BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ANALYSIS OF THE MOVE PROGRAMMING LANGUAGE FOR BLOCKCHAIN PLATFORMS

ANALÝZA PROGRAMOVACÍHO JAZYKA MOVE PRO BLOCKCHAINOVÉ PLATFORMY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ADAM ŠMEHÝL**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN PEREŠÍNI**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148522

Institut:	Department of Intelligent Systems (UITS)
Student:	**Šmehýl Adam**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Analysis of the Move Programming Language for Blockchain Platforms**
Category:	Security
Academic year:	2022/23

Assignment:

1. Get familiar with the principles of blockchains and smart contracts.
2. Study the programming model of Move language and blockchain platforms such as Aptos, and Sui and compare it with the programming model of Ethereum.
3. Study and compare Move language related to other smart contract languages such as Solidity (EVM, Ethereum) and Rust (Solana).
4. Propose at least 3 use cases on how to compare Move language with others.
5. Implement various smart contract scenarios in different smart contract languages.
6. Evaluate and analyze the security, and performance of Move in these scenarios.
7. Discuss results and usability of Move language.

Literature:

- I. Homoliak, S. Venugopalan, D. Reijsbergen, Q. Hum, R. Schumi and P. Szalachowski, "The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses," in *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 341-390, Firstquarter 2021, doi: 10.1109/COMST.2020.3033665. https://doi.org/10.1109/COMST.2020.3033665
- The Move Book, The Move Programming Language: https://move-book.com/index.html
- Move: A Language With Programmable Resources, https://github.com/diem/diem/blob/main/developers.diem.com/static/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf
- Ethereum Virtual Machine (EVM): https://ethereum.org/en/developers/docs/evm/
- Move Language Introduction: https://move-language.github.io/move/introduction.html

Requirements for the semestral defence:
1-4

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

Supervisor:	**Perešíni Martin, Ing.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	3.11.2022

Abstract

This thesis studies the Move programming language, focusing on its usability for developing applications (smart contracts or programs) on blockchain platforms. Two key aspects are considered: first, a comparison of programming models using Move to widely used models of EVM-compatible platforms like Ethereum and the popular Solana blockchain; and second, the implementation of the same program in Solidity on Ethereum, Rust on Solana, and Move on Aptos. Criteria for comparison include deployment and execution costs, processing speed, code readability, and overall development experience. A detailed analysis of Move's unique features, such as resource management, the use of generics, and other security enhancements in programming, is conducted. The results demonstrate Move's potential for extensive use in the blockchain field, with its strong emphasis on secure coding and resource management contributing to the growing interest within the blockchain community.

Abstrakt

Tato práce se zabývá zkoumáním programovacího jazyka Move z hlediska jeho použitelnosti pro vývoj aplikací (smart kontraktů či programů) na blockchainových platformách. Práce zahrnuje analýzu dvou klíčových aspektů. Prvním z nich je porovnání programovacích modelů platforem používajících jazyk Move s běžně používanými modely EVM-kompatibilních platforem (jako je Ethereum) a stále populárnějšího blockchainu Solana. Druhou částí práce je implementace stejného programu v Solidity na Ethereu, Rustu na Solaně a Move na Aptosu. Mezi kritéria pro porovnání těchto tří řešení patří transakční náklady, rychlost zpracování, čitelnost kódu a zkušenosti z vývoje. V rámci této práce byla provedena podrobná analýza unikátních vlastností jazyka Move, jako je správa zdrojů, používání generik a další zlepšení bezpečnosti při programování. Výsledky práce ukazují potenciál jazyka Move pro rozsáhlé použití v oblasti blockchainu, přičemž jeho silnou stránkou je bezpečnost (angl. secure coding), což přispívá k rostoucímu zájmu blockchainové komunity.

Keywords

Move programming language, Move practical applications, Move evaluation, smart contracts, smart contract development, decentralized applications, blockchain platforms, Ethereum, Solana, Aptos, Sui, blockchain security

Klíčová slova

programovací jazyk Move, jazyk Move v praxi, zhodnocení jazyka Move, smart kontrakty, programování smart kontraktů, vývoj smart kontraktů, decentralizované aplikace, blockchainové platformy, Ethereum, Solana, Aptos, Sui, bezpečnost blockchainu

Reference

ŠMEHÝL, Adam. *Analysis of the Move Programming Language for Blockchain Platforms.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Perešíni

Rozšířený abstrakt

Tato práce se zabývá podrobným studiem programovacího jazyka Move s ohledem na jeho použitelnost pro vývoj aplikací na blockchainových platformách. Pro korektní zhodnocení je nutné se zaměřit na dvě hlediska. Prvním je porovnání programovacího modelu při použití jazyka Move s běžně používanými modely EVM-kompatibilních platforem, jako je Ethereum, a také stále populárnějšího blockchainu Solana. Druhým hlediskem je otestování jazyka v praxi pomocí implementace stejného smart kontraktu, programu nebo modulu v každém z těchto jazyků – Solidity na Ethereum, Rustu na Solaně a jazyku Move na Aptosu.

Kritéria pro zhodnocení jsou založena na několika faktorech, jako jsou náklady na nasazení, transakční poplatky, rychlost zpracování, čitelnost kódu, celková zkušenost s vývojem a dostupné nástroje. Tato analýza se snaží poskytnout podrobné porozumění odlišujících charakteristik jazyka, jako je správa zdrojů, generika nebo datový model, a jak tyto charakteristiky přispívají k jeho silným a slabým stránkám a jejich implikaci na vývojáře pracující s blockchainovými technologiemi.

Práce také zkoumá vliv zralosti platformy na vývojový proces. To zahrnuje vývojářské výzvy spojené s omezenými nástroji, nedostatkem vzdělávacích zdrojů a častými změnami, které narušují kompatibilitu. Všechny tyto faktory mají vliv na rychlost a úroveň adopce nových technologií.

Výsledky studie demonstrují potenciál jazyka Move pro rozšiřující se adopci skrze různé blockchainové platformy. Silné stránky jazyka, jako je bezpečnost, správa zdrojů a typová kontrola, přispívají k rostoucímu zájmu blockchainových vývojářů. Práce také zmiňuje aktuální vývoj v ekosystému, například nadcházející spuštění mainnetu Sui nebo snahu solanových vývojářů integrovat podporu pro programy v jazyce Move. To naznačuje rozsáhlé příležitosti pro expanzi v oblasti blockchainu.

Závěrem lze říci, že tato práce poskytuje komplexní zdroj informací pro porozumění programovacího jazyka Move, jeho výhod a výzev a příležitostí v kontextu moderních blockchainových platforem. Závěry přispívají do rozšiřující se kolekce znalostí o vývoji blockchainu a poskytují pevný základ pro další výzkum a praktické implementace s využitím jazyka Move.

Analysis of the Move Programming Language for Blockchain Platforms

Declaration

I hereby declare that this term project was prepared as an original work by the author under the supervision of Mr. Perešíni. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Adam Šmehýl
May 17, 2023

Contents

Chapter 1

Introduction

For quite some time, and even today, blockchain technology developers are still using classic programming languages. However, these languages may pose risks due to their potential to overlook critical aspects, leading to an increased count of vulnerabilities. Security is the most crucial aspect of this industry, so it must be robust. That might lead developers to build a superstructure on top of the programming language core with specific restrictions to lower the chance of introducing errors in the written code.

As with other fields, specialized solutions tend to be the most effective. Therefore, a programming language custom-made for blockchain development not only makes the work more accessible but also addresses specific security criteria with restrictions directly at the language level, lowering the chance of developers introducing errors while creating complex solutions, such as smart contracts.

Over a year ago, several new platforms built around Move emerged in the blockchain space. At the time, Move was a relatively unknown programming language. However, the Move whitepaper outlined promising advancements and techniques that, upon further examination, inspired the creation of this thesis, which showcases Move's potential and its appeal to developers.

This thesis focuses on demonstrating the practical use case of Move on a simple application rather than delving deep into the details of related technologies. Please refer to the official documentation or other sources referenced in this work for detailed information.

Chapter 2

Blockchain Basics and Principles

This chapter delves into the basics and principles of blockchain technology. It explores its origins, core concepts, and structure, including the roles of smart contracts, fees, and incentives. Also, it examines different types of blockchain systems and their increasing adoption across industries, concluding with a discussion on blockchain in the Web 3.0 era.

The content presented is based on the experience and knowledge I have gathered over several years while working with blockchain technology. Nevertheless, some small parts have been adjusted and supplemented using the information and interpretation I have found in this source [25].

2.1 Origin of Blockchain Technology

In 2008, a person under the pseudonym of Satoshi Nakamoto published a paper called "Bitcoin: A Peer-to-Peer Electronic Cash System" [32]. In this paper, Satoshi proposed a system that would become a digital payment system when trusted by enough third parties. The underlying idea to achieve this goal is for this system to be fully transparent, with transaction records publicly available for review by anyone, with a guarantee of finality for the processed transactions. To achieve this, Satoshi introduced an underlying solution – a brand-new technology called a blockchain.

2.2 Bitcoin

Bitcoin is the system that represents the *first* decentralized cryptocurrency. The main reason for its creation was the idea of digital currency not being governed by any authority. Therefore, its value and usage are based on trust and cannot be easily manipulated (at least to some extent). Bitcoin was the first system to utilize blockchain technology.

Within 14 years of its existence, Bitcoin has become a globally recognized currency. As a digital currency not governed by any state and available to anyone worldwide, Bitcoin has already become a standard part of daily life for people in some countries, serving as an alternative to their national currency due to high inflation levels. For instance, Turkey is facing a year-to-year inflation rate of around 80% on its national currency. Reports suggest that Bitcoin has become widely used among Turkish residents precisely because of the ongoing inflation [29].

However, although highly secure, Bitcoin has a slow throughput, which results in its use case being more like a store of value than for active usage. This limitation has led to the creation of other blockchain platforms.

2.3 Blockchain Concepts

This section discusses the core principles characteristic of blockchain systems, followed by the technical details describing the blockchain structure comprised of blocks. With that in mind, it continues through transaction processing and network operation, explaining the importance of consensus and incentives needed to reach a correct consensus between nodes participating in the network.

Core Principles: Availability, Immutability, Scarcity, and Ownership

The principles of availability, immutability, scarcity, and ownership are central to blockchain operation. In a blockchain network, the data is distributed across multiple nodes rather than stored in a single central database. This data distribution is a prominent feature in blockchain technology as it drastically improves the system's availability. If some nodes go offline or are compromised, others still maintain a full copy of the blockchain. Therefore the network can continue to operate normally. It is characteristic for blockchain networks to be immutable. Therefore, once the data is recorded in a block, it cannot be altered or deleted.

Data gains value through its scarcity and the concept of ownership. Scarcity ensures that data is neither created nor destroyed arbitrarily; creating an asset is a privileged operation, and its successful destruction adheres to pre-set conditions. Ownership assigns each data unit an owner who retains the exclusive right to modify its contents unless they grant explicit permission to another party. Unlike traditional finance, where a central authority manages the property, data modification agreed upon by two parties in blockchain equates to a transaction. Under certain conditions, this renders blockchain a distinctive form of digital currency.

Blockchain Structure

Blockchain is a data structure containing many smaller data structures, often called blocks. When creating a new block, the node takes the entire content of the previous block and hashes it as a hash pointer. For the new block to be valid, it must comply with the defined structure, and every block, except the genesis block, must contain a hash pointer to the previous block. The entire blockchain forms a distributed ledger, a consensus of replicated digital data distributed across many nodes worldwide. This ledger is distributed to newly joining nodes in a peer-to-peer manner.

Transaction Processing

A transaction in a blockchain network represents a set of instructions that, when executed, modify the state of the blockchain. For a transaction to be successful, all its instructions must be executed successfully. However, suppose the execution of even a single instruction fails. In that case, the transaction is considered unsuccessful, and all previous changes are reverted, leaving the blockchain in its original state before the transaction attempt. In other words, transactions are atomic – they are either completed as a whole or not at all.

Due to the limited computing resources available, the execution process involves a fee to reduce the likelihood of transaction starvation. The transaction fee size depends on the number of computing resources used for the transaction's execution. The usage of the computation resources is metered in a unit called "gas." Typically, each instruction has associated execution costs in gas units.

When a transaction is processed, it is stored in a block, a data structure that groups transactions executed within the same time window. Each block contains a pointer to a previous block, resulting in a visual representation of the chain of blocks – hence the term "blockchain."

When a transaction is submitted to the network, it is distributed among the nodes, which add it to their current block. Then, depending on the consensus protocol, the network chooses a node (called a leader) to broadcast its block onto the network. The other nodes then verify the block's consistency and the validity of the consensus protocol. If the block is valid, it is added to the local copy of the blockchain stored by each node.

Network Operation: Consensus and Incentives

The effective operation of the network relies on consensus and trust. Nodes participating in the network evaluate the validity of the current network state through consensus. The consensus algorithm used by the particular network determines the specific method. Nevertheless, the network's correct state is typically a result of the consensus by the majority of the nodes.

In the case of an incorrect state proposal, perhaps in an attempt to exploit the network, the proposal is rejected since it deviates from the consensus of the majority. If a single entity gains control over the majority of nodes, it could manipulate the network. Consequently, the network provides incentives to deter malicious activities and promote the proper behavior of participating nodes. These incentives typically take the form of a collection of transaction fees or newly minted coins as rewards, or they may involve both.

2.4 Smart Contracts

Smart contracts are transactions that function as programs. Once deployed, nodes store these transactions in the blockchain, ensuring their persistent availability for execution. Furthermore, nodes execute these transactions within a virtual environment (such as the Ethereum Virtual Machine (EVM) in the case of Ethereum) to guarantee consistent results across all nodes. The primary purpose of smart contracts and decentralized apps is to facilitate complex data modifications or transfer of assets and tokens across the network (blockchain).

2.5 Fees and Incentives

Since computational resources are limited, validators must be incentivized to process transactions. These incentives typically take the form of transaction fees. However, due to limited resources, validators can only process a certain number of transactions within a specified period. Therefore, validators determine the order in which transactions are processed by prioritizing those with higher fees. Consequently, users submitting transactions compete by bidding on the gas price to gain the attention of validators.

On some chains, such as Ethereum, network congestion or high usage can lead to fierce bidding wars, resulting in transaction fees that amount to hundreds or even thousands of U.S. dollars when denominated in that currency.

2.6 Types of Blockchain Systems

Blockchain systems can be categorized based on their properties as either public or private:

1. Public – Blockchain records are publicly available to anyone.
2. Private – Blockchain records are accessible only within the company or entity.
   In addition, the data is encrypted to prevent unauthorized review by any third party in case of a leak.

Furthermore, blockchain networks can be classified based on permissions, either as permissioned or permissionless:

1. Permissioned – A centralized entity decides who can participate as a node in the network.
2. Permissionless – Anyone can become a node and participate in network operation.

2.7 Adoption

More than ten years later, blockchain technology is revolutionizing the world, and its presence is becoming increasingly prevalent in our daily lives. In the face of global competition among companies in various industries, these organizations seek new opportunities to seize. As the scale and wealth of the industry continue to grow, the world's largest corporations are starting to take notice – evidenced, for instance, by Facebook's rebranding to Meta [26].

2.8 Blockchain in the Era of Web 3.0

It is essential to understand that Web 3.0 represents an era of transactional freedom. The idea of Web 3.0 is to follow up on the commerce and people-to-people connection (Web 2.0) by allowing people to transact with whoever other people they want to transact with, instantly, whenever they want, without any governing entity overseeing the transaction. This teardown of the financial borders with blockchain and Web 3.0 solves a couple of problems:

1. the issue of tracking ownership (of property)
2. the need to transact in a specific denominated currency
3. cross-border transactions

Besides financial data, blockchain technology enables the storage of various other forms of digital data, including digital ledgers and digital logging, offering a versatile platform for a wide range of applications.

Chapter 3

Overview of Existing Solutions

Chapter 3 presents an overview of existing solutions for blockchain platforms and decentralized application development, focusing on Ethereum and Solana. A brief comparison of Ethereum and Solana is showcased in Table 3.1, with this chapter later delving into the core components of each platform, including their execution environments, storage models, transaction fees, and runtime.

To provide accurate and comprehensive descriptions of each platform, I have relied on official documentation, research papers, and expert articles found online [7, 17, 24, 12].

property	**Ethereum**	**Solana**
consensus	Proof-of-Stake	Proof-of-Stake
language	Solidity	Rust, C++ and Python
scalability	sharding and layer 2s	codebase and hardware
TPS	30 – 50 TPS	up to 100k TPS
TTF	≈15 minutes	≈2.5 seconds
fees	tens of dollars	fraction of a cent

Table 3.1: Comparison of main properties between both Ethereum and Solana.

3.1 Ethereum

While Bitcoin is often referred to as a distributed ledger with replicated data on many nodes, Ethereum represents a single instance of an object resembling a replicated (distributed) state machine, often called "one big distributed computer."

Ethereum has a large data structure encompassing all account data and the machine state. This massive data structure, representing the machine (network) state, is implemented using a modified Merkle Patricia Trie [21]. The typical EVM blockchain state is depicted in Figure 3.1.

This machine state changes from block to block due to the predefined behavior that emerges from executing transactions. The EVM (Ethereum Virtual Machine) defines this behavior by specifying rules. Every validator (node) in the network is aware of these rules, as every possible state transition is defined by a set of state transition functions with a single output. During transaction execution, nodes independently follow the rules. Upon successful execution, they all reach the same new state defined by the state transition

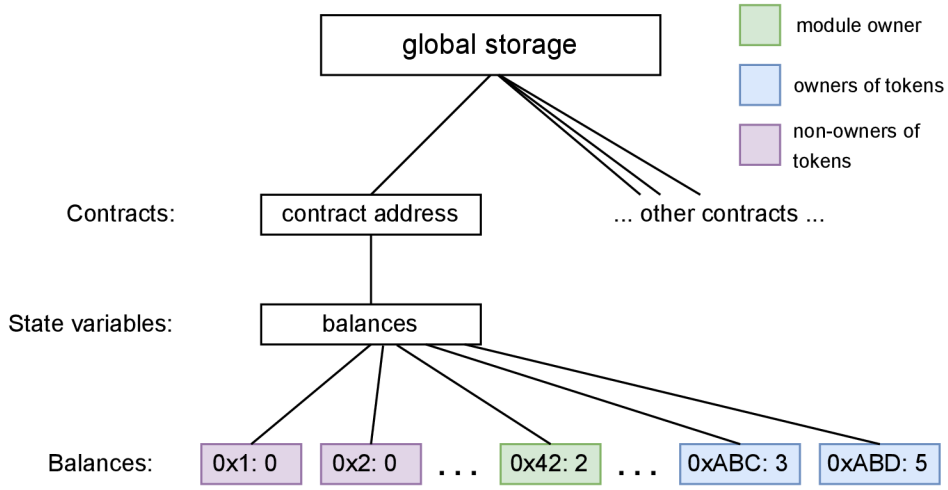


Figure 3.1: Solidity (EVM) blockchain state[1].

function. This is why the network state is referred to as a replicated state machine, as state machine transitions are consistently replicated across every node.

On Ethereum, there are a few transactions types:

- Those resulting in message calls – such as regular transactions or smart contract execution.
- Those resulting in contract creation (deployment).

The contract creation process leads to the creation of a new contract account containing executable compiled smart contract bytecode. This bytecode is available for execution whenever it is called by a message call.

Ethereum Execution Pipeline

The Ethereum virtual machine (EVM) is an isolated runtime environment where execution clients (nodes) execute EVM-compatible bytecode. It behaves as a stack-based machine with a maximum stack size of 1024 items. Each item is a 256-bit word. For execution purposes, the EVM maintains volatile (transient) memory in the form of a word-addressed byte array. However, as part of the smart contract account, non-volatile (persistent) storage is available for storing data if the smart contract needs to retain any information between transactions.

A smart contract is essentially compiled EVM-compatible bytecode stored in a "smart contract account" after being deployed onto the network by a developer. This bytecode consists of numerous EVM opcodes, and the execution of such code is triggered by transactions targeting a specific smart contract account. The full execution pipeline is depicted in Figure 3.2.

[1]Image adapted from https://github.com/move-language/move/tree/main/language/documentation/tutorial.

Gas

To incentivize the EVM to execute transactions, the user must pay an execution fee. Each executed code has a specified compute resource budget to prevent unnecessary over-usage of computing resources. The EVM measures the computing resource consumption in a unit called gas. Each EVM instruction (opcode) has a set gas cost reflecting its computational complexity. As a result, the EVM consumes gas from the provided gas limit during code execution. The consumption of gas is illustrated in Figure 3.2. If the EVM runs out of gas during execution, it raises an "Out-of-gas" exception, resulting in transaction processing failure and the restoration of the original state. Upon transaction submission, the user (sending account) specifies this computing budget by prepaying a specific amount of gas in ether (ETH). The total transaction fee cost is the *gas price * gas units*, where the user sets the gas price, effectively stating how much they are willing to pay for the gas.

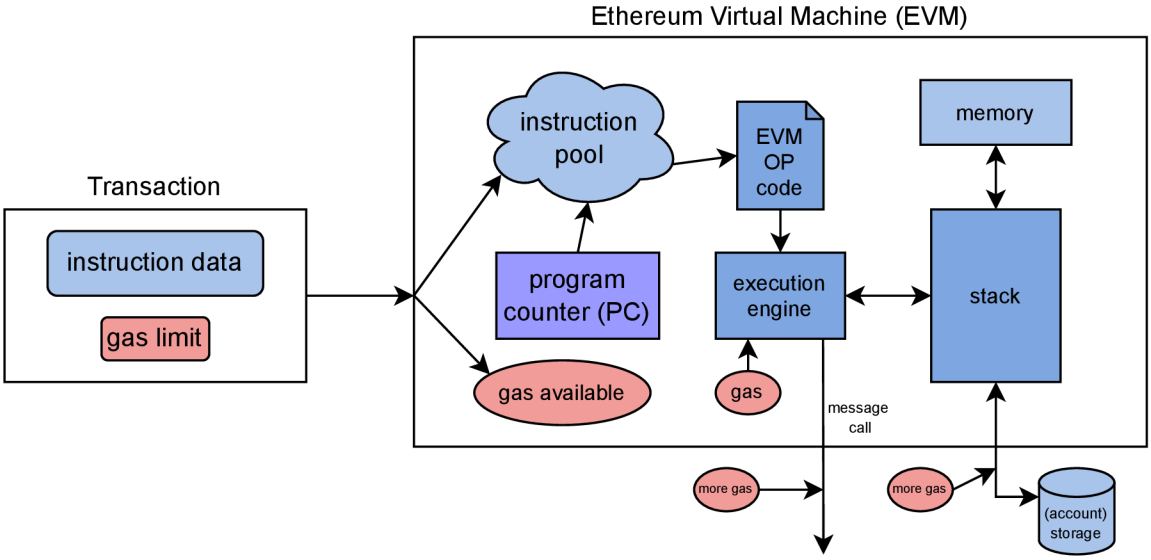


Figure 3.2: EVM transaction execution pipeline.

3.2 Solana

"A Solana cluster is a set of validators working together to serve client transactions and maintain the integrity of the ledger."[16] In the case of Solana, the network selects one node to be a leader for a fixed time interval called a "slot." Within the duration of a slot, only the appointed leader is expected to produce a block. Currently, the network rotates a leader every four slots following an order set in advance for the entire epoch. An epoch on Solana consists of 432,000 slots, with a leader schedule determined at the epoch's start.

"Clients send transactions to any validator's Transaction Processing Unit (TPU) port. If the node is in the validator role, it forwards the transaction to the designated leader. If in the leader role, the node bundles incoming transactions, timestamps them creating an entry, and pushes them onto the cluster's data plane. Once on the data plane, the transactions are validated by validator nodes, effectively appending them to the ledger."[16] The validator rejects blocks bearing the signature of anyone other than the current slot leader.

The On-chain Storage Model of Solana

Data storage on the Solana network takes the form of an "account." An account is an arbitrary location on the network designed to store persistent data, provided that it pays a "rent" for the storage resources used. More details on the rent are discussed in the next section.

Alongside the data itself, the account includes metadata for access control. There are three types of accounts on Solana:

- Program accounts — store executable code, which is equivalent to a program (often referred to as a smart contract)
- Storage accounts — store data connected to a program
- Token accounts — store the account balance of a specific token (e.g., SPL token account)

In contrast to Ethereum, where smart contract code and data are stored in the same account, Solana's program (executable code) is stored in a different account than its data.

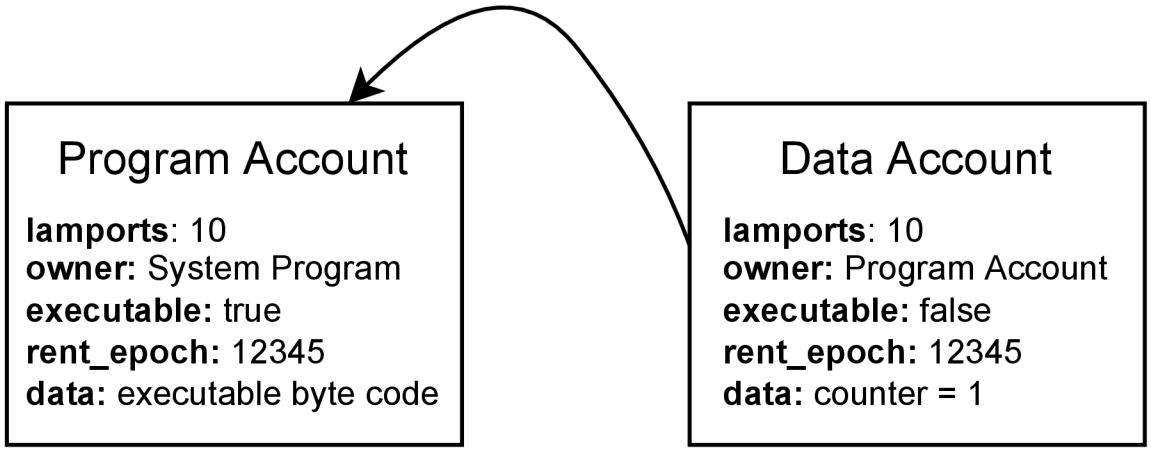


Figure 3.3: Solana on-chain storage model – a program account and associated storage account[2].

Overall, all accounts are either executable or non-executable. An executable account storing executable bytecode is a program. Every non-executable account has an associated owner program address, and only the specified owner can modify the stored data. Otherwise, the storage account is transparent, meaning any program on the Solana network can read the account data. The content of accounts and the mutual connection of data accounts and program accounts is visualized in Figure 3.3.

Transaction Fees and Rent

Solana has two types of fees: transaction fees and storage rent. Users must pay a small transaction fee in the form of the network's native token ($SOL) to incentivize validators to process their transactions. The standard transaction fee is currently static at 0.000005 $SOL per signature.

[2]Image source: https://solanacookbook.com/core-concepts/accounts.html#account-model.

With the introduction of fee markets on Solana in the autumn of 2022, users can now add a prioritization fee. "The prioritization fee is calculated by multiplying the requested maximum compute units by the compute-unit price (specified in increments of 0.000001 lamports per compute unit), rounded up to the nearest lamport."[18] One lamport is one billionth of $SOL. Paying a prioritization fee ensures faster execution, as the nodes prioritize transactions with higher fees.

When developers or users want the clusters to keep accounts and their data persistently in memory and not be lost, they must fund a time and space-based fee called rent.
"The Solana rent rate is set on a network-wide basis, primarily based on the set lamports per byte per year."[13] However, it is not a feasible model to theoretically pay an infinite amount of $SOL for renting space for an infinite amount of time. The developers solved this with mechanics called rent exemption. An account qualifies to be rent-exempt by having a balance equal to at least two years of rent. If an account fails to meet this condition, a garbage collector will clean up its data. The current rent rate is available in the rent sysvar. As of December 2022, the current rent rate is static at 0.00000348 $SOL per byte per year. As an example, a classic token account takes up 165 bytes. That means when opening a new token account, the user has to fund this account with at least 0.00203928 $SOL to be rent-exempt.

The Solana's "Sealevel" Runtime

Each instruction identifies a specific program and provides a selection of the transaction's accounts that need to be transferred to the program, along with a data byte array. The program deciphers the data array and interacts with the accounts outlined by the instructions. The program can either produce a successful result or generate an error code. In case of an error, the entire transaction is immediately deemed unsuccessful [19].

Solana's on-chain programs utilize the LLVM compiler infrastructure, which compiles them into an Executable and Linkable Format (ELF) containing a version of the Berkeley Packet Filter (BPF) bytecode. This approach allows developers to write programs in any language that can target LLVM's BPF backend. Solana currently supports Rust and C/C++, and ongoing research aims to add support for the Move programming language. Python can also be used for program development through the Seahorse framework. BPF provides efficient instructions that can be run in an interpreted virtual machine or as high-performance just-in-time compiled native instructions.

3.3 Use Cases and Adoption: Ethereum and Solana

Although Solana boasts high throughput and low latency, making it an attractive platform as of Q1 2023, it has struggled to draw a more extensive user base to its DeFi platforms. The network's impressive performance has facilitated the development of decentralized exchanges (DEXs) and advanced liquidity and lending pools with rapid rebalancing in just a matter of seconds. Nevertheless, users continue to prefer the stability and provenance of the Ethereum network, despite the high fees and extended transaction confirmation times.

Due to the high fees and slow transactions on Ethereum, developing highly interactive games on the platform becomes nearly infeasible. Consequently, Solana gains a competitive advantage in the realm of game development, offering seamless on-chain interaction capa-

bilities. As a result, Solana has firmly established itself as the premier choice for gaming applications and is poised to maintain this position for the foreseeable future.

One sector has successfully gained traction on both Ethereum and Solana – trading with non-fungible tokens (NFTs). Ethereum boasts a higher trading volume, which could be attributed to its wealthier user base that can afford the higher fees. On the other hand, Solana has more active wallets involved in NFT trading, as its lower fees create a more accessible entry point for users. Consequently, the adoption of NFTs can be considered relatively balanced between the two platforms.

Chapter 4

Emerging New Solutions

The Move programming language offers a fresh take on a different approach to smart contract programming. It focuses on addressing pitfalls developers may have encountered in the past. Alongside its rise in popularity, a few blockchain platforms with native support for Move have become prominent. The information in this chapter was sourced online from [22, 3, 9, 1, 4, 11, 20, 8, 5, 14, 27].

4.1 Move: A Programming Language for Blockchain Development

Move is an executable bytecode language explicitly designed for blockchain developers to implement custom transactions and smart contracts. While Rust inspired its basic concepts, Move's standout feature is its ability to define custom resource types. These resources have unique properties that will be discussed later. Typically, marking data as a resource is used for data representing something of value, such as the number of tokens a user has. Additionally, Move comes with a built-in formal verification checker called Move Prover.

First-Class Resources

In contrast to other programming languages, Move introduces a feature that allows developers to define their custom resource types. Developers can define any common program values as a resource. Nevertheless, doing so applies a whole suite of protection approaches to the variable. To highlight a few, as mentioned in the Move whitepaper [22].

> A resource can never be copied or implicitly discarded, only moved between program storage locations.

However, a resource cannot simply appear out of nowhere or disappear at any time, so we need to maintain a certain level of control over its creation and destruction. Move uses modules to ensure just that. In normal development in Move, a programmer declares a resource type (variable) and resource managing procedures (create, modify, and destroy) inside a module. A Move module looks and behaves similarly to smart contracts created in other blockchain languages. Likewise, a module can invoke procedures defined by other modules. The resource within a module can only be modified by the procedures specified in the module that defines that particular resource.

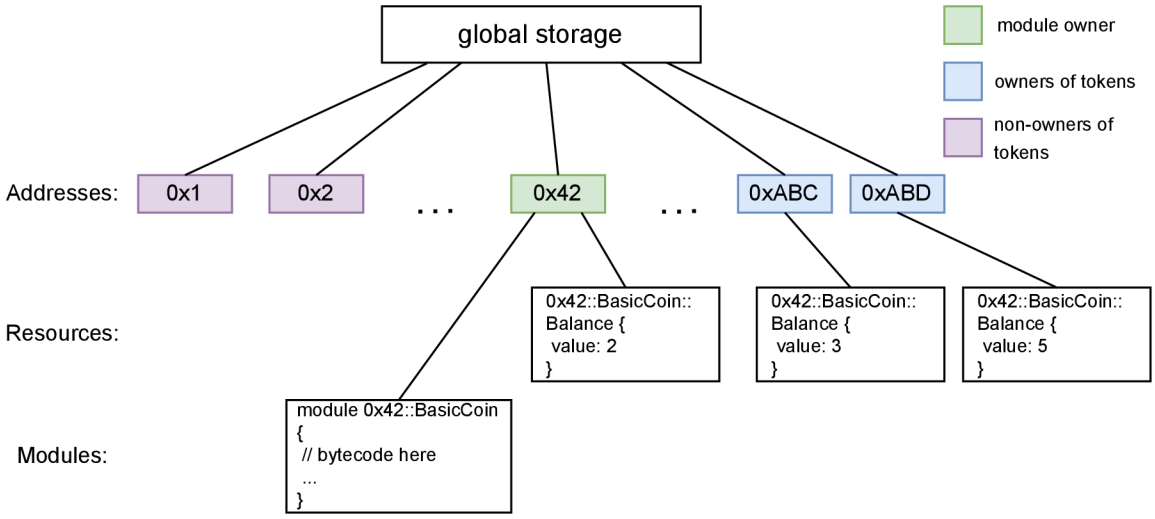


Figure 4.1: Move blockchain state[1].

Each address on the Move-native blockchain represents an account. In contrast to Ethereum, all accounts can store data under their addresses, either resources or deployed modules containing the executable bytecode, as depicted in Figure 4.1.

This approach separates the architectures mentioned earlier, with the Move solution offering better security. As the module does not control any data structures (resources), it cannot modify them, as they are stored under a user account. Modifying or withdrawing resources from the account is only possible with the user's (owner's) signer capability.

Code Security and Move Intermediate Representation

When developing in Move, programmers typically write code in Move Intermediate Representation (IR), which should possess the following qualities [22].

> Move IR is high-level enough to write human-readable code yet low-level enough to have a direct translation to Move bytecode.

However, the final executable Move source code format is a typed bytecode.

Checks by the bytecode verifier are performed at the bytecode level, examining the bytecode for resource, type, and memory safety directly on-chain. This memory safety includes preventing dangling references and memory leaks. Only after passing these checks does the bytecode interpreter execute the bytecode. The complete execution pipeline is depicted in Figure 4.2. Since it is computationally feasible to perform only some checks during every transaction execution, Move developers designed the language to support advanced off-chain static verification tools.

Formal Verification using Move Prover

Since data on blockchain systems often represent real-world value, platforms must be as secure as possible to prevent the potential theft of assets from their owners. Formal verification is an excellent tool for developers to verify that their program behaves as intended.

[1]Image adapted from https://github.com/move-language/move/tree/main/language/documentation/tutorial.

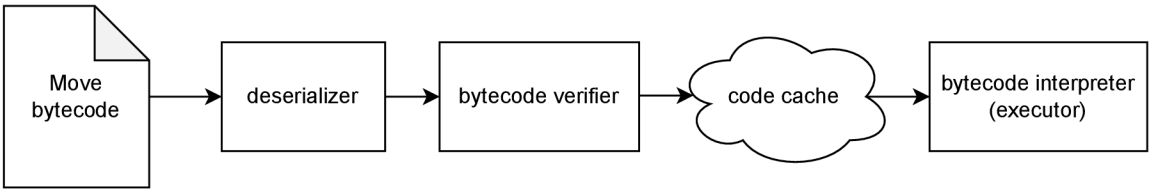


Figure 4.2: Move execution pipeline.

First, the developer specifies the desired behavior using a mathematical expression for formal verification. Then, the formal verification tool checks that the code behaves as specified. Unlike simple code testing, formal verification examines the code behavior under every possible scenario, providing greater security assurance than testing alone.

4.2 Move-based Platforms: Aptos and Sui

The inception of Move dates to 2018 when developers at Meta (formerly Facebook) sought a programming language to power Meta's project Diem (previously known as Libra). Unfortunately, Meta later canceled this project due to regulatory issues. However, the developers behind Move wanted to preserve their years of work. As a result, they left Facebook and formed two independent groups of former Meta employees to create their own Move-based blockchain platforms. This led to the creation of the Aptos and Sui blockchain platforms. The main differences between Aptos and Sui are stated in Table 4.1, with detailed explanation in the following sections.

Criterion	**Aptos**	**Sui**
Consensus Mechanism	Proof-of-Stake	Proof-of-Stake
Move Language Variant	Diem's Move	Sui Move
Storage Model	Account-centric	Object-centric
Transaction Capacity	Up to 160k TPS	10k to 300k TPS
Time to Finality	$<$ 1 second	$<$ 1 second
Module Deployment	Under account	As object
Resource Distribution	Two-step (offer and claim)	Unilateral transfer

Table 4.1: Comparison of key properties between Aptos and Sui.

Aptos

Aptos is a standalone layer-one blockchain with a Proof-of-Stake (PoS) consensus mechanism. With a mainnet launch on October 12th, Aptos became the first Move-native blockchain to launch a mainnet and the only one to do so in 2022. Aptos describes itself as a reliable, secure, scalable, and upgradeable blockchain. The developers designed Aptos with native support for Move, alongside the development of Move itself. In addition to smart contracts, the blockchain even uses Move internally for fast and secure transaction execution.

The blockchain achieves high throughput and low latency by parallelizing transaction execution as much as possible. The network accomplishes this through batch processing

and parallel transaction execution. During the transaction dissemination phase, each validator groups transactions into batches and combines them into blocks during consensus. Transactions without data resource conflicts can execute in parallel.

Aptos stores data on-chain using accounts, which consist of a set of values and key-value data structures. These data structures take the form of Binary Canonical Serialization format (BCS). Move modules are stored similarly but under a separate namespace.

A notable feature of Aptos is its ability to support private key rotation and native multi-signature capabilities. Aptos achieves this by providing on-chain mapping through an account lookup address. In Aptos, the account address shared with other users differs from the public key in a signature pair (public_key, private_key).

During the account creation process, the concatenation of all public keys (a single one or multiple in the case of multi-signature) is hashed using a cryptographic hash to form an authentication key. The public account address is then set to match the authentication key. However, the Aptos blockchain includes a function that allows users to update the authentication key associated with their account address at a later time.

These technological advancements significantly differentiate the Aptos blockchain from existing platforms, capturing the interest of venture capital investment firms and leading to a staggering $350 million funding raise [35]. These figures are considered an overwhelming success for a raise conducted in 2022.

Sui

Sui is a Move-native, permissionless, proof-of-stake, layer-one blockchain designed from the ground up to achieve a near-instant, high-throughput network. Unlike Aptos, which uses the Diem version of Move, Sui's developers took Move and made a few modifications, resulting in a version they call "Sui Move."

Resource Distribution

In the original Move (used by Aptos), transferring a resource to a user account is impossible without the user's consent. While this can be beneficial in preventing spam or scam NFTs from being sent to users' wallets across various blockchains, it poses a problem when distributing resources (e.g., NFTs) to a predetermined list of addresses. To address this, Sui implemented a unilateral resource transfer (similar to other blockchains) with the function `transfer(resource, recipient_addr)`. As a bonus, the execution of a function implementing multi-item distribution does not collide with others; therefore, the Sui runtime forwards this transaction via the "fast path" broadcast that does not need consensus, resulting in parallel execution.

However, this issue is not unsolvable. Aptos addresses this (in their framework) by splitting the resource transfer into two steps:

1. The distributing module makes a "transfer" offer to the user account:
   `token_transfers::offer(&module_sig, receiver_addr, token_id, 1);`

2. The offers will be displayed to the user in their wallet, allowing them to send a "claim" transaction to claim the resource into their account.

Sui Storage and Native Asset Ownership

Sui employs a distinct approach to native asset ownership and transfers compared to Aptos, which uses account-centric global storage. Sui does not utilize the built-in global storage in the core Move language. Instead, the platform has developed its own storage system called Sui Storage. This system is object-centric, with each address representing a globally unique object ID. Generally, this should be fine during development in Sui, as it is easy to detect if a generated address (ID) already exists on the blockchain.

This storage model simplifies object ownership and transfers. In Sui Move with Sui Storage, module functions (entry points) already take object references as input arguments. Therefore, this design eliminates the need for developers to use `borrow_global_mut` (Aptos) and incorporates checks to ensure that the transaction sender is the object owner into the Sui runtime.

One fascinating aspect I will explore later in this thesis is that with this change, modules are published into Sui Storage and not under an existing account (as in Aptos). Consequently, developers inadvertently avoid the module deployment challenges I experienced while working with Aptos. As mentioned, more on this topic will be discussed later.

Simplified Gas Consumption in Sui

Like Ethereum, Sui measured computation resource consumption in computation units for each instruction execution. However, in late October, Sui modified the gas fees for each instruction to a flat value. The creators behind Sui attributed this change to developer behavior, stating that overly fine-grained per-instruction metering encouraged unnecessary optimization (gas-golfing). The hope is that this decision will improve code readability and prevent unnecessary computation resource wastage in the future.

Chapter 5

Experiments with Current and New Platforms

The idea behind Move is promising, but a proper evaluation must be based on more than just theory. So instead, the focus is on using Move, Solidity, and Rust in practice to build a fully functional application called *The Deposit Box*. The development process will be done on Ethereum using Solidity, Solana using Rust, and Aptos using Move. This chapter will discuss the development experience and challenges faced and evaluate each platform based on execution cost, processing speed, and code readability. By the end of this chapter, readers will have a deeper understanding of the strengths and weaknesses of these platforms, as well as insights into their practical use. The information presented in this chapter, and used during development was sourced from [2, 11, 4, 34, 25, 23, 6, 9, 10, 30, 15, 33].

5.1 Implementation – *The Deposit Box.*

I needed to develop an application – an application that would put to the test several features commonly used by various smart contracts. Eventually, I have devised an idea for an application I call "The Deposit Box." I implemented this application in Solidity on Ethereum, Rust on Solana, and Move on Aptos. To simplify things, I will try to refer to the application backend, implemented as a smart contract, program or module, uniformly as a "program."

How the Application Works

Users will visit the application and deposit selected assets. In exchange, they will receive a new non-fungible token representing the ownership of the deposited assets, sometimes referred to as a token backed by those assets. Whoever holds the token can then exchange it back for the underlying assets. The token holder is free to transfer the token to any wallet they want. The described application use case is illustrated in Figure 5.1. This approach tests several aspects:

1. Handling of multiple assets
2. New token minting (emission)
3. Privileged recovery of assets from the program

If the application behaves as intended, the functionality can be expanded by adding the possibility of setting up a time-lock on the deposit. When depositing, the user is free to choose an optional lock time. If the user has done so, the token holder can withdraw the assets only after the specified time expires.

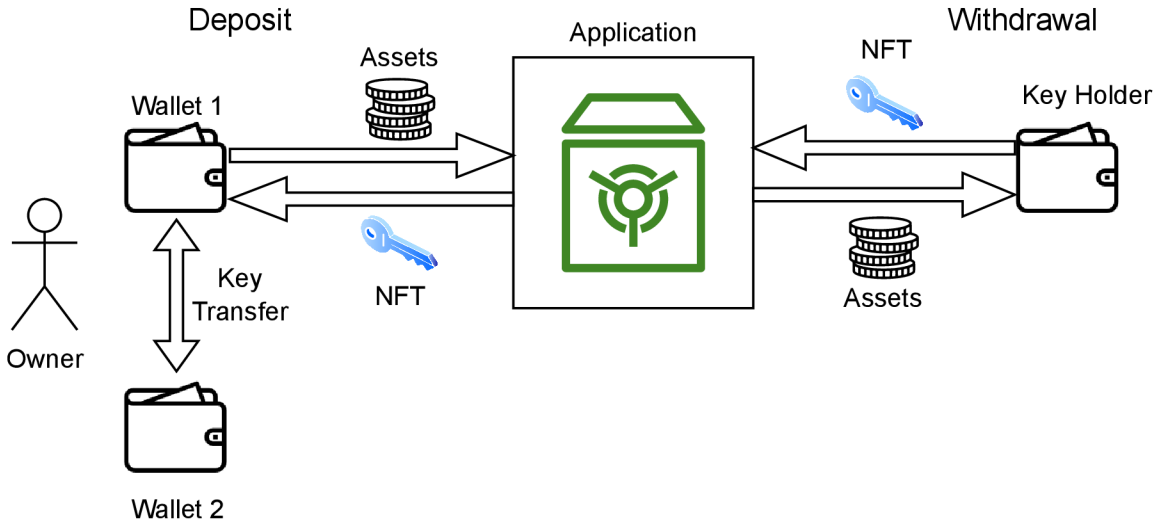


Figure 5.1: Visualization of the application behavior.

5.2 Evaluation Criteria

After considering various ideas, I have decided to settle on three criteria for testing the different implementations in each programming language and corresponding platform. I have kept it simple as these criteria will cover the fundamental questions:

- How much – objective comparison based on fees associated with transaction execution
- How fast – objective comparison based on network throughput and time-to-finality (TTF)
- How – subjective evaluation of code readability, available tooling and supporting resources

Execution Cost

In contrast to user experience, in a majority of cases, the primary concern of the developer is not the transaction execution fee but the program deployment cost. For a program to become usable, the developer must first upload it to the chain. A completed smart contract usually consists of many instructions processing somewhat complex logic. As the smart contract is submitted onto the network for it to be available for execution, the execution client must keep a copy of it in its memory. In blockchain systems, the most scarce resource is memory (storage) capacity. With many instructions, the executable bytecode becomes of non-negligible size. Therefore, the developer must pay a significant transaction fee (on contract deployment) on Ethereum or charge up an account with enough $SOL for the account to become rent-exempt on Solana.

Processing Speed

Currently, the choice of programming language is platform-dependent, as neither platform supports other programming languages. However, this might change as Solana announced that their developers are working on native support for Move on Solana. Therefore, the evaluation of this criterion will mainly reflect the platform's efficiency. However, the very nature of the programming language might impact the resulting processing speed, as particular actions may consist of different amounts of instructions, resulting in varying execution times. Nevertheless, this criterion is a vital part of the technology research process. Evaluation of processing speed will depend on time-to-finality and network throughput.

Code Readability

Code readability is the primary differentiation a programming language choice will make. It influences the security and adaptability of the code. If the code is easily readable and understandable, the probability that other developers and security researchers will spot any potential bugs and security holes within the source code before any attacker increases. Easy-to-read code is necessary for sophisticated static analysis. Similarly, new developers can use existing source code as a reference or learn from it.

5.3 The Development Experience – Ethereum, Solana, and Aptos

In this section, I will cover the development process, its nuances, the platform differences, and the design aspects I encountered while developing the application on each blockchain platform.

The Ethereum EVM and Solidity

Smart contract development was carried out in the Remix IDE[1]. The online in-browser Remix IDE is a comprehensive suite of tools available to developers working in Solidity. It contains a decent source code editor and a Solidity code compiler readily accessible via a keyboard shortcut with various compiler versions. With an injected wallet extension, developers can easily deploy the smart contract on-chain directly inside the IDE. As on-chain deployment can become quite inconvenient (due to online on-chain transactions taking time), developers can use the built-in virtual sandbox Remix VM, which mimics the real-world blockchain.

The Challenges Faced by Ethereum Developers with Testnets

Currently, two public Ethereum testnets are available for developers to test their code: the Sepolia network, explicitly recommended for smart contract testing, and the Goerli network. In order to test anything on either network, testnet tokens are required. However, obtaining testnet tokens from faucets can be challenging due to a shortage of tokens and mandatory anti-robot checkups.

For example, on Goerli, after being forced to create an Alchemy account, I received 0.2 goerliETH (the daily faucet allowance), which I depleted within an hour by sending

[1] Remix – Ethereum IDE available at https://remix.ethereum.org/

contract deployment and interaction transactions. I then switched to the Sepolia network, which had cheaper transaction fees and higher faucet allowances (up to 5 ETH/day).

Nevertheless, the struggle is significant enough that Ethereum core developers plan to launch a new testnet called Holli specifically designed to address testnet token distribution and struggling faucets [31].

ERC-20 Token Allowances and Approvals

To understand how token allowances work under the ERC-20 standard, we must first examine the ERC-20 token implementation itself. Each ERC-20 token is technically a smart contract that uses the ERC-20 standard library to inherit functionality. The two mapping functions are the most important in that inherited code:

```
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256)) private _allowances;
```

The initial function maps balances to their corresponding addresses, whereas the second function maintains allowances a user (address) has granted to other addresses to make transactions on their behalf. Whenever a transfer transaction occurs, these two mappings are updated accordingly.

The shared interface provides two functions to initiate token transfers from one address to another:

1. `transfer(recipient, amount)`
2. `transferFrom(sender, recipient, amount)`

In the first case of `transfer(recipient, amount)`, the function implementation expects to be called directly by the sender (user's wallet). Therefore, a simple transfer deducts the balance from the sender's address and adds the same amount to the recipient's address.

When a smart contract is required to respond to the token transfer, it is necessary to call the `transferFrom(sender, recipient, amount)` function inside the smart contract code to transfer the token and follow it with more code. However, in doing so, the smart contract effectively carries out the transfer on behalf of the user. For this to succeed, the contract must be explicitly authorized with the specific spending allowance.

It is unclear why it was designed like this in the first place, but essentially the user is forced to make two separate transactions. The first transaction grants the spending allowance to the specific smart contract, and only then can the second transaction interact with the custom smart contract. This transaction flow can be seen in Figure 5.2. Both transactions make data changes on the blockchain, meaning both must pay a transaction fee. This can become unnecessary and quite expensive, especially considering that transaction fees on the Ethereum blockchain are usually high.

Nevertheless, for the smart contract to use the `transferFrom()` function, it must first be an approved spender. There is a catch: the `approve(spender, amount)` function takes only two arguments – the spender and the amount. However, we must specify the owner who grants the approval, right? The implementation of this specific function solves that by appointing `msg.sender` as the owner. Nevertheless, if the function call is sent from the smart contract application, the `msg.sender` becomes the smart contract itself. As a result, the only consequence of this call is that the smart contract provides transfer approvals to itself. The smart contract is unable to handle the approval. Consequently,

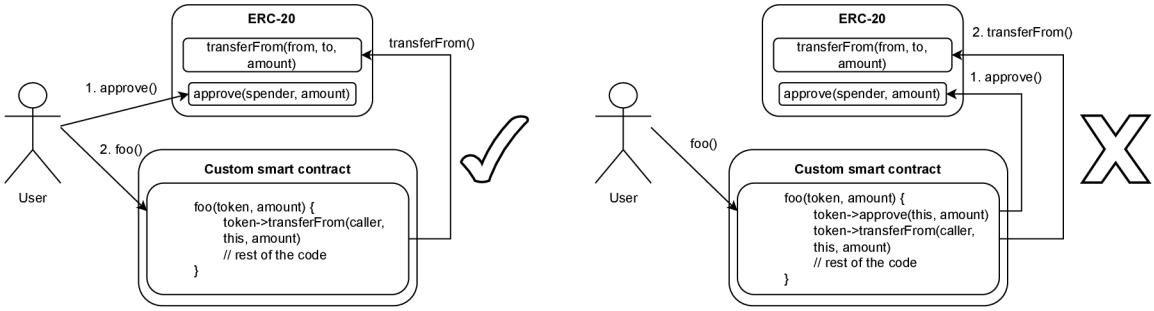


Figure 5.2: The correct and wrong transaction flow of ERC-20 approvals.

without extensions, the user is compelled to execute two separate transactions. However, a possible solution I encountered during my search was a project named Permit2.

Permit2[2] takes advantage of the EIP-2516 extension to the ERC-20 standard to solve the issue of recurring approval transactions for each smart contract. The user only needs to approve the Permit2 contract once, and the Permit2 contract will then serve as an intermediary for token transfers for other smart contracts using this system. When interacting with a smart contract application, the user first signs an off-chain (gasless) message confirming their intention to transfer tokens and then passes this signed Permit2 message as a parameter to the smart contract function containing the token transfer. Nevertheless, incorporating Permit2 seems rather complicated and is likely beyond this project's scope.

Addressing Reentrancy Vulnerabilities

On Ethereum, it is typical to provide a lot of the functionality via cross-program calls, and it is not abnormal that many of the functions are handling some ether at some point. However, calling an external contract requires performing an external call. The potential attacker can exploit this as they can force a contract to execute additional code, for example, via callback functions. Furthermore, like that, they can do recursive calls to the function itself. Nevertheless, we do not need to worry about this vulnerability as long we use the ReentracyGuard by OpenZeppelin.

Development on Solana

The development experience on the Solana network was quite distinct from EVM and had unique nuances. While some aspects might be better on Solana, it can be challenging for beginners. Additionally, it is worth noting that many articles, tutorials, and courses covering development on the Solana network contained deprecated code, which needed to be adjusted to work with up-to-date versions of libraries (e.g. `@solana/web3.js`) or the Anchor framework.

The Anchor Framework

I developed the Solana program using the Anchor framework [2]. Initially introduced by Armani Ferrante in 2021, Anchor is a development framework for Solana's Sealevel

[2]Learn more about Permit2 here: https://github.com/dragonfly-xyz/useful-solidity-patterns/tree/main/patterns/permit2

runtime. The development of the Anchor framework is an open-source project housed under the Coral company, with more than 80 community contributors at the time of writing this thesis. This framework aims to accelerate the development process of Solana programs by providing various boilerplate codes and macros for easy (de)serialization of accounts and instruction data. It also includes features like generating the program IDL specification (IDL = Interface Description Language), which enables developers to effortlessly use the IDL to run a TypeScript test suite for program testing. Furthermore, creating a nested React front-end application is also possible within the environment.

The Accounts and Addresses

In section 3.2, I outlined the primary distinctions of Solana's on-chain storage model. Now, I will add more detail: Data accounts on Solana have a fixed size (storage capacity) allocated during account creation, which cannot be increased later. However, due to Solana's network design, wherein the program must be provided with interacted accounts from the front end (i.e., transaction sender), the program does not keep any memory of all owned accounts. Therefore, an address must be supplied during program interaction to initialize a data storage account. When additional data needs to be added to an account created earlier, a new account must be established instead, leading to a cumulative number of accounts linked to the program. As I mentioned earlier, the front end must provide the addresses of the accounts to the program, meaning these addresses must be easily obtainable. This is achieved through a technique known as *program-derived addresses* (PDAs).

PDA (Program Derived Addresses)

A program-derived address is obtained through a derivation process using a set of *seeds*. These seeds are arrays of strings, keys, and other elements, converted into byte form, which are then passed together with the program ID through the SHA256 hashing algorithm. Unlike traditional keypairs, PDAs do not have a private key. However, there is approximately a 50 % chance that the hashing output may end up on the ED25519 elliptic curve, which is undesirable. Therefore, an additional value known as a *bump* is added to the hashing process to ensure an output outside this curve. This value starts at number 255 and is decremented each time the output lands on the curve until an address lying off the curve is obtained. In specific scenarios, there may be a need to enable users to create an unlimited number of accounts.

Transaction Complexity and Limitations

Transactions on the Solana network are collections of instructions, each consisting of:

- Invoked program identifier
- An array of accounts to read from or write to
- Data (as a byte array) specific to the RPC (remote procedure call)

With a program providing somewhat complex functionality, the array of accounts can become quite long. In the case of the first use-case of the program, where the user deposits an SPL token (the asset) into the escrow program and receive an honorary NFT back, there is a considerable number of accounts that our program will interact with. All actions in a simplified summary include:

1. Escrow (data) account creation (account storing the data about the escrow)
2. Token account of the escrowed token (owned by the escrow data program above)
3. NFT Mint account creation
4. NFT user token account creation

In addition to the list of required accounts for transaction execution, I had to add these accounts (read-only):

1. User account (our main wallet account)
2. Mint account (of the escrowed token)
3. User token account (holding token balance before escrow)
4. User escrow counter (escrow account PDA seed)
5. Token program
6. Associated token program
7. Rent program
8. System program

Unfortunately, transactions have a specified maximum size. With that many accounts specified, I simply ran out of available space within a single transaction. Therefore, there were only two solutions for this:

1. Split the interaction into two transactions
2. Implement versioned transactions

I chose the former due to the additional complexity of the implementation and the lack of available resources illustrating the usage of the new solution. Also, support for versioned transactions in the Anchor framework was added just recently on the 8th of March[3].

Versioned Transaction

Each Solana transaction, regardless of its content, is limited to a maximum of 1232 bytes. Solana transactions consist of an array of signatures (each signature takes 64 bytes) and the message. The message itself comprises a header (3 bytes), an array of account addresses (32 bytes per address), a recent block hash (32 bytes), and an array of instructions. In the best-case scenario, addressing up to 35 accounts in a single transaction is possible. However, this theory does not consider various instructions and the corresponding instruction data. Therefore, developers might encounter the transaction size limit when addressing 10 to 20 accounts.

Before late 2022, there was only one version of the transaction format, the legacy version, which had no solution to the size limitation issue. Recognizing the need for an improved system, Solana developers introduced a new transaction format that supports both

[3] Anchor v0.27.0 changelog:
https://github.com/coral-xyz/anchor/blob/master/CHANGELOG.md#0270---2023-03-08

the legacy version for backward compatibility and new versions. When writing this thesis, the latest transaction version is 0 ("zero"), which includes utilizing address lookup tables for more efficient account addressing. The exact differences are illustrated in Figure 5.3. The lookup tables are separate data accounts that allow storing up to 255 addresses.

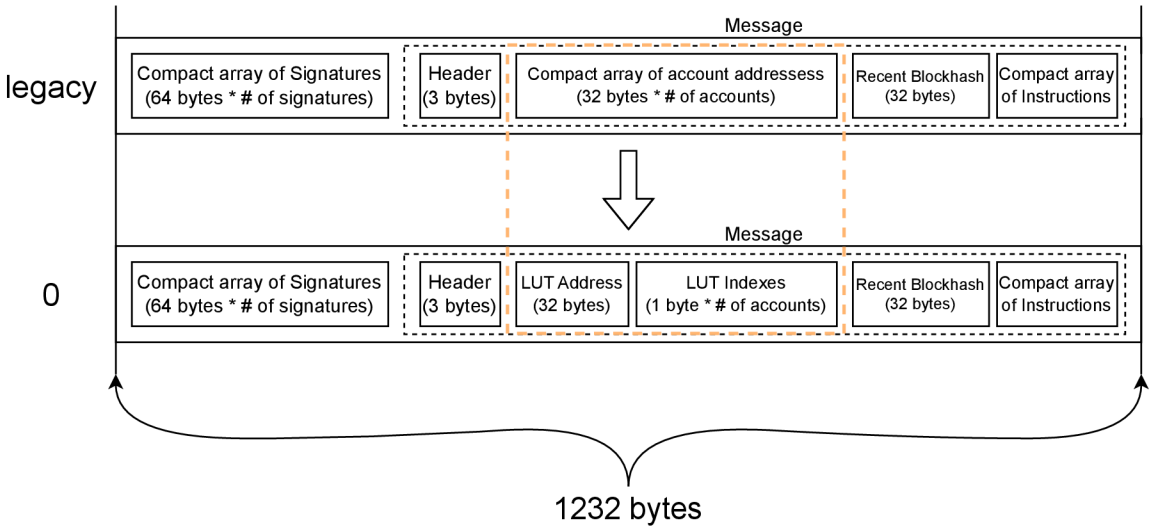


Figure 5.3: Depiction of differences between transaction versions of legacy and 0.

Developers must first initialize the lookup table, but then a single account address can be addressed with just a 1-byte unsigned integer index. For example, in a transaction with ten account addresses, the legacy transaction version would require 320 bytes, while in comparison, version 0 would require only 42 bytes (32 bytes LUT address + 10 indexes). A similar comparison with an example of 5 account addresses can be seen in Figure 5.4.

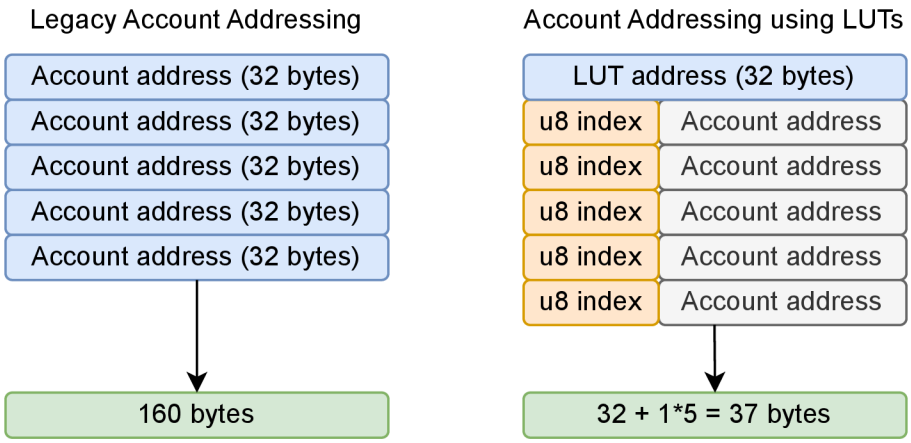


Figure 5.4: Comparison of space taken by addressing five accounts in versions legacy a 0[4].

[4]Image adapted from https://solanacookbook.com/guides/versioned-transactions.html#address-lookup-tables-lut.

Challenges and Insights from Aptos Blockchain Development

In the following subsections, I will delve into some aspects of developing on the Aptos blockchain, providing more detail on Move than in section 4.1 where I gave a high-level overview of Move. Most of these insights come from my hands-on experience with Aptos development.

The Hurdles of New and Seemingly Unfinished Platforms

Aptos was the last of the three blockchains I developed on, and as I progressed, I brought with me some programming practices common in blockchain development from my previous experiences. So, what challenges did I face? I wanted to retrieve some data from the blockchain, so it seemed natural to me to use and implement a view function. However, I could not find anything when I searched for references on how to do this. I found it strange at the time.

Nonetheless, after discovering a few view functions implemented in the Aptos Core module, I pieced it all together. Only a few weeks later, I learned that view functions were not initially available on Aptos until recently. The Aptos development team introduced view functions with the Aptos Move 1.2.4 update[5] in February 2023.

Strict Resource Management Policies in Move

Move enforces strict rules regarding permitted actions when manipulating values of a specific type. These permissions can be controlled using a typing feature called abilities. Without explicit assignment, for example, a value (struct) cannot be copied or dereferenced, dropped by going out of scope or stored inside other structs in the global storage. This limitation is essential for ensuring the safety and correctness of Move programs. However, in some cases, developers may need to manipulate specific values in ways that are not allowed by default. This is where abilities come in.

Abilities allow developers to assign specific permissions to values using keywords such as "copy," "drop," "store," and "key."

- Keyword *copy* – the value can be copied.
- Keyword *drop* – the value can be dropped.
- Keyword *store* – the value can be stored inside other structs in global storage.
- Keyword *key* – the type can be used as a key in global storage.

By assigning these abilities, developers can expand the range of actions allowed for specific values while maintaining the overall safety and correctness of the Move program.

For example, in the following code snippet, the "Box" struct has been assigned the "key," "store," "copy," and "drop" abilities using the "has" keyword.

```
struct Box has key, store, copy, drop {
    value: u64
}
```

This allows values of type "Box" to be used as keys in the global storage, stored inside other structures, copied, and dropped as needed.

[5]Move 1.2.4 release notes – https://github.com/aptos-labs/aptos-core/releases/tag/aptos-node-v1.2.4

References and Ownership

In common programming languages like C, C++, or Python, variables exist within their defined scope and all nested scopes. These variables can be passed as arguments to other scopes (such as functions) as copies or references. However, these variables and references remain available for the rest of the scope, which may sometimes lead to a phenomenon called a *dangling reference*. In this situation, the variable may be deallocated inside the external scope, causing the reference to point to a value (data) that is no longer in memory.

Move VM handles references and ownership similarly to Rust. Each variable has an associated owner scope. A variable can be defined within or passed into a scope as an argument. Creating a reference to the variable does not pass the ownership of the variable; instead, it only grants the right to read or write data to it. References must not outlive the variable's owner scope. When a variable is passed to a function, that function takes ownership of the variable. Like Rust, explicitly stating if a mutable reference is wanted promotes more robust code security. All of these rules are enforced through reference checking by the compiler. To obtain a reference to a resource, Move includes functions with keywords "borrow" and "mut", such as `borrow_global_mut` or `borrow_mut`. Additionally, Move contains mechanisms that aim to prevent developers from making mistakes while using references. One of these mechanisms is known as borrow checking. When a function attempts to obtain a mutable reference to a resource, it must be explicitly specified in the function declaration, as shown in the example below:

```
struct ModuleDatah has key {value: u64}
public entry fun foo(sender: &signer) acquires ModuleData {
    let _module_data = borrow_global_mut<ModuleData>(@module_addr);
}
```

In this example, I specified that the function *acquires* the resource `ModuleData`, which implies that a mutable reference to this resource is created within the code, making it highly likely that the resource data will be modified.

Generics

In order to enhance the security of the source code, it is preferable to reuse the same code rather than writing similar and redundant code for different types. In Move, both functions and structs can utilize generics. Generics can be employed by placing a placeholder "<T>" next to the name, where "T" represents *any type*. The type of "T" can be used like any other type within the body of the function or struct. Here is a simple example to illustrate this concept:

```
struct Data<T> {
    value: T
}
public fun save_data(value: u64): Data<u64> {
    Data<u64> {value}
}
```

In this example, the `save_data` function returns a new `Data` struct containing a value of type `u64`.

The automation with Resource Accounts

On Aptos, two categories of functions handle a significant amount of functionality. The first group comprises essential functions and types in the built-in standard library. The second category includes extended functionality and entire modules deployed at the `0x1` address directly on the blockchain. On Aptos, any module can use any function implemented in other modules deployed on the blockchain if marked public. The only requirement is that the developer knows the module's deployment address during their own module's development.

Many of these functions (from either category) take a signer as a parameter, which is necessary for their operation. In most cases, these functions modify or withdraw resources from the user account, the most privileged operation in the Move programming language.

If a module is deployed under an existing account, it cannot access the account signer capability. Consequently, if it is desired to withdraw coins of any type from the module and send them to the caller, limitations would arise.

```
coin::withdraw<CoinType>(&module_account, amount)
coin::deposit(sender_addr, coins);
```

The `withdraw` function requires `&module_account` (signer); however, the Aptos virtual machine only passes the sender (caller) as a signer to the function during execution. Consequently, the function does not have the `module_account` signer capability and cannot process a withdrawal.

Resource accounts can be utilized to achieve module self-sufficiency and autonomy. Like Solana, a resource account is published under an address derived from the deployer's address using *seeds*. Consequently, the resource account does not have an associated private key, ensuring that the published module remains immutable since no one has the right to update it. Despite the absence of a private key, the resource account still possesses a signer capability that the published module can utilize.

Creating a resource account and publishing a module (package) under it from within another module should be possible. However, due to the lack of available references on how to accomplish this, I opted to deploy my module using the Aptos CLI with the following command (memory mapping details omitted):

```
aptos move create-resource-account-and-publish-package --seed <seed>
```

This command creates a resource account and publishes a package using the specified seed. The complete instructions can be found in the application's source code. By following this approach, the module has been successfully made self-sufficient and autonomous, leveraging the capabilities of resource accounts on the Aptos blockchain.

Upon successful deployment of each module, the Move VM carries an automatic execution of a function named `init_module` if it is present in the bytecode. This function is not an entry function; therefore, it will be executed only once in the module's lifetime. With that, the Move VM automatically passes the resource account as a signer to this function. Furthermore, that behavior allows us to store the resource account signer capability that will be later used in each action that needs it, allowing us to automatize the module's code execution independent of the module storing account. The complete code needed to store the signer capability would look like this.

```
struct ModuleData has key {res_sig_cap: account::SignerCapability}
fun init_module(resource_account: &signer){
    let res_signer_cap = resource_account::retrieve_resource_account_cap(
    resource_account, @dev_address);
    move_to(resource_account, ModuleData {res_sig_cap: res_signer_cap});
}
```

The `@dev_address` is the original account that this resource account was eventually derived from using the seeds chosen earlier. The `@res_address` is the resource account address. This allows the resource account to be available in any entry function as needed.

```
    public entry fun foo(sender: &signer) acquires ModuleData {
        let module_data = borrow_global_mut<ModuleData>(@res_address);
        let resource_signer =
        account::create_signer_with_capability(&module_data.res_sig_cap);
        // resource_signer is now available
        // ...
    }
```

By following this approach, it is possible to efficiently store and retrieve the resource account signer capability, making the module self-sufficient and autonomous, leveraging the capabilities of resource accounts on the Aptos blockchain.

Developer Adoption and Scaling Challenges for New Blockchain Platforms

The process of adopting new blockchain platforms, and indeed any new technology, can often seem like a vicious circle. As more developers use a particular technology or platform, many resources such as examples, guides, and courses become available. However, the challenge is to convince developers to choose a specific platform over others. To achieve this, it is necessary to provide them with the necessary resources to get started.

It can be informative to compare the developer counts of different blockchain platforms. For example, Ethereum boasts an estimated developer count in the tens of thousands. With such a significant number of developers, there is a high likelihood that someone has already tackled a similar challenge to what others are trying to achieve. Additionally, there is a greater probability of finding experienced developers who can offer guidance and support.

On the other hand, Solana, with its recent growth, has a substantially lower developer count than Ethereum[6]. While the resource availability is reasonably good at this point, it is far from Ethereum's level. Additionally, the pace of core upgrades for the Solana network is significantly higher, leading to code-breaking changes and deprecated old methods over time.

In the case of Aptos, the developer count is more likely to be in the hundreds rather than thousands. Consequently, if resource availability is linked to the total developer count, fewer resources are undoubtedly available. This emphasizes the challenge of attracting developers to new platforms and highlights the importance of providing comprehensive resources to kickstart developer adoption and, ultimately, the platform's success.

[6]Reference – https://twitter.com/solana/status/1615398642717687842

Chapter 6

Comparison and Evaluation of Move, Solidity, and Rust Implementations

In this chapter, I will analyze and compare the implementation of the application in three different languages: Move, Solidity, and Rust. Therefore, I will summarize and assess the role of languages in blockchain development by evaluating code complexity, readability, transaction fees, development time, and developer experience. This comparison aims to provide insights into the strengths and weaknesses of each language. To present correct information in this Chapter, I used the following sources [28, 4, 34, 5, 27].

6.1 Comparing Code Complexity and Readability

In this subsection, I compare the code complexity and readability of the Move, Solidity, and Rust implementations by looking at code length, naming conventions, and error handling to show how these stack against each other. Code complexity and readability play a crucial role in software development, as they directly impact the ease of understanding, maintainability, and collaboration among developers.

Code Length

Code length is not a perfect indicator, but shorter code is often easier to understand and maintain. However, it is crucial to maintain readability even if it comes at the expense of more extended code. The results from the Succinct Code Counter[1] can be found in Table 6.1.

Language	Lines	Code
Solidity	321	196
Rust	437	299
Move	247	154

Table 6.1: Comparison of lines of code of implementation in each language.

[1] Learn more about SCC – https://github.com/boyter/scc

These results show the differences in code length among Rust, Move, and Solidity implementations.

Naming Conventions

Solidity uses CamelCase, while both Rust and Move employ snake_case. Solidity also follows specific guidelines regarding underscores. For example, private variables begin with an underscore (e.g., `uint256 private _myPrivateVariable`), and private or internal functions start with an underscore. Move function names are in lower snake case, and constant names are in upper snake case. Unused variables must start with an underscore, or the compiler will not compile the source code. Rust generally uses the snake case convention, but Anchor's context and structs adopt CamelCase.

Error Handling

Error handling is a crucial aspect of any programming language, as it allows developers to manage unexpected situations and provide meaningful feedback to users. In this section, I will compare the error-handling mechanisms in Move, Solidity, and Rust.

Solidity

Solidity handles errors using a combination of require, assert, and revert functions. These functions allow developers to validate conditions and revert transactions if they fail. The require function is used for input validation and throws an error message when a condition is not met, while the assert function is used to check internal errors and consumes all the remaining gas when a condition fails. Revert functions can be used to revert state changes in case of errors.

Move

In addition to the built-in "assert!" macro, Move also provides an "abort" operation that can halt the execution of a script or function when an error occurs. The abort operation takes an error code as an argument, which can be later used for debugging purposes. Example of using "assert!" macro:

```
/// Item does not exist
const Item_NOT_FOUND: u64 = 1;
assert!(search_result, error::permission_denied(ITEM_NOT_FOUND));
```

Additionally, the wallet displays the comment above the constant definition as an error message to the user.

Rust

In Rust, the default behavior on Solana is to print "panic!" or "assert!" outcomes to the program logs. Additionally, developers commonly utilize the Anchor framework's custom error handling macro "err!". It can be used in the following manner:

```
err!(MyError::TransferFailed);

#[error_code]
pub enum MyError {
    #[msg("TransferFailer")]
    TransferFailed
}
```

6.2 Transaction Fees (Gas and Rent)

I deployed the created application on Goerli, Sepolia, Arbitrum, Solana, and Aptos to highlight the differences in transaction fees across the compared blockchains. I recorded the deployment fees, alongside the application deposit and withdrawal fees, in the following Table 6.2.

Blockchain	**Deployment**	**Deposit**	**Withdraw**	TTF	Network TPS
Goerli	41.29 ($76626)	2.116 ($3934)	0.385 ($716)	12 secs	≈30 TPS
Sepolia	0.0112 ($21,36)	0.0001 ($1.00)	0.0001 ($0,20)	12 secs	≈15 TPS
Arbitrum	0.0134 ($25,69)	0.0003 ($0.51)	0.0002 ($0.37)	1 sec	Up to 4.5k TPS
Solana	6.4297 ($136,8)	0.0108 ($0,23)	-0.008 ($0,17)	0.4 sec	Up to 100k TPS
Aptos	0.00869 ($0,07)	0.0022 ($0,02)	0.000019 ($0)	0.4 sec	Up to 160k TPS

Table 6.2: Comparison of transaction costs, time-to-finality (TTF), and network throughput. The values are stated in the blockchain native currency with conversion to dollars at rates on May 10th.

The application was deployed on the following testnets and mainnets, with the respective application addresses.

- Goerli (Ethereum testnet) – `0x8Cc21861f2a5238F4A4FEc636621C53a8C01Cac0`
- Sepolia (Ethereum testnet) – `0x244416d09b1AeaFdf6fD7ceB6bc000b84DdAAF70`
- Arbitrum One mainnet – `0x244416d09b1AeaFdf6fD7ceB6bc000b84DdAAF70`
- Solana mainnet – `SAFEEinTUX1Eqkx4nCMit5qvKzAPkKQW8XM53QNHsZZ`
- Aptos mainnet – `0x2530b03fb0ce8e58b824f47f63124c9f87ec861bad0f7611a8699b7de0b721f9`

Hence, verifying the exact values presented in Table 6.2 using blockchain-specific explorers is possible. For Goerli, Sepolia, and Arbitrum One, I verified the contract code on *Etherscan* and *Arbiscan*, respectively. Consequently, these explorers provide public interfaces for these contracts.

Fee Differences between Solana and Other Blockchains

As discussed in chapter 3, a considerable difference exists between fees on Solana and those on EVM-compatible blockchains. On Solana, rent represents the most significant portion of any noticeable transaction fee. However, closing the accounts and reclaiming the $SOL

required for rent exemption is typically possible. As depicted in Table 6.2, the deployment cost on Solana might seem substantial, but unlike on Ethereum, developers can recover most of the $SOL by closing the account if they delete the program. This principle applies to deposit and withdrawal fees as well.

Upon depositing to the program, the user effectively opens an escrow data account and a token account for the escrow program, depositing enough $SOL to attain rent-exempt status. When withdrawing from the program, these accounts are closed, and the user receives a refund of the deposited rent. The only non-refundable part is the rent-fee deposit needed to initialize the *escrow counter*, amounting to just a few cents in value at current prices. Therefore, I deemed it unnecessary to include this in my application.

6.3 Development Time

To objectively assess or compare development time is almost impossible. However, in this section, I will discuss some factors that influence development time and share my experiences working with Solidity, Rust, and Move.

Before this thesis, I had no experience with Solidity, Rust, or Move, but I was proficient in C/C++ and Python and familiar with JavaScript. As a result, I found Solidity, which resembles C++, easier to adapt to than Rust or Move. Thankfully, Rust and Move share similarities, as Move is based on Rust.

Tooling and Resources

Navigating Solidity was relatively straightforward due to the great quantity of documentation from various sources. The Remix IDE is an exceptional tool for starting with Solidity development, as it eliminates the need to install utilities, tools, or compilers. Additionally, the IDE allows for quick contract deployment to real or virtual blockchains and provides a simple interface for interacting with deployed contracts.

The Anchor framework is a game-changer for Solana, as it simplifies development by automating tedious tasks and enhancing security. It handles (de)serialization processes and implements measures to prevent security issues. However, finding resources for building cross-program invocation transactions took much work.

While the Aptos blockchain provides some useful on-chain modules under the address `0x1`, the developer resources explaining its use could be improved as they should be extended and revised.

In terms of familiarizing myself with the languages and platforms, I found Solidity the easiest and quickest to learn. For example, the development of the Solana program took roughly twice, almost triple the time it took to develop the Ethereum contract, with the Aptos module falling somewhere in between. However, it is challenging to predict the difficulty and time required for future development now that I am familiar with each language.

Debugging and Testing

Debugging the Solidity code was quick and easy using the Remix IDE's interface. However, the Solidity compiler offered little guidance regarding code functionality.

In contrast, debugging the Rust program on Solana was more challenging. I had to build my front end, implement error handling, and troubleshoot issues independently.

Furthermore, the feedback from the explorer was limited to messages like "Transaction failed" or, from the Anchor framework, "constraint violation". I often had to return to code instrumentation and go through the code repeatedly.

Similar to Solana, I had to build my front end for Aptos. However, the Petra Aptos wallet offered a more informative error message directly in the wallet extension, which allowed me to identify and fix issues in the Move code more quickly than with Rust on Solana.

By analyzing the development time, tooling, and resources for Solidity, Rust, and Move, insights can be obtained into each language's learning curve and ease of use. This facilitates informed decision-making when selecting a language for blockchain projects.

Becoming a Full-Stack Blockchain Developer

Developing smart contracts, programs, or modules (the back end) for blockchain applications is challenging. While writing a back-end that compiles might seem straightforward, deploying the back end to the network and sending actual transactions is essential to ensure everything functions as intended. Consequently, a blockchain developer often takes on the full-stack developer role.

The primary domain for user interaction is a wallet, typically in the form of a web browser extension. To interact with the program using your wallet, you need a front end that can connect to your wallet extension and the blockchain, enabling effective communication with the back end.

Fortunately for Ethereum developers, tools that simplify their work are available, thanks to Ethereum's long-standing presence in the industry. Two such examples are the Remix IDE and Etherscan. However, similar tools still need to be created for developers working on Solana or Aptos. As a result, they must build their front end to test and complete their back-end development successfully.

This process includes integrating front-end and back-end communication, as the front end requires data to accurately construct a proper transaction. As a full-stack blockchain developer, one must be adept at handling both aspects of the development process.

6.4 Developer Experience and Available Tooling

This subsection will provide an overview of the developer experience and the tooling available for Ethereum, Solana, and Aptos, which can significantly impact the ease and efficiency of working with these blockchains.

1. **Ethereum**: Ethereum has a rich ecosystem of developer tools, including IDEs like Remix, testing frameworks like Truffle, and numerous libraries and SDKs. Various resources are available for learning and troubleshooting, such as documentation, tutorials, and community support via forums and chat channels. This comprehensive set of tools and resources has contributed significantly to Ethereum's popularity among developers.

2. **Solana**: While Solana's developer ecosystem is not as mature as Ethereum's, it has been growing tremendously lately. Tools such as the Anchor framework simplify program development. Additionally, Solana has been improving its documentation and supporting developers that create tutorials and sample projects to help new developers get started.

3. **Aptos**: As a newer platform, Aptos has fewer tools and resources for developers. To attract more developers, Aptos must focus on creating comprehensive documentation, expanding its developer toolset, and building a supportive community where developers can exchange knowledge and experiences.

By comparing the developer experience and available tooling across these three blockchains, we can better understand the challenges that newer platforms face in attracting and retaining developers and the importance of a robust ecosystem for driving the growth and success of a blockchain platform.

6.5 Chapter Conclusion

In conclusion, by examining factors such as code complexity, readability, transaction fees, development time, and developer experience, we gain valuable insights into the strengths and weaknesses of each language.

Being the most mature language with the most extensive ecosystem, Solidity offers the advantages of easy adoption, extensive tooling, and a large community. Before introducing the Anchor framework, developing in pure Rust on Solana was quite challenging. Thankfully, the Anchor framework has emerged, simplifying program development and enhancing security. However, working with Rust and the Anchor framework has a steeper learning curve than other languages. As a newer language, Move needs more resources and tooling but benefits from its close relationship with Rust and the increasing interest in the Aptos blockchain.

Ultimately, the choice of language and platform for blockchain development may not always be optional. For example, developers might lean towards Solana or Aptos for their superior transaction throughput and low latency. However, understanding the pros and cons of each language is still crucial for making informed decisions when developing projects on these platforms. In addition, the project's specific requirements and the developer's familiarity and comfort with the language should also be considered in this decision-making process.

Chapter 7

Conclusion

The primary objective of this thesis is to explore various blockchain platforms and the Move programming language, focusing on gaining a deeper understanding of the advantages and challenges developers encounter when working with these technologies. In this study, I have clarified the fundamentals of blockchain technology and examined the unique features of Ethereum, Solana, and Move-native platforms such as Aptos and Sui.

Additionally, I have conducted an in-depth analysis of the Move programming language, concentrating on its distinctive characteristics. Finally, I have created and executed a testing scenario using each language, evaluating code complexity, transaction fees, development experience, and available tooling.

My work has revealed that the Move programming language has several key strengths that distinguish it from other languages in the blockchain domain:

1. Move enables any value to be designated as a resource, providing a comprehensive set of protections for the variable.

2. It enforces strict resource management policies, ensuring the secure handling of these assets and minimizing the risk of accidental loss or duplication.

3. Move supports generics, facilitating the development of reusable and adaptable code while preserving type safety.

4. Move organizes the blockchain state by storing resources under individual accounts, enhancing security, and providing a solid foundation for developing safe and reliable modules.

These features have contributed to Move's growing adoption and interest within the blockchain community.

My investigation has shown that the future of Move is promising, with a growing number of platforms adopting it and existing platforms integrating support for it. For instance, Sui is launching its mainnet, and Solana is working on supporting Move programs. These developments reflect the increasing interest in Move and its potential for widespread use.

My work has also demonstrated that developing smart contracts, programs, or modules on any blockchain platform is a complex process that requires source code writing, deployment, and comprehensive testing. In addition, working with new technologies is often challenging due to limited resources for learning and adaptation of the technology.

In the future, I would like to continue exploring the practical applications of Move. It would also be worthwhile to investigate how other blockchain platforms can benefit from integrating Move or similar languages, expanding the scope of blockchain development or cross-platform implementation.

In summary, this thesis provides an overview of blockchain platforms and the Move programming language, offering valuable insights for developers and promoting a better understanding of the technologies involved. As blockchain continues to evolve, it is critical to remain informed and adaptable.

Bibliography

[1] *Accounts in Aptos* [online]. Martian Wallet, 2022 [cit. 2023-01-07]. Available at: https://medium.com/@martian-wallet/accounts-in-aptos-1ecc3f0b1213.

[2] *Anchor – introduction* [online]. Coral, 2022 [cit. 2023-03-14]. Available at: https://www.anchor-lang.com/.

[3] The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. [online]. Aptos Foundation. August 2022, [cit. 2022-12-27]. Available at: https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf.

[4] *Aptos Developer Documentation* [online]. Aptos Foundation, 2022 [cit. 2023-01-07]. Available at: https://aptos.dev/.

[5] Aptos: Solving the Layer 1 blockchain trilemma. [online]. State io. July 2022, [cit. 2022-12-27]. Available at: https://medium.com/@state_xyz/aptos-a-formidable-layer-1-addressing-the-blockchain-trilemma-398ff9be62d7.

[6] *Cooking with Solana | Solana Cookbook* [online]. Solana Developers, 2022 [cit. 2023-02-28]. Available at: https://solanacookbook.com/.

[7] *Ethereum Development Documentation* [online]. 2022 [cit. 2023-01-07]. Available at: https://ethereum.org/en/developers/docs/.

[8] *How Sui Move differs from Core Move* [online]. Sui Foundation, 2022 [cit. 2023-04-23]. Available at: https://docs.sui.io/devnet/learn/sui-move-diffs.

[9] *An Introduction to Move* [online]. CertiK, November 2022 [cit. 2023-01-07]. Available at: https://www.certik.com/resources/blog/3o4Cg1cjQH4IwA88aT8OwT-an-introduction-to-move.

[10] *Introduction to Smart Contracts* [online]. 2022 [cit. 2023-04-22]. Available at: https://docs.soliditylang.org/en/latest/index.html.

[11] *Learn about Sui | Sui Docs* [online]. Sui Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.sui.io/learn.

[12] *Overview | Solana Docs* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/developing/on-chain-programs/overview#berkeley-packet-filter-bpf.

[13] *Rent | Solana Docs* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/implemented-proposals/rent.

[14] Securing Move. [online]. Aptos Labs. December 2022, [cit. 2022-04-13]. Available at: https://medium.com/aptoslabs/securing-move-f81099f5e08c.

[15] *Solana | Developers: Resources and Information for Building on Solana* [online]. Solana foundation, 2022 [cit. 2023-03-23]. Available at: https://solana.com/developers.

[16] *A Solana Cluster | Solana Docs* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/cluster/overview.

[17] *Solana Documentation* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/.

[18] *Transaction Fees | Solana Docs* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/transaction_fees.

[19] *Transactions | Solana Docs* [online]. Solana Foundation, 2022 [cit. 2023-01-07]. Available at: https://docs.solana.com/developing/programming-model/transactions#instructions.

[20] Why We Created Sui Move. [online]. Mysten Labs. July 2022, [cit. 2022-12-17]. Available at: https://medium.com/mysten-labs/why-we-created-sui-move-6a234656c36b.

[21] *Merkle Patricia Trie* [online]. ethereum.org, 2023 [cit. 2023-02-13]. Available at: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[22] Blackshear, S., Cheng, E., Dill, D. L., Gao, V., Maurer, B. et al. Move: A Language With Programmable Resources. [online]. May 2020, [cit. 2022-12-23]. Available at: https://github.com/diem/diem/blob/main/developers.diem.com/static/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf.

[23] Dabit, N. *The Complete Guide to Full Stack Solana Development with React, Anchor, Rust, and Phantom* [online]. DEV Community, 2022 [cit. 2023-03-24]. Available at: https://dev.to/edge-and-node/the-complete-guide-to-full-stack-solana-development-with-react-anchor-rust-and-phantom-3291.

[24] Dan, D. *An Introduction to the Solana Account Model* [online]. QuikNode, 2022 [cit. 2023-01-07]. Available at: https://www.quicknode.com/guides/solana-development/an-introduction-to-the-solana-account-model.

[25] Hutchinson, K. and Barrios, J. *Forked One | Solana Blockchain Development* [online]. Forked One, 2022 [cit. 2023-04-04]. Available at: https://adept.at/forked/solana-blockchain-development.

[26] Kolakowski, M. Facebook (FB) Rebrands Itself as Meta. [online]. Investopedia. October 2021, [cit. 2022-12-17]. Available at: https://www.investopedia.com/facebook-fb-rebrands-itself-as-meta-5207628.

[27] Lobo, G. Diving into the Aptos Network. [online]. The Tie Research. August 2022, [cit. 2022-03-29]. Available at: https://research.thetie.io/aptos-network-deep-dive/.

[28] McKie, S. Solidity Learning: Revert(), Assert(), and Require() in Solidity, and the New REVERT Opcode in the EVM. [online]. Medium. September 2017, [cit. 2022-05-02]. Available at: https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e.

[29] Michaelson, R. Tales from the crypto: lira crisis fuels Bitcoin boom in Turkey. [online]. Guardian News & Media. January 2022, [cit. 2022-12-17]. Available at: https://www.theguardian.com/business/2022/jan/21/tales-from-the-crypto-lira-crisis-fuels-bitcoin-boom-in-turkey.

[30] Milano, A. *How to Use Account Constraints in Your Solana Anchor Program* [online]. QuikNode, 2023 [cit. 2023-03-08]. Available at: https://www.quicknode.com/guides/solana-development/anchor/how-to-use-constraints-in-anchor/.

[31] Muriuki, L. Ethereum plans to launch Holli Testnet. [online]. Cryptopolitan. February 2023, [cit. 2022-03-05]. Available at: https://www.cryptopolitan.com/ethereum-plans-to-launch-holli-testnet/.

[32] Nakamoto, S. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. Dec 2008 [cit. 2022-12-15]. Available at: https://bitcoin.org/bitcoin.pdf.

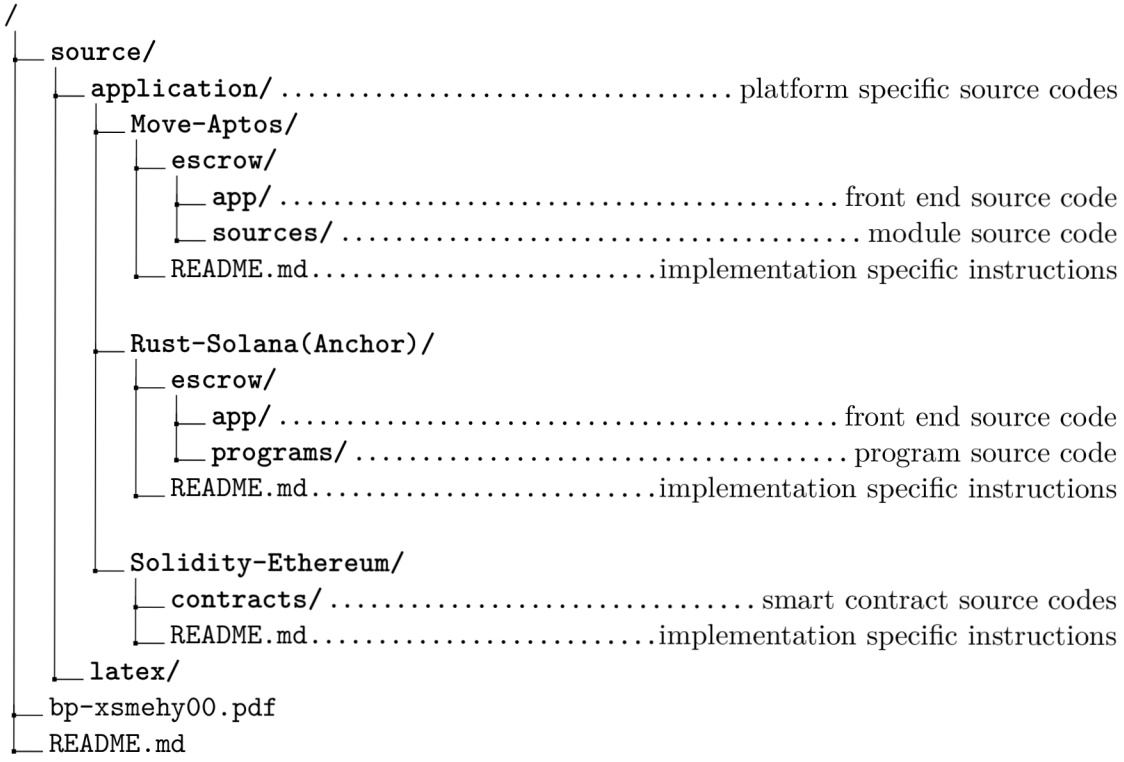
[33] Pacheco, J. *SolDev – Solana Development Course* [online]. Ironforge, 2022 [cit. 2023-03-27]. Available at: https://www.soldev.app/course.

[34] Shamanaev, D. *Getting Started – The Move Book* [online]. 2022 [cit. 2023-03-28]. Available at: https://move-book.com/introduction/getting-started.html.

[35] Stankovic, S. Layer 1 Blockchain Aptos Raises $150M From FTX, Jump Crypto. [online]. Crypto Briefing. July 2022, [cit. 2022-12-17]. Available at: https://cryptobriefing.com/layer-1-blockchain-aptos-raises-150m-from-ftx-jump-crypto/.

Appendix A

Storage medium



Directory `application/` contains three subfolders for the three implementations – in Move on Aptos, in Rust on Solana, and in Solidity on Ethereum. Each subfolder for a specific platform contains platform-specific source codes and `README.md` containing all instructions regarding usage and a brief explanation of the implementation.

Directory `latex/` contains LaTeX source files and images used in the thesis.

File `bp-xsmehy00.pdf` is a PDF file containing the final version of the thesis text.