Others allow writing Turing-complete code, but limit how much CPU time a function can consume until it gets forcibly terminated and its effects reverted. Ethereum and EOS present different approaches to denial of service prevention, which will be described in the next chapters.

2.4.4 Tokens

Before smart contracts became popular, every cryptocurrency needed to have its own blockchain. With the rise of popularity of Bitcoin, many people wanted to create their own currencies, but lacked the technical skills to do it themselves. Moreover, if they wanted to use Proof of Work, they would have needed a large user base from the very beginning to lower the risk of overtaking

the network by much more powerful miners. The cryptocurrency enthusiasts were looking for a better and easier way to create new currencies.

It became possible with smart contracts. Because they can contain any logic, it didn't take long until programmers started to program their own currencies that were managed by a smart contract. Currencies that do not have their own blockchain but exist on another blockchain in the form of smart contract are called *tokens*. Tokens store users' balances inside the contract's storage. Moreover, tokens' transactions are performed by sending transactions to the contract implementing the token. Therefore, they are subject to the consensus mechanism used by the parent cryptocurrency. For this reason, the above mentioned problem with young PoW-based cryptocurrencies and dominant miners does not apply to tokens. As long as the parent cryptocurrency works properly, the token works properly as well.

Both Ethereum and EOS (and even Bitcoin, to a lesser extent) host various tokens and new ones are created on a daily basis.

Chapter 3

Description of evaluated smart-contract platforms

Both Ethereum and EOS are cryptocurrencies that focus on providing an environment for smart contracts execution. However, these environments differ in many areas. This chapter briefly describes these projects and the logic of smart contracts that were written for the purpose of this thesis.

Before describing the projects, it should be emphasized, that Ethereum is a name of both the crypocurrency and the software powering it. On the other hand, EOS is a name of the cryptocurrency, but the software is called EOSIO. However, for the simplicity and the common naming scheme with Ethereum, both the EOS cryptocurrency and its software will be referred to as EOS. Should this distinction matter for any of these platforms, it will be clarified in the text.

3.1 Ethereum

Ethereum is a cryptocurrency created by Vitalik Buterin, Gavin Wood and Joseph Lubin in 2005. It uses Proof of Work for reaching consensus and has a block time of 15 seconds. It executes the contract bytecode inside the stack-based *Ethereum Virtual Machine (EVM)*, which is Turing-complete.

Ethereum officially supports few programming languages created specifically for developing on this platform. The first one is called *Serpent*, and it is considered as deprecated now. The second one is *Solidity*, and is currently recommended and most commonly used language. It has a syntax similar to JavaScript, and is fairly high-level. However, it also has an allows to inline EVM assembly for low-level optimizations. The last supported language is still under development and is called *Vyper*. It has a Python-like syntax and is not Turing-complete, as it was purposefully designed to constrain programmers so they could not use more advanced features that Solidity offers, like recursion, infinite-length loops, class inheritance, function and operator overloading and inlining assembly. The imposed limitations were intended to reduce potential bugs and vulnerabilities and increase code readability. However, it is clear that Vyper will not replace Solidity, as it does not (and will not) support all capabilities of the former and was designed to be used for writing simpler contracts in a safer, more constrained environment. More advanced contracts would still need to be programmed using Solidity.

Ethereum was build around the "code is law" dogma. It does mean that in case of a found and exploited vulnerability in the contract, it is not possible to replace its code with a patched version. It is also impossible to reverse attacker's transactions and return the stolen funds to their

rightful owners. This strictly follows the principles of a cryptocurrency as envisioned by Satoshi Nakamoto, who created Bitcoin in a way that the funds can only be controlled by a private key of an account they belong to.

3.2 EOS

EOS is a cryptocurrency created under the lead of Daniel Larimer – an industry veteran who also developed highly successful blockchain platforms: *BitShares* [SL17] and *Steem* [LSZ+16]. EOS uses Delegated Proof of Stake (also invented by Larimer) as its consensus protocol. The users vote for witnesses (called *block producers* in EOS) who are responsible for confirming transactions. Moreover, they can make vital changes to the system, including freezing accounts and reverting previously confirmed transactions. This approach is a completely opposite of Ethereum's (and most cryptocurrencies') *code is law* paradigm. EOS gives block producers very powerful abilities, which could be used for doing justice by reverting effects of various attacks. However, with great power comes great responsibility and it cannot be ruled out that the witnesses will be overusing their power.

EOS also provides a *constitution* which users and block producers oblige to respect. This document defines, among other things, the circumstances under which block producers can freeze accounts and reverse transactions. Such limitations, however, are purely theoretical, and from the technical point of view it only requires two thirds of the witnesses to agree on such action for it to take effect. The constitution can also be changed with the acceptance of two thirds of the block producers.

The concept of mixing politics with technological enforcement is definitely an interesting one. The users, however, are not defenseless against the bad-behaving block producers. If some of them act against the community, they can be voted out, and thus they lose rewards from block production. Therefore, they are economically incentivised to play by the rules. There are 21 witnesses who are actively producing blocks and much more backup ones. Every producer have a designated time slot, when they should produce a block. If they miss their time slot, they are automatically swapped for one of the backup producers.

In EOS, smart contracts are written in C++, compiled to WebAssembly (WASM) and then executed in the EOS Virtual Machine. The choice of a well-known and established programming language allows for easier formal verification and potential use of external C++ libraries.

3.3 The smart contract

In order to gain a deeper understanding of smart contract development on Etherem and EOS, a sample contract has been written on both of these platforms. It enables users to choose a number from 0 to 9 and bet any amount of cryptocurrency on their choice. The cryptocurrency associated with every bet gets added to the prize pool accumulated in the contract. Then, any user can advance the contract to the next phase, in which the pseudo-random winning number in a range of 0 to 9 is generated. Subsequently, the contract selects the users who voted for a winning number and divides the prize pool among them proportionally to the their bet amounts. The winners can then withdraw their part of the prize. The calculation of the final prizes relies on integer division, which "cuts off" the fractional part of a result. For this reason, there could be a small amount of the prize in the pool, which has not been assigned to any of the winners. These leftovers are sent to the contract's owner (an account who uploaded the contract to the blockchain). Then,

the contract returns to the first phase, in which it waits for the users' bets and for some user to initiate a resolution procedure.

Both contracts support betting with the platform's native currency or with tokens. However, the contracts were designed in such way that only one token could be accepted as a payment method by a single contract instance. One instance could support many tokens at once, however that would severely complicate the code. It was been assumed that more complex code would not help in pointing out the differences between Ethereum and EOS, and, in fact, could even be harder to follow. Hence, in order to support more token in this simplified design, a separate instance of the smart contract would have to be deployed for each of them.

For the good user experience, no user should be tasked with sending the appropriate transactions manually. It is cumbersome and it could lead to the loss of funds due to the human error. For this reason, a web frontend has been created for both contracts, which expose a graphical user interface for the easier interaction with the contracts.

It should be noted, that the Ethereum version of the smart contract and its frontend was originally co-created with Tomi Wójtowicz and Marcin Hrycaj. However, for the purpose of this thesis, the contract was refined and upgraded to Solidity 5.x language revision. Additionally, more tests were added, various bugs were fixed and the contract's frontend was mostly rewritten from the ground up. The EOS contract and its frontend were build from scratch solely by the thesis author.

The full source code of all smart contracts written for the purpose of this thesis can be found in the appendix.

3.4 Toolchain

In order to build the smart contract on both platforms, the following tools were used:

3.4.1 Ethereum

- Compiler: solc 0.5.8
- Runtime: Ganache CLI 6.4.5 (ganache-core 2.5.7)
- EVM version: *byzantium*
- Framework: Truffle 5.0.27
- Frontend integration: web3.js 2.0.0-alpha
- Wallet: Metamask 7.1.1 (browser extension)

3.4.2 EOS

- Compiler: eosio.cdt 1.6.2
- Runtime: eosio 1.8.1
- Framework: zeus 1.8.2057
- Frontend integration: eosjs 20.0.0
- Wallet: Scatter Desktop 11.0.1

Chapter 4

Comparison of evaluated smart-contract platforms

The criteria used to compare Ethereum and EOS platforms were divided into two main categories: *functional* and *technical*. Functional criteria focus on areas that impact the usability of the platform and highlight the fundamental differences resulting from a distinct visions and choices made by the platform developers. This category focuses more on the high-level capabilities of the platform and puts an emphasis on the end user's point of view. Accordingly, technical criteria focus on the low-level details the programmers could struggle with and explains how both platforms achieve the similar capabilities despite their different approaches to the same problems.

4.1 Functional criteria

4.1.1 Accounts and permissions

In most cryptocurrencies that support smart contracts, there are two types of accounts, which were described in previous chapters – an externally owned account and a contract account.

In Ethereum, an account has a one-to-one relationship with a private-public key pair or a smart contract code. In case of an externally owned account, its address is derived from the last 20 bytes of the public key of a key pair controlling the account. Therefore, the public key determines the account's address, which is used for receiving funds.

In EOS, an account is a much more advanced concept. The official developer guide explains it the following way:

> "An account is a human-readable name that is stored on the blockchain. It can be owned through authorization by an individual or group of individuals depending on permissions configuration. An account is required to transfer or push any valid transaction to the blockchain." [eos19a]

Thus, it is a layer of abstraction over the plain public-private key pair. In EOS, the transactions happen between accounts, not key pairs. The account name is an Ethereum account's address equivalent with one important difference – it can be chosen by the user and is not determined by the keys that control the account.

Moreover, an EOS account does non necessarily have a one-to-one relationship with a key pair. One private key could control multiple accounts, and also one account could be set up in a way, that certain transactions need to be signed by multiple private keys (*multisig*). Such a system allows for a very flexible access control. It is very useful for accounts that need to be

managed by more than one person. The *permission* system is responsible for specifying what kind of authorization (signature(s)) should different contract's functions (called *actions* in EOS) require for successful execution. An interested reader can read more about the permission system in the official developer guide [eos19a].

4.1.2 Tokens handling

Ethereum makes a distinction between its native currency – *Ether* (abbreviated with ETH) and the tokens, which are additional currencies implemented as smart contracts. This distinction forces programmers to write different code for handling both types of payment. Tokens can be programmed in various ways. However, the community quickly realized that having a uniform interface to interact with the tokens is beneficial. It enables tools like block explorers and cryptocurrency wallets to automatically support every token that conforms to the standard.

There are two very popular token standards: ERC-20 and ERC-721. The main difference between them is *fungibility*. ERC-20 tokens are fungible, which means that their individual units are interchangeable and indistinguishable. For this reason, fungible tokens are the most commonly used for implementing currencies. On the other hand, ERC-721 tokens are non-fungible, which make them very useful for digitizing ownership of real-world assets, such as real estate, artwork and other goods. It is also used for gaming, where non-fungible tokens represent the items the player possesses. The player can then trade these tokens on the blockchain, thus changing the ownership of an the asset they represent.

Moreover, a new standard ERC-1155 was recently introduced. It differs from the previously mentioned standards by two means. Firstly, it allows creating multiple tokens that are managed by a single smart contract. ERC-20 and ERC-721 standards require deploying a separate contracts for every created token, which unnecessarily bloats the blockchain and involves cost for every deployed contract. ERC-1155, however, involves paying only once for submitting one contract, which could govern various tokens. Secondly, the new standard allows creating both fungible and non-fungible tokens, thus it combines functionalities of both ERC-20 and ERC-721. Because both fungible and non-fungible tokens could be managed by the same contract, it is possible to easily exchange various tokens, even those of different types. It is very useful for implementing games on the blockchain – the players could buy items (non-fungible tokens) for gold (fungible token) or do barter trades with other players by exchanging some items directly without any gold involved.

EOS treats its native currency (*EOS*) the same way as any other token. In fact, the native currency is also implemented in the form of smart contract. Therefore, from now on, the EOS's native currency will be referred to as the *EOS token*.

The standard for fungible tokens is an eosio.token contract, which is used to implement the EOS token. If programmers want to create their own tokens, they can just use the source code of this contract (which is available) and deploy it under a different name. Then, they can create as many fungible tokens as they wish, which will all be managed by the same deployed contract. Thus, the eosio.token contract is similar in that regard to the Ethereum's ERC-1155 standard. However, it does not support creating non-fungible tokens. Because EOS native currency is a token itself, it makes no difference for programmers to accept the EOS token or any other fungible token as a form of payment in their contract, as from the programming perspective, they are treated exactly the same way. This is the reason, why there are two Ethereum smart contracts included with this thesis and only one EOS contract. The requirement for writing a different code for accepting an ERC-20-compliant token and Ether resulted in splitting the Ethereum contract into two separate

contracts for better readability (listings A.4 and A.5). However, as most of the code was common in both of them, the shared code was moved to the base abstract contract (listing A.3), which the two above mentioned contracts derive from and implement the functions related to cryptocurrency transfers differently.

EOS does not currently have one *de facto* standard for non-fungible tokens. There are some competing implementations, however there is no clear and widely recommended way of creating these kind of tokens.

4.1.3 Denial of service prevention

As mentioned in the previous chapters, every cryptocurrency has to implement a denial of service mechanism. Otherwise, malicious users could just spam the network with large numbers of transactions, or create malicious contracts with infinite loops, thus preventing legitimate users from using the network.

In Ethereum, execution of every transation requires its sender to pay a fee. The fee essentially prevents the above mentioned attacks, because the sender would eventually run out of money. Before explaining how the fee is calculated, it is essential to understand the notion of *gas* in Ethereum.

Gas is a unit that measures the computational effort that it will take to execute operations required by a transaction. Both simple money transfers and interactions with smart contracts are done via transactions. For this reason, both require paying the gas needed for their execution. Every EVM instruction consumes a precise amount of gas, which is defined in the Ethereum Yellow Paper [W+14]. Intuitively, executing complex smart contract logic should cost more gas than a simple money transfer, and that logic is also reflected in the paper.

Gas is a unit with a dynamic relation to ETH, which is a native Ethereum currency. When constructing a transaction, a user has to choose how much ETH is one unit of gas worth to them. The chosen amount is called a *gas price*. Hence, the fee that they have to pay is expressed in ETH, and is a result of multiplication of the chosen amount of gas units and the gas price.

The pricing of gas is completely up to the user to decide. However, transactions with low-priced gas are going to wait a lot longer to be confirmed, because miners earn the fees of transactions they are including in the blockchain. For this reason, miners would prioritize transactions with higher-priced gas. Because of that, the gas in Ethereum acts as both denial of service prevention mechanism and a transaction prioritization technique.

While constructing a transaction, the user has to specify how many units of gas are they willing to spend on its execution. If the user chooses a gas amount that is larger than the transaction requirements, the unused gas will be returned to them. However, if the user chooses too little amount of gas, the transaction will fail, its effects will be reverted, and the whole fee will be transferred to the miner executing the transaction. Essentially, the chosen amount of gas units is an upper bound that the user accepts to lose for executing the transaction.

Therefore, it is in the user's interest to properly estimate how much gas the transaction is going to consume. It is easy for simple money transfers whose cost is fixed regardless of the amount of transferred cryptocurrency. However, estimating exact gas requirements for smart contract interactions can be tricky. Even invoking the same contract function multiple times could require different amounts of gas as the execution could depend on the contract state which can change over time. Fortunately, most wallets and frontend libraries help estimating the costs by automatically proposing the required gas amount, which the user can accept or modify as they wish.

Some operations, like deleting values from contract storage, have a negative gas cost. It does mean, that if a user invokes a smart contract function, whose execution would delete some value the contract's storage, the user will be refunded the gas that was used for storing this value in the first place.

These negative costs have to be considered when setting the gas units, as the tools that estimate the gas usage of a particular contract's invocation are just summing the costs of all operations that would have been executed. They would also subtract the negative cost, which will in turn lower the overall estimate. However, for the transaction to be properly executed, the user would still need to specify an amount of gas as if there were no refunds, because they are processed after the transaction has successfully executed. It will not execute correctly if the user does not attach sufficient gas units to reach the end of transaction where the refunds take place.

EOS proposes totally different solution for preventing denial of service. Its authors decided that all transactions should be free in order to encourage users to interact with the system as it would not cost them anything. For this reason, transacting in EOS cannot involve fees of any kind. Thus, the developers came up with an idea, that every user should be able to use the network resources proportionally to the amount of cryptocurrency they own. The logic is, the more the user is invested in the system, the more they are entitled to consume its resources. The exact implementation is the following.

EOS resources are represented in the form of three categories: CPU, NET and RAM. CPU resources are needed to execute transactions. The more complex it is, the more CPU is consumed. NET resources are consumed by sending transactions regardless of the complexity of their execution. Finally, RAM resources are used for contract's storage.

To be able to interact with the EOS blockchain, the user needs to *stake* some of his EOS tokens to CPU and NET resources. These resources are most essential, because the simplest transactions would require sending some bytes over the network (NET) and processing the transaction (CPU). However, they might not involve saving anything into the storage, thus RAM might not be required.

Staking in the context of EOS is understood as making some of the cryptocurrency unavailable for transacting in exchange for being entitled to use some of the system resources. Users can stake their EOS tokens for CPU and NET resources. The more cryptocurrency they stake for the particular resource, the more are they guaranteed to be able to use when the system is under full load. However, if the network is not operating at its fullest capacity, users can use more resources than they have staked for. For example, let us assume that there is 1000 EOS staked in NET by all users combined. Let us also assume, that the user has 100 EOS on their account. If the user stakes 10 EOS for the NET, they are guaranteed to be able to use 1% of the total system networking capacity, in case the system is under full load. However, if the block producers' network infrastructure is not heavily utilized, the user can consume more than the guaranteed 1% of their share. Moreover, the user have now only 90 EOS available for transacting. If the user decides that they do not send transactions often enough to justify freezing 10 EOS, they can unstake 9 EOS, giving them only 0,1% guaranteed bandwidth but allowing them to spend 99 EOS. However, unstaking is not instantaneous – the user has to three days until the unstaked EOS become available for transacting. The same logic applies to CPU.

Because of the nature of CPU and NET, both of these resources are 'regenerating' over time. For example, if the network infrastructure is under full load and the user sends too many transactions in a short period of time, they will exceed their guaranteed network bandwidth. However, if they wait some time, they will be able to send transactions again, as their allowed usage of network

increases by not sending transactions. However, these resources will only regenerate up to the starting point resulting from their stake.

RAM is a different kind of resource. RAM cannot be staked, it has to be bought instead. For this reason, there is a RAM market, where EOS users can trade this resource. Its units are bytes, thus a user can buy a precise number of bytes they could use for writing to various contracts' storage. This resource also does not regenerate automatically. The only way to reclaim RAM is to delete previously saved data from the contract's storage.

What also makes RAM different from CPU and NET, is the ability to be payed by the contract itself. The programmer of the contract is able to specify, whether a particular write to the storage should be covered from the contract's RAM supply or the user's RAM supply. This feature enables programmers to create a truly free-to-use applications on the blockchain. If all writes to the storage are billed by the contract, the only thing the user has to do is to stake some amount of CPU and NET, which is required for sending any transaction in EOS anyway. However, they do not lose any resource that is scarce, like RAM.

That approach is totally different from Ethereum's and enables programmers to create contracts that prioritize the user experience. It is similar to how todays web applications work. If users had to pay for to use Facebook or Twitter, these services would not be as popular as they are. However, users still need computers (CPU) and an Internet connection (NET) to access these services. Therefore, the resource model in EOS closely resembles the model that most applications use – their creators cover the cost of running the service and storing users' information. EOS creators decided that this approach would be more appealing to the users than a need to irreversibly lose some of the cryptocurrency in the form of fees for every sent transaction.

4.1.4 Ricardian contracts

Ricardian contracts were originally proposed by Ian Grigg and defined as:

> "A Ricardian Contract can be defined as a single document that is a) a contract offered by an issuer to holders, b) for a valuable right held by holders, and managed by the issuer, c) easily readable by people (like a contract on paper), d) readable by programs (parsable like a database), e) digitally signed, f) carries the keys and server information, and g) allied with a unique and secure identifier. and described in his paper" [Gri04]

Implementations of this idea can vary, however in most cases a Ricardian contract is a document written in a legal prose, containing some machine-readable tags to assist in the parsing process].

Because Ricardian contracts can be interpreted by both humans and machines, they could be used to compliment smart contracts by providing a legally binding description of the code. Ricardian contracts have been implemented in EOS, however Ethereum does not support them.

In EOS, a smart contract could optionally have a matching Ricardian contract. Ricardian contract specifies the legally binding behavior associated with each action of the smart contract. Thus, it explains what a user should expect by executing the action, and what was the programmer's intent. Not every EOS user is a programmer, therefore such a document enables the user to know what is going to happen if they execute a smart contract's action. If the Ricardian contract is available for a given smart contract, the wallet could use the included machine-readable tags to find the fragment of a legal prose, that concerns the smart contract's action that the user is about to execute. Then, it would display it in a prompt and ask the user to confirm that they acknowledge and agree to the terms displayed on screen before signing and broadcasting the transaction.

Signing a transaction addressed to the smart contract legally binds the sender to the terms of Ricardian contract associated with the invoked smart contract's action.

Therefore, the terms written in Ricardian contract are more important, as they are legally binding, and the smart contract is just a mechanism of their enforcement. However, it is possible that due to the programmer's error, the code does not precisely reflect what has been written in the Ricardian contract. If the exploitation of the bug lead to stealing the users' funds, it is possible for block producers to compare the code and its expected behavior defined by the Ricardian contract. If they confirm that the bug exist, they could return the funds to their rightful owners and patch the code or ask an original programmer to do so.

4.1.5 Transaction expiration time

The EOS block time is just half a second, however it is still possible that due to the high volumes of circulating transactions, some of them would be queued and not immediately included in the next block. Some of these transactions might be time-critical and their inclusion to the blockchain after the deadline would make no sense or could even lead to errors. To solve this problem, EOS allows specifying an expiration time when constructing a transaction. If a specified time is exceeded, the transaction cannot be included in the blockchain. Unfortunately, Ethereum does not support expiration times for transactions.

4.1.6 Upgradability of smart contracts

Following the *code is law* paradigm, Ethereum does not allow for contract code upgrades. However, through the clever design of the contract, it is possible to work around this limitation. The mentioned design has to use one of the *proxy patterns*. One of the simplest proxy pattern use the following structure.

The contract is split into two – the first one contains the logic, and the second one is used for the contract's storage and keeping the address of a logic contract. If the user wants to interact with the contract, they send a transaction to the storage contract, which redirects the call to the logic contract. If the logic needs to be updated, the new contract with the updated logic is uploaded to the blockchain. It receives a different address, therefore it is required to modify the address kept in the storage contract. After this change, the storage contract would redirect any subsequent calls to the new logic contract, while the storage would remain the same.

However, this approach would not work if the storage structure also needs updating. In this case a more advanced proxy patterns would have to be used.

On the contrary, EOS natively supports updating the contracts' logic. There is no need for uploading updated version as a new contract. The old code gets completely replaced, and the new transactions addressed to the contract are automatically executing the latest code.

Regardless of the update method, it is possible that the new version of the code is malicious. The users are not informed that the code update took place, therefore upgradable smart contracts are both a blessing (for programmers who can fix bugs) and a curse (for the users who cannot be sure what code are they executing). However, Ricardian contracts in EOS should reflect the code changes, so the users should know what they are executing, assuming the programmer is honest. If they are not, the block producers could intervene as described previously. Thus, in EOS, there are some defense mechanisms against the malicious code updates. In Ethereum, however, there are none, so the users should be very careful when dealing with contracts that utilize the proxy pattern.

4.1.7 Blockchain logic upgrades

The blockchain logic define the rules and conditions under which the blockchain operates. It could include the various parameters, such as block time, block size and consensus-related settings.

In Ethereum (and most cryptocurrencies), a *hardfork* is required to apply vital changes to the blockchain logic. A hardfork is a network split that occur due to the non-backwards compatibile changes to the blockchain logic. After releasing the new version of the blockchain software which includes the incompatible changes, the nodes that upgraded the software would start processing the blocks according to the new rules. However, the nodes that did not update the software would still be working according to the old rules. For this reason, the network becomes split. When most of the nodes upgrade the software, the network stabilizes again. However, during the split period the users are encouraged to not use the blockchain as their transactions might not be not become confirmed. Therefore, cryptocurrencies which use hardforks as an upgrade mechanism cause troubles for their users during the transition period.

In EOS, the vast majority of the blockchain logic is enforced by a smart contract called *system contract*. Because smart contracts are upgradable on EOS, the system contract can also by updated without a hard fork. The EOS's blog post explains which parts of the system are governed by the smart contract:

> "The only thing implemented in the core blockchain code is the permission system which includes the ability to create accounts, deploy contracts, and enforce resource quotas. Everything that makes the blockchain Delegated Proof of Stake including the token, voting, staking, and resource allocation is now defined by the Web Assembly based system contract." [eos19b]

For this reason, blockchain logic upgrades on EOS are smooth and should not cause much inconvenience for the end users.

4.1.8 Performance

Ethereum, being limited by the Proof of Work consensus, the relatively small block size (about 20 - 30 kB) and a block time of about 17 seconds, achieves a throughput of about 15 processed transactions per second.

However, EOS uses a very fast DPoS consensus algorithm, does not have a limited block size and has block time equal to half a second. For this reason, the practical throughput EOS currently can achieve oscillates around 4000 processed transactions per second.

It has to be noted though, that Ethereum plans to change its consensus protocol to Proof of Stake, which along with some architectural changes like *sharding* could substantially increase its transaction throughput in the future.

4.2 Technical criteria

4.2.1 Floating point mathematics

In Ethereum, floating point mathematics is not supported. It was done on purpose, to avoid the problems arising from the limited precision of floating point representation and the potentially non-deterministic calculations when run on different processor architectures. Therefore, in Ethereum, every numeric operation has to be performed using 256-bit integers.

EOS developers took a different approach. To avoid the problems of diverse hardware implementations of the floating-point mathematics, they have implemented it in software inside the EOS VM. The Virtual Machine, that runs smart contracts, provide deterministic execution of floating point operations regardless of the underlying hardware. The problem with the limited precision of floating point representation still exists, however it also exist in non-blockchain application. Therefore, the EOS developers decided, that the having deterministic floating point operations is sufficient for providing a reliable smart contract execution environment.

4.2.2 Access modifiers

EOS enables programmers to use access modifiers that are present in most object-oriented programming languages - *public*, *protected* and *private*. These modifiers work like in any language, with one caveat. The *public* modifier is not sufficient to make the function callable externally (i.e. by users' transactions or another contracts). In order to achieve this, the function has to be marked by an `eosio::action` annotation or by an `ACTION` macro, which resolves to the former.

Ethereum exposes a different set of access modifiers. However, before explaining them, let us introduce the two kinds of function calls that Ethereum supports – *internal* and *external* calls.

Internal calls do not create an actual EVM call (which consume more gas). Instead, they are implemented as jumps in the code. The jumps require that the call arguments are present in the EVM memory. For this reason, that kind of call is useful for helper functions, which are not exposed to the end user, and which operate on data that is already present in the memory.

On the other hand, external calls are implemented using an EVM's CALL opcode, which costs more gas than a simple jump. In these kind of calls the arguments can be directly read from the *calldata*, which is a read-only byte-addressable space where the arguments of transaction or a call are stored [eth19].

Any user-initiated call (a transaction) is an external call, due to its nature. Moreover, any call from one contract to another is an external call. However, a smart contract's call to one of its own functions could be done by both internally (`f()` notation) and externally (`this.f()` notation). However, the external call overhead makes it a less attractive option in for this case. The variety of choices in regard to function calling is the reason why Ethereum provides not three, but four access modifiers.

The *private* modifier works as expected, and the *internal* modifier is the *protected* equivalent. Two last modifiers are more interesting. The first one is *public*, which allows calls from the users, from the same contract and from other contracts. Therefore, it is the most permissive modifier. Moreover, public functions can be called both internally and externally. In order to support both kinds of calls, the compiler generates a bytecode that expects the arguments to be present in memory, just like in case of the internal call. For this reason, the arguments of external calls would have to be copied from the calldata to the memory, which is an expensive operation. Therefore, the public modifier should only be applied to functions that require compatibility with both kind of calls.

The last modifier, *external*, like the name suggests, allows only for the external calls. For this reason, the arguments of external functions would always be read from the calldata, and never from the memory. For this reason, if it is known that the function will only be called externally, it is recommended to mark it as *external* instead of *public*. This will avoid copying the arguments to memory like it is done in public function, thus saving the gas.

Access modifiers do not only apply to functions. Both Solidity and C++ are object-oriented languages, therefore the access modifiers can also be used on contract's fields. In Ethereum, the field modifiers are used to define if other contracts can access particular elements of the contract's storage. In EOS, they don't make any difference as the contract's storage is not exposed in the form of contract fields. The differences in handling the storage by Ethereum and EOS will be discussed later in this chapter.

However, it is essential to emphasize the point that was made in the theoretical chapter. The data stored in the contract is visible to anyone participating in the network regardless of the access modifiers. Programmers can use the modifiers only to limit the ability of other contracts to read from their contract's storage. The data is still accessible to anyone having access to the blockchain.

What is also very important, Ethereum uses *public* as its default access modifier. Therefore, if a programmer does not specify any modifier to the function, it would be callable by any Ethereum user. It could have devastating consequences if a developer assumes that a default modifier is *private*, which is true for most programming languages. Similarly, not specifying any modifier to the contract's field will make it readable by any contract.

4.2.3 Data storage

Ethereum exposes its storage in a very intuitive way – in a form of contract's fields. It directly resembles the way the programmers are used to work with object-oriented languages. Every class could have fields with different access modifiers to provide encapsulation. It works exactly the same in Ethereum.

Ethereum enables programmers to use few data structures which are available in Solidity. The list of provided structures is short and consists of fixed-size arrays, dynamically-sized arrays and maps (called *mappings*). As for types, the most important ones are booleans, integers, fixed point numbers, strings, enums and structs.

The limited number of data structures is not the only problem of data storage in Ethereum. The first problem is the lack of *null* value, which could be used to indicate the lack of data. Instead of the *null* value, Ethereum returns a type's default value when accessing uninitialized or missing data. This problem is easier to explain on an example.

```
1   enum Phase { NO_BET, BET_MADE }
2   struct Bet {
3      uint number;
4      uint value;
5   }
6
7   uint constant internal minAllowedNumber = 0;
8   uint constant internal maxAllowedNumber = 9;
9   uint internal withdrawalBalance = 0;
10
11  Phase internal currentPhase;
12  mapping(address => Bet) internal bets;
13  // Stores the addresses that put a bet on a given number
14  mapping(uint => address[]) internal numberMappedBetters;
```

Listing 4.1: Fragment of Ethereum smart contract, which presents some of the user-defined types and contract's fields

Let us examine the line 12 of a listing 4.1. It is a declaration of a map, where keys are of type *address*, and values are of type *Bet*, which is a struct defined at line 2. When a user bets on a number, it is saved into this map where the address of the better is saved as a key, and the bet details are saved as a value in a form of a struct. If a contract tries to get a bet details of an

address that is not present in the map, it would return not a *null*, but a *Bet* struct with *number* equal to 0 and *value* equal to 0, as it is the default value of any integer type in Solidity. Therefore, it is not possible to tell if the address was not present in the map or it was, but its value was a *Bet* struct with all fields equal to 0. Fortunately, this smart contract does not allow to bet with no cryptocurrency at all, so the *value* field equal to 0 could be used as an evidence that the address did not exist in the map. However, in the general sense, this behavior is troublesome if the range of accepted values contains the default value as well. A possible workaround could be to add an additional boolean field to the struct. This field would need to be set to *true* on the struct creation to indicate that it is a real record. Then, if the struct received as a value from the map have this boolean field set as *true*, it means that the key was present in the map. Otherwise the key was not present in the map, as the default value of boolean is *false* in Solidity.

Another problem with maps is that they are not iterable. It is not possible to get the list of keys from the map and then read every value stored in the map. A possible workaround would be to create a separate dynamically-sized array which will store the keys of the map. Arrays are iterable, so it would be possible to get every key saved in the array and query the map using these keys. However, arrays are also not without their own problems.

Arrays do not behave intuitively. For instance, removing an element in the middle of the array does not move every succeeding elements to the left and the array size is decremented. Instead, there is a gap where the element was removed and the array size remains the same. In order to fix the this issue and make the array continuous again, the programmer could manually shift the array, element by element, which would consume a enormous amount of gas. Fortunately, if the contract's logic does not require the array to preserve the order of insertion, there is a trick that would make the array continuous with very little gas usage. After deleting an element from the middle of the array, the last element from the array is moved to the location of the gap, thus creating a new gap at the end of the array. Then, the array's length is decremented, which removes the last element of the array, and as a result, gets rid of the gap. This approach was used in the smart contract written for this thesis (lines 106-117 of listing A.3 found in the appendix).

The last problem of the storage in Ethereum is its waste of space. EVM is a 256-bit Virtual Machine, hence it operates on a 256-bit words. For this reason, every type in Ethereum consumes 256 bits by default, even the boolean type. The Solidity compiler tries to pack adjacent contract's fields if they would fit (for instance, two fields of type uint128) in one 256-bit storage space. Therefore, the order of fields declaration is important from the optimization perspective. However, if the programmer did not properly order their fields declarations, every non-dynamic type (i.e. every type besides mapping and dynamicly-sized array) would consume 256 bits of storage. Such waste does not only bloat the blockchain, but also incur significantly higher gas costs.

On the contrary, EOS does not use contract's fields to expose contract's storage to programmers. In EOS, data can be stored using the `eosio::multi_index` data structure, which is based on the `boost::multi_index` implementation [boo19]. According to Boost developers, a Multi-Index is a structure which enables the construction of containers maintaining one or more indices with different sorting and access semantics [boo19].

```
1   TABLE bet_table {
2      name better;
3      bet_number number;
4      amount value;
5
6      [[nodiscard]] uint64_t primary_key() const {
7        return better.value;
```

```
8       }
9
10      [[nodiscard]] uint64_t by_number() const {
11        return number;
12      }
13    };
14    using bet_index = eosio::multi_index<"bets"_n, bet_table, indexed_by<
15      "bynumber"_n, const_mem_fun<bet_table,uint64_t,&bet_table::by_number>
16    >>;
```

Listing 4.2: Fragment of EOS smart contract, which presents a table definition

Instead of contract's fields, EOS uses tables as an interface to the contract's storage. An example table definition is presented in listing 4.2. The EOS tables are similar to the tables found in relational databases. A table can have multiple fields (columns), must have a primary key and optinally up to the 16 secondary indices defined.

In Ethereum contract, various maps and arrays had to be used in order to fulfill the requirements with the limited number of available data structures. However, in EOS, the table defined in listing 4.2 contains everything related to the users' bets. Every record consists of three attributes – a better's account, a chosen number and an amount of cryptocurrency that has been bet on the choice. Primary key (line 6) is defined on the better names (`better.value` is an `uint64_t` representation of an account name, as primary key currently only supports this type). A secondary index is used on the *number* column (line 10), which makes it possible to get all users, who have bet on a particular number. Lastly, lines 14-16 define an alias for the multi_index object created using the table for more concise code.

Operations on the multi_index are much more verbose compared to the simple contract's field access that is present in Ethereum. However, the multi_index allows for a much greater flexibility and a database-like access with user-defined indices.

```
1   struct state_table {
2      phase phase;
3      /** An owner of the contract **/
4      name owner;
5      /** A token that the contract is configured to work with **/
6      extended_symbol token;
7      /** How much of the contract balance is reserved for withdrawals **/
8      amount withdrawal_balance;
9   };
10  using state_singleton = singleton<"state"_n, state_table>;
11  state_singleton state_db;
```

Listing 4.3: Fragment of EOS smart contract, which presents a singleton definition

The smart contract included with this thesis does also use `eosio::singleton`, which is a wrapper for a table with just one record. The singleton is usually used for storing the contract's state and configuration, therefore one record with multiple columns is sufficient. Listing 4.3 shows the definition of the singleton in the contract.

When it comes to storage efficiency, EOS is much better than Ethereum. Its VM is 32-bit (with efficiently implemented 64-bit operations), therefore it does not waste so much space like 256-bit EVM. Moreover, the types are automatically packed into the optimal serialized format, so a `short` would occupy 16 bits of storage, instead of 32 bits, like it would do on Ethereum if the programmer did not optimize the layout of fields.

Moreover, EOS does not also suffer from problems like the non-existence of *null*. The C++ language supports *null* values, and so does EOS.

4.2.4 Contract initialization

In Ethereum, a contract's constructor is invoked only once – during its deployment on the blockchain. For this reason, the constructor is used to set up the initial state of the contract.

```
1    IERC20 public token;
2
3    constructor(address _tokenAddress) BaseGambling() public {
4        token = IERC20(_tokenAddress);
5    }
```

Listing 4.4: Constructor of Ethereum smart contract, which accepts an ERC-20 token as payment

In case of the contract which supports ERC-20 token payments, the constructor is used to remember the address of the token which will be accepted as a currency for betting. Listing 4.4 presents the code of this constructor. The token address passed as an argument is used to construct a token variable, which will be saved in the contract's storage. Every future operations performed by the contract that are related to the token, such as transferring, receiving or checking balances, would be referencing this variable.

As mentioned in the previous chapter, this design implies that one contract instance is able to work only with a particular ERC-20 token, which has to be specified at the time of deployment. It should also be noted, that the constructor shown in listing 4.4 also invokes the parameterless constructor of the *BaseGambling* class, which contains the logic shared by the both Ether and ERC-20 variants of the Ethereum smart contract.

```
1  gambling::gambling(const name& receiver, const name& code,
2                     const datastream<const char*>& ds)
3     : contract(receiver, code, ds), state_db(_self, _self.value) { }
4
5  void gambling::init(const name& owner,
6                      const extended_symbol& token_to_accept) {
7     require_auth(get_self());
8     check(not state_db.exists(), "Contract already initialized");
9     auto initial = state_table {
10       .phase = phase::NO_BET,
11       .owner = owner,
12       .token = token_to_accept,
13       .withdrawal_balance = 0
14    };
15    state_db.set(initial, get_self());
16 }
```

Listing 4.5: Constructor and init function of EOS smart contract

In EOS, a constructor is invoked every time a contract's function is called. Therefore, it is not suitable for setting up the initial state of the contract. However, it can be used to execute the logic that would otherwise have to be included at the beginning of every function . For instance, in line 3 of listing 4.5, a state singleton is initialized. The state singleton is used in every contract's function, therefore including its initialization in the constructor prevents code duplication.

In order to perform a contract's primary initialization, a special action can be created. In case of the contract included with the thesis, this function is called *init* (line 5), and it sets up the initial contract state by writing to the state singleton. One of its fields is the token that the contract will accept as payment. In order to support multiple tokens, like in Ethereum variant, multiple instances of the contract would have to be deployed and initialized appropriately. It is

also essential to ensure that the state has not been already initialized and revert the transaction if necessary (line 8).

4.2.5 Referencing function invoker

Smart contract often need to know which user (or contract) invoked a particular contract's function. Some of these functions could make major changes to the contract configuration, like the *init* function presented in listing 4.5). There could also be functions whose execution does not need to be limited to certain privileged accounts, but the information about the sender is needed to access appropriate data. These are the two most common use cases that require the knowledge about the contract's invoker. In fact, the mentioned security-critical function *init* does indeed check if the invoker is the account that has deployed the contract(or an account that was authorized by it (line 7). However, because the *init* function is not present in the Ethereum variants, let us use an another example to compare the differences in the way a function's invoker can be identified in both platforms.

```
1    function withdraw() external {
2       require(toWithdraw[msg.sender] > 0);
3       uint reward = toWithdraw[msg.sender];
4       toWithdraw[msg.sender] = 0;
5       withdrawalBalance = withdrawalBalance.sub(reward);
6       transfer(msg.sender, reward);
7    }
```

Listing 4.6: Withdraw function of Ethereum smart contract

Listing 4.6 presents the Etherem's version of the *withdraw* function. It is supposed to be called by a user who won a bet in order to withdraw their rewards. In order to check if the invoker of the function has any rewards available to withdraw, it is needed to check the value in the *toWithdraw* map under the key equal to the invoker's address. Therefore, line 2 does exactly that. The *msg.sender* is a global variable which is always set to the address of an account, that invoked the currently-executing function. If the *require* check is not satisfied, the transaction will be reverted.

```
1  void gambling::withdraw(const name& to) {
2     require_auth(to);
3     auto withdrawals = withdrawal_index{get_self(), to.value};
4     auto reward =
5           withdrawals.require_find(to.value, "No balance available to withdraw");
6     auto state = state_db.get();
7     state.withdrawal_balance -= reward->value;
8     state_db.set(state, same_payer);
9     withdrawals.erase(reward);
10    transfer(to, reward->value.to_asset(state.token.get_symbol()),
11             state.token.get_contract());
12 }
```

Listing 4.7: Withdraw function of EOS smart contract

Listing 4.7 shows the EOS's version of the *withdraw* function (or more precisely – action, as it is callable from the users and other contracts). The first thing that is different from the Ethereum's version is the function signature. It expects one argument of type *name* while the function of Ethereum contract is parameterless. The *name* tyle is a built-in type of EOS and is used to represent account's name. This parameter is required, as EOS does not provide any way of checking which account has invoked the action. Therefore, this information has to be provided by

the caller – they need to pass their own account name as an function argument. However, if this information is provided by the user, and not the execution environment itself, there is a possibility of spoofing another account's identity. For instance, let us assume that two people, Alice and Bob, have put their bets on different numbers. One of them initiates the resolution procedure, which results in Alice winning the prize pool and Bob not winning anything. Theoretically, only Alice should be able to withdraw her rewards. However, nothing prevents Bob from executing the *withdraw* action and passing Alice's account name as an argument. As a result, the function would behave like it was Alice who requested a withdrawal of her rewards. In case of this particular function it would not be very harmful, as the funds would be sent to Alice's account. However, if the function had accepted an additional parameter specifying the account that the rewards should be transferred to, it would have been a much more dangerous.

Fortunately, EOS allows to check if the account executing the action has an authority to do so. This check is performed in line 2 of listing 4.7. The `require_auth(to)` checks Authority can be obtained in two ways. Firstly, an account has an authority of itself. Therefore, `require_auth` check will be successful if its argument is a name of an account that currently executes the action. Secondly, an account can use the permission system to grant another contract an authority of itself. In this case, the `require_auth` check will also be successful if the account that currently executes the action has been granted an authorization of an account whose name is specified as the check argument.

With this explanation, let us go back to the example with Bob pretending to be Alice when executing the *withdraw* function. He would not be able to impersonate Alice, because he would need to sign his transaction with Alice's private key (which he has no access to) or he should be granted an authority of Alice (which has to explicitly granted by Alice). Therefore, the main difference between Ethereum's and EOS's approach to determining the function's invoker is that Ethereum provides this information in a form of a variable which is provided by the execution environment, thus it cannot be faked. On the other hand, EOS does not provide such information and require users or contract that invoke an action to provide their identity as a function argument. Then, the authenticity of this claim is verified using a built-in `require_auth` function.

4.2.6 Token handling

Smart contracts often accept the native cryptocurrency or tokens as a payment method. They also often perform various mathematical operation on the amount of received cryptocurrency or token. For example, in the case of a gambling contracts written specifically for this thesis, the bet function accepts the native cryptocurrency or a token as a payment. Let us first examine the Ethereum version of the contract.

As mentioned previously, because Ethereum treats tokens differently to its native currency, Ether, two smart contracts were created. One of them allows paying with Ether, and the other one supports payments with any ERC-20 compliant token. Let us begin with an Ether-compatibile contract.

```
1  function bet(uint _number) external payable {
2     require(msg.value > 0 && _number >= minAllowedNumber &&
3             _number <= maxAllowedNumber);
4     updateBet(_number, msg.value);
5  }
```

Listing 4.8: Bet function of Ethereum smart contract, which accepts Ether as payment

Listing 4.8 starts with a function signature, which expects one argument – a number that the user wants to bet on. However, the function does not expect a second argument specifying how much Ether the user wants to bet on their choice. It is done this way, because Ether transfers are not processed on the smart contract level. If the transaction addressed to the smart contract has some Ether attached, the Ether gets added to the contract's balance first, then the contract function is called, and the amount of received Ether is available as a global variable *msg.value*. In order to invoke a contract's function via a transaction that has some Ether attached, the function has to be marked as *payable*. For this reason, the *bet* function presented in the listing 4.8 has been declared using this keyword.

Lines 2 and 3 check if some Ether was sent along with the transaction and the number passed as an argument is within the allowed boundaries. Otherwise, the transaction will be reverted. Line 4 is an invocation of *updateBet* internal function, which contains a logic shared by both Ether and ERC-20 variant of the *bet* function. It saves the information about the bet made by the user to the contract's storage.

```
1  function bet(uint _number, uint _value) external {
2      require(_value > 0 && _number >= minAllowedNumber &&
3              _number <= maxAllowedNumber);
4      uint beforeTransferBalance = getBalance();
5      require(token.transferFrom(msg.sender, address(this), _value));
6      uint realValue = getBalance() - beforeTransferBalance;
7      updateBet(_number, realValue);
8  }
```

Listing 4.9: Bet function of Ethereum smart contract, which accepts an ERC-20 token as payment

Listing 4.9 presents the *bet* function of the ERC-20-enabled contract. It does expect two arguments. The first one is the same as in the contract which supports Ether payments. The second argument is the amount of an ERC-20 token that the user wants to bet on their choice. This design is required, because the token transfers are, as opposed to Ether transfers, processed on the smart contract level. For this reason, the user has to specify an amount of tokens to bet and the actual transfer of these tokens happens at line 5. The *require* function makes sure that the transfer was successful (the *transferFrom* function returned *true*), otherwise the transaction will be reverted. Then, in line 7, the *updateBet* function is called like in the Ether-accepting version.

However, one could ask why does the *updateBet* function not use the passed *_value* argument, and use a computed *realValue* instead. The computation involves checking the actual balance of the contract (line 4) and subtracting it from the balance after the token transfer is performed (line 6). Therefore, the result of the computation is a real amount of the token that has been received by the contract. In the simplest ERC-20 implementations these calculations would not be necessary, as the real amount received would probably be the same as expressed in the *bet* function's *_value* argument. However, the ERC-20 standard does only define a token interface, and not its implementation. For this reason, it is possible that some tokens have their own fee system in addition to the standard gas cost that has to be payed for every transfer. For instance, a programmer could have implemented their token's *transferFrom* function in such way that the 5% of the transferred amount would be sent to the programmer, and the remaining 95% would be sent to the recipient. Because the exact implementation of the token is unknown to the contract, it is safer to not make any assumptions and check how many units of the token were actually added to the contract's balance as a consequence of executing the *transferFrom* function.

Additionally, it is important to notice that the function is not *payable*, as it does not accept Ether.

As mentioned before, EOS implements its own currency in a form of smart contract. Therefore, its native currency (*EOS*) is a token itself. For this reason, unlike in case of Ethereum, there has been only one EOS smart contract written, as it supports any token that complies to the eosio.token interface (which also includes the native EOS token).

```
1  void gambling::bet(const name& from, const name& to,
2                     const asset& quantity, const std::string& memo) {
3     if (to != get_self() or from == "eosio"_n or
4         from == "eosio.ram"_n or from == "eosio.stake"_n) {
5        return;
6     }
7     require_auth(from);
8     auto state = state_db.get();
9     extended_asset value = {quantity, get_first_receiver()};
10    extended_symbol expected_token = state.token;
11    check(value.get_extended_symbol() == expected_token,
12          "Tried to bet with a not supported token");
13    check(memo.length() > 0, "Fill the memo with a number you want to bet on");
14
15    auto passed_number = std::stoi(memo);
16    check(passed_number >= min_allowed_number and
17          passed_number <= max_allowed_number,
18          "Passed number exceeds the allowed boundaries");
19    auto number = static_cast<bet_number>(passed_number);
```

Listing 4.10: Fragment of bet function of EOS smart contract

Listing 4.10 shows a fragment of the *bet* function of the EOS contract. The whole function is much longer, however it also includes the code responsible for persisting the information about the user's bet. In Ethereum variants, this code was not presented, as it was a part of *updateBet* helper function. Therefore, for simplicity and consistency, this code is also not present in listing 4.10. An interested reader is encouraged to study the whole source code, which can be found in the appendix.

The bet function has a totally different signature compared to Ethereum variants of the contract. This particular signature and lines 3-6 arise from the fact, that *bet* is not an action but a notification handler. Notifications will be discussed in the later chapters, therefore let us ignore these lines for now.

The signature, despite being different from the previous examples, contain all of the needed arguments. The *memo* argument is used to pass a chosen number, and the *quantity* argument is used to inform the contract about the amount of tokens the user wants to bet on their choice. Additionally, a *from* field is required to identify the sender of the transaction.

In line 8, the contract reads the information from the state singleton. This information is used to check if the token that the user wanted to pay with is the the one, that the contract has been configured to support (line 11). Lines 13-19 are responsible for parsing the number from the memo (which is a *string*) and checking if it is within the allowed boundaries.

It is worth noting, that the *quantity* argument is not a plain unsigned integer like it was on Ethereum. EOS uses an *asset* type for representing units of tokens. The asset type has few advantages over the unsigned integer for used in Ethereum.

First of all, it can represent the tokens which have decimal places. Because Ethereum represents token units as integers, developers have to store the number of decimal places in a different variable, and always have to remember to convert the integer according to the number of decimal places before presenting it to the end user. It is a lot of hassle for developers, and EOS does much better job at handling of tokens with decimal places.

Secondly, it also provides arithmetic operations that can be performed of units of the same token. These operations have built-in safeguards against under- and overflows, which are a common source of vulnerabilities in contracts.

Lastly, it stores information not only about the amount of the token, but also about its name (e.g. *EOS*) and number of decimal places. This is both an advantage and a disadvantage. It makes code cleaner, because every information regarding the token is available in the *asset* type. However, if the contract has to persist a lot of assets in its storage and all of them concern the same token, the information about the name and decimal places is redundant, as the only thing that will be different in all these stored assets will be amounts. This problem has been encountered when writing the gambling contracts for this thesis. The contract could be storing potentially unlimited number of user bets (stored as assets), which will all be of the same name and amount of decimal places, as the contract supports only one token. Because RAM is a resource that does not regenerate automatically over time like CPU and NET, it was essential not to waste the RAM on storing the same data over and over again. However, the *amount* type had implemented the safe arithmetics, which was desirable. Therefore, instead of completely resigning from storing an *asset* types, a custom *amount* type was written, which implements all arithmetic operations and delegates their implementation to the *asset* type. However, it does not store any information about the token name and its decimal places. This information is stored separately in the *state* singleton, so the full information could be reconstructed if needed. However, this approach does not waste RAM by saving duplicated data.

It is important to notice, that there is no *transferFrom* equivalent found in this listing. Both Ethereum and EOS tokens are processed on the smart contract level, therefore some operation which moves funds from the invoking user's account to the contract account should be executed. However, it is not present in this listing. The reason for its absence is that the *bet* is a notification handler, therefore it will be executed after the transfer has been completed.

4.2.7 Invoking other contracts

In Ethereum, contracts can invoke other contracts via external calls, which were already described in the *Access modifiers* section. Let us recall that the notation for doing an external call is `contract.function(arguments)`, for example `token.transfer(from, to, value)`, where *token* is a variable of type `IERC20` instantiated by an address on the contract is deployed (listing 4.4). Therefore, a public interface of a contract is needed in a from of Solidity source code in order to interact with such contract programatically from another smart contract. It also also worth noting, that all Ethereum calls are *synchronous*, which means that the invoking contract stops its execution until the called contract's function finishes.

In EOS, contracts can invoke other contracts' functions using one of three mechanisms: *inline actions*, *deferred actions* (which will be deprecated soon) and *notifications*. They are all *asynchronous*, which means that they do not block currently running action.

Before describing each of the methods, it is important to emphasize that these three methods are only working for executing actions from the smart contract. It does not matter if an action to run is located in the same contract or in a different one. What is important is that actions (functions exposed to users and another contracts) need be be called using one of these methods. However, if a function is not marked as action (e.g. a helper function not exposed to the world) it can be called directly with a usual `f()` notation and not by the three mentioned mechanisms. Direct calls are synchronous and apply only to the non-action functions withing the same contract.

Inline actions share some similarities with a direct method calls, because they are processed (almost) immediately. While direct calls are synchronous and executed immediately when invoked, inline actions are asynchronous, and they are scheduled to be processed just after the current action finishes its execution. Moreover, it is guaranteed that an inline action would be processed within the same transaction, therefore if the inline action fails for some reason, the whole transaction fails and is reverted. The contract does need to have an *eosio.code* permission added to its account in order to use inline actions.

Deferred actions are also asynchronous but they are not executed within the same transaction as its caller. Therefore, any errors that happens inside the deferred action does not influence the caller in any way. Moreover, they are not even guaranteed to run. They are scheduled to run at later time, at the block producer's discretion. Therefore, they should only be used for non-essential tasks. The common use case for deferred actions is splitting the complex transactions into few smaller ones. EOS requires that execution time of every transaction does not exceed 30ms. To get around this limitation it is possible to split one complex action into two smaller ones, where at the end of the first action a deferred action targeting the second action is spawned. Because the deferred action does not run in the same transaction as the calling action, it does have its own 30ms limit of execution. However, not every action can be split this way, because of the lack of error propagation and guarantees to run. Deferred actions can also be canceled by the contract that requested them (if they haven't been executed yet, of course). It is going to be a deprecated feature starting from EOS software v2.0.0-rc1 release.

Notifications are a mechanism allowing contracts to inform other contracts that a certain thing has happened. Other contracts can listen for particular types of notification and trigger their own actions in response to a captured notification. The illustrate how this system works, let us examine the most common notification use case – informing the sender and a recipient about a successful token transfer.

```
1  void token::transfer(const name& from, const name& to,
2                       const asset& quantity, const string& memo) {
3
4     check(from != to, "cannot transfer to self");
5     require_auth(from);
6     check(is_account(to), "to account does not exist");
7     auto sym = quantity.symbol.code();
8     stats statstable(get_self(), sym.raw());
9     const auto& st = statstable.get(sym.raw());
10
11    require_recipient(from);
12    require_recipient(to);
13
14    check(quantity.is_valid(), "invalid quantity");
15    check(quantity.amount > 0, "must transfer positive quantity");
16    check(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
17    check(memo.size() <= 256, "memo has more than 256 bytes");
18
19    auto payer = has_auth(to) ? to : from;
20    sub_balance(from, quantity);
21    add_balance(to, quantity, payer);
22 }
```

Listing 4.11: Transfer function of eosio.token smart contract

Listing 4.11 shows an implementation of an *transfer* action of the eosio.token standard which most tokens just use without any modification. Lines 11 and 12 are responsible for notifying both

the sender and the recipient that a successful transfer took place. These notifications will be sent after the action successfully finishes its execution.

By default, these notification would not have any effect as the contract has to explicitly listen for certain notification to be able to react to them.

To be able to respond to a notification, a contract needs to have a function that accepts the same arguments as the action that sends the notification. Moreover, it has to be marked with a *eosio::on_notify* annotation that specifies which received notifications should result in execution of the annotated function.

```
1   [[eosio::on_notify("*::transfer")]]
2   void bet(const name& from, const name& to, const asset& quantity,
3            const std::string& memo);
```

Listing 4.12: Declaration of bet function of EOS smart contract

The annotation presented in line 1 of the listing 4.12 could be translated to "invoke this function with the same arguments as the notifying function if the name the notifying function is *transfer* and argument types of both functions match". In general, the annotation also allows to specify which contract should be the origin of a notification. However, the contract was supposed to work with any token that is compatible with the *eosio.token* interface. Therefore, an asterisk has been put instead of specifying a name of the contract.

The idea is, that if a user wants to bet on a number, they should transfer some amount of tokens to the contract and input their number of choice as a memo. Then, if the transfer is successful, a *bet* function will be called as a result of a received notification. This design makes the contract simpler and safer to use, because the user does not need to grant any permissions to the contract. An alternative approach would be similar to how Ethereum handles tokens. A user would need to invoke a *bet* action, which would have to create an inline action to transfer the tokens on behalf of the user. It would not be possible unless the user had granted appropriate permissions to the contract.

Let us go back to the listing 4.10. With the knowledge of how notifications work in EOS, the reason why the signature of a *bet* function is so different from the Ethereum variants is obvious. However, lines 3-6 were previously left without an explanation. These lines are safeguards against the malicious contracts, which could notify our contracts that a transfer happened without actually sending any tokens. The `to != get_self()` check ensures that the transfer has to be addressed to the contract. The remaining check are used to ignore notifications of transfers resulting from changing the staking settings and other transfers originating from the system contracts.

Another very important check is performed at line 9 and 11. In line 9, an *extended_asset* is constructed, which contains all information of an asset (amount, name, number of decimals) and additionally also stores the account that the token contract is deployed to. The account of the token is determined by the *get_first_receiver* function, which returns an account which was the original recipient of the transaction, which was the token account. The user had to first send the transaction to the token account in order invoke the *transfer* method, which in turn invoked the *bet* of a gambling contract by a notification. Therefore, in this chain of actions, the first receiver of the user's transaction was the token account. Then, the determined token is compared with the token saved during the contracts initialization. If its symbol (name and number of decimals) and account matches, the execution of the function can proceed.

The increased number of security precautions that a programmer have to take when developing on EOS is a result of more flexible design of this platform. With more ways to interact with the contracts, the more attack vectors are possible.

4.2.8 Standard library

Ethereum's Solidity language does not include any standard library. Even the simplest operation on strings have to be implemented manually. Fortunately, a company called OpenZepplin manages an impressive amount of example contracts and utility functions that can be used freely under the MIT license [ope19]. Their code is audited and well-tested, what is crucial for Ethereum contracts, which cannot be patched without using the proxy pattern.

On the other hand, EOS has ported the vast majority of C++ Standard Template Library (STL) to the EOS Contract Development Toolkit (CDT). It does mean that the contract programmers can use most data structures and algorithms present in the STL, which are also very throughly tested and performant. Of course, the STL data structures could not be used for persistent storage, as EOS only supports its tables and the Multi-Index for this purpose. However, these structures could be used temporarily to assist with the computations and the results could still be stored in the Multi-Index.

4.2.9 Read-only calls

In Ethereum's Solidity, it is possible to apply *state modifiers* to the function declarations. Functions marked as *view* cannot modify the state of the blockchain. Therefore, they cannot make any writes to the storage or send transactions. The advantage of marking a function as view is the ability to execute it without any gas cost, as there is no modification to the state, hence the inclusion of such transaction to the blockchain is pointless. Functions marked as *pure* cannot both read and write to the contract's storage. Such function can only access its own arguments and nothing else. They are also totally free to execute.

In EOS, a *view* function equivalent is a *const* function (a C++ feature). EOS also supports context-free actions, which just like the *pure* functions are limited to accessing their own arguments and not the blockchain storage.

4.2.10 Contract ABI handling

In order to send transaction to the contract from the frontend, a contract's Application Binary Interface (ABI) has to be put somewhere inside the source of the frontend.

In Ethereum, it is not possible to invoke a contract's function without the knowledge of its API or an ABI.

On the other hand, EOS is able to dynamically fetch the contract's ABI from the nodes which form the network. Therefore, there is no need to hardcode the ABI inside the frontend code.

4.2.11 Getting information from smart contract

In order to present valuable information to the users, the frontend applications would need to be able to query the smart contracts. Frontend applications that the users will use to interact with smart contracts would need to query some data from the contracts.

Ethereum programmers can query the data from the contract's storage using the getter functions. For every contract's field that should be accessible to the frontend there should be a *view* function which returns this field. Alternatively, a field can be made *public*, which will generate the getter function with the same name as the field automatically.

On the contrary, EOS does not support returning values from actions. However, the tables can be queried with advanced filtering options, like specifying an index to use and setting the lower and the upper bound of the key values, as well as setting a maximum number of rows to return.

4.2.12 Contract inheritance

Ethereum supports contract inheritance, which was used in the gambling contracts written for the purpose of this thesis. The common logic was places inside an abstract contract and the two concrete contract variants (that supports Ether and ERC-20 token payments respectively) derive from the abstract contract.

EOS also theoretically supports contract inheritance, however in case of the gambling contract it did not generate the contract ABI correctly. It is possible, that with a hand-written ABI file it would work properly.

4.2.13 Error handling

Ethereum does not support reverting transactions with a customized message that could suggest what is wrong. Developers can only force revert the currently executing transaction with a generic error message.

EOS, however, allows for customized error messages, which can be used for debugging or instructing a user what he should do to get rid of the error.

4.2.14 Debugging

Truffle, a framework used for developing smart contracts on Ethereum platform has a built-in command-line debugger. In most cases it works, but it cannot be described as reliable. Ethereum does not support any other debugging methods, like printing arbitrary messages to the console or defining custom error messages.

EOS has much less mature frameworks because it is a newer project than Ethereum. However, Zeus, which was used for developing the gambling EOS contract, is a very promising framework, despite being in the early development phase. Zeus does not provide any debugger, however EOS itself allows printing arbitrary messages to the console during contract's execution and allows for customized error messages, which are very helpful for debugging. Zeus, a very promising

4.2.15 Wallet integration

In order to accomodate real-life use cases, smart contracts need to be accessible to everyone, not just tech-savvy people. Therefore, the develoment of tools which bridge the world of HTML and JavaScript with the hermetic blockchain environment is as important as the smart contracts development tools.

Ethereum uses *web3.js* frontend library which can work various wallet such as a browser-based Ethereum wallet *Metamask* and the hardware wallets. On the other hand, EOS uses eosjs frontend library, which also integrates with wallets, such as *Scatter*. One of the latest innovations regarding the wallet integration in EOS is an *Universal Authenticator Library*, which greatly simplify the process of adding support for multiple wallet by providing an unified API to a developer and hiding the complexity and differences in API's of different wallet providers. However, the frontend created for this thesis uses only Metamask for Ethereum and Scatter for EOS.

4.2.16 Events

Ethereum allows contract's functions to emit *events* which the frontend can listen for. Events can also carry values so the frontend could update their state based on the values contained in the events.

EOS does not support events, therefore currently the only way for a frontend to be up-to-date with the state of the contract is to constantly poll for the new blocks and analyse them. Alternatively, in order to reduce the stress on the frontend, the backend could be polling the blockchain and the frontend could subscribe to the events pushed by the backend.

4.2.17 Smart contracts testing

The most popular Ethereum framework – *Truffle* – allows writing contract tests in both Solidity and JavaScript. Moreover, EOS-based smart contracts can be tested using C++. The *zeus* framework extends these capabilities by enabling programmers to write their tests also in JavaScript. Therefore, contracts developed for any of these platforms can be written in their native language or JavaScript.

Writing tests using JavaScript is more common, because these test have to use the same JavaScript libraries which are used by the frontend to interact with the contracts. Therefore, the tests interact with the contracts just like it is done by the browser in the real environment.

Before the test suite is run, the framework runs a software that simulates the real blockchain and processes the transactions sent from the tests. After sending the transactions, various assertions can be used to confirm that the state of the local blockchain has changed as expected.

The software used for creating a temporary local blockchain is different for every platform. Truffle framework uses *Ganache*, which has some features that are very convenient for testing purposes. The first one concerns the transaction confirmation time. In the real Ethereum network, the Proof of Work consensus has to be performed, and transactions need some time to propagate through the network. For this reason, the real Ethereum network has a block time of 17 seconds and achieves a throughput of about 15 transactions per second. If the testing blockchain would also emulate this performance figures, it would severely increase the time needed for the tests to complete, which could consists of hundreds or thousands of transactions depending on the complexity of the test suite.

Fortunately, Ganache confirms transactions immediately as it receives them. Moreover, to support testing of smart contracts, which rely on the fact that the blocks are produced with an perceptible delay, Ganache exposes an *evm_increaseTime* RPC call, which can be used to programistically advance the time of the test blockchain.

Another great feature of Ganache is the ability to snapshot the blockchain at any given time and restore it later. This feature can be used to save the state of the blockchain before running the test suite and restore it after every finished test. This approach enables programmers to use the same user accounts (which are generated by the framework) and contract accounts in every test. If the snapshotting feature was not available, the state changes introduced by one tests could influence the result of the another test. Therefore, restoring the snapshot after every test ensures a consistent starting point for every test in the suite.

A zeus framework used for development on EOS uses *nodeos* as its test blockchain software. However, nodeos is not a blockchain software designed with testing in mind. In fact, it is the software that powers the real EOS network – it is used by every block producer. For this reason, nodeos does not have any features that make testing easier. It produces blocks with the same speed as the real network. Therefore, new blocks are created every half a second. It is not a long delay, however it is sufficient to make the test significantly slower than the tests which use Ganache.

Moreover, nodeos does not support blockchain snapshotting. Therefore, to make the tests not dependent on each other, a new set of user and contract accounts has to be created for every executed test, which further increases their total execution time.

Additionally, nodeos rejects transactions that have already been included in the block, which causes problems for some tests. Let us imagine a smart contract which has an action that should not be invoked twice by the same account. In order to verify this behavior we create a test, which consists of two identical invocations of the mentioned action and the two assertions. The first assertion ensures that the first invocation was successful, and the second one ensures that the second invocation was not successful and the transaction got reverted. After running this test several times it turns out that sometimes the test passes and sometimes it fails.

The result of the test depends on whether these two invocations are received by nodeos during the half-second slot in which it has to produce a block. If both invocations are received in the same block production time slot, nodeos will reject the second transaction, because it treats it as a duplicate of the first transaction. Rejecting a transaction is not the same as processing and reverting it, which is expected by the test. Nodeos treats the second transaction as a duplicate, because transactions in EOS do not have a *nonce* field, which is present in Ethereum, and could help in differentiating the transaction with otherwise the same data. If the second transaction is included in the next block after the first transaction, it will be referencing a different block than the first transaction. Therefore, the second transaction will not be treated as a duplicate, and this is the scenario when the test passes.

To work around this issue, one could write a custom wrapper for sending transactions, which would be able to detect that a transaction was rejected due to being a duplicate and resend it until it finally gets accepted. The wrapper which was written for the purpose of this thesis is presented in listing 4.13. The `txFunc` argument should a be lambda function which sends a transaction using the *eosjs* library.

```
async function resendWhileDuplicated(txFunc) {
    do {
        try {
            return await txFunc();
        } catch (e) {
            if (e.search('tx_duplicate') === -1) {
                throw e;
            }
            console.debug('Duplicate transactions in block found - resending...');
        }
    } while (true);
}
```

Listing 4.13: Transfer function of eosio.token smart contract

Chapter 5

Final evaluation of the platforms and possible improvements

5.1 Final evaluation of the platforms

Both Ethereum and EOS enable decentralized and trustless enforcement of the logic encoded inside the smart contract. However, they differ in capabilities that they offer to both programmers and end users.

Ethereum is an older cryptocurrency, which is based on the traditional and well-tested approaches. For example, it uses Proof of Work, which is slow and energy-inefficient but also the most battle-tested consensus mechanism that powers Bitcoin since its inception and has been already working properly for 10 years. It is also based on a concept of one-to-one relationship between a public-private key pair and a cryptocurrency account, which is also true for most cryptocurrencies. It does require users to pay a fee for every transaction, just like almost any other cryptocurrency.

Moreover, Ethereum seems not to make the full use of its smart contract capabilities. For instance, its own native currency and the rules governing the blockchain are implemented in the blockchain software and not in the form of smart contracts, which would enable seamless upgrades to the most fundamental cryptocurrency logic.

It also treats the code of smart contracts as immutable, which has its advantages. However, many programmers want to their contracts to be upgradable in order to fix bugs, therefore they have to work around the platform limitations by using a proxy pattern.

On the contrary, EOS starts from the assumption that a cryptocurrency cannot get adopted on a massive scale if it does not respond to the users expectations. First of all, paying for every interaction with the system certainly does not encourage people to use it. Therefore, EOS introduced a completely new resource system, which allows to use this cryptocurrency basically for free.

Moreover, since it is natural for people to make mistakes, EOS allows the programmers to update the code of their smart contracts, and allows the block producers to reverse transactions and freeze accounts to help users recover their funds in case of an vulnerability found in the smart contract they were using. Additionally, smart contracts can also provide Ricardian contracts, which help users who are not programmers to understand what is the logic of the smart contract.

EOS also proposes an account system, which breaks with a traditional one-to-one relationship of a public-private key pair with a particular account. The new system allows for setting up advanced permission schemes, like requiring signatures of multiple users to perform some actions. Additionally, accounts have human-readable names as opposed to other cryptocurrencies, in which

the users are identified by a public key, which is long and looks like a pseudo-random string.

When it comes down to performance, EOS also shines compared to Ethereum, mostly due to the Delegated Proof of Stake consensus. However, the performance figures are likely to change in the near future, as Ethereum is preparing to switch to the Proof of Stake consensus, and EOS is promising a new Virtual Machine in the upcoming 2.0 version of the software. According to the the official news [eos19d], the new VM is 8 times faster than a current VM.

Taking above mentioned aspects into consideration, EOS platform seems to be more suitable for handling real-world use cases for smart contracts, as it addresses three most common problems with smart contract-enabled cryptocurrencies – performance, ease of use, and costs. However, the stellar performance of EOS mostly arises from the fact, that only a small subset of all nodes participate in the consensus. There are 21 block producers who can not only confirm transactions, but can also do things that are not possible in any other cryptocurrency, such as freezing accounts and reversing transactions. Therefore, EOS breaks two of the most fundamental rules of cryptocurrencies, which are finality of transaction after they are confirmed, and that the only way to control an account is to possess its private key.

On the contrary, in Ethereum, no node has a special role, and all of them can participate in the consensus. Ethereum also conforms to the aforementioned fundamental rules of cryptocurrencies. Therefore, when choosing a right platform for developing a smart contract-based application, the technical capabilities should not be the only criteria to consider. One should be informed about the way these capabilities are achieved and decide which features of the system are more important for their project.

5.2 Possible improvements to the contracts

The smart contract prepared for this thesis could be improved in few areas.

5.2.1 Unpredictable pseudo-random number generation

Currently, both Ethereum and EOS gambling contracts use deterministic values like a hash of the previous block or other data that is present on the blockchain to generate pseudo-random numbers. It is suitable for research and development purposes. However, it is not suitable for real-world applications, as these values are known also to any cryptocurrency user. Therefore, they can predict or influence the next returned 'random' number.

In order to reliably generate random numbers in smart contracts, one of two sources can be used:

- a distributed seed that is generated from data sent by multiple users
- an oracle service

The first approach is descried in the EOS developer guide. It is presented as a way to randomly select a winner and a loser assuming the smart contract is a two-player game. However, it can also be used for more complex contracts as well.

> "In the example of two users wanting to play a game with 50/50 odds, each player must first submit a hash of their secret. By submitting their hash, they then become eligible to play the game. Once both players have submitted the hash of their secret, they are effectively engaged in the game. It's only when both players reveal their secrets that both of their submitted hashes and recently submitted secrets are hashed. From

> the result of this hash, two numbers are selected to determine a winner and a loser." [eos19c]

This approach provides a deterministic seed, which is a combination of both secrets submitted by the users. It also prevent the scenario when the last user who submits their secret simulates the generation process and by trial and error determines the secret which will generate a number that they want. The prevention is provided by the necessity of submitting the hashes of the secrets before being able to take part in the generation process. This solution is completely decentralized, however it requires multiple steps to be taken by multiple users who could not be at their computers at the same time. Therefore, it is not suitable for interactive environments where the users expect fast responses.

The second approach involves relying on a trusted third party to generate a random number for the smart contract. Oracle service providers often attach a proof that the generation process was not tampered with, and that no generated numbers were discarded. For instance, many providers perform the generation process inside a secured enclave embedded in the hardware, such as *Intel SGX* technology. This solution delivers generated numbers much faster than the decentralized approach, however it depends on a proper functioning of a third party service.

5.2.2 Time delays between betting and resolving procedures

Currently, the smart contracts do not impose any limitation on how fast a *resolve* function can be called after the first bet in the round has been made. However, it makes most sense to initiate the resolution procedure when there is a high number of users who made their bets or a sufficiently long time has elapsed.

5.2.3 Fair distribution of costs

Cryptocurrency users should not pay for computations that are not related to their own actions. For instance, let us examine the *bet* function in the provided smart contracts. It checks whether the user-provided arguments are correct and saves the information about the bet to the contract's storage. All these operations are performed only to handle to user request.

However, there is also a *resolve* function, which generates the random number and computes the rewards for every user who took place in the recent betting round. Therefore, the user who invoked the *resolve* function will be billed for computing not only their, but also everyone else's rewards.

This problem concerns Etherem much more than EOS, because an EOS user would only use lose some of the CPU and NET resources, which will regenerate over time. Moreover, the contract has been written in such way, that the RAM costs of executing the *resolve* function will be covered by the contract itself. Therefore, an EOS user would only temporarily lose the resources consumed by the resolving procedure. However, Ethereum gas costs will incur a permanent loss of Ether used for computing someone else's balance.

Ethereum and EOS do not support spreading the costs of functions execution across multiple users. Therefore, let us discuss how this issue could be mitigated on the smart contract level. One way to solve this problem would be to calculate how much gas is used for computing reward for a single participant of the betting round. Then, in the *bet* function, the users would have to pay not only an amount that they want to bet, but also an amount that would cover the costs of computing they rewards in case they will win. With this modification, when some user invokes the

resolve function, the smart contract will refund them for computing everyone else's rewards from the additional amounts of cryptocurrency that were taken during their bets.

However, this approach has its own issues, like the dynamic gas prices and the fact, that every user who makes a bet has to pay an extra price for the potential costs that would need to be covered in case they win. Therefore, they have to pay the extra price regardless if they won or not.

Another approach would be to redesign the whole resolving procedure in such way, that every betting user would be calculating their own rewards, if possible. It has to be remembered though, that the random winning number would still have to be the same by every user despite the resolving procedures being performed separately.

Chapter 6

Conclusions and future work

The primary goal of this thesis was to analyze and compare Ethereum and EOS blockchain platforms in regard to their capabilities and limitations from both user and developer perspective. For this reason, two smart contracts were designed and implemented, and every contract targeted a different platform. Moreover, several automatic tests and a web-based GUI was created, as the thesis aimed to evaluate not only the platforms themselves, but also the whole developer ecosystem around them.

The main conclusion that can be drawn is that although both platforms are Turing-complete, which make them both suitable for executing any logic, EOS is more developer- and user-friendly, and offers much better performance compared to Ethereum. However, EOS also introduced a human element to the governance instead of relying solely on mathematics and cryptography. It did that by giving the block producers much greater powers than it is available in any other cryptocurrency, including freezing accounts and reversing transactions. Whether this approach is positive or negative should be decided on per-project basis.

Future research on this topic may be devoted to evaluating and comparing other smart contract-enabled cryptocurrencies like *NEO* [neo19] (which is already functioning) and *Cardano* [car19] (which has its smart contract-execution engines still in the testing phase). As mentioned in the introduction, smart contracts are one of the most desirable features in the cryptocurrency space. Therefore, it is safe to assume, that more and more cryptocurrencies would start supporting them. Every new implementation could introduce new solutions to the well-known problems, therefore new analysis and comparisons would certainly benefit programmers who are looking for the best platform for developing their smart contract-based applications.

Appendix A

Source code of smart contracts

```
#pragma once

#include <eosio/singleton.hpp>
#include "../token/token.hpp"
#include "../utils/amount.hpp"
#include "../utils/Random/Random.hpp"

using namespace eosio;
using bet_number = uint8_t;

enum class phase : bool {
   NO_BET, BET_MADE
};

CONTRACT gambling : public contract {
public:
   using contract::contract;

   gambling(const name& receiver, const name& code,
            const datastream<const char*>& ds);

   ACTION init(const name& owner,
               const extended_symbol& token_to_accept);

   [[eosio::action("setowner")]]
   void set_owner(const name& new_owner);

   [[eosio::on_notify("*::transfer")]]
   void bet(const name& from, const name& to, const asset& quantity,
            const std::string& memo);

   ACTION resolve();

   ACTION withdraw(const name& to);

private:
   static const bet_number min_allowed_number = 0;
   static const bet_number max_allowed_number = 9;

   static bet_number generate_winning_number();

   void transfer(const name& recipient, const asset& value,
                 const name& token_account);
```

```
44
45    TABLE bet_table {
46       name better;
47       bet_number number;
48       amount value;
49
50       [[nodiscard]] uint64_t primary_key() const {
51         return better.value;
52       }
53
54       [[nodiscard]] uint64_t by_number() const {
55         return number;
56       }
57    };
58    using bet_index = eosio::multi_index<"bets"_n, bet_table, indexed_by<
59       "bynumber"_n, const_mem_fun<bet_table,uint64_t,&bet_table::by_number>
60    >>;
61
62    TABLE withdrawal_table {
63       name better;
64       amount value;
65
66       [[nodiscard]] uint64_t primary_key() const {
67         return better.value;
68       }
69    };
70    using withdrawal_index = eosio::multi_index<"withdrawals"_n,
71                                                withdrawal_table>;
72
73    struct state_table {
74       phase phase;
75       /** An owner of the contract **/
76       name owner;
77       /** A token that the contract is configured to work with **/
78       extended_symbol token;
79       /** How much of the contract balance is reserved for withdrawals **/
80       amount withdrawal_balance;
81    };
82    using state_singleton = singleton<"state"_n, state_table>;
83    state_singleton state_db;
84 };
```

Listing A.1: [Full source code - EOS] gambling.hpp

```
1  #include "gambling.hpp"
2
3  gambling::gambling(const name& receiver, const name& code,
4                     const datastream<const char*>& ds)
5     : contract(receiver, code, ds), state_db(_self, _self.value) { }
6
7  void gambling::init(const name& owner,
8                      const extended_symbol& token_to_accept) {
9     require_auth(get_self());
10    check(not state_db.exists(), "Contract already initialized");
11    auto initial = state_table {
12       .phase = phase::NO_BET,
13       .owner = owner,
14       .token = token_to_accept,
15       .withdrawal_balance = 0
```

```
16      };
17      state_db.set(initial, get_self());
18  }
19
20  void gambling::set_owner(const name& new_owner) {
21      auto state = state_db.get();
22      require_auth(state.owner);
23      state.owner = new_owner;
24      state_db.set(state, get_self());
25  }
26
27  void gambling::bet(const name& from, const name& to,
28                     const asset& quantity, const std::string& memo) {
29      if (to != get_self() or from == "eosio"_n or
30          from == "eosio.ram"_n or from == "eosio.stake"_n) {
31          return;
32      }
33      require_auth(from);
34      auto state = state_db.get();
35      extended_asset value = {quantity, get_first_receiver()};
36      extended_symbol expected_token = state.token;
37      check(value.get_extended_symbol() == expected_token,
38            "Tried to bet with a not supported token");
39      check(memo.length() > 0, "Fill the memo with a number you want to bet on");
40
41      auto passed_number = std::stoi(memo);
42      check(passed_number >= min_allowed_number and
43            passed_number <= max_allowed_number,
44            "Passed number exceeds the allowed boundaries");
45      auto number = static_cast<bet_number>(passed_number);
46
47      /** Get the table handle **/
48      bet_index bets = {get_self(), get_self().value};
49      auto previous_bet = bets.find(from.value);
50
51      if (previous_bet == bets.end()) {
52          /** No bet by this account exists in this round **/
53          bets.emplace(get_self(), [&](auto& row) {
54              row.better = from;
55              row.number = number;
56              row.value = value.quantity.amount;
57          });
58      } else {
59          /** Previous bet by this account exists in this round **/
60          bets.modify(previous_bet, same_payer, [&](auto& row) {
61              row.number = number;
62              row.value += value.quantity.amount;
63          });
64      }
65
66      state.phase = phase::BET_MADE;
67      state_db.set(state, same_payer);
68  }
69
70  void gambling::resolve() {
71      auto state = state_db.get();
72      check(state.phase == phase::BET_MADE, "No bet made, cannot resolve");
73      bet_number winning_number = generate_winning_number();
74      bet_index bets = {get_self(), get_self().value};
```

```
   auto bets_by_numbers = bets.get_index<"bynumber"_n>();
   auto winning_bets_begin = bets_by_numbers.find(winning_number);
   amount contract_balance =
         token::get_balance(state.token.get_contract(), get_self(),
                            state.token.get_symbol().code()).amount;

   /** Amount of tokens that is going to be divided among the winners **/
   amount current_resolve_balance = contract_balance - state.withdrawal_balance;
   amount remainder = current_resolve_balance;

   if (winning_bets_begin != bets_by_numbers.end()) {
      /** There are some accounts with winning bets **/
      amount winning_bets_sum = 0;

      for (auto bet = winning_bets_begin; bet != bets_by_numbers.end(); ++bet) {
         winning_bets_sum += bet->value;
      }

      for (auto bet = winning_bets_begin; bet != bets_by_numbers.end(); ++bet) {
         amount reward = (current_resolve_balance * bet->value) / winning_bets_sum;
         withdrawal_index withdrawals = {get_self(), bet->better.value};
         auto account_withdrawal = withdrawals.find(bet->better.value);

         if (account_withdrawal == withdrawals.end()) {
            /** Account has no pending withdrawal balance **/
            withdrawals.emplace(get_self(), [&](auto& row) {
               row.better = bet->better;
               row.value = reward;
            });
         } else {
            /** Account has pending withdrawal balance **/
            withdrawals.modify(account_withdrawal, same_payer, [&](auto& row) {
               row.value += reward;
            });
         }

         state.withdrawal_balance += reward;
         remainder -= reward;
      }
   }

   /** Prepare for the next round by cleaning the bets table **/
   auto it = bets.begin();
   while (it != bets.end()) {
      it = bets.erase(it);
   }

   /** Add the remaining balance to the owner withdrawal balance **/
   auto owner_withdrawal = withdrawal_index {get_self(), state.owner.value};
   auto owner_withdrawal_iter = owner_withdrawal.find(state.owner.value);
   if (owner_withdrawal_iter == owner_withdrawal.end()) {
      /** Owner has no pending withdrawal balance **/
      owner_withdrawal.emplace(get_self(), [&](auto& row) {
         row.better = state.owner;
         row.value = remainder;
      });
   } else {
      /** Owner has pending withdrawal balance **/
      owner_withdrawal.modify(owner_withdrawal_iter, same_payer, [&](auto& row) {
```

```
134             row.value += remainder;
135         });
136     }
137
138     state.withdrawal_balance += remainder;
139     state.phase = phase::NO_BET;
140     state_db.set(state, same_payer);
141 }
142
143 void gambling::withdraw(const name& to) {
144     require_auth(to);
145     auto withdrawals = withdrawal_index{get_self(), to.value};
146     auto reward =
147           withdrawals.require_find(to.value, "No balance available to withdraw");
148     auto state = state_db.get();
149     state.withdrawal_balance -= reward->value;
150     state_db.set(state, same_payer);
151     withdrawals.erase(reward);
152     transfer(to, reward->value.to_asset(state.token.get_symbol()),
153              state.token.get_contract());
154 }
155
156 bet_number gambling::generate_winning_number() {
157     eosblox::Random rand;
158     return rand.nextInRange(min_allowed_number, max_allowed_number + 1);
159 }
160
161 void gambling::transfer(const name& recipient, const asset& value,
162                         const name& token_account) {
163     token::transfer_action action(token_account, {get_self(), "active"_n});
164     action.send(get_self(), recipient, value, "");
165 }
```

Listing A.2: [Full source code - EOS] gambling.cpp

```
1  pragma solidity ^0.5.0;
2
3  import 'openzeppelin-solidity/contracts/math/SafeMath.sol';
4
5  contract BaseGambling {
6      using SafeMath for uint;
7
8      enum Phase { NO_BET, BET_MADE }
9      struct Bet {
10         uint number;
11         uint value;
12     }
13
14     uint constant internal minAllowedNumber = 0;
15     uint constant internal maxAllowedNumber = 9;
16     uint internal withdrawalBalance = 0;
17
18     Phase internal currentPhase;
19     mapping(address => Bet) internal bets;
20     // Stores the addresses that put a bet on a given number
21     mapping(uint => address[]) internal numberMappedBetters;
22     /** Stores the index at which a given address is stored in the
23         'numberMappedBetters' value */
24     mapping(address => uint) internal addressIndices;
```

```
25
26    mapping(address => uint) internal toWithdraw;
27    address payable internal ownerAddress;
28
29    event BetAcceptedEvent(address indexed better, uint number, uint value);
30    event BetEndEvent(uint indexed number);
31
32    modifier onlyOwner() {
33       require(msg.sender == ownerAddress);
34       _;
35    }
36
37    constructor() internal {
38       ownerAddress = msg.sender;
39       currentPhase = Phase.NO_BET;
40    }
41
42    function setOwner(address payable _owner) external onlyOwner {
43       ownerAddress = _owner;
44    }
45
46    function resolve() external returns (uint) {
47       require(currentPhase == Phase.BET_MADE);
48
49       uint winningNumber = generateWinningNumber();
50       address[] memory winningAddresses = numberMappedBetters[winningNumber];
51
52       uint currentResolveBalance = getBalance().sub(withdrawalBalance);
53       uint remainder = currentResolveBalance;
54
55       if (winningAddresses.length > 0) {
56          uint winningBetsSum = 0;
57          for (uint i = 0; i < winningAddresses.length; ++i) {
58             winningBetsSum = winningBetsSum.add(bets[winningAddresses[i]].value);
59          }
60
61          for (uint j = 0; j < winningAddresses.length; ++j) {
62             uint reward =
63                   currentResolveBalance.mul(bets[winningAddresses[j]].value)
64                                        .div(winningBetsSum);
65             toWithdraw[winningAddresses[j]] =
66                   toWithdraw[winningAddresses[j]].add(reward);
67             withdrawalBalance = withdrawalBalance.add(reward);
68             remainder = remainder.sub(reward);
69          }
70       }
71
72       // Cleaning the mappings to prepare them for the next round of betting
73       for (uint number = minAllowedNumber; number <= maxAllowedNumber; ++number) {
74          address[] memory addresses = numberMappedBetters[number];
75          for (uint addr = 0; addr < addresses.length; ++addr) {
76             delete bets[addresses[addr]];
77          }
78          delete numberMappedBetters[number];
79       }
80
81       toWithdraw[ownerAddress] = toWithdraw[ownerAddress].add(remainder);
82       withdrawalBalance = withdrawalBalance.add(remainder);
83
```

```
84        currentPhase = Phase.NO_BET;
85        emit BetEndEvent(winningNumber);
86        return winningNumber;
87    }
88
89    function withdraw() external {
90        require(toWithdraw[msg.sender] > 0);
91        uint reward = toWithdraw[msg.sender];
92        toWithdraw[msg.sender] = 0;
93        withdrawalBalance = withdrawalBalance.sub(reward);
94        transfer(msg.sender, reward);
95    }
96
97    function readyToWithdraw() external view returns (uint) {
98        return toWithdraw[msg.sender];
99    }
100
101   function updateBet(uint _number, uint _value) internal {
102       uint previousBetValue = bets[msg.sender].value;
103       uint previousBetNumber = bets[msg.sender].number;
104       bets[msg.sender] = Bet(_number, _value.add(previousBetValue));
105
106       if (previousBetValue > 0) {
107           // Previous bet exists
108           address[] storage bettersOnPrevious =
109                 numberMappedBetters[previousBetNumber];
110           // Removing previous bet
111           if (bettersOnPrevious.length > 1) {
112               uint existingBetIndex = addressIndices[msg.sender];
113               bettersOnPrevious[existingBetIndex] =
114                     bettersOnPrevious[bettersOnPrevious.length - 1];
115           }
116           bettersOnPrevious.length--;
117       }
118
119       uint index = (numberMappedBetters[_number].push(msg.sender)) - 1;
120       addressIndices[msg.sender] = index;
121
122       currentPhase = Phase.BET_MADE;
123       emit BetAcceptedEvent(msg.sender, _number, msg.value);
124   }
125
126   function generateWinningNumber() internal view returns (uint) {
127       return uint(blockhash(block.number - 1)) % (maxAllowedNumber + 1);
128   }
129
130   function getBalance() internal view returns (uint);
131
132   function transfer(address payable _recipient, uint _value) internal;
133 }
```

Listing A.3: [Full source code - Ethereum] BaseGambling.sol

```
1  pragma solidity ^0.5.0;
2
3  import "./BaseGambling.sol";
4
5  contract GamblingEther is BaseGambling {
6
```

```
7     function bet(uint _number) external payable {
8        require(msg.value > 0 && _number >= minAllowedNumber &&
9                _number <= maxAllowedNumber);
10       updateBet(_number, msg.value);
11    }
12
13    function getBalance() internal view returns (uint) {
14       return address(this).balance;
15    }
16
17    function transfer(address payable _recipient, uint _value) internal {
18       _recipient.transfer(_value);
19    }
20 }
```

Listing A.4: [Full source code - Ethereum] GamblingEther.sol

```
1  pragma solidity ^0.5.0;
2
3  import 'openzeppelin-solidity/contracts/token/ERC20/IERC20.sol';
4  import 'openzeppelin-solidity/contracts/token/ERC20/ERC20.sol';
5  import 'openzeppelin-solidity/contracts/token/ERC20/SafeERC20.sol';
6  import "./BaseGambling.sol";
7
8  contract GamblingERC20 is BaseGambling {
9     using SafeERC20 for IERC20;
10
11    IERC20 public token;
12
13    constructor(address _tokenAddress) BaseGambling() public {
14       token = IERC20(_tokenAddress);
15    }
16
17    function bet(uint _number, uint _value) external {
18       require(_value > 0 && _number >= minAllowedNumber &&
19               _number <= maxAllowedNumber);
20       uint beforeTransferBalance = getBalance();
21       require(token.transferFrom(msg.sender, address(this), _value));
22       uint realValue = getBalance() - beforeTransferBalance;
23       updateBet(_number, realValue);
24    }
25
26    function getBalance() internal view returns (uint) {
27       return token.balanceOf(address(this));
28    }
29
30    function transfer(address payable _recipient, uint _value) internal {
31       require(token.transfer(_recipient, _value));
32    }
33 }
```

Listing A.5: [Full source code - Ethereum] GamblingERC20.sol

Listings

Bibliography

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. *IACR Cryptology ePrint Archive*, 2014:349, 2014.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[bit19a] Bitcoin wiki - mining. [on-line] `https://en.bitcoin.it/wiki/Mining`, accessed: 23 September 2019.

[bit19b] Bitcoin wiki - proof of work. [on-line] `https://en.bitcoin.it/wiki/Proof_of_work`, accessed: 23 September 2019.

[bit19c] Bitcointalk.org - proof of stake. [on-line] `https://bitcointalk.org/index.php?topic=27787.0`, accessed: 23 September 2019.

[boo19] Boost multiindex. [on-line] `https://www.boost.org/doc/libs/1_71_0/libs/multi_index/doc/index.html`, accessed: 26 September 2019.

[car19] Cardano documentation. [on-line] `https://cardanodocs.com/introduction`, accessed: 28 September 2019.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. 1999.

[coi19] Coinmarketcap. [on-line] `https://coinmarketcap.com`, accessed: 9 September 2019.

[CPV+16] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[DD18] Evan Duffield and Daniel Diaz. Dash: A payments-focused cryptocurrency. Whitepaper, [online] `https://github.com/dashpay/dash/wiki/Whitepaper`, 2018. accessed: 26 September 2019.

[eos19a] Eos.io developer guide - accounts and permissions. [on-line] `https://developers.eos.io/eosio-nodeos/docs/accounts-and-permissions`, accessed: 26 September 2019.

[eos19b] Medium - eosio dawn 3.0 now available. [on-line] `https://medium.com/eosio/eosio-dawn-3-0-now-available-49a3b99242d7`, accessed: 26 September 2019.

[eos19c] Eos developer guide - random number generation. [on-line] `https://developers.eos.io/eosio-cpp/v1.3.2/docs/random-number-generation`, accessed: 28 September 2019.

[eos19d] Eosio 2.0. [on-line] `https://eos.io/news/introducing-eosio-2`, accessed: 28 September 2019.

[ES18] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[eth19] Openzepplin blog - ethereum in depth, part 2. [on-line] `https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9`, accessed: 26 September 2019.

[Gri04] Ian Grigg. The ricardian contract. In *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004.*, pages 25–31. IEEE, 2004.

[Jep15] Christina Jepson. Dtb001: Decred technical brief. [online] `https://coss.io/documents/white-papers/decred.pdf`, 2015. accessed: 26 September 2019.

[KMVOV96] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[L+18] Daniel Larimer et al. Eos.io technical white paper v2. [on-line] `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`, 2018. Accessed on 18 September 2019.

[Lar19] Daniel Larimer. Delegated proof-of-stake (dpos). accessed: 23 September 2019.

[LeM18] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. [on-line] `https://nano.org/en/whitepaper`, 2018. accessed: 26 September 2019.

[LSZ+16] Daniel Larimer, Ned Scott, Valentine Zavgorodnev, Benjamin Johnson, James Calfee, and Michael Vandeberg. Steem: An incentivized, blockchain-based social media platform. *March. Self-published*, 2016.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[neo19] Neo whitepaper. [on-line] `https://docs.neo.org/docs/en-us/basic/whitepaper.html`, accessed: 28 September 2019.

[Noe15] Shen Noether. Ring signature confidential transactions for monero. *IACR Cryptology ePrint Archive*, 2015:1098, 2015.

[ope19] Openzeppelin contracts github repository. [on-line] `https://github.com/OpenZeppelin/openzeppelin-contracts`, accessed: 27 September 2019.

[ora19] Bitcoinhub berlin - blockchain oracles. [on-line] `https://blockchainhub.net/blockchain-oracles`, accessed: 26 September 2019.

[Pop16] Serguei Popov. The tangle. *cit. on*, page 131, 2016.

[SL17] Fabian Schuh and Daniel Larimer. Bitshares 2.0: General overview. 2017.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[Vuk15] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.

[W+14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

Poznan University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
    author = "Krzysztof Wencel",
    title = "{Comparison of programming capabilities of Ethereum and EOS blockchain platforms}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2019",
}
```