FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Developer tooling for Solana
Student:	Bc. Lukáš Kozák
Supervisor:	Ing. Josef Gattermayer, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2022/2023

Instructions

Solana is a blockchain (or more precisely a global state machine) that excels in speed and scalability over other blockchains that allow to run programs such as Ethereum. But the ecosystem is very new and lacks some standard development tooling.

Commands:
- Analyse the Solana blockchain and its ecosystem.
- Analyse Trdelnik, the Solana Rust client.
- Propose a new functionality for Trdelnik (after discussion with the supervisor).
- Implement given functionality.
- Test correctness of your implementation.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 10 December 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Developer tooling for Solana

Bc. Lukáš Kozák

Department of Computer Systems
Supervisor: Ing. Josef Gattermayer, Ph.D.

April 26, 2022

Acknowledgements

I would like to thank Ackee Blockchain for providing guidance early with the Rust and Solana development, especially Tibor Tribus, the Solana Tech Lead of Ackee Blockchain, and Vladimír Marcin, a Solana auditor in Ackee Blockchain. I would also like to express my gratitude to my supervisor, Ing. Josef Gattermayer, Ph.D., who is also the CEO of Ackee Blockchain, for leading my thesis and giving me a chance to work on an open-source project with a practical impact.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 26, 2022 . . . .

Citation of this thesis

Kozák, Lukáš. *Developer tooling for Solana.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce představuje čtenáři Solana blockchain. Slouží jako vstupní bod pro nové Solana vývojáře nebo blockchainové nadšence chtějící se dozvědět o Solaně, jelikož práce detailně vysvětluje veškeré klíčové koncepty a její unikátní programovací model. Práce také ukazuje některé z nejdůležitějších projektů Solana ekosystému, které jsou dnes dostupné. Zbytek práce je zaměřen na vývojářské nástroje a Trdelnik, nový Rust testovací framework v raném vývoji pro Solana programy. Součástí práce je navržení nové funkcionality pro Trdelnik, samotná implementace, otestování a diskuze budoucí práce na projektu.

Klíčová slova Blockchain, Solana, Trdelnik, Explorer, Solana programy, Chytré kontrakty, Rust programovací jazyk, Vývojářské nástroje pro Solanu

Abstract

This thesis introduces the reader to the Solana blockchain. It serves as an entry point for new Solana developers or blockchain enthusiasts wishing to learn about Solana. The thesis thoroughly explains all core concepts and its unique programming model. It also showcases some of Solana's most crucial ecosystem projects today. The rest of the thesis is focused on the developer tooling and Trdelnik, the new Rust Testing Framework for Solana programs. As a part of the thesis, new functionality for Trdelnik is proposed, implemented, tested, and future work on the project is discussed.

Keywords Blockchain, Solana, Trdelnik, Explorer, Solana programs, Smart contracts, Rust programming language, Developer tooling for Solana

Contents

List of Figures

List of Tables

List of Source Codes

Introduction

Scalability

One of the most discussed scientific topics in the cryptocurrency and base layer blockchain area during the last few years has been something called *blockchain* or *scalability trilemma* [1]. It is a term coined in 2017 by Vitalik Buterin, the co-founder of Ethereum[1], and describes three main features of blockchains – *decentralization*, *scalability* and *security* – of which one must be sacrificed in order to make room for the other two.

Decentralization means the network is not operated by a single entity but by a group of nodes that can join in a permissionless manner. Scalability indicates the network can support an ever-growing increase in transaction demand without major changes or a slowdown. And the last of the three features, security, conveys it takes enormous resources to censor, double-spent, or in any other way attack the network. Thus, security in the blockchain is a kind of similar concept to computational infeasibility known from cryptography.

Several so-called "Ethereum killers" have appeared since the definition of the term in 2017, claiming to have solved the said scalability trilemma successfully. Notable mentions are *Avalanche*[2], *Cardano*[3], or *Solana*[4]. All with their specific approaches to tackle the problem – be it sharding, novel consensus mechanisms, or optimizing and re-engineering the way blockchain network nodes can communicate and execute transactions.

Whether they keep up to their promises and the scalability trilemma is really solved is yet left to be seen. Even though the mentioned blockchains have already been publicly launched and are in use today, they are still in heavy development, and often new challenges arise unexpectedly.

[1]https://ethereum.org
[2]https://avax.network
[3]https://cardano.org
[4]https://solana.com

Goal of this Thesis

This thesis will be primarily focused on the new Solana blockchain with comparisons with Ethereum. It is a semi-analytical and semi-implementation work. Therefore it does not solely focus on the implementation part but tries to understand the underlying technology. This is especially important as there is little to no officially-provided correct information, and currently, it is not easy to understand Solana from the ground up without digging into the source code or personally asking the developers how some of the specifics work. Therefore this thesis should be a good starting point to start learning about Solana.

Now for the implementation part – since Solana is a relatively new blockchain, it lacks most of the standard development tooling known from Ethereum, which is still the go-to standard. In fact, most other blockchains copy its execution model and *EVM* (Ethereum Virtual Machine) as the execution environment. Therefore the tooling can be shared and reused.

The practical part of the thesis consists of implementing a new feature for *Trdelnik*[5], which is a new open-source testing framework for Solana programs based on Anchor[6], which is yet a new development framework that helps to simplify the process of building new and secure Solana on-chain programs. Because writing raw Solana programs is a tedious and error-prone process, Anchor, together with Trdelnik, aim to fill most of the gaps in the current development ecosystem of Solana and bring better developer experience to everyone.

Thesis Structure

The Chapter 1, “Analysis”, introduces the reader to the Solana blockchain. Its core concepts and programming model are thoroughly explained. A sample of the ecosystem projects is showcased so that a reader can get an idea of how the blockchain is used in practice. Then Trdelnik, the Rust Testing Framework for Solana programs, is analyzed.

The Chapter 2, “Function Proposal for Trdelnik”, proposes a new feature, researches it and examines the functional and non-functional requirements. Use cases for the said feature are presented in this chapter as well.

The Chapter 3, “Implementation”, and Chapter 4, “Testing”, deal with the implementation and testing phase of the said functionality.

Finally, “Conclusion” evaluates the whole thesis and addresses the future work on Trdelnik and the newly implemented feature.

[5]`https://github.com/Ackee-Blockchain/trdelnik`
[6]`https://github.com/project-serum/anchor`

Chapter **1**

Analysis

The chapter provides an in-depth look into the Solana blockchain and its ecosystem. After that, Trdelnik, the Rust Testing Framework for Solana programs, is introduced and analyzed.

1.1 Solana

First, blockchain technology is introduced, and key terminology is explained. Then the Solana blockchain is introduced, and its core concepts are thoroughly explained. The unique programming model is shown, and at the end of the chapter, the current ecosystem of Solana is evaluated.

1.1.1 Terminology

Blockchain

Solana is, simply put, another blockchain like Bitcoin [2] or Ethereum [3]. A blockchain can be thought of as a series of blocks or an append-only data structure that resembles an ordered back-linked linked list, which uses hashes as pointers to previous blocks (Figure 1.1). This structure is comprised of blocks, which form a chain, hence the term blockchain. It can be easily concluded that on its own, it is a very simple data structure.

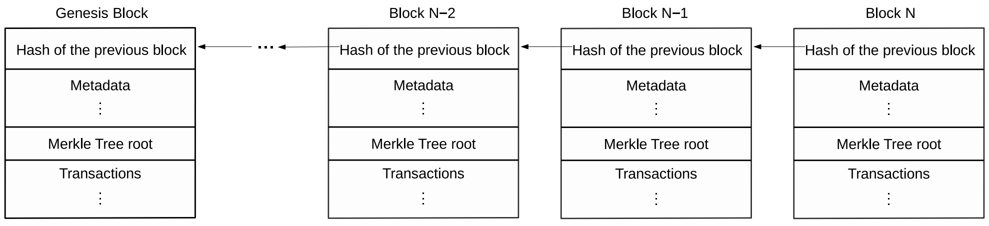


Figure 1.1: General Scheme of Blockchain [4]

Block

Block is a data structure that contains a header, which is comprised of three items – hash of the header of the previous block, metadata, and a Merkle root [5]. Metadata depends on the protocol. The Merkle root is a root of the well-known Merkle tree that can be used to verify later that transactions in a block have not been tampered with. After the header comes the core part of the block, transactions.

Transaction

Transaction is a protocol-defined message that is stored as a part of a block, which is then stored as a part of a blockchain. The content usually consists of some kind of value transfer or on-chain program execution. Transactions are cryptographically signed by their authors, proving their authenticity, and in the case of a value transfer, ownership of the funds or tokens often represents some value in the real world.

Protocol

Protocol is a common set of rules network nodes need to adhere to. It defines things like communication between P2P (Peer-to-Peer) nodes, transaction format for everyone intending to use the network, any special features, and everything else for the network to operate correctly and for the users to know how to transact over the network. An important feature of a good protocol for a decentralized blockchain network is to set up incentives properly. The need for its native coin arises.

Coin

Coin incentivizes participation in the network. It is usually paid with every new block to miners or validators for their help in securing the network. Without proper incentives, any decentralized blockchain network falls apart.

Nodes

Node is a term from the graph theory or distributed systems, is a single participant in a network. Nodes communicate with each other according to the protocol and in a P2P manner forming the whole blockchain network. There might be more kinds of nodes that are not equal, e.g., validator nodes securing the network or pure RPC (Remote Procedure Call) nodes used only to query the network and post new transactions. They can overlap in features.

Consensus

In an effort to agree on a certain state of a blockchain, network nodes need to come to a consensus. We assume there are malicious nodes in the network. Therefore the system must be able to withstand attacks to a certain extent, not just simple failures of nodes. BFT (Byzantine Fault Tolerant) is thus a desired property of such distributed system.

There are currently essentially only three viable consensus families that can be used in practice. The first is the classic PBFT-like (Practical BFT) algorithm family [6]. The second is a so-called Nakamoto consensus, which couples a Sybil protection mechanism of Proof-of-Work with the longest-chain rule, a novel consensus invented by Satoshi Nakamoto for Bitcoin in 2008 [2]. The third and newest family of consensus protocols known today is called Snow, but it is more known under its implementation name – Avalanche Consensus [7], introduced in 2018 and used for the Avalanche cryptocurrency.

Sybil Resistance

In order to prevent a single entity from taking over the network, there must be a mechanism put in place so that no one can just spawn more nodes that can mine or vote, depending on the network, subverting the reputation system of the network. These dishonest nodes would be able to out-vote honest nodes and start censoring transactions or even approving invalid transactions, or changing the whole protocol.

The two most common Sybil resistance mechanisms today are PoW (Proof-of-Work) and PoS (Proof-of-Stake). The former employs a model where miners in the network are given a chance to mine a block that is proportional to their hashing power in the network and is used in Bitcoin [8]. The latter is a new type of model for voting-based networks where a validator is given the power of their vote proportionally to staked coins.

Security

Consensuses and Sybil resistance mechanisms are often confused as the same, which is not true and is worth pointing out. One works when coupled with the other. Let's see how this works in both PoW-based and PoS-based networks.

Consider what makes Bitcoin, which is a PoW-based network, theoretically secure – it is the fact that only the longest chain is respected, also commonly known as the longest chain rule. This is the reason why the consensus is actually called, as mentioned before, the Nakamoto consensus.

For a PoS-based network, the Sybil resistance mechanism is usually coupled with a variant of a PBFT-like algorithm or the novel Avalanche consensus.

Smart Contracts

Many blockchains allow the deployment of so-called Smart Contracts or, in other words, on-chain programs. Smart contracts were introduced in Ethereum.

A smart contract is a piece of code deployed to the blockchain with a cryptographically signed transaction. Users are then able to interact with it by sending transactions that invoke a specific function defined in the smart contract, and the business logic is executed as stated in the deployed code [9].

Data relevant to the state of the smart contract are also stored on the blockchain. Hence we can look at smart contracts as programs on a decentralized computer that accesses files in its file system and changes them according to the predefined rules. If such a contract is made immutable, we can trust the smart contract will not do anything else than it was supposed to.

It is worth noting that apart from storing the blockchain itself, each node creates a state as a result of transaction execution. The final state is the result of all processed transactions and can always be deterministically recreated from the blockchain history.

Code is compiled for a predefined ISA (Instruction Set Architecture) and executed in a VM (Virtual Machine) which understands it. The mentioned VM is a special runtime environment similar to well-known VMs such as JVM (Java Virtual Machine) or CLR (Common Language Runtime) from Microsoft's .NET ecosystem. The most commonly known VM for smart contracts, which is used by Ethereum, is EVM (Ethereum Virtual Machine) and includes its very own instruction set specialized for the needs of smart contracts.

Only transactions involving smart contract execution need to be processed by the VM. The common execution path is to prepare the relevant smart contract data and smart contract byte code, launch the VM with said data and code and observe possible failures. If the execution results in success, take the changes to the smart contract data made in the VM and change the state outside the VM, otherwise discard the changes and continue with another transaction.

1.1.2 Solana Introduction

Solana is a single-chain blockchain using a slightly changed PBFT consensus called Tower BFT with Proof-of-Stake as a Sybil protection mechanism. Leaders are known in advance, their rotation is the function of the blockchain data, and they are known one full epoch before.

Epoch is a series of 432,000 slots, where the slot is a term for the time period the block is in the making by the leader.

The blocks are actually streamed as something called entries, so the creation of the block by the leader and verification of the block by others can happen in parallel.

Solana officially launched its mainnet, still labeled as beta, in March 2020. The native coin that incentivizes validator nodes and protects the network from spam by paying transaction fees with it is named SOL.

Solana's main value proposition is solving the blockchain trilemma, i.e., delivering scalability, decentralization, and security without sacrificing any of the three mentioned features.

Solana was founded by Anatoly Yakovenko in 2017 when the Solana Whitepaper [10] was published. Describing a novel clock mechanism for distributed systems called PoH (Proof-of-History) as a technique for keeping time between computers that do not trust each other. With this mechanism, they were able to demonstrate [11] on a testnet with a gigabit network and 150 nodes processing 500,000 TPS (Transactions per Second). Compared with Bitcoin's maximum throughput of 7 TPS and Ethereum's maximum capacity of 15 TPS [12].

Since this proof of concept, Solana has been developed into a fully functional blockchain smart contract platform and strives towards adoption. It is worth noting that often laboratory experiments are vastly different from the real world and it remains to be seen if Solana will be able to scale for universal adoption outside of speculations. Nevertheless, new contenders like Solana help drive the research and development forward and anyone is free to build on top of what has been invented already and improve it.

Ethereum Killer

Solana is a smart contract platform. The community defines direct competitors of Ethereum in this sense as Ethereum killers. Solana has been considered to be a new potential Ethereum killer.

Compared to Ethereum, smart contracts on Solana are called simply Programs. They can be executed in parallel. Parallelization is one of the key differences from other platforms. While Ethereum can be thought of as a single-threaded distributed computing platform, Solana can be considered a multi-threaded one.

Solana makes itself clear to focus on improving scalability from the engineering perspective. It is rethinking and reengineering core parts that were first seen in Ethereum and making them parallel and optimized, including the usage of Nvidia CUDA to speed up certain parts of the code and inventing its own specialized horizontally scalable database system for state storage and many other things that are supposed to make it possible to reach maximum TPS practically only bounded by the network throughput, memory throughput and the number of CUDA cores in modern Nvidia GPUs. Therefore over time, it should scale with better hardware available on the market and internet connectivity in the world.

Rust Development

Solana's ecosystem revolves around the Rust programming language and its ecosystem. The main and only implementation of the node software is written in it. Also, Solana programs are almost exclusively written in Rust. Even though there is no technical barrier preventing from using C or C++, Rust is the most supported language for developing on Solana, and all the libraries and supporting code that can be found are written in it, leaving practically no other choice.

1.1.3 Core Concepts

There are eight main core concepts introduced in Solana that are supposed to make it as fast as developers claim. This section will try to cover all of them in maximum detail. Unfortunately, you cannot always find a proper explanation of some of the details, and some of these are not yet or not fully implemented, so the source code does not answer the questions that arise while studying them.

Proof-of-History (PoH) – Virtual Clocks

Agreement on time in distributed systems has always been problematic. First, a high-level overview of this concept is described and followed by a more in-depth description.

Solana leverages the so-called Proof-of-History (PoH) mechanism to synchronize local virtual clocks on all nodes [10]. PoH makes sure the timestamp in any message can be trusted, and any timeouts in the consensus protocol can be avoided as everyone knows the time and knows if they should start a new round consensus round or not. PoH allows minimizing the block time as there's no waiting overhead. In other words, with synchronized clocks, we can replace communication with local computation.

To prevent validators from skipping the validator that comes before them, PoH is used to force all validators to spend a minimum amount of time before they could even submit their block. Thus if validator B follows validator A, B cannot try to skip A by chaining off its previous block because B has to run the Proof of History algorithm at least as long as A does, so A will get a fair chance of submitting their block.

Verifiable Delay Function (VDF)

PoH is based on a Verifiable Delay Function (VDF). More concretely, Solana uses a recursive pre-image resistant SHA256 VDF, where the output of one SHA256 iteration is used as an input of the next iteration recursively.

In order to create a block, the producer needs to compute the VDF with all new messages to be included in the block:

$$\begin{aligned} Message_1 &\rightarrow Hash_1 \\ Hash_1 + Message_2 &\rightarrow Hash_2 \\ &\dots \\ Hash_{n-1} + Message_n &\rightarrow Hash_n \end{aligned}$$

Observations:

1. From PoH we have a proof for the **Lower Bound** on Time of Time of $Message_i$ (i.e. $Message_i$ must have taken place after $Hash_{i-1}$).
2. From PoH we have a proof for the **Upper Bound** on Time of Time of $Message_i$ (i.e. $Message_i$ must have taken place after $Hash_{i+1}$).
3. Points 1 and 2 imply the exact ordering of messages, which then implies that VDF not only provides us virtual clocks, but everyone can trust the order of events.

Phases of PoH:

1. **Evaluation phase (leader):** computation on only one CPU core as it is a strictly sequential computation by definition. This takes:

$$\frac{\text{Total number of hashes}}{\text{Hashes per second for 1 core}}$$

2. **Verification phase (voters):** the block can be checked in parallel using GPU with thousands of cores as it can be easily sliced and the intermediate hashes are known, this takes:

$$\frac{\text{Total number of hashes}}{\text{Hashes per second for 1 core * Number of cores available}}$$

It can be concluded that PoH is therefore hard to produce but easy to verify. These are two important factors that are critical for the use of PoH, as it is not easy to fake the PoH, but once it is finished, any validator can verify the results very quickly.

Tower BFT (TBFT) – PoH-based PBFT

As a consensus algorithm, Solana uses the Tower BFT (TBFT), which is a custom implementation of a well-known Practical Byzantine Fault Tolerance (PBFT) algorithm published in 1999 by Miguel Castro and Barbara Liskov [6].

PBFT consensus rounds are broken into three main phases (pre-prepare, prepare and commit), see Figure 1.2. The exact description is out of the scope of this work.

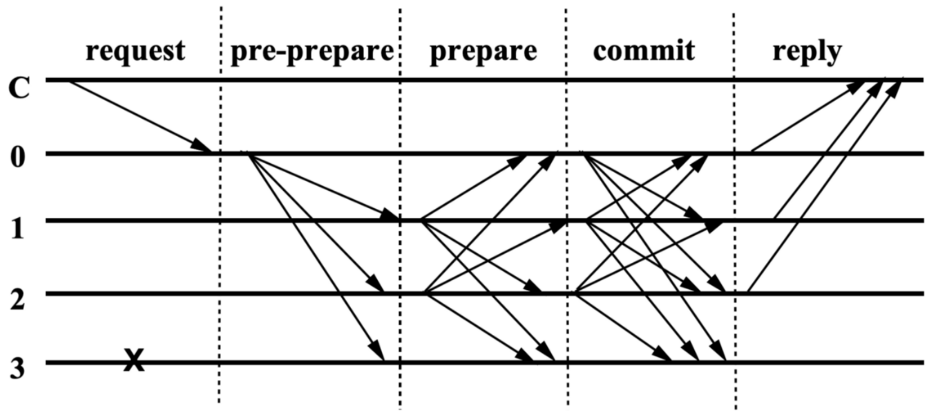


Figure 1.2: Normal operation of PBFT [6]

PBFT is focused on satisfying **safety** (results are valid and identical at all non-faulty nodes) and **liveness** (nodes that don't fail always produce a result) properties. The safety guarantee is possible due to the deterministic nature of the process (executed on every node). Liveness guarantee is possible due to the View-change process. The network will not be stopped unless there are too many byzantine nodes. View-change allows nodes to switch leaders when it seems to be malicious or faulty.

View-change

View-changes are carried out when it appears that the leader has failed, so another node tries to take over his place by starting an election process. It gets triggered by timeouts that prevent nodes from waiting indefinitely for requests to execute.

In addition, timeout is postponed each time that the protocol detects that nodes are reaching an agreement on the current block.

TBFT vs. PBFT

TBFT is a derivation of PBFT, which differs in one fundamental thing. PoH provides a global source of time before consensus and can be therefore used to enforce the exponentially-increasing timeouts introduced in the original PBFT algorithm. No messages are needed as it is enforced by the PoH itself.

The way it is done follows. Voting on a new block is restricted to a fixed period of time counted in hashes, this unit of time is called a slot. As of this moment and with the current network settings, if we translate the number of PoH hashes into time, it is around 400ms for one slot. Thus every 400ms, the new potential rollback point occurs, but every subsequent new block that is voted on doubles the amount of time that the network would have to stall before unrolling the original vote.

Consider that each validator has voted 32 times in the last few ~12 seconds $(32 \cdot 0.4)$. The vote 12 seconds ago now has a timeout of 2^{32} slots, which translated into years with a constant time of a slot of 400ms, is roughly 54 years $(2^{32} \cdot 0.4/86400/365)$. A transaction with 32 confirmations is also considered finalized.

Turbine – Block Propagation Protocol

Turbine is a name for a smart block propagation protocol that reduces the time needed for block propagation and the overall message complexity reducing the communication overhead of a node.

Turbine is a multi-layer propagation protocol. First, nodes in the network are divided into small partitions called neighborhoods. Nodes within a particular neighborhood are responsible for sharing data received with other nodes in the same neighborhood and propagating the data to a small number of nodes in other neighborhoods (Figure 1.3 and 1.4). The data unit shared is called a shred, and one block is constituted of many shreds.

The partitioning of nodes into neighborhoods and how exactly are shreds shared within and out of their neighborhoods are implementation details.

Since we are in an adversarial environment, any node can decide not to rebroadcast the received shreds or broadcast incorrect data. These are two problems solved with a series of countermeasures:

- Forward Error Code (FEC), more concretely Erasure Code, helps by broadcasting a block with more shreds than initially needed to reconstruct the entire block without errors, even if some shreds are lost along the way. With $N = 6$ data shreds and additional $K = 3$ shreds, we can lose up to 1/3 of the shreds and still be able to fully reconstruct the whole block.
- Propagation is prioritized accordingly to their stake. Validators with the most stake are put closer to the current leader. A stake-weighted

selection algorithm is used to create such a tree where the risk of faulty or malicious nodes is minimized.

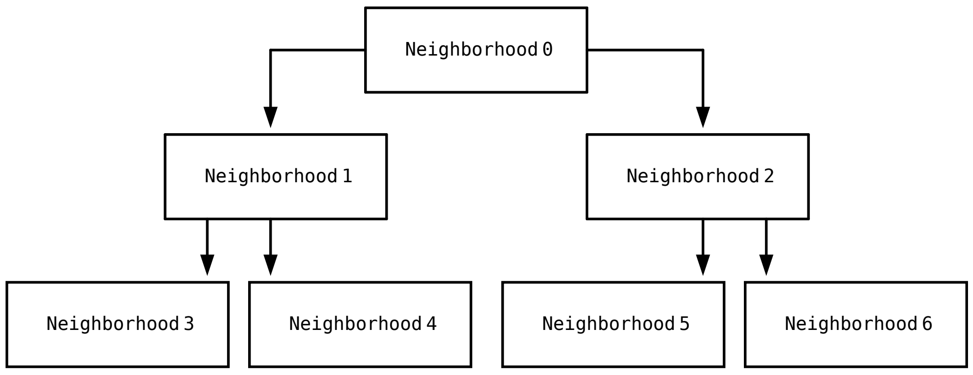


Figure 1.3: Shred propagation diagram [13]

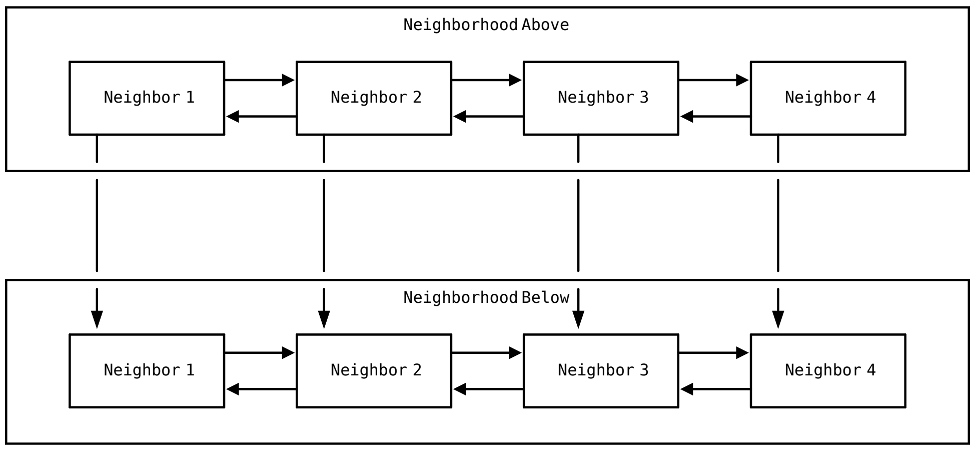


Figure 1.4: Shred propagation between two neighborhoods [13]

Gulf Stream – Transaction Forwarding Protocol

Gulf Stream is Solana's mempool-less solution to forward and store transactions before processing them.

In traditional blockchains, each node reserves a part of its memory for a memory pool. This memory pool, more commonly referred to as mempool, is used to store transactions being currently broadcasted over the network but have not been processed and added to the blockchain as a part of a new block yet.

This implies a huge communication overhead where any transaction needs to reach all other nodes in the network. Even though not necessarily everyone needs to be aware of all transactions in the mempool, they are most important for miner and validator nodes (depending on the type of a network), which need to include them in new blocks.

If there are more transactions in the mempool than can fit in a block, the backlog of transactions is created. This can generally lead to increased transaction fees for users, who need to push their transaction ahead of other transactions, as it is economically viable for the nodes securing the network to prefer the transaction with higher fees. This is currently not possible on Solana, but on the other hand, the network is so fast with its ~400ms block rate that the aim is to process all remaining transactions almost instantaneously anyway. As an example, see Figure 1.5.

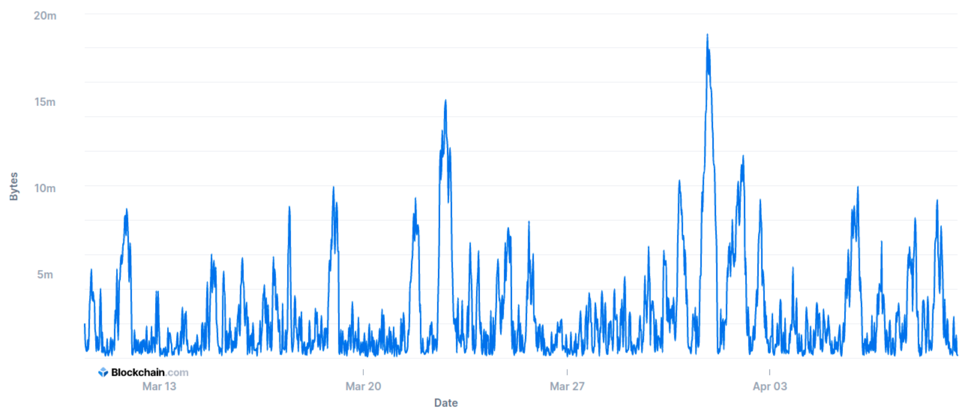


Figure 1.5: Ethereum mempool in bytes [14]

With the aim of Solana to process potentially transactions in hundreds of thousands, the common gossip protocols used in other blockchains to propagate transactions to all nodes are infeasible.

The Solution

The solution that Solana thought of is avoiding having a single shared mempool and instead of pushing transactions to the edge of the network to the expected leader. The leader receives the transaction as quickly as possible and can process it immediately.

This solution has a catch, though. The expected leader must be known ahead. Leaders are known in advance, and their rotation is the function of the blockchain data, and known one full epoch before. An epoch is a number of slots that one leader's schedule is valid for. It is set to 432,000 slots, and with ~400ms block rate, it takes about two days.

Sealevel – Parallel Smart Contract Runtime

Other blockchains are single-threaded global state machines. The only thing they might do in parallel is signature verification. Solana introduced Sealevel, a parallelized transaction processing engine designed to scale horizontally across GPUs and SSDs.

Sealevel can theoretically process as many transactions as many cores are available to the system. According to the source code, Sealevel is not parallelized on the GPU level yet.

This is a major improvement, which makes Solana a multi-threaded global state machine, a thing not seen until Solana. Other blockchains, including the leading Ethereum, can be considered single-threaded global state machines, as, at one time, only one smart contract invocation can be processed.

The reason why it is possible with Solana is that each and every Solana transaction describes all the states required to read and write to. Sealevel can then choose non-overlapping instructions to execute in parallel and not just that. Transactions that only read certain states can be executed in parallel as well. This is a high-level description of how it works:

1. Sort millions of pending transactions.

2. Schedule all the non-overlapping transactions in parallel.

SIMD approach with GPUs

There is a big potential for GPU parallelization and leveraging its SIMD capability. For example, in Nvidia CUDA, modern cards have thousands of CUDA cores and tens of Streaming Multiprocessors.

When a CPU invokes a kernel grid, the blocks of threads are distributed among streaming multiprocessors and executed using specific ALU execution units, usually called CUDA cores and other SFUs (special function unit).

The executed code is the same for all cores. Imagine a situation where there is a single smart contract invocation but with numerous different inputs. This is the exact workload that can be efficiently executed on GPU architectures, such as Nvidia CUDA.

Since Sealevel is not yet optimized for GPU offloading, GPUs are today used only to accelerate PoH verification and signature verification and only if it is available to the system and the algorithm decides it is worth the overhead of launching the kernel grid.

BPF – Berkeley Packet Filter

There is one important thing that was not covered in Sealevel yet. What actually executes the code, and how it is done. The standard way is to use some sort of a Virtual Machine (VM) and compile the code for it from any supported language. This code gets deployed to the blockchain, and when a user sends a transaction invoking this contract, this code gets loaded into the VM and executed.

Ethereum does this with its own Ethereum Virtual Machine (EVM). Some other blockchains make use of Web Assembly (WASM). Solana iterated over all possible solutions and chose an unexpected VM called Berkeley Packet Filter (BPF).

Sealevel hands off transactions to be executed on hardware natively using an industry-proven bytecode called the Berkeley Packet Filter (BPF), which is designed for high-performance packet filters. It can be used for non-networking purposes. BPF and the extended BPF (eBPF) are basically in-kernel VMs available in most UNIX-like operating systems, and they are very performant because their primary use was for packet matching, which needs to be as fast as possible. It also has had tens of years of development behind it.

The original version of BPF is now called classic BPF (cBPF), and this one could not be used for anything other than packet matching. Linux kernel now includes only extended BPF (eBPF), which is a virtual machine with 64-bit registers. The eBPF is now normally called just BPF.

It is worth mentioning that new modern firewalls are being built on top of the extended BPF. Execution of BPF is currently parallelized only on the CPU level. What is really used is a modified version of BPF called rBPF, which is launched in the userspace instead of the kernel. This was important as the kernel version of the BPF would not be able to facilitate certain operations.

Pipelining – Transaction Processing Optimizations

It is not enough to be able to form a consensus and share a block with the rest of the network quickly. A node must validate and execute all those transactions in received blocks before another block comes.

For this reason, the Solana team developed something they call a Transaction Processing Unit (TPU) [15]. The TPU works as a processor and makes heavy use of pipelining – a common CPU optimization that helps to keep the chip more utilized via staging an instruction execution into stages. It is a general way to keep all the hardware parts busy instead of being idle. This concept of pipelining was borrowed, and that is how the TPU was born.

The pipeline stages of TPU are following (Figure 1.6):

1. Data fetch in kernel space via network card (I/O)
2. Signature verification using GPU (very computation heavy if not offloaded)
3. Change of the state using CPU (banking)
4. Write to the disk in kernel space and send out via network card (I/O)

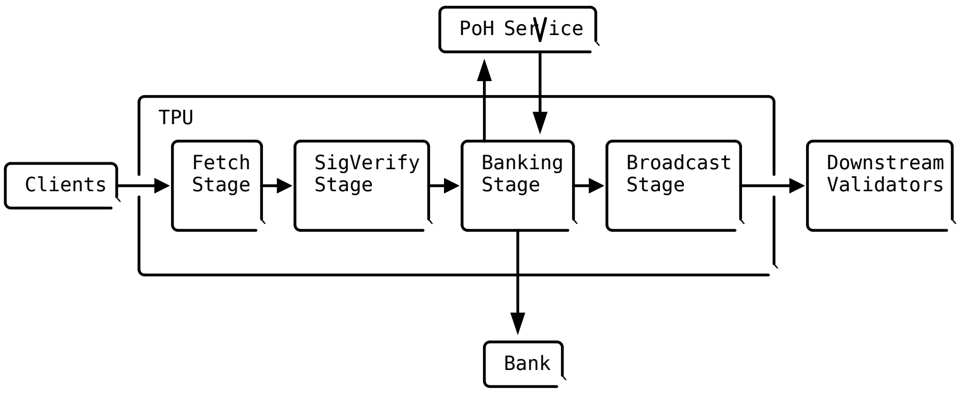


Figure 1.6: Transaction processing unit [15]

In fact, there are two TPUs in the Solana node software. The second one is called TVU, where the V stands for validator or validation. The one called TPU is used for creating new blocks, and the TVU one is used for validating. They might slightly differ. However, the concept and functionality are very similar.

Cloudbreak – Horizontally-scalable Database

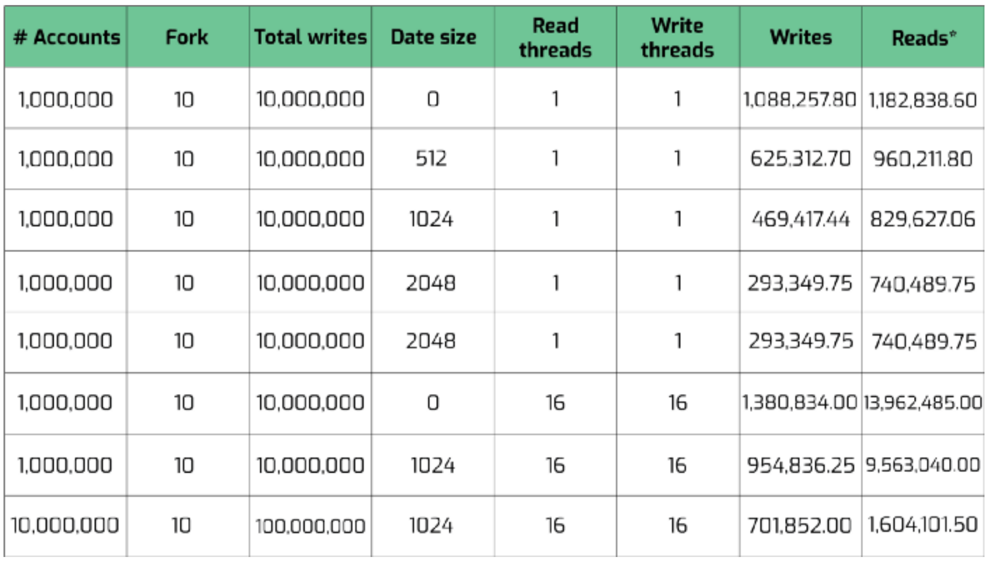
With fast computation, the obvious thing that becomes the new bottleneck is the memory. For example, the industry-standard local database for storing blockchain and state, LevelDB, does not support parallel reads and writes. For Bitcoin or Ethereum, that is fine, not for a massively parallel system like Solana.

We could say, why not store everything in RAM? It is too large, and even for enterprise machines and large servers, this becomes impossible over time. Therefore Solana had to invent its own database system that supports parallel reads and writes and scales easily with more disks.

This new database system is called Cloudbreak and makes use of memory-mapped files. The data is therefore stored in files that can be accessed independently. A memory-mapped file is a file that is mapped to the process' virtual memory address space and can be accessed directly without further system calls. The speed is still limited by the disk I/O, but we get less overhead, and the kernel can keep a part of it in its page cache (also known as file cache).

Reads in Cloudbreak are randomly distributed among available disks, as the data is stored uniformly. Writes in Cloudbreak use the Copy-on-Write semantics and are appended to a random disk. Hence we get the speed of sequential writing. This is all possible because of a clever system of bookkeeping. Old data entries are also garbage collected for future use.

The design of Cloudbreak makes it ideal for hardware setups, such as RAID 0 with fast NVMe SSDs. The Cloudbreak database has been benchmarked by the Solana team (Table 1.1). The results show that even with 10 million accounts (unit of data storage on Solana that will be described in the Programming model), which is a size that will not fit in the RAM (i.e., cannot be cached by page cache in the kernel), Cloudbreak still achieves reads and writes close to 1 million with a single SSD [16].



# Accounts	Fork	Total writes	Date size	Read threads	Write threads	Writes	Reads*
1,000,000	10	10,000,000	0	1	1	1,088,257.80	1,182,838.60
1,000,000	10	10,000,000	512	1	1	625,312.70	960,211.80
1,000,000	10	10,000,000	1024	1	1	469,417.44	829,627.06
1,000,000	10	10,000,000	2048	1	1	293,349.75	740,489.75
1,000,000	10	10,000,000	2048	1	1	293,349.75	740,489.75
1,000,000	10	10,000,000	0	16	16	1,380,834.00	13,962,485.00
1,000,000	10	10,000,000	1024	16	16	954,836.25	9,563,040.00
10,000,000	10	100,000,000	1024	16	16	701,852.00	1,604,101.50

Table 1.1: Cloudbreak benchmark [16]

Archivers – Distributed Ledger Storage

Since the Solana blockchain can grow at enormous speed, considering the full capacity of 1 Gbps (with no overhead) for 365 days, it is roughly 4 petabytes of data that each node would need to store to have a complete history. There is a concept of distributed ledger storage that would store this data for everyone else in a decentralized fashion.

The idea is to offload the data from validators to these specialized network nodes. The data is broken into many small pieces and replicated so that the full state can always be reconstructed. These special nodes are also contested on the protocol level to make sure they store the data they are supposed to store, and the loss of data is prevented.

This concept is yet to be implemented. A potential implementation might be using a new decentralized protocol for permanent storage Arweave[7] or Filecoin[8].

[7] `https://www.arweave.org/`

[8] `https://filecoin.io/`

1.1.4 Programming Model

This section explains the programming model of Solana. There are a few fundamental topics that any programmer, who wishes to use Solana, needs to know and study beforehand.

communication with the network

Any user, when they decide to interact with the network, needs to interact with any of the network's nodes over either a JSON-RPC or a WebSocket endpoint. The available methods are all listed publicly in the Solana documentation[9].

The methods range from queries, such as specific account information, the network state (an example shown in Listings 1.1 and 1.2) to sending transactions.

```
curl http://localhost:8899 -X POST -H "Content-Type: application/json" -d '
  {"jsonrpc":"2.0","id":1, "method":"getBlockHeight"}
'
```

Listing 1.1: Request of the getBlockHeight method

```
[{ "jsonrpc": "2.0", "result": 1233, "id": 1 }
```

Listing 1.2: Response of the getBlockHeight method

What really matters is the ability to send transactions. Sending a transaction is the only way we can change the data on the Solana blockchain. Any write operation is done through the means of transactions.

Users are not required to use the RPCs directly. There are multiple libraries that provide convenient interfaces for languages, such as Javascript, Rust, and Python.

Overview

The following steps can be thought of as an overview of what happens when an app or any user interacts with the Solana network by sending a transaction. The terms, such as instruction, account, or program, will be explained shortly, and a more in-depth explanation will follow.

1. An app or a user sends a transaction with one or more instructions to a Solana node that accepts RPC requests.

[9] https://docs.solana.com/developing/clients/jsonrpc-api

2. The transactions get validated and forwarded according to the deterministic leader schedule to the next leader.

3. The transaction is validated and processed by the leader and included in a new block, which is then streamed to all other validators, who validate and process the transaction as well, coming to the same final state.

4. During processing, the instructions in transactions are passed to programs deployed by developers beforehand. This is the job of the Sealevel runtime. The relevant accounts get modified by code in those programs. All happens isolated in the VM. Instructions are executed sequentially and atomically, meaning either all instructions finish successfully or all changes introduced by any instruction within the transaction are discarded.

Transaction Key Elements

Some of the transaction key elements should be explained first:

Signature
Each digital signature is in the ed25519 binary format consuming 64 bytes.

Account
A key-addressable record on Solana ledger.

Compact array
An array-like data structure that begins with a specially encoded array length in the first 16 bits, followed by the array items.

Blockhash
A unique hash that identifies a block produced as a part of the Proof-of-History algorithm.

Program id
The address of an account containing a program.

Instruction
A structure specifying a program id for execution, relevant accounts, and opaque instruction data that can be interpreted by the program.

Transaction Anatomy

A Solana transaction (Figure 1.7) is comprised of two major parts in the following order:

1. A compact array of signatures.

2. A message, which contains a compact array of account addresses, followed by a recent blockhash and ending with a compact array of instructions.

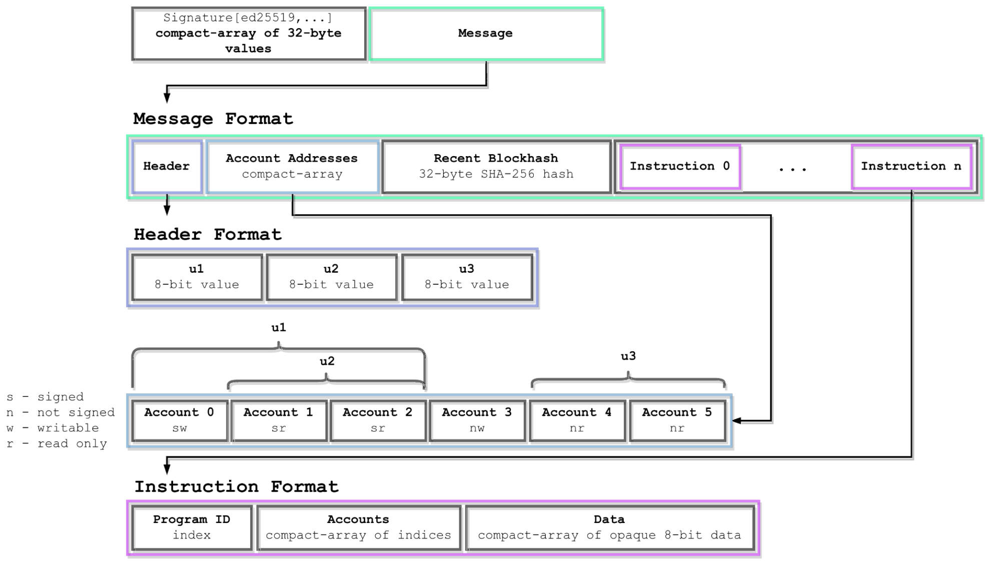


Figure 1.7: Transaction anatomy

Signatures

For signatures in the compact array of signatures, the Solana runtime verifies the following:

- The number of signatures must match the first 8 bits of the message header.

- The signature is verified against the public key at the same index in the message's account addresses array.

Message

The layout of a message is shown in the following table:

Field	Description
Header	Message metadata
Accounts	Compact array of account addresses
Recent blockhash	Blockhash of recently produced block
Instructions	Compact array of instructions

Table 1.2: Message layout

1. Header

   a) # of required signatures in the transaction (8 bits).

   b) # of read-only accounts requiring signatures (8 bits).

   c) # of read-only accounts not-requiring signatures (8 bits).

2. Accounts

   a) Addresses that require signatures with read-write access.

   b) Addresses that require signatures with read-only access.

   c) Addresses that do not require signatures with read-write access.

   d) Addresses that do not require signatures with read-only access.

3. Recent blockhash

   - Transaction lifetime: transaction is deemed invalid if the blockhash is older than 32 blocks.
   - Transaction replay: identical txs get rejected, you can change the blockhash and repeat the exact same action. Works in a similar way as nonce in Ethereum.

4. Instructions with the following instruction anatomy (Figure 1.8):

   a) Program id index (index to Accounts).

   b) Compact-array of account address indices (indices to Accounts).

   c) Compact-array of opaque 8-bit data (what operations to perform and any additional data).

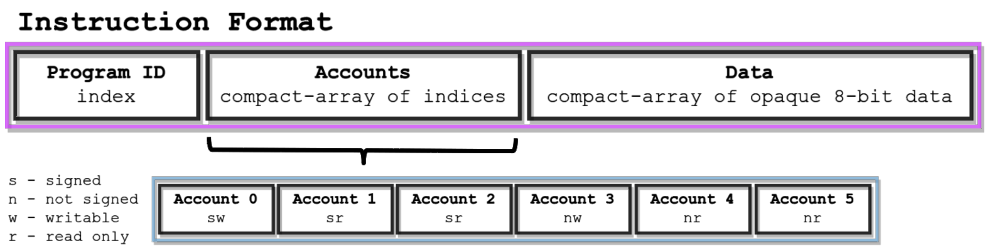


Figure 1.8: Instruction anatomy

Account Anatomy

Similar to UNIX's philosophy where everything is a file", on Solana, the statement everything is an account" holds true. In other words, an account is a memory buffer, an equivalent of a file in any file system. Its main purpose is to store states between instructions and transactions.

To look up an account, an address is used, often referred to as a public key or pubkey. Solana's account system can be therefore considered a key-value database system.

The key may be one of the following:

- An ed25519 public key.
- A program-derived account address (32byte value forced off the ed25519 curve).
- A hash of an ed25519 public key with a 32-character string.

The structure of an account is shown in the following table:

Field	Description
Lamports	Lamports in the account
Data	data held in this account
Owner	the program that owns this account. If executable, the program that loads this account.
Executable	this account's data contains a loaded program (and is now read-only)
Rent Epoch	the epoch at which this account will next owe rent

Table 1.3: Account layout

1. Lamports
    - Balance of the account in Lamports.
    - 1 lamport = 10^{-9} SOL.
2. Data
    - Vector of bytes.
    - Maximum size of 10 MB (10 KB for PDAs).
3. Owner
    - The owner is a program id or a loader in case of an executable account.
    - If the owner matches the program id, the program is granted write access. Otherwise, it is only permitted to read its data and credit the account
    - All new accounts are owned by the System program that allows transfers of Lamports, allocating data, and assigning ownership to a different program id.
    - An account is always owned by a program or a loader.
4. Executable
    - Turning a non-executable account into an executable one is a one-way only operation.
    - The account becomes read-only.
    - The owner of such an account is a loader that will load the code from the data field of the account and start executing it if invoked.
5. Rent Epoch
    - Keeping accounts alive on Solana incurs a fee called rent.
    - An account is considered rent-exempt if it holds at least two years worth of rent.
    - Rent Epoch is the epoch number when the runtime will check again whether an account should pay rent or is rent-exempt.

Account Types

There are three basic types of accounts on Solana. Note that this is not any sort of official classification.

- Data account storing data, also user wallets with an empty data field.
  - System owned accounts.
  - Program owned PDA (Program Derived Address) accounts.
- Program accounts storing user-deployed executable bytecode.
- Native accounts indicating native programs or special runtime accounts
  - System – lamports transfers, data allocation and ownership assignment.
  - BPF Loader – uploading and launching executable programs.
  - BPF Upgradeable Loader – uploading and launching of upgradeable executable programs.
  - Stake – program for staking SOL as a part of Proof-of-Stake mechanism.
  - Vote – program for voting as a part of the Tower BFT consensus.
  - Native Loader – owner of native programs and their loader.
  - SysVar – special runtime accounts with blockchain-related information

Runtime Policy

Runtime policy or Sealevel runtime account rules are a set of rules enforced by the Sealevel runtime to protect the security of the system and make Solana a safe and predictable environment for its users. The following list of rules is taken from the official documentation [17]:

- Only the owner of the account may change owner.
  - And only if the account is writable.
  - And only if the account is not executable.
  - And only if the data is zero-initialized or empty.
- An account not assigned to the program cannot have its balance decrease.
- The balance of read-only and executable accounts may not change.
- Only the system program can change the size of the data and only if the system program owns the account.

- Only the owner may change account data.
  - And if the account is writable.
  - And if the account is not executable.
- Executable switch is a one-way (false->true) operation, and only the account owner may set it.
- No one can make modifications to the rent_epoch associated with this account.

Compute Budget

To prevent abuse of the Solana nodes' resources that could potentially lead to network failures or denial of service, each transaction is given a compute budget. When the program consumes its entire compute budget or exceeds certain bounds, the runtime halts the currently running instructions and returns an error.

Cross Program Invocation (CPI)

Cross Program Invocation (CPI) is a facility allowing us to call other programs from within an instruction. The caller is halted until the execution returns back from the callee. An important term connected with CPIs is a Program Derived Address (PDA).

Program Derived Address (PDA)

Programs can issue instructions that contain signed accounts that were not signed in the original transaction by using Program Derived Addresses (PDA). These accounts are called PDA accounts. Program derived addresses allow programmatically generated signatures to be used when calling between programs.

PDA is an address deterministically derived from the program id and supplied keywords (Figure 1.9). The resulting address is checked against the Ed25519 curve and bumped off it with so-called bump seeds if needed. Hence, there is no private key.

When a program tries to invoke a CPI with such address, the runtime takes the supplied keywords and bump seeds, uses the caller's program id, and repeats the process. If the resulting PDA matches, then the account is considered to be signed.

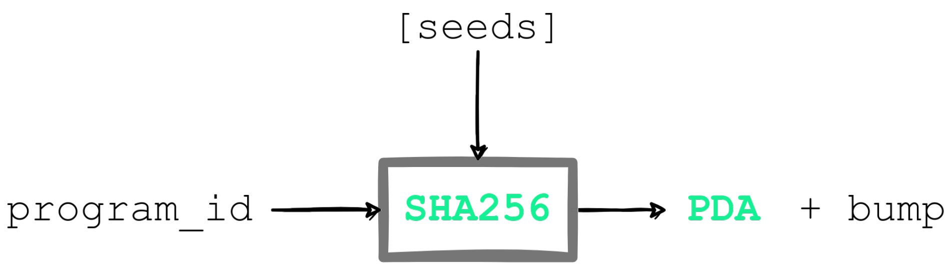


Figure 1.9: PDA generation

1.1.5 Ecosystem

Wallets

The Solana ecosystem consists of various user-facing products like wallets and tools, allowing anyone to create their own token or a Non-Fungible Token (NFT) easily and use the network without much hassle.

There are various web wallets, Android and iOS app wallets for smartphones, browser extensions, and also official CLI tools. They differ in capabilities and out-of-the-box support for various Solana projects or being developer-oriented.

The major ecosystem wallet providers include:

- Phantom (iOS/Android apps and all major browser extensions)
- Solflare (Web wallet, iOS/Android apps, and chrome-only extension)
- Sollet (Developer-oriented web wallet and chrome-only extension)

The only supported hardware wallets to safely interact and store keys to access cryptocurrencies and other assets now are products by the company Ledger:

- Ledger Nano S
- Ledger Nano X

Decentralized Finance (DeFi)

There are many financial products available, bringing the traditional banking but in a decentralized coat – known as Decentralized Finance (DeFi).

Decentralized exchanges (DEX) and swapping services offer a way to trade SOL or any available token into any other token easily:

- Project Serum
- Bonfida DEX

- Raydium Swaps
- Orca Swaps
- Jupiter Aggregator
- Mango Markets

Decentralized bridges offer a way to bridge your tokens from blockchains such as Bitcoin or Ethereum to Solana, allowing them to be used in the Solana's DeFi products:

- Wormhole Bridge
- DeBridge Finance

Liquid staking of SOL and synthetic assets:

- Marinade Finance
- Synthetic
- Sypool

NFT Marketplaces and Metaverse

There are multiple NFT marketplaces to trade NFTs, which are in theory supposed to represent a certificate of ownership of something, these days usually of some kind of an auto-generated picture.

The trend of NFTs is tightly coupled with an old-new concept of Metaverse, an artificial world where people could meet as virtual avatars and show off their collectible NFTs.

NFTs might find a useful use case in the future for representing real-world items, such as concert tickets, but it remains to be seen if they offer in practice any real advantage or if the technology stays being used mostly only for speculations.

Some of the projects include:

- Metaplex Protocol
- OpenSea Marketplace
- Solanart Marketplace

Web3 and Games

First games built on Solana are starting to appear, using the blockchain as a back end – this type of development paradigm is referred to as Web3.

In general, any application using the blockchain as a data and business logic layer can be considered Web3.

When a user visits such a website, they are usually able to connect their wallet with the website and interact with it by sending transactions that could, e.g., post a message on a Web3 social media platform or post an order o a decentralized exchange.

1.2 Trdelnik

Trdelnik is a new Rust-based testing framework for Solana programs written in the Anchor framework. This section explains the motivation behind creating Trdelnik and the essential features provided by Trdelnik to make the developer experience smoother and safer from a security standpoint. First, some essential concepts are covered.

1.2.1 Present State

Regular Solana on-chain programs, referred to often as Raw Solana programs, are developed in the plain Rust programming language, following only the formal ABI prescribed by the BPF Loader, specifying things like what the entry point looks like and some special types provided by a library.

A programmer of a Raw Solana program needs to handle all of the following:

1. Deserialize the instruction data.
2. Decision what operation of the program to do based on the instruction data.
3. Deserialize and perform security checks on input accounts. This is important because a user could create accounts with arbitrary data and pass those accounts in place of the valid accounts. The maliciously fabricated input can cause unexpected results and unintended behavior.
4. Perform the required business logic.
5. Serialize the changed data required to handle the state for future program invocations and save it.

The typical structure of a Raw Solana program is shown below:

```
.
├─ src
│   ├─ lib.rs -> module registration
│   ├─ entrypoint.rs -> required entrypoint defined by the BPF Loader
│   ├─ instruction.rs -> program API and instruction data deserialization
│   ├─ processor.rs -> program business logic
│   ├─ state.rs -> program structs and (de)serialization of accounts
│   ├─ error.rs -> program-specific errors
├─ .gitignore
├─ Cargo.lock
├─ Cargo.toml
├─ Xargo.toml
```

The development of Raw Solana programs is a tedious and error-prone experience. And in a world where one single mistake can cost millions of dollars of damage, and the environment is full of adversaries scanning through every new contract with some monetary value deployed on the Solana blockchain to exploit them, this becomes a huge problem.

Anchor Framework

Anchor framework aims to help with the described problem. Anchor offers a set of convenient libraries (crates in the Rust terminology) and tools to help developers with creating their Solana programs in a fast, secure and manageable way.

The most important attributes of the Anchor framework include:

- An embedded Domain-Specific Language (eDSL) exploiting the advanced Rust macro system, making the development easier by adding simple macro attributes to specific parts of the code or using special types.
- An Interface Description Language (IDL) defining a public interface of the Solana program, saying how to properly form a Solana instruction to perform a specific operation of the invoked Solana program and also the description of the relevant accounts used to store data for the program.
- Javascript/Typescript library handling the code generation based on the program's IDL.
- A CLI workspace management tool to help with building, testing, and deploying Solana programs.

Anchor eDSL

The Listing 1.3 shows, how the account structures are defined using the Anchor's eDSL, e.g. Rust macros like *#[derive(Accounts)]*, *#[account]* and special types like `Account`, `Signer` or `Program` driven by Rust generics and lifetimes.

```rust
#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(
        init,
        payer = user,
        space = 8 + 2
    )]
    pub state: Account<'info, State>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}
#[derive(Accounts)]
pub struct UpdateState<'info> {
    #[account(mut)]
    pub state: Account<'info, State>,
}
#[account]
pub struct State {
    pub locked: bool,
    pub res: bool,
}
```

Listing 1.3: Definition of accounts using Anchor's eDSL

The Listing 1.4 demonstrates business logic of a simple turnstile program, which can be initialized with the `initialize` operation. After inserting a coin, the turnstile state changes to unlocked with the `coin` operation, and when the turnstile is pushed via `push` operation, the turnstile gets locked again. Notice the Anchor-provided macro *#[program]* and also that nowhere within the code you can see explicit deserialization or serialization and no input account checking. This is all done internally because the macro expansion of *#[program]* is performed on the module, leaving less room for mistakes and potential loopholes in Solana programs.

Compared to the "Raw Solana way," a programmer using the Anchor framework only describes the business logic, the data needed to perform the business logic, and everything else, including the deserialization of instruction data, deserialization and serialization of accounts, security checks on the accounts, etc., is handled for them automatically via expansion of the Rust macros (essentially an eDSL) and the semantic meaning of various special types provided by the Anchor framework.

```rust
#[program]
pub mod turnstile {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        let state = &mut ctx.accounts.state;
        state.locked = true;
        state.res = false;
        Ok(())
    }
    #[allow(unused_variables)]
    pub fn coin(ctx: Context<UpdateState>, dummy_arg: String) -> Result<()> {
        let state = &mut ctx.accounts.state;
        state.locked = false;
        Ok(())
    }
    pub fn push(ctx: Context<UpdateState>) -> Result<()> {
        let state = &mut ctx.accounts.state;
        if state.locked {
            state.res = false;
        } else {
            state.locked = true;
            state.res = true;
        }
        Ok(())
    }
}
```

Listing 1.4: Definition of business logic using Anchor's eDSL

Anchor's IDL

When a program is compiled to the BPF bytecode and ready for deployment, an IDL is generated. This IDL (example in Listing 1.5) can be further used to generate code for tests or front-end applications interacting with the Solana blockchain using JavaScript on the client side.

```json
{
  "version": "0.1.0",
  "name": "turnstile",
  "instructions": [
    {
      "name": "initialize",
      "accounts": [
        {
          "name": "state",
          "isMut": true,
          "isSigner": true
        },
        {
          "name": "user",
          "isMut": true,
          "isSigner": true
        },
        {
          "name": "systemProgram",
          "isMut": false,
          "isSigner": false
        }
      ],
      "args": []
    },
  // ... rest of the IDL
  ]
}
```

Listing 1.5: An example of Anchor's IDL

1.2.2 Identified Problem

Anchor is able to generate Javascript/Typescript client program code from the IDL. The problem is it takes enormous effort to even send a single instruction calling one of the instructions.

The process of testing Solana programs is following:

1. Create a Solana program using Rust and Anchor Framework.
2. Compile into bytecode and generate an IDL.
3. Launch a local cluster and deploy the compiled program to it.
4. Write boilerplate code to prepare sending a transaction (Listing 1.6).

5. Use the generated IDL to generate an RPC client to be used within the boilerplate (Listing 1.6).

6. Execute the RPC with correct arguments.

```
const  anchor  =  require('@project-serum/anchor');
const  fs  =  require('fs')
// Configure the local cluster.
anchor.setProvider(anchor.Provider.local());
const  getPublicKey = (name) =>
        new anchor.web3.PublicKey(...);
const  getPrivateKey = (name) =>
        Uint8Array.from(...);
const  getKeypair = (name) =>
        new anchor.web3.Keypair(...);
async  function  coin()  {
        // Read the generated IDL.
        const  idl  =  JSON.parse(require('fs').readFileSync(
                    './target/idl/turnstile.json',  'utf8'));
        // Address of the deployed program.
        const  programId  =  getPublicKey("program");
        // Generate the program client from IDL.
        const  program  =  new anchor.Program(idl, programId);
        // Get keypair of the state as it is the input to the coin instr
        const  state  =  getKeypair("state");
        // Finally we execute the rpc
        const  tx  =  await program.rpc.coin({
                        accounts: {
                                state: state.publicKey,
                        },
                        signers: [],
                });
}
coin();
```

Listing 1.6: Client code in coin.js

This is a significant overhead for a developer. In order to send a single transaction executing one single instruction, you need to write about 30 lines of code in Javascript, as the other languages are not supported by the Anchor Framework.

In addition, the sequence of steps below is required for us to be able to call this script:

```
$ solana-keygen [OPTIONS] # keys generation into a ./keys dir
$ anchor build # builds an Anchor program and creates its IDL
$ solana-test-validator -C ./config.yml # starts a local chain
$ solana airdrop -C ./config.yml 1 ./keys/id.json # airdrop SOL to an account,
                                                   # so it can pay for deployment
$ solana program deploy -C ./config.yml --program-id ./keys/program.json
./target/deploy/turnstile.so # deploys a compiled program
$ ANCHOR_WALLET=./keys/id.json node coin.js # calls the script
```

Listing 1.7: Sequence of steps needed for coin.js execution

1.2.3 Value Proposition

Trdelnik, the Rust Testing Framework for Solana, aims to solve the problem by doing all of those things automatically. It consists of a CLI tool and a library with API to test Solana programs written in Anchor offering also various convenience features.

One of the major problems in the blockchain world is security and reliability are more and more prioritized. Trdelnik is written in Rust and helps to test code directly in Rust, which is also the language of the Solana programs, instead of using Javascript. A safe language like Rust should help minimize issues that would be left undiscovered using any other language for testing.

1.2.4 Features in Development

Trdelnik is still very early in development but already open-sourced and available on GitHub[10] for a download. It can be compiled from the source code, which, thanks to the Rust's Build and Package Management Tool Cargo, is quite a convenient process. The name Trdelnik is inspired by a tool called Brownie from the Ethereum ecosystem.

Furthermore, Trdelnik desires to be a set of convenient tools for developers, offering more features that help developers code Solana programs.

Trdelnik's developer tools currently in development include:

Trdelnik Client (first version delivered)
: Build and deploy an Anchor program to a local cluster, simplify the process of writing tests and run the test suite against the deployed program. This is the main motivation behind Trdelnik, as explained earlier.

[10]`https://github.com/Ackee-Blockchain/trdelnik`

Trdelnik Explorer (function proposed by **this thesis**)
Transaction and blockchain inspector, which helps to examine changes on a local cluster. Trdelnik Explorer is introduced in Chapter 2, "Function Proposal for Trdelnik".

Trdelnik Fuzz (currently being researched)
First built-in fuzzy tester of Solana programs. It automatically generates random data as input, so that developers do not need to write them themselves.

Trdelnik Console (currently being researched)
Built-in console to give developers a command prompt for quick program interaction allowing developers to call individual transactions or perform deployment interactively. See Listing 1.8 for an example.

```
$ trdelnik console
>>> program = Turnstile.deploy(...)
>>> program.initState()
>>> program.coin({accounts: {state: accounts[0]}})
>>> ...
```

Listing 1.8: Example from Trdelnik Console

Trdelnik Client

Trdelnik Client, as the main driving force behind Trdelnik and the only so-far delivered part of Trdelnik before this thesis.

Compare the following workflow with the previous way of testing Solana programs shown in Subsection 1.2.2, "Identified Problem":

1. Create a Solana program using Rust and Anchor Framework.
2. Run `trdelnik build` to create a `program_client` crate containing an auto-generated code for easy invocation of program instructions.
3. Write tests using the said crate and using `#[trdelnik_test]` macro (see an example in Listing 1.9).
4. Run `trdelnik test` to automatically launch a local cluster, deploy the program and run your test suite against it.

As can be seen, only the very first step stays the same. The rest of the process is conveniently streamlined and handled by Trdelnik. Developers can therefore focus only on the program itself and writing tests in the same language as their programs are written in without the previously needed Javascript boilerplate.

```rust
#[trdelnik_test]
async fn test_happy_path() {
    // create a test fixture
    let mut fixture = Fixture {
        client: Client::new(system_keypair(0)),
        program: program_keypair(1),
        state: keypair(42),
    };
    // deploy a tested program
    fixture.deploy().await?;

    // init instruction call
    turnstile_instruction::initialize(
        &fixture.client,
        fixture.state.pubkey(),
        fixture.client.payer().pubkey(),
        System::id(),
        Some(fixture.state.clone()),
    )
    .await?;
    // coint instruction call
    turnstile_instruction::coin(
        &fixture.client,
        "dummy_string".to_owned(),
        fixture.state.pubkey(),
        None,
    )
    .await?;
    // push instruction call
    turnstile_instruction::push(&fixture.client, fixture.state.pubkey(), None).await?;

    // check the test result
    let state = fixture.get_state().await?;

    // after pushing the turnstile should be locked
    assert!(state.locked);
    // the last push was successfull
    assert!(state.res);
}
```

Listing 1.9: Testing with Trdelnik Client

Chapter 2

Function Proposal for Trdelnik

This chapter introduces a proposal for a new function, Trdelnik Explorer, to be implemented as the second major feature of Trdelnik, the Rust Testing Framework for Solana, presented in the previous chapter.

2.1 The General Idea Behind Explorer

Trdelnik Client, as explained earlier, solves the problem of testing Solana programs. However, what remains complicated is observing the changes on a blockchain for the purpose of developers and security auditing companies. The program testing was just one part of the mosaic.

Some of the current obstacles when testing includes issues observing:

- Changes to accounts between transactions.
- How the programs were deployed and the implications.
- Identifying possible risk.
- Inspecting details of individual transactions.

Even though there are some consumer-facing explorers available on the market already, there is still a need for a convenient tool that can get all the important information from the blockchain, analyze it, and show it to the user in a way and a form suited for developers and security researchers. Consumers and developers generally differ in their needs, and the focus until now has been mostly on consumers.

This thesis proposes a new function called Trdelnik Explorer, which aims to solve this issue by providing its own library as a part of Trdelnik to analyze the ledger changes using a convenient API. Some of the parts of the library could be directly integrated with the Trdelnik CLI tool and used by the end-user.

2.2 Explorer Requirements

2.2.1 Functional Requirements

FR-01 – User Input Format

A user is able to insert any valid addresses for accounts:

- An Ed25519 public key.
- A program-derived account address (32byte value forced off the ed25519 curve).
- A hash of an ed25519 public key with a 32-character string.

When inspecting programs, any program id, which is an Ed25519 Public Key, is also processed.
For transactions, any valid transaction id is processed. That is always the first signature of a transaction.
In case of an invalid input or an account, program id, or transaction id posing no information, the user is always informed about this fact.

FR-02 – User Output Format

A user can choose from multiple output formats.
This list of output formats is comprised of:

- CLI format as the standard human-readable way to read the results.
- JSON format as both human-readable, but mainly machine-readable data format that can be further processed.
- JSONPretty format as the prettified variant of the JSON format modified by adding spaces and newlines to make this an easy-to-read pretty-printed JSON.

FR-03 – Account Inspection

The explorer is able to fully inspect accounts given their address. That means all information about any account can be easily observed, and changes before and after a transaction are analyzed.

FR-04 – Program Inspection

The explorer is able to fully inspect programs given their program id. Furthermore, the explorer is able to identify the way the program was deployed and show further information based on this analysis.

This is especially important for distinguishing between non-upgradeable and upgradeable programs, which pose additional risk to end-users because of a single point of failure and required trust.

FR-05 – Transaction Inspection

The explorer is also able to fully inspect all transactions given their transaction id. This is by far the most complicated and most insightful inspection.

The complexity of Solana transactions sometimes requires multiple different views of a transaction from a different perspective. Therefore, the explorer provides two ways of inspecting transactions:

- Raw View to show the transaction in its raw form without any possibly misleading processing or interpretation of data.
- Interpreted View to show a processed transaction that tries to provide as much interpreted information about the transaction, invoked programs, the actual instructions, and logs.

FR-06 – Account Differences

The explorer shall be able to show differences before and after a transaction happens to observe the effects of a program instruction invocation or be able to show differences between any two accounts.

FR-08 – Instruction Parser

As there is no standardized way to publish your program API, the explorer during the transaction inspection described in FR-05 should still try to deserialize and parse as many program instructions as possible using heuristics or other methods.

FR-09 – Custom Program Instruction Parser

The explorer shall present a way to add any program to the instruction parser, for example, for purposes of auditing undisclosed and yet unreleased programs.

FR-10 – Account Deserialization

If the account structure and serialization format is known, the explorer should be able to deserialize the account data.

FR-11 – Cluster Support

The explorer works with all available Solana clusters, and users can change the targeted cluster easily. The most common clusters include:

- Beta Mainnet – the official Solana cluster that is not meant to be reverted in the future.
- Devnet – an official Solana cluster to help developers test their programs on a mainnet-like network, sometimes with new features that are yet to be introduced to the mainnet.
- Testnet – an official Solana cluster for testing new Solana versions and features.
- Localnet – any local Solana cluster launched, usually used for testing before going to Devnet.

FR-12 – Granular Visibility

The explorer allows setting certain granularity to the output presented to the user. The purpose of this requirement is to hide not needed information and allow the user to focus only on what currently matters.

2.2.2 Non-functional Requirements

NFR-01 – Programming Language

The explorer shall be programmed in the Rust programming language so that it can be easily integrated with the Trdelnik CLI or used by other tools in the Solana ecosystem, which is focused on the Rust ecosystem.

NFR-02 – Platform

Any desktop computer with an architecture supported by the Rust compiler and LLVM backend.

NFR-03 – Usage

The explorer is meant to be used within the Trdelnik CLI or standalone as a library.

NFR-04 – Demands

The explorer shall be very lightweight, always running locally and without the need for a database.

NFR-05 – Extensibility

It should be relatively easy to extend the explorer with more features in the future.

NFR-06 – Logging

Invalid inputs, non-existing accounts, programs or transactions, failures during node communication, or any other problems are reported to the user.

2.3 Use Cases

Use cases are summarized in Figure 2.1 as a use case diagram.

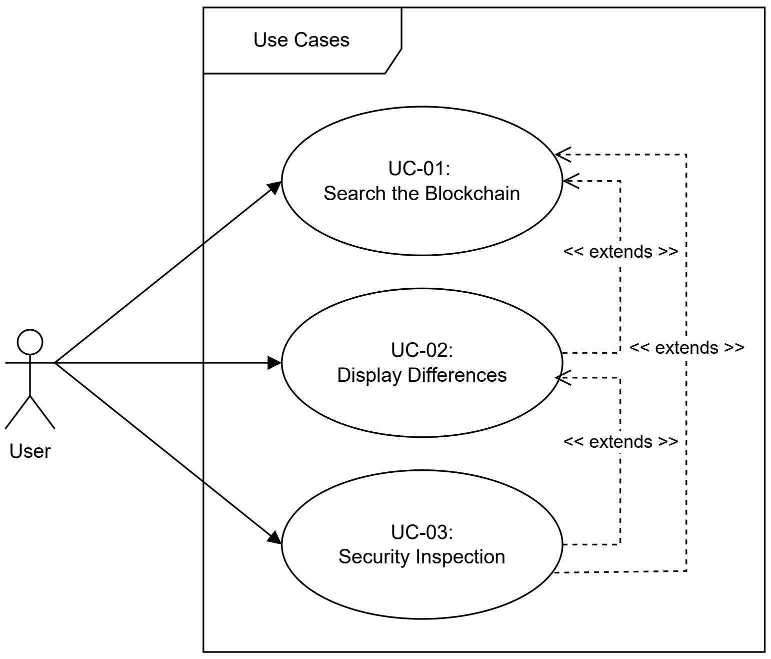


Figure 2.1: Use case diagram

2.3.1 UC-01 – Search the Blockchain

The use case enables the user to search for and show parts of the Solana blockchain, such as accounts, programs, and transactions.

2.3.2 UC-02 – Display Differences

The use case enables the user to compare changes to the Solana ledger and how transactions affect the state.

2.3.3 UC-03 – Security Inspection

The use case enables the user to inspect details of the program instruction invocations, state changes, and adding unknown instructions in the case of undisclosed programs being audited and alerted about risks.

Chapter 3

Implementation

This chapter covers the architecture and implementation of the Explorer functionality for Trdelnik, the Rust Testing Framework for Solana, and its integration with the Trdelnik CLI tool. Trdelnik Explorer is meant to be used as both a library and directly from the Trdelnik CLI.

3.1 Technology Used

Rust Programming Language

Rust is a new modern programming language designed for both performance and safety at the same time. Its design allows programmers not to worry about manual memory management, yet there is no Garbage Collection. Thanks to the design of the language, the mistakes and common undefined behavior known from languages, such as C or C++ are caught during compilation time. Rust's zero-cost abstractions result in comparable performance to code written in C or C++ and are slowly becoming their contender even in the field of High-Performance Computing.

Rust has recently become very popular in cryptocurrency projects for the said safety and performance. In Solana, the whole codebase, all on-chain programs, and most tooling are written in Rust. Therefore it makes sense to continue in the same manner, as others are able to easily use your work in their projects. Also, for integration with Trdelnik, it is a necessity to code the Explorer in Rust, so it is an obvious choice here.

As not many are familiar with Rust, to not get lost in its terminology, let some of the key terms of a typical Rust project be repeated. Simply put, the compilation unit in Rust is called a crate. A library is typically called a library crate, and an executable is called a binary crate. Each crate consists of modules, which can be considered namespaces to a certain extent and help divide the crate into many logical parts. Crates can be a part of a larger Rust project that we call a Workspace. The whole project management and the building are handled by the tool named Cargo.

Tokio Runtime
Tokio is an asynchronous runtime for the Rust programming language ideal for writing network applications.

Solana Blockchain
Solana blockchain is used whenever the tool is used, mainly by connecting to one of the RPC listening nodes and talking to them via their JSON-RPC protocol or WebSocket endpoint. Implementing the explorer requires extensive knowledge of Solana internals, programming model, and some of its core concepts. Without a Solana cluster (network), the tool has no use.

Solana Client
A convenient wrapper around the JSON-RPC protocol and the WebSocket protocol used by Solana nodes. Developers are thus not required to write their own clients using HTTP libraries to communicate with nodes when using Rust to write their tools.

Solana SDK
An important dependency of `Solana Client` contains most of the important type definitions, constants, and convenient functions.

3.2 Explorer Crate

The explorer crate is comprised of multiple modules, each serving as a logical part of the explorer. The crate also tries to make use of the Rust generics to simplify the code, readability and future refactoring, and easy extensibility:

Config module
Module serving for the purpose of creating a configuration (a snippet shown in Listing 3.1) that can be used later within the Explorer API.

Two main configuration options include:

- Creation and setting up of an Async RPC client to talk to the predefined Solana cluster via an RPC node. By default using the developer's preferred cluster set in the configuration file of the official Solana Tool Suite[11].
- Logger settings, by default set to the `ERROR` level.

[11] `https://docs.solana.com/cli/install-solana-cli-tools`

```
...
pub struct ExplorerConfig {
    json_rpc_url: String,
    rpc_client: RpcClient,
}
pub fn setup_logging(level: LogLevel) {
    match level {
        LogLevel::ERROR => solana_logger::setup_with_default("error"),
        LogLevel::WARN => solana_logger::setup_with_default("warn"),
        LogLevel::INFO => solana_logger::setup_with_default("info"),
        LogLevel::DEBUG => solana_logger::setup_with_default("debug"),
        LogLevel::TRACE => solana_logger::setup_with_default("trace"),
    }
}
...
```

Listing 3.1: Config module snippet

Error module

Module serving as a place for error definitions and the common return type definition of fallible functions (a snippet shown in Listing 3.2). Transformation of errors returned from the libraries to the common error type used throughout the whole explorer crate is handled by this module. The errors can be caught by a user of the explorer's crate or shown to the user of the Trdelnik CLI in case of a failure.

```
pub type Result<T> = std::result::Result<T, ExplorerError>;
#[derive(Debug, Error)]
pub enum ExplorerError {
    #[error("{0}")]
    SolanaClient(#[from] ClientError),
    #[error("{0}")]
    SerdeJson(#[from] SerdeError),
    #[error("{0}")]
    Fmt(#[from] FmtError),
    #[error("{0}")]
    Instruction(#[from] InstructionError),
    #[error("{0}")]
    Custom(String),
}
```

Listing 3.2: Error module snippet

Display module

Module using generics to allow pretty printing. As can be seen in Listing 3.3, there are three variants of the `DisplayFormat` enum allowing to get a string output of any explorer item `T` implementing `fmt::Display` and `Serialize` traits.

```
#[derive(Clone, Copy)]
pub enum DisplayFormat {
    Cli,
    JSONPretty,
    JSON,
}
impl DisplayFormat {
    pub fn formatted_string<T>(&self, item: &T) -> Result<String>
    where
        T: fmt::Display + Serialize,
    {
        match self {
            DisplayFormat::Cli => Ok(format!("{}", item)),
            DisplayFormat::JSONPretty => Ok(serde_json::to_string_pretty(&item)?),
            DisplayFormat::JSON => Ok(serde_json::to_string(&item)?),
        }
    }
}
```

Listing 3.3: Display module snippet

Output module

Module with the main public API of the explorer library, consisting of many public functions that can be used as needed for specific use cases. Some of them are explicitly used in the Trdelnik CLI tool integration.

The snippet shown in Listing 3.4 manifests the public API of this module and one full function signature and function definition of the `print_transaction` as an example.

It can be observed it takes a signature that uniquely identifies a transaction. Then it takes the next parameter defining what to show, another parameter that defines the display format, and finally, the explorer configuration specifying the Solana cluster to talk to.

```
...
pub async fn print_transaction(
    signature: &Signature,
    visibility: &TransactionFieldVisibility,
    format: DisplayFormat,
    config: &ExplorerConfig,
) -> Result<()> {
    let transaction_string = get_transaction_string(signature, visibility,
                                                    format, config).await?;
    println!("{}", transaction_string);
    Ok(())
}
pub async fn print_raw_transaction(...) -> Result<()> {...}
pub async fn print_account(...) -> Result<()> {...}
pub async fn print_program(...) -> Result<()> {...}
pub async fn get_transaction_string(...) -> Result<String> {...}
pub async fn get_raw_transaction_string(...) -> Result<String> {...}
pub async fn get_account_string(...) -> Result<String> {...}
pub async fn get_program_string(...) -> Result<String> {...}
pub fn classify_account(...) -> String {...}
pub fn calculate_change(...) -> String {...}
pub fn pretty_lamports_to_sol(...) -> String {...}
pub fn change_in_sol(...) -> String {...}
pub fn status_to_string(...) -> String {...}
```

Listing 3.4: Output module snippet

Account, Program and Transaction modules

Modules with important data structures regarding Account, Program, and Transaction representations, granular visibility, and interpretation of data. The most complicated one is the Transaction module, as the transaction is the most complex structure in Solana and things like instruction deserialization are not trivial (without it, it is just raw binary data). The transaction module makes use of the parser module for this reason.

Parser module and its submodules

Module dedicated to parsing of program instructions invoked in transactions, an important part of the Transaction deserialization and interpretation.

This is especially a tricky part, as there is no standard way to publish your program API in Solana, and without proper interpretation, we just see some raw data, not knowing what it actually does.

The chosen approach to tackle this problem is to remember the most used programs in Solana as a lookup table (Listing 3.6), which is a hashmap built during compilation time that helps identify program instructions when processing transactions. Recognized program instructions are then decoded and interpreted according to their known attributes. If the program cannot be recognized, we only show the information about which accounts are used within the instruction and the raw instruction data.

Deserialization and interpreting logic of the known programs is implemented as submodules of the Parser module.

```
pub enum DisplayInstruction {
    Parsed(DisplayParsedInstruction),
    PartiallyParsed(DisplayPartiallyParsedInstruction),
}
impl DisplayInstruction {
    fn parse(instruction: &CompiledInstruction, account_keys: &[Pubkey]) -> Self {
        let program_id = &account_keys[instruction.program_id_index as usize];
        if let Ok(parsed_instruction) = parse(program_id, instruction, account_keys) {
            DisplayInstruction::Parsed(parsed_instruction)
        } else {
            DisplayInstruction::PartiallyParsed(partially_parse(
                program_id,
                instruction,
                account_keys,
            ))
        }
    }
}
```

Listing 3.5: Instruction parser logic

There are some first initiatives, started mainly by the Anchor Framework, to upload a program's IDL to deterministic addresses on blockchain, which could help us to automatize the whole process, and all such programs could be automatically recognized, decoded, and interpreted in the future. The explorer can be easily extended to support this feature in the future. Listing 3.5 shows the processing logic of an instruction parser.

```
static PARSABLE_PROGRAM_IDS: phf::Map<&'static str, ParsableProgram> = phf_map!
{
    // System
    "11111111111111111111111111111111" => ParsableProgram::System,
    // BPF Loader Deprecated
    "BPFLoader1111111111111111111111111111111111" =>
    ParsableProgram::BPFLoaderDeprecated,
    // BPF Loader
    "BPFLoader2111111111111111111111111111111111" =>
    ParsableProgram::BPFLoader,
    // BPF Loader Upgradeable
    "BPFLoaderUpgradeab1e11111111111111111111111" =>
    ParsableProgram::BPFLoaderUpgradeable,
    // Stake
    "Stake11111111111111111111111111111111111111" => ParsableProgram::Stake,
    // Vote
    "Vote111111111111111111111111111111111111111" => ParsableProgram::Vote,
    // SPL Memo v1
    "Memo1UhkJRfHyvLMcVucJwxXeuD728EqVDDwQDxFMNo" => ParsableProgram::SPLMemoV1,
    // SPL Memo (current)
    "MemoSq4gqABAXKb96qnH8TysNcWxMyWCqXgDLGmfcHr" => ParsableProgram::SPLMemo,
    // SPL Token
    "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA" => ParsableProgram::SPLToken,
    // SPL Associated Token Account
    "ATokenGPvbdGVxr1b2hvZbsiqW5xWH25efTNsLJA8knL" =>
    ParsableProgram::SPLAssociatedTokenAccount
};
```

Listing 3.6: Lookup table of parsable programs

3.3 Trdelnik Integration

The explorer can be used standalone as a library. However, it is desirable to have some of the features available even as a part of the Trdelnik CLI tool.

The Trdelnik CLI tool has been, therefore, properly extended in its `trdelnik-cli` crate to support the most frequent use cases from the command line. Concrete examples can be seen in Appendix A, “CLI Examples”.

Chapter 4

Testing

4.1 Testing in Rust

Rust natively supports both unit and integration testing. In Rust, tests are just special user-written functions that verify the correctness of the non-test code.

The only difference is that in order to get skipped during normal compilation, they must be marked accordingly. An example of a unit test of a simple module consisting of one function is shown in Listing 4.1.

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;
    // additional use statements

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }

    // additional tests
    // #[test]
    // ...
}
```

Listing 4.1: Rust unit test example

4.2 Unit Testing

Parser is an important part of the explorer and handles deserialization and interpretation of parsable program instructions as explained in Chapter 3.

To verify the correctness of the implementation of the parser, every submodule of the parser module has been properly covered by unit tests to make sure that all kinds of instruction types are recognized properly.

Table 4.1 shows the test coverage of all parser submodules. It can be observed that the coveraged has reached 97.72% and Listing 4.2 demonstrates the testing in practice.

The only missing unit test is a unit test of a new marginal instruction of the stake program, which has not been yet properly documented. The test can be added when there is more information available to construct a proper unit test.

```
$ cargo test -p trdelnik-explorer
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests src/lib.rs (target/debug/deps/trdelnik_explorer-
                                                  839c9fa65fe2aebe)
running 43 tests
... (skipped for the purpose of this thesis) ...
test parse::vote::test::test_parse_vote_switch_ix ... ok
test parse::vote::test::test_parse_vote_withdraw_ix ... ok
test parse::vote::test::test_parse_vote_update_validator_identity_ix ... ok

test result: ok. 43 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.01s
```

Listing 4.2: Unit testing in practice

Submodule	# of IXs Tested	# of IXs	Test Coverage
associated_token_account	1	1	100%
bpf_loader	2	2	100%
bpf_upgradeable_loader	7	7	100%
memo	1	1	100%
stake	12	13	92.3%
system	12	12	100%
token	1	1	100%
vote	8	8	100%
Overall	43	44	97.72%

Table 4.1: Parser submodules coverage

4.3 Integration Testing

Given the nature of public blockchains and possible state changes for items being tested, which can change at any moment, it is not trivial to automatize testing.

One way to do it would be to simulate the whole blockchain locally with certain testing transactions.

Intending to avoid that and because of major coding overhead and intangible value of these tests as just described, it has been concluded that except for the Parser unit testing, the explorer would had undergone mostly manual and semi-manual testing to validate the overall correctness of the output.

4.4 Functional and User Testing

The explorer has been tested against the Solana beta mainnet for all types of blockchain items, i.e., accounts, programs, transactions, and for all kinds of output to evaluate correctness.

- Account testing:
  - System owned data accounts (with no data, ≤ 64B, > 64B)
  - Program owned PDA accounts (with no data, ≤ 64B, > 64B)
  - Program accounts (owned by BPF Loader deprecated, BPF Loader and BPF Upgradeable Loader)
  - Native accounts (native programs and special runtime accounts)
- Testing of programs deployed using:
  - BPF Loader deprecated
  - BPF Loader
  - BPF Upgradeable Loader
- Transaction testing:
  - Transactions with program instructions matching known parsable programs according to Listing 3.6.
  - A random sample of transactions from the network.

The explorer has also undergone internal user testing and has been improved according to the remarks of Ackee Blockchain who helped to supervise this thesis.

Conclusion

The goal of the thesis was to introduce the Solana blockchain and Trdelnik, the Rust Testing Framework for Solana programs. Furthermore, new functionality for Trdelnik was proposed and implemented.

The reader is first introduced to some of the key blockchain terms, followed by an introduction to Solana, where the original motivation and proposition of Solana are described. The analysis chapter then explains all core technical concepts Solana is based on and how it differs from other blockchains, mainly Ethereum. For new Solana programmers, the unique programming model is essential, as it is very different from other blockchains, and for example, programmers with experience in developing on Ethereum cannot simply switch to Solana without changing the way they think about smart contracts. As the core reason why Solana even exists and what is currently possible, some of the most important Solana ecosystem projects are presented.

The reader is also introduced to Trdelnik, the Rust Testing Framework for Solana programs. It should be obvious that the current state of development on Solana is an error-prone, tedious, and lengthy process, which is what frameworks such as Anchor, together with Trdelnik, try to change. Trdelnik focuses on the testing part of the process, as a single mistake in this adversary space can easily cost millions of dollars in losses. The features currently being worked on Trdelnik are introduced.

The new functionality for Trdelnik is proposed. This feature is called Trdelnik Explorer, or simply put explorer, and aims to be a swiss knife for inspecting the blockchain changes between transactions. It offers convenient ways for security inspection of transactions and the effects they have on the data stored in accounts, which is roughly an equivalent of files from file systems, but on Solana. Most importantly, functional and non-functional requirements and use cases are presented.

Finally, the proposed explorer feature is implemented. The description includes the technology used and reasoning. All modules in the explorer crate are examined with some code snippets to provide a better explanation. The explorer is implemented as a library, but since Trdelnik has its own CLI tool, some of its most frequent functions are integrated with it.

Future Work

Trdelnik Explorer and overall the whole Trdelnik Framework are still early in development, and there is a lot to be researched and developed. Especially for the Trdelnik Fuzz and Trdelnik Console features that are yet to be started being developed.

Even though the development is not finished, with the solid ground created by the first two major features, Trdelnik Client and Trdelnik Explorer, proposed and implemented in this thesis, it has a good starting point and future potential to become one of the go-to standard tools of any Solana developer. As an open-source tool with a public repository on GitHub[12], any contributors are welcome to join the project and help with development.

[12]`https://github.com/Ackee-Blockchain/trdelnik`

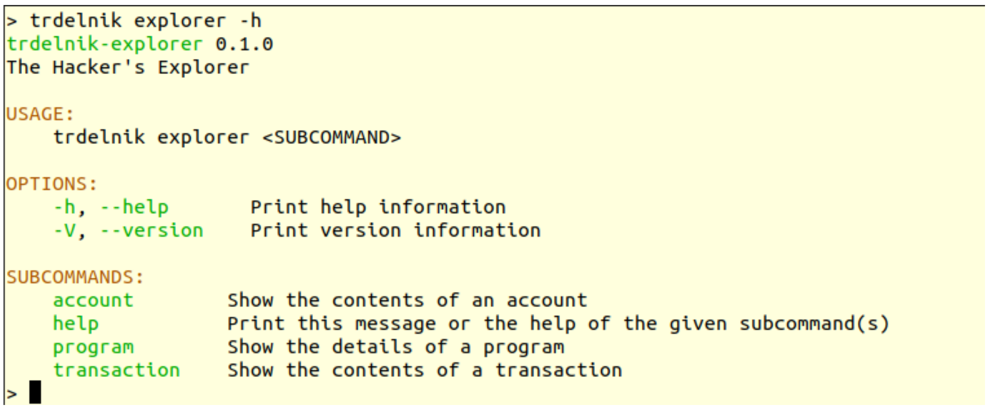
Bibliography

1. BUTERIN, Vitalik. *Sharding FAQ* [online]. 2017 [visited on 2022-04-07]. Available from: `https://vitalik.ca/general/2017/12/31/sharding_faq.html`.
2. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008 [visited on 2022-04-07]. Available from: `https://bitcoin.org/bitcoin.pdf`.
3. BUTERIN, Vitalik. *Ethereum Whitepaper* [online]. 2014 [visited on 2022-04-12]. Available from: `https://ethereum.org/en/whitepaper/`.
4. KOZÁK, Lukáš. *Security Analysis of Hardware Crypto Wallets*. 2020. Bachelor's Thesis. Czech Technical University in Prague, Faculty of Information Technology.
5. BECKER, Georg. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep.* 2008, vol. 12, p. 19.
6. CASTRO, Miguel; LISKOV, Barbara, et al. Practical byzantine fault tolerance. In: *OsDI*. 1999, vol. 99, pp. 173–186. No. 1999.
7. ROCKET, Team; YIN, Maofan; SEKNIQI, Kevin; RENESSE, Robbert van; SIRER, Emin Gün. *Scalable and Probabilistic Leaderless BFT Consensus through Metastability*. arXiv, 2019. Available from DOI: `10.48550/ARXIV.1906.08936`.
8. ANTONOPOULOS, Andreas M. *Mastering Bitcoin: Programming the Open Blockchain*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1491954386.
9. ANTONOPOULOS, Andreas M.; WOOD, Gavin. *Mastering Ethereum: Building Smart Contracts and DApps*. Sebastopol, CA: O'Reilly Media, 2018. ISBN 978-1491971949.
10. YAKOVENKO, Anatoly. *Solana: A new architecture for a high performance blockchain* [online]. 2017 [visited on 2022-04-07]. Available from: `https://solana.com/solana-whitepaper.pdf`.

11. SOLANA. *Documentation: History* [online]. 2020 [visited on 2022-04-08]. Available from: `https://docs.solana.com/history`.
12. BACH, Leo Maxim; MIHALJEVIC, Branko; ZAGAR, Mario. Comparative analysis of blockchain consensus algorithms. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 1545–1550.
13. SOLANA. *Documentation: Turbine Block Propagation* [online]. 2022 [visited on 2022-04-09]. Available from: `https://docs.solana.com/cluster/turbine-block-propagation`.
14. BLOCKCHAIN.COM. *Ethereum Mempool* [online]. 2022 [visited on 2022-04-09]. Available from: `https://www.blockchain.com/charts/mempool-size`.
15. SOLANA. *Documentation: Validator – TPU* [online]. 2022 [visited on 2022-04-09]. Available from: `https://docs.solana.com/validator/tpu`.
16. YAKOVENKO, Anatoly. *Cloudbreak: Solana's Horizontally Scaled State Architecture* [online]. 2019 [visited on 2022-04-09]. Available from: `https://solana.com/news/cloudbreak---solana-s-horizontally-scaled-state-architecture`.
17. SOLANA. *Documentation: Runtime* [online]. 2022 [visited on 2022-04-10]. Available from: `https://docs.solana.com/developing/programming-model/runtime`.

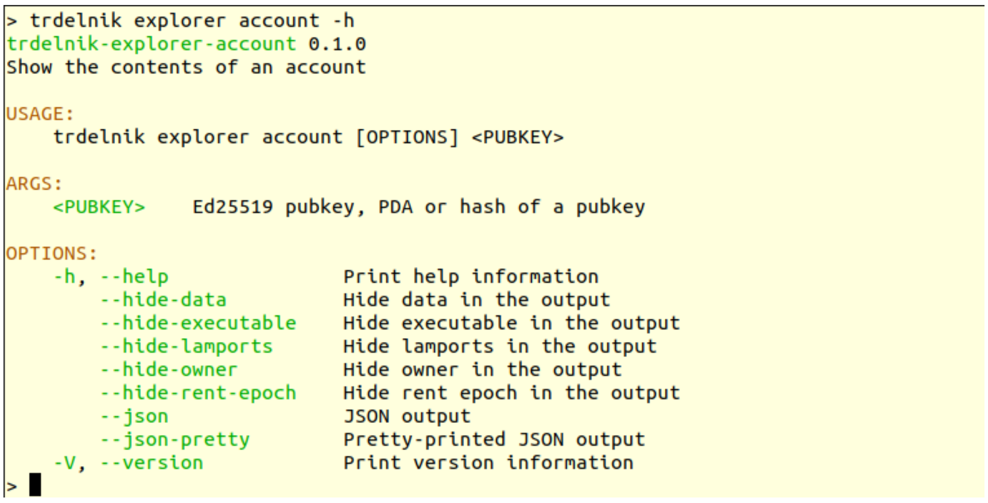
Appendix A

CLI Examples



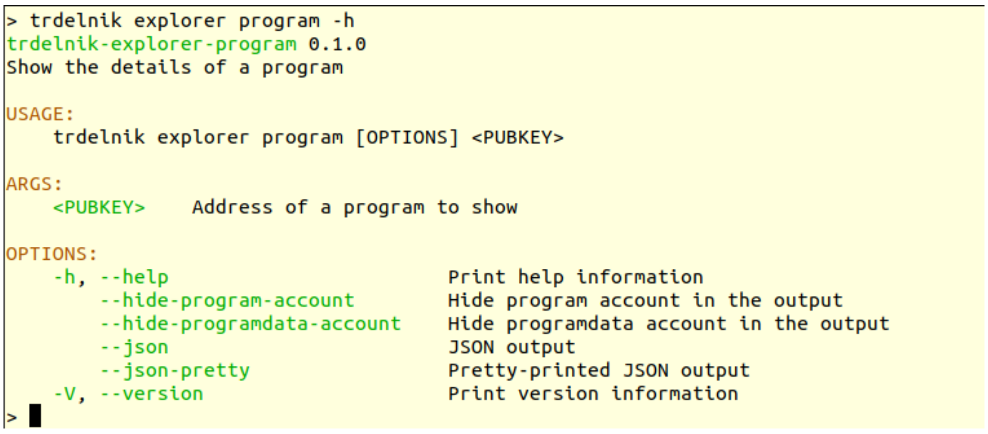
```
> trdelnik explorer -h
trdelnik-explorer 0.1.0
The Hacker's Explorer

USAGE:
    trdelnik explorer <SUBCOMMAND>

OPTIONS:
    -h, --help       Print help information
    -V, --version    Print version information

SUBCOMMANDS:
    account        Show the contents of an account
    help           Print this message or the help of the given subcommand(s)
    program        Show the details of a program
    transaction    Show the contents of a transaction
>
```

Figure A.1: Trdelnik Explorer help message



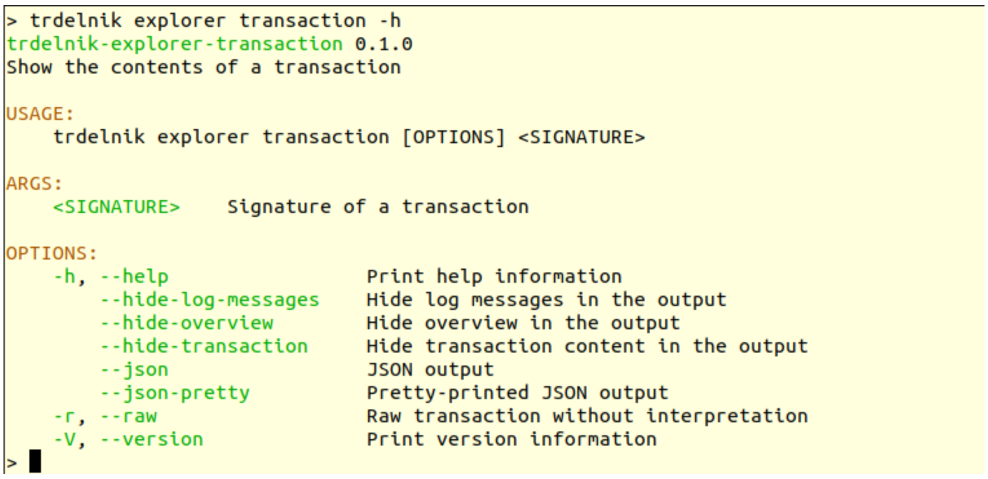
```
> trdelnik explorer account -h
trdelnik-explorer-account 0.1.0
Show the contents of an account

USAGE:
    trdelnik explorer account [OPTIONS] <PUBKEY>

ARGS:
    <PUBKEY>    Ed25519 pubkey, PDA or hash of a pubkey

OPTIONS:
    -h, --help                 Print help information
        --hide-data            Hide data in the output
        --hide-executable      Hide executable in the output
        --hide-lamports        Hide lamports in the output
        --hide-owner           Hide owner in the output
        --hide-rent-epoch      Hide rent epoch in the output
        --json                 JSON output
        --json-pretty          Pretty-printed JSON output
    -V, --version              Print version information
> █
```

Figure A.2: Account subcommand help message



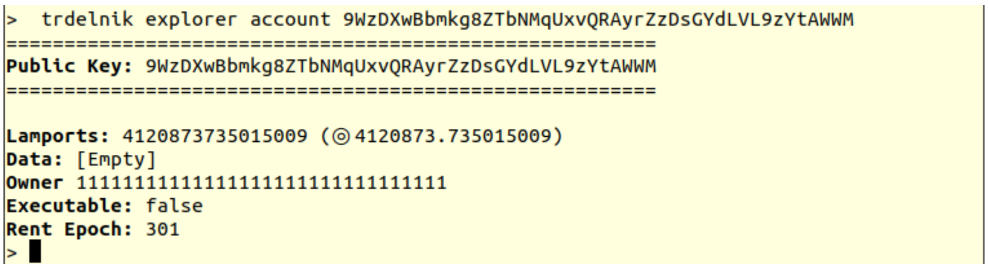
```
> trdelnik explorer program -h
trdelnik-explorer-program 0.1.0
Show the details of a program

USAGE:
    trdelnik explorer program [OPTIONS] <PUBKEY>

ARGS:
    <PUBKEY>    Address of a program to show

OPTIONS:
    -h, --help                          Print help information
        --hide-program-account          Hide program account in the output
        --hide-programdata-account      Hide programdata account in the output
        --json                          JSON output
        --json-pretty                   Pretty-printed JSON output
    -V, --version                       Print version information
> █
```

Figure A.3: Program subcommand help message



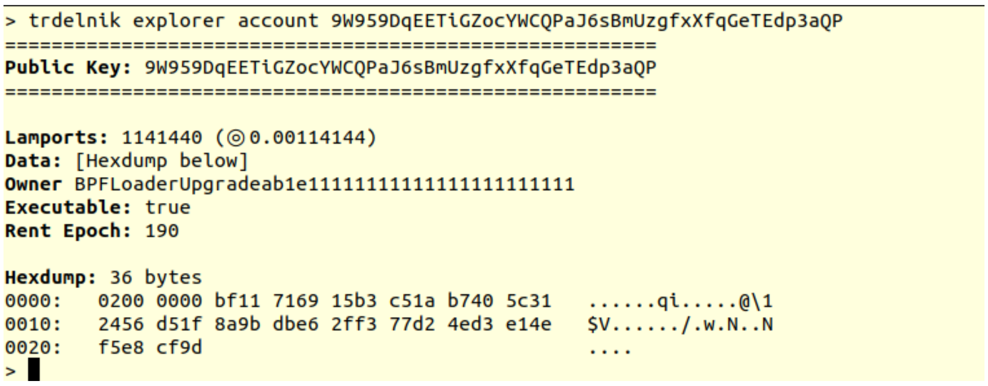
```
> trdelnik explorer transaction -h
trdelnik-explorer-transaction 0.1.0
Show the contents of a transaction

USAGE:
    trdelnik explorer transaction [OPTIONS] <SIGNATURE>

ARGS:
    <SIGNATURE>    Signature of a transaction

OPTIONS:
    -h, --help                 Print help information
        --hide-log-messages    Hide log messages in the output
        --hide-overview        Hide overview in the output
        --hide-transaction     Hide transaction content in the output
        --json                 JSON output
        --json-pretty          Pretty-printed JSON output
    -r, --raw                  Raw transaction without interpretation
    -V, --version              Print version information
> 
```

Figure A.4: Transaction subcommand help message



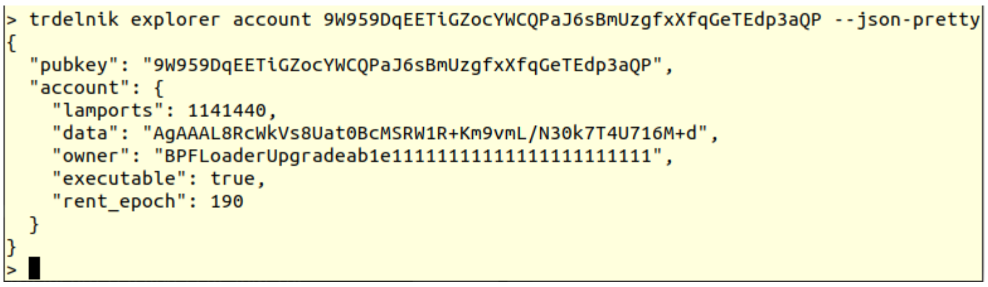
```
>  trdelnik explorer account 9WzDXwBbmkg8ZTbNMqUxvQRAyrZzDsGYdLVL9zYtAWWM
========================================================
Public Key: 9WzDXwBbmkg8ZTbNMqUxvQRAyrZzDsGYdLVL9zYtAWWM
========================================================

Lamports: 4120873735015009 (◎4120873.735015009)
Data: [Empty]
Owner 11111111111111111111111111111111
Executable: false
Rent Epoch: 301
> 
```

Figure A.5: Account subcommand example with empty data



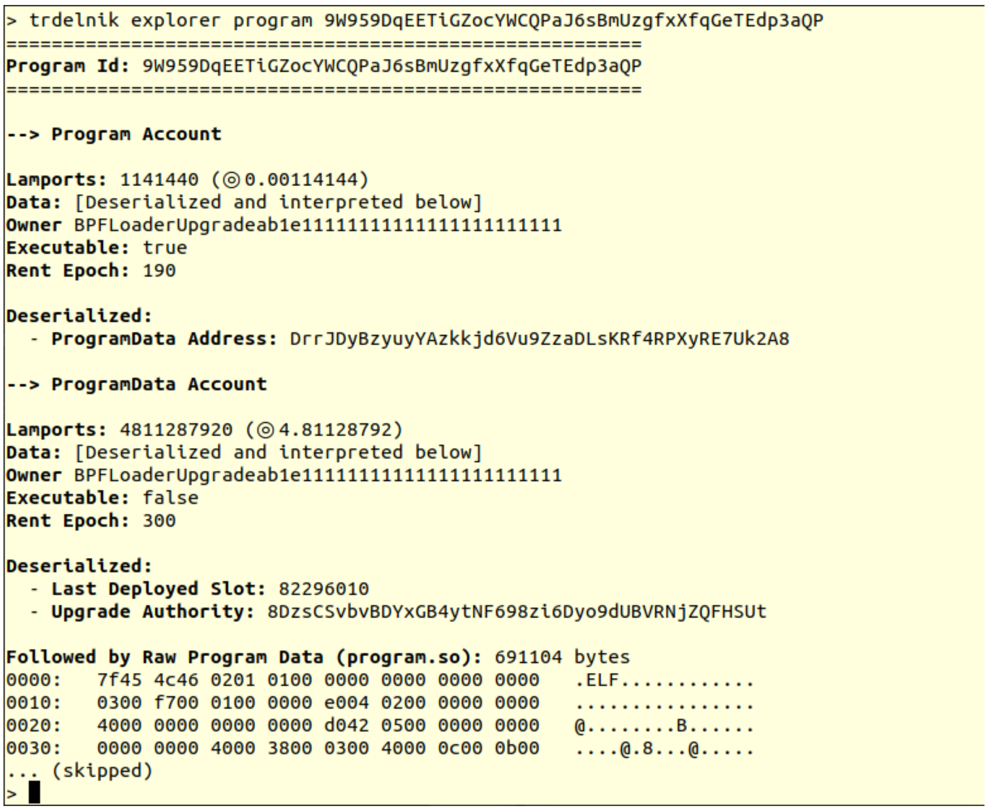
```
> trdelnik explorer account 9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP
========================================================
Public Key: 9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP
========================================================

Lamports: 1141440 (◎0.00114144)
Data: [Hexdump below]
Owner BPFLoaderUpgradeab1e11111111111111111111111
Executable: true
Rent Epoch: 190

Hexdump: 36 bytes
0000:   0200 0000 bf11 7169 15b3 c51a b740 5c31   ......qi.....@\1
0010:   2456 d51f 8a9b dbe6 2ff3 77d2 4ed3 e14e   $V....../.w.N..N
0020:   f5e8 cf9d                                 ....
> 
```

Figure A.6: Account subcommand example with data



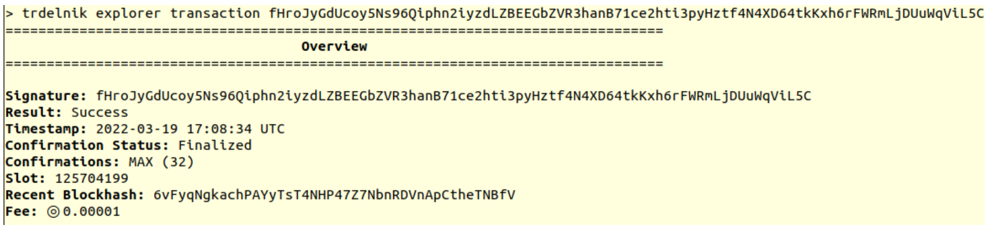
```
> trdelnik explorer account 9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP --json-pretty
{
  "pubkey": "9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP",
  "account": {
    "lamports": 1141440,
    "data": "AgAAAL8RcWkVs8Uat0BcMSRW1R+Km9vmL/N30k7T4U716M+d",
    "owner": "BPFLoaderUpgradeab1e11111111111111111111111",
    "executable": true,
    "rent_epoch": 190
  }
}
> 
```

Figure A.7: Account subcommand example with JSONPretty output



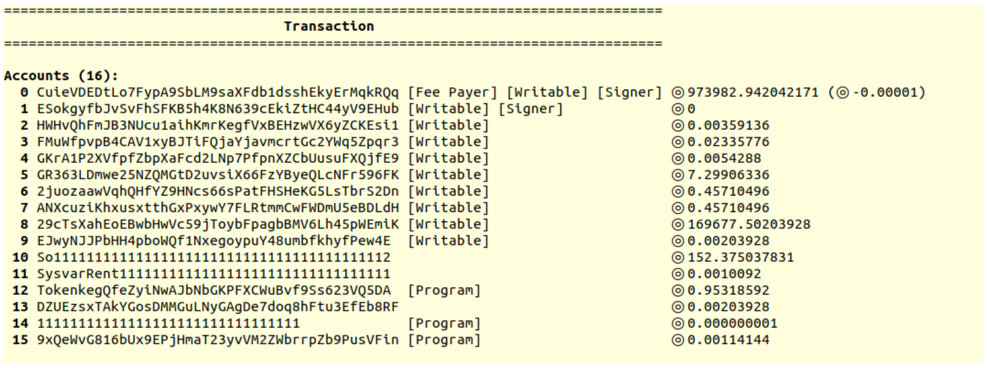
```
> trdelnik explorer program 9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP
========================================================
Program Id: 9W959DqEETiGZocYWCQPaJ6sBmUzgfxXfqGeTEdp3aQP
========================================================

--> Program Account

Lamports: 1141440 (◎0.00114144)
Data: [Deserialized and interpreted below]
Owner BPFLoaderUpgradeab1e11111111111111111111111
Executable: true
Rent Epoch: 190

Deserialized:
  - ProgramData Address: DrrJDyBzyuyYAzkkjd6Vu9ZzaDLsKRf4RPXyRE7Uk2A8

--> ProgramData Account

Lamports: 4811287920 (◎4.81128792)
Data: [Deserialized and interpreted below]
Owner BPFLoaderUpgradeab1e11111111111111111111111
Executable: false
Rent Epoch: 300

Deserialized:
  - Last Deployed Slot: 82296010
  - Upgrade Authority: 8DzsCSvbvBDYxGB4ytNF698zi6Dyo9dUBVRNjZQFHSUt

Followed by Raw Program Data (program.so): 691104 bytes
0000:   7f45 4c46 0201 0100 0000 0000 0000 0000   .ELF............
0010:   0300 f700 0100 0000 e004 0200 0000 0000   ................
0020:   4000 0000 0000 0000 d042 0500 0000 0000   @........B......
0030:   0000 0000 4000 3800 0300 4000 0c00 0b00   ....@.8...@.....
... (skipped)
> 
```

Figure A.8: Program subcommand example



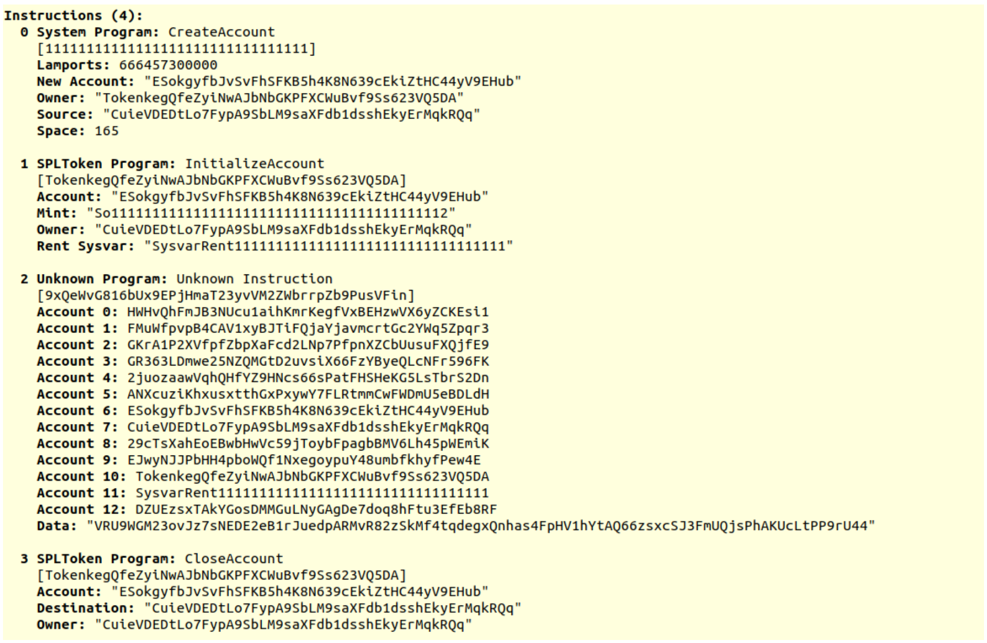
```
> trdelnik explorer transaction fHroJyGdUcoy5Ns96Qiphn2iyzdLZBEEGbZVR3hanB71ce2hti3pyHztf4N4XD64tkKxh6rFWRmLjDUuWqViL5C
=================================================================================
                                    Overview
=================================================================================

Signature: fHroJyGdUcoy5Ns96Qiphn2iyzdLZBEEGbZVR3hanB71ce2hti3pyHztf4N4XD64tkKxh6rFWRmLjDUuWqViL5C
Result: Success
Timestamp: 2022-03-19 17:08:34 UTC
Confirmation Status: Finalized
Confirmations: MAX (32)
Slot: 125704199
Recent Blockhash: 6vFyqNgkachPAYyTsT4NHP47Z7NbnRDVnApCtheTNBfV
Fee: ◎0.00001
```

Figure A.9: Overview part of the transaction subcommand



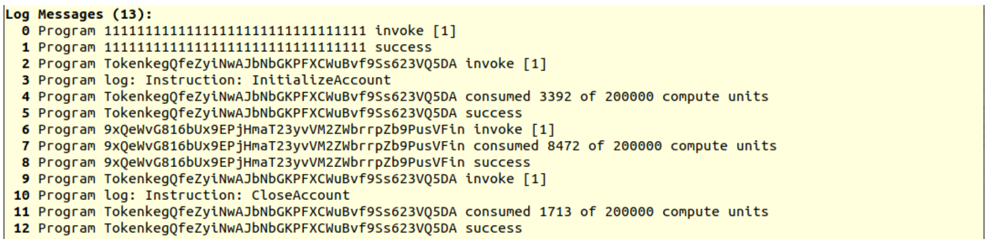
```
================================================================================
                                  Transaction
================================================================================

Accounts (16):
  0 CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq [Fee Payer] [Writable] [Signer] ◎973982.942042171 (◎-0.00001)
  1 ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub [Writable] [Signer]             ◎0
  2 HWHvQhFmJB3NUcu1aihKmrKegfVxBEHzwVX6yZCKEsi1 [Writable]                      ◎0.00359136
  3 FMuWfpvpB4CAV1xyBJTiFQjaYjavmcrtGc2YWq5Zpqr3 [Writable]                      ◎0.02335776
  4 GKrA1P2XVfpfZbpXaFcd2LNp7PfpnXZCbUusuFXQjfE9 [Writable]                      ◎0.0054288
  5 GR363LDmwe25NZQMGtD2uvsiX66FzYByeQLcNFr596FK [Writable]                      ◎7.29906336
  6 2juozaawVqhQHfYZ9HNcs66sPatFHSHeKG5LsTbrS2Dn [Writable]                      ◎0.45710496
  7 ANXcuziKhxusxtthGxPxywY7FLRtmmCwFWDmU5eBDLdH [Writable]                      ◎0.45710496
  8 29cTsXahEoEBwbHwVc59jToybFpagbBMV6Lh45pWEmiK [Writable]                      ◎169677.50203928
  9 EJwyNJJPbHH4pboWQf1NxegoypuY48umbfkhyfPew4E  [Writable]                      ◎0.00203928
 10 So11111111111111111111111111111111111111112                                  ◎152.375037831
 11 SysvarRent111111111111111111111111111111111                                  ◎0.0010092
 12 TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA  [Program]                       ◎0.95318592
 13 DZUEzsxTAkYGosDMMGuLNyGAgDe7doq8hFtu3EfEb8RF                                 ◎0.00203928
 14 11111111111111111111111111111111             [Program]                       ◎0.000000001
 15 9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin [Program]                       ◎0.00114144
```

Figure A.10: Accounts part of the transaction subcommand



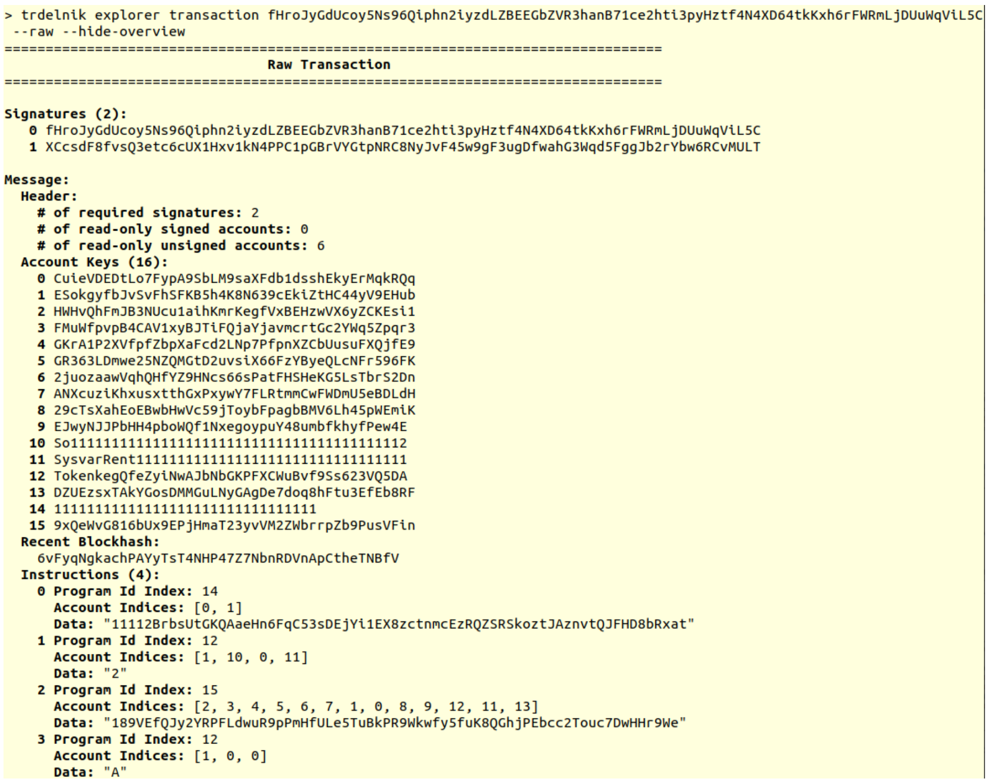
```
Instructions (4):
  0 System Program: CreateAccount
    [11111111111111111111111111111111]
    Lamports: 666457300000
    New Account: "ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub"
    Owner: "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA"
    Source: "CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq"
    Space: 165

  1 SPLToken Program: InitializeAccount
    [TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA]
    Account: "ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub"
    Mint: "So11111111111111111111111111111111111111112"
    Owner: "CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq"
    Rent Sysvar: "SysvarRent111111111111111111111111111111111"

  2 Unknown Program: Unknown Instruction
    [9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin]
    Account 0: HWHvQhFmJB3NUcu1aihKmrKegfVxBEHzwVX6yZCKEsi1
    Account 1: FMuWfpvpB4CAV1xyBJTiFQjaYjavmcrtGc2YWq5Zpqr3
    Account 2: GKrA1P2XVfpfZbpXaFcd2LNp7PfpnXZCbUusuFXQjfE9
    Account 3: GR363LDmwe25NZQMGtD2uvsiX66FzYByeQLcNFr596FK
    Account 4: 2juozaawVqhQHfYZ9HNcs66sPatFHSHeKG5LsTbrS2Dn
    Account 5: ANXcuziKhxusxtthGxPxywY7FLRtmmCwFWDmU5eBDLdH
    Account 6: ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub
    Account 7: CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq
    Account 8: 29cTsXahEoEBwbHwVc59jToybFpagbBMV6Lh45pWEmiK
    Account 9: EJwyNJJPbHH4pboWQf1NxegoypuY48umbfkhyfPew4E
    Account 10: TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA
    Account 11: SysvarRent111111111111111111111111111111111
    Account 12: DZUEzsxTAkYGosDMMGuLNyGAgDe7doq8hFtu3EfEb8RF
    Data: "VRU9WGM23ovJz7sNEDE2eB1rJuedpARMvR82zSkMf4tqdegxQnhas4FpHV1hYtAQ66zsxcSJ3FmUQjsPhAKUcLtPP9rU44"

  3 SPLToken Program: CloseAccount
    [TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA]
    Account: "ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub"
    Destination: "CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq"
    Owner: "CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq"
```

Figure A.11: Instruction part of the transaction subcommand



```
Log Messages (13):
  0 Program 11111111111111111111111111111111 invoke [1]
  1 Program 11111111111111111111111111111111 success
  2 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [1]
  3 Program log: Instruction: InitializeAccount
  4 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 3392 of 200000 compute units
  5 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
  6 Program 9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin invoke [1]
  7 Program 9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin consumed 8472 of 200000 compute units
  8 Program 9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin success
  9 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA invoke [1]
 10 Program log: Instruction: CloseAccount
 11 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA consumed 1713 of 200000 compute units
 12 Program TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA success
```

Figure A.12: Logging part of the transaction subcommand



```
> trdelnik explorer transaction fHroJyGdUcoy5Ns96Qiphn2iyzdLZBEEGbZVR3hanB71ce2hti3pyHztf4N4XD64tkKxh6rFWRmLjDUuWqViL5C
 --raw --hide-overview
================================================================================
                                Raw Transaction
================================================================================

Signatures (2):
   0 fHroJyGdUcoy5Ns96Qiphn2iyzdLZBEEGbZVR3hanB71ce2hti3pyHztf4N4XD64tkKxh6rFWRmLjDUuWqViL5C
   1 XCcsdF8fvsQ3etc6cUX1Hxv1kN4PPC1pGBrVYGtpNRC8NyJvF45w9gF3ugDfwahG3Wqd5FggJb2rYbw6RCvMULT

Message:
  Header:
    # of required signatures: 2
    # of read-only signed accounts: 0
    # of read-only unsigned accounts: 6
  Account Keys (16):
    0 CuieVDEDtLo7FypA9SbLM9saXFdb1dsshEkyErMqkRQq
    1 ESokgyfbJvSvFhSFKB5h4K8N639cEkiZtHC44yV9EHub
    2 HWHvQhFmJB3NUcu1aihKmrKegfVxBEHzwVX6yZCKEsi1
    3 FMuWfpvpB4CAV1xyBJTiFQjaYjavmcrtGc2YWq5Zpqr3
    4 GKrA1P2XVfpfZbpXaFcd2LNp7PfpnXZCbUusuFXQjfE9
    5 GR363LDmwe25NZQMGtD2uvsiX66FzYByeQLcNFr596FK
    6 2juozaawVqhQHfYZ9HNcs66sPatFHSHeKG5LsTbrS2Dn
    7 ANXcuziKhxusxtthGxPxywY7FLRtmmCwFWDmU5eBDLdH
    8 29cTsXahEoEBwbHwVc59jToybFpagbBMV6Lh45pWEmiK
    9 EJwyNJJPbHH4pboWQf1NxegoypuY48umbfkhyfPew4E
   10 So11111111111111111111111111111111111111112
   11 SysvarRent111111111111111111111111111111111
   12 TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA
   13 DZUEzsxTAkYGosDMMGuLNyGAgDe7doq8hFtu3EfEb8RF
   14 11111111111111111111111111111111
   15 9xQeWvG816bUx9EPjHmaT23yvVM2ZWbrrpZb9PusVFin
  Recent Blockhash:
    6vFyqNgkachPAYyTsT4NHP47Z7NbnRDVnApCtheTNBfV
  Instructions (4):
    0 Program Id Index: 14
      Account Indices: [0, 1]
      Data: "11112BrbsUtGKQAaeHn6FqC53sDEjYi1EX8zctnmcEzRQZSRSkoztJAznvtQJFHD8bRxat"
    1 Program Id Index: 12
      Account Indices: [1, 10, 0, 11]
      Data: "2"
    2 Program Id Index: 15
      Account Indices: [2, 3, 4, 5, 6, 7, 1, 0, 8, 9, 12, 11, 13]
      Data: "189VEfQJy2YRPFLdwuR9pPmHfULe5TuBkPR9Wkwfy5fuK8QGhjPEbcc2Touc7DwHHr9We"
    3 Program Id Index: 12
      Account Indices: [1, 0, 0]
      Data: "A"
```

Figure A.13: Transaction subcommand with --raw switch

Appendix **B**

Acronyms

CLI Command-Line Interface

CLR Common Language Runtime

DeFi Decentralized Finance

DEX Decentralized Exchange

eDSL Embedded Domain Specific Language

EVM Ethereum Virtual Machine

IDL Interface Description Language

JSON JavaScript Object Notation

JVM Java Virtual Machine

NFT Non-Fungible Token

P2P Peer-to-Peer

PoS Proof-of-Stake

PoW Proof-of-Work

RPC Remote Procedure Call

SDK Software Development Kit

SOL Solana Coin

TPS Transactions per Second

VM Virtual Machine

Appendix C

Contents of Enclosed SD Card

```
README.txt .................. the file with SD Card contents description
src ................................... the directory with source codes
  trdelnik ........... the directory with the whole Trdelnik workspace
    Cargo.toml.....................Trdelnik workspace definition file
    crates ..................... the directory with all Trdelnik crates
      cli ..................................... Trdelnik CLI crate
      client.................................Trdelnik Client crate
      explorer............................Trdelnik Explorer crate
      test...................................Trdelnik Test crate
    HOWTO.md...............................Installation instructions
text ........................................ the thesis text directory
  latex.............the directory with LaTeX source codes of the thesis
  thesis.pdf..........................the thesis text in PDF format
```